從你口袋中的手機談編譯器技術的多元應用

How Compilers are Used

Jim Huang (黃敬群) <jserv.tw@gmail.com>

Aug 1, 2016

關於黃敬群（@jserv)

- 成功大學資訊工程系 / 兼任專家
- 交通大學資訊工程系 / 兼任教師
- 從事消費性電子產品開發十餘年
  近年投入工業控制領域
  - 台達電子 / 顧問
  - 台灣工研院資通所 / 顧問
  - 台灣聯發科技 / 顧問
  - 台灣晶心科技 / 幕僚工程師

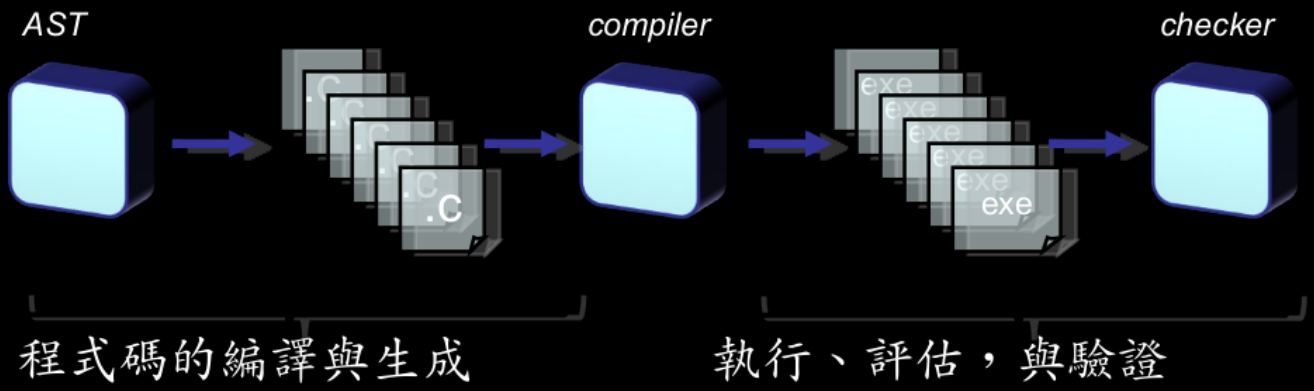
多項世界級開源項目開發者

- 新酷音輸入法
- Android Open Source Project
- GCC / GNU Classpath
- LXDE
- Kaffe
- pcman(x)
- Linaro

本簡報的目的

- 小小一只 Android 手機，裡頭竟然自帶 8 種以上動態編譯器和虛擬機
- 程式語言理論到實踐
- 理解從 C 程式原始碼到二進制的過程，從而成為電腦的主人
- 引導不具備編譯器背景知識的聽眾，得以對編譯器最佳化技術有概念

編譯器流程::JIT



建立 IR
載入必要的函式庫
連結程式模組
Optimizations + Transforms
codegen

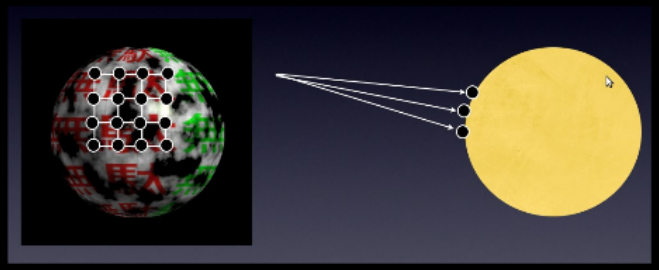
3D/Shader 的考量點

光

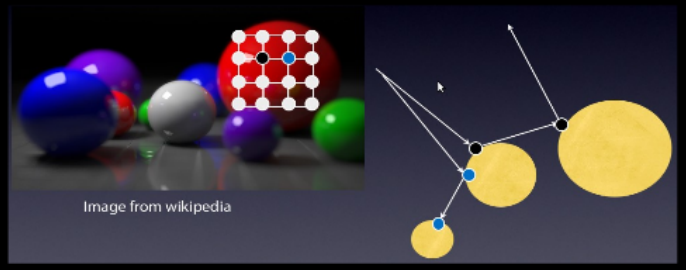
反射

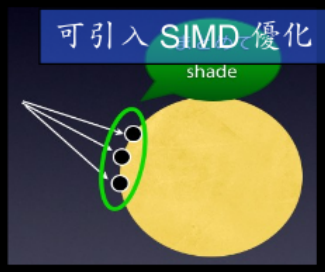
移動的演算法

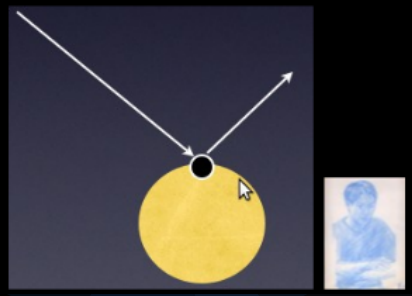
Reyes(scanline,polygon)



Raytracing





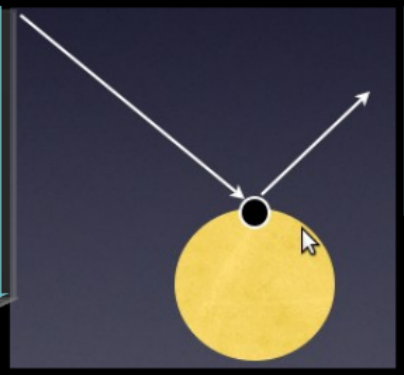


```
surface
matte(float Ka = 1.0; float Kd = 1.1;)
{
  normal Nf = faceforward(
              normalize(N), I);
  Oi = Os;
  Ci = Os * Cs * (Ka * ambient() +
                  Kd * diffuse(Nf));

}
```

push N
normalize
push I
faceforward
...

```
surface
matte(float Ka = 1.0; float Kd = 1.1;)
{
  normal Nf = faceforward(
              normalize(N), I);
  Oi = Os;
  Ci = Os * Cs * (Ka * ambient() +
                  Kd * diffuse(Nf));
}
```



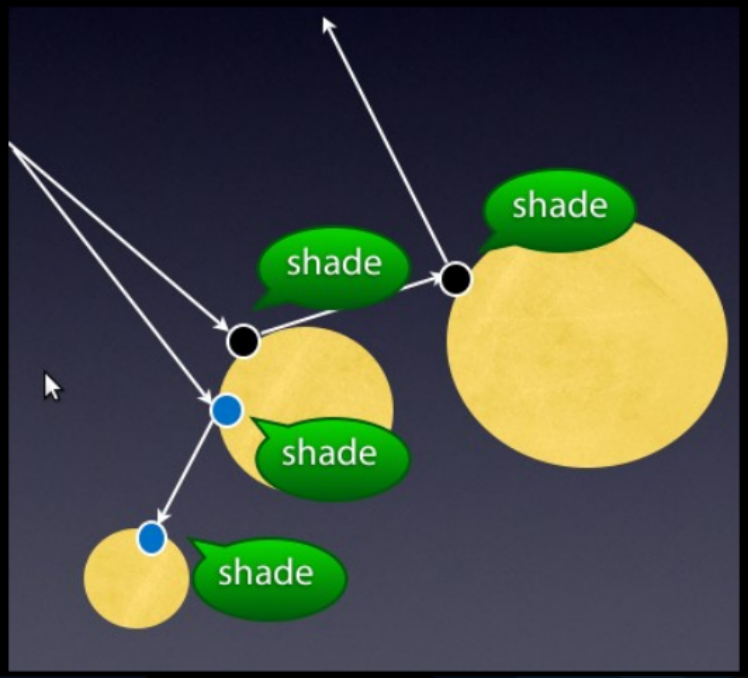
```
push N
normalize
push I
faceforward
...
```

Raytracing 的難題

無法善用 SIMD

運算相依性高且繁瑣

需要動態調整快速運算的路徑



Specialize 技巧

以 color space 轉換來說，相當大量且繁瑣的運算，如
BGRA 444R --> RGBA 8888

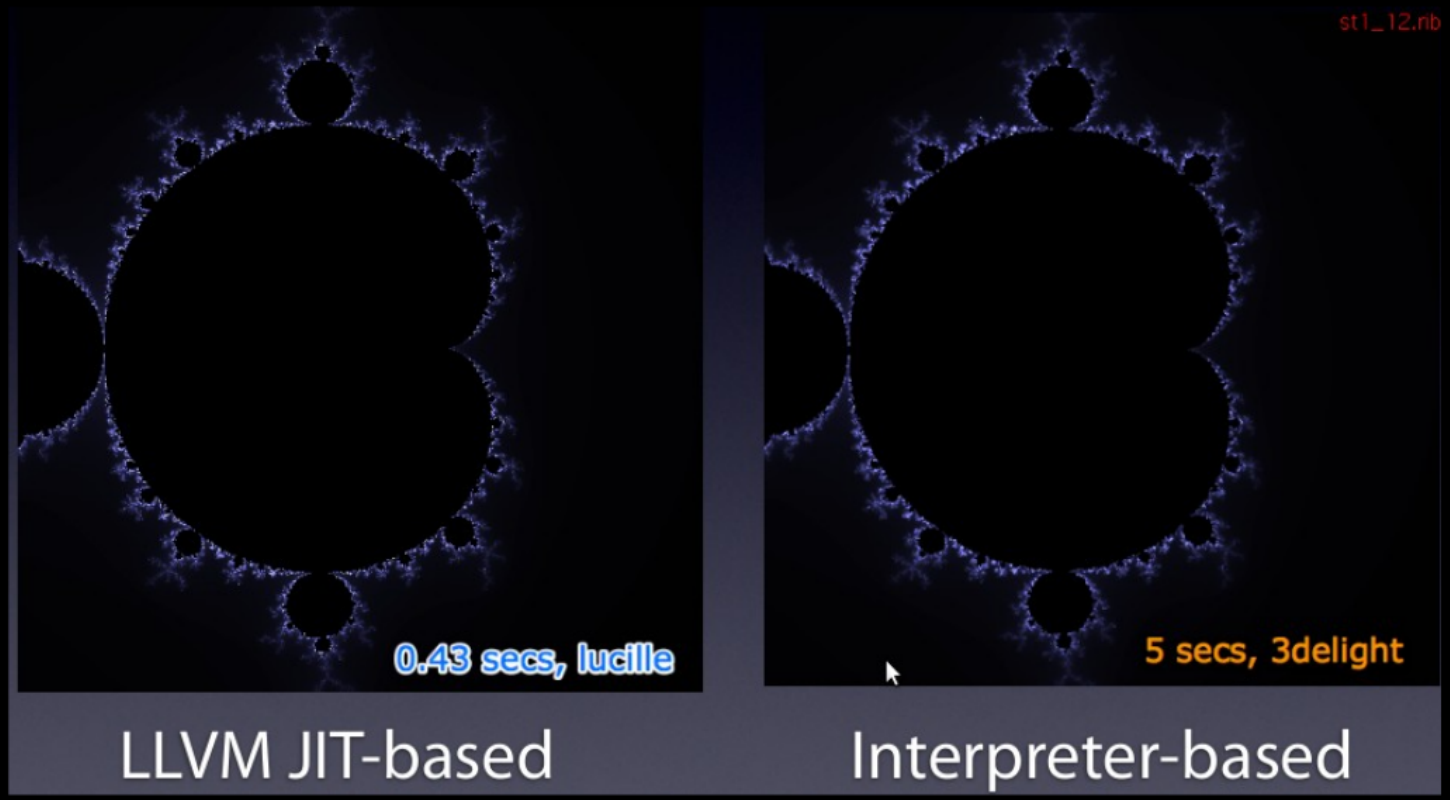
```
for each pixel {
  switch (infmt) {
  case RGBA 5551:
    R = (*in >> 11) & C
    G = (*in >> 6) & C
    B = (*in >> 1) & C
  ... }
  switch (outfmt) {
  case RGB888:
    *outptr = R << 16 |
              G << 8 ...
  }
}
```

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 on average, 19.3x max speedup: (13.3MB/s to 257.7MB/s)



Mandelbort 碎形運算透過 LLVM JIT 後，
提昇效能達到 11 倍

Programming Language Theory (PLT)

- Essential component of a programming language: type theory, variable scoping, language semantics, etc.
- How do people reason and compose a program?
- Create an abstraction that is understandable to human and tracable to computers.

Example: Dynamic Scoping in Bash

```
#!/bin/sh
v=1                     # Initialize v with 1
foo () {
  echo "foo:v=${v}"     # Which v is referred?
  v=2                   # Which v is assigned?
}
bar () {
  local v=3
  foo
  echo “bar:v=${v}”     # What will be printed?
}
v=4                     # Assign 4 to v
bar
echo “v=${v}”           # What will be printed?
```

Example: Dynamic Scoping in Bash

```
#!/bin/sh
v=1                          # Initialize v with 1
foo () {
  echo "foo:v=${v}"          # Which v is referred?
  v=2                        # Which v is assigned?
}
bar () {
  local v=3
  foo
  echo “bar:v=${v}”          # What will be printed?
}
v=4
bar
echo “v=${v}”                # What will be printed?
```

foo:v=3

Surprisingly, foo is accessing local v in bar instead of the global v.

bar:v=2

v=4

Example: Subtype and Mutable Records

- Why you can't perform following conversion in C++?

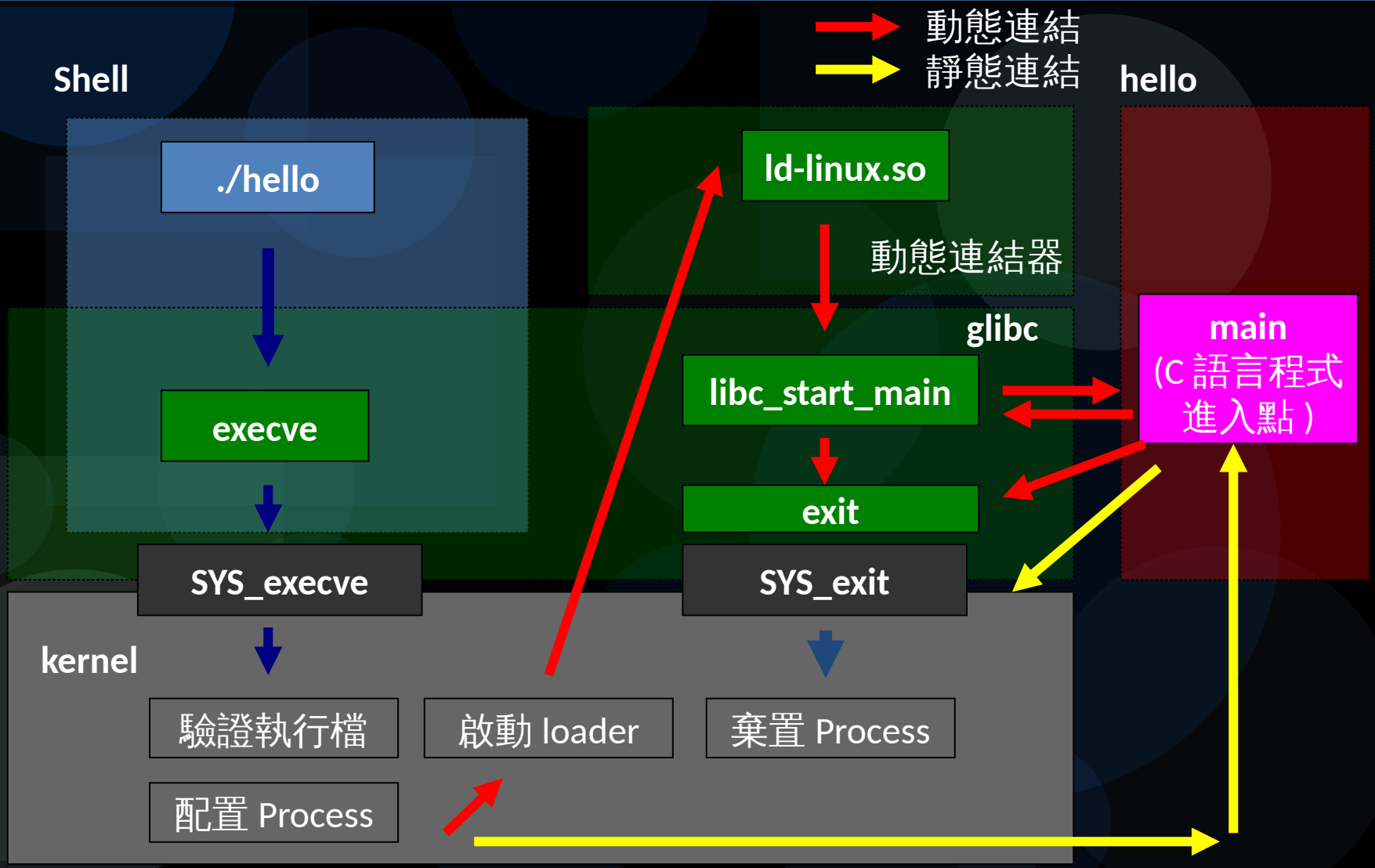
```
void test(int *ptr) {
  int **p = &ptr;
  const int** a = p;  // Compiler gives warning
  // ...
}
```

Example: Subtype and Mutable Records

- This is related to covariant type and contravarience type. With PLT, we know that we can only choose two of
  - (a) covariant type
  - (b) mutable records
  - (c) type consistency

```
void test(int *ptr) {
  const int c = 0;
  int **p = &ptr;
  const int** a = p;  // If it is allowed,
                      // bad programs will pass.
  *a = &c;
  *p = 5;  // No warning here.
}
```

現代 GNU/Linux 環境，程式執行流程



執行 execve 系統呼叫

在 shell 中執行 `hello` 這個執行檔時，實際上 shell 會呼叫 `fork` + `execve` 系統呼叫，讓核心準備建立新的 process

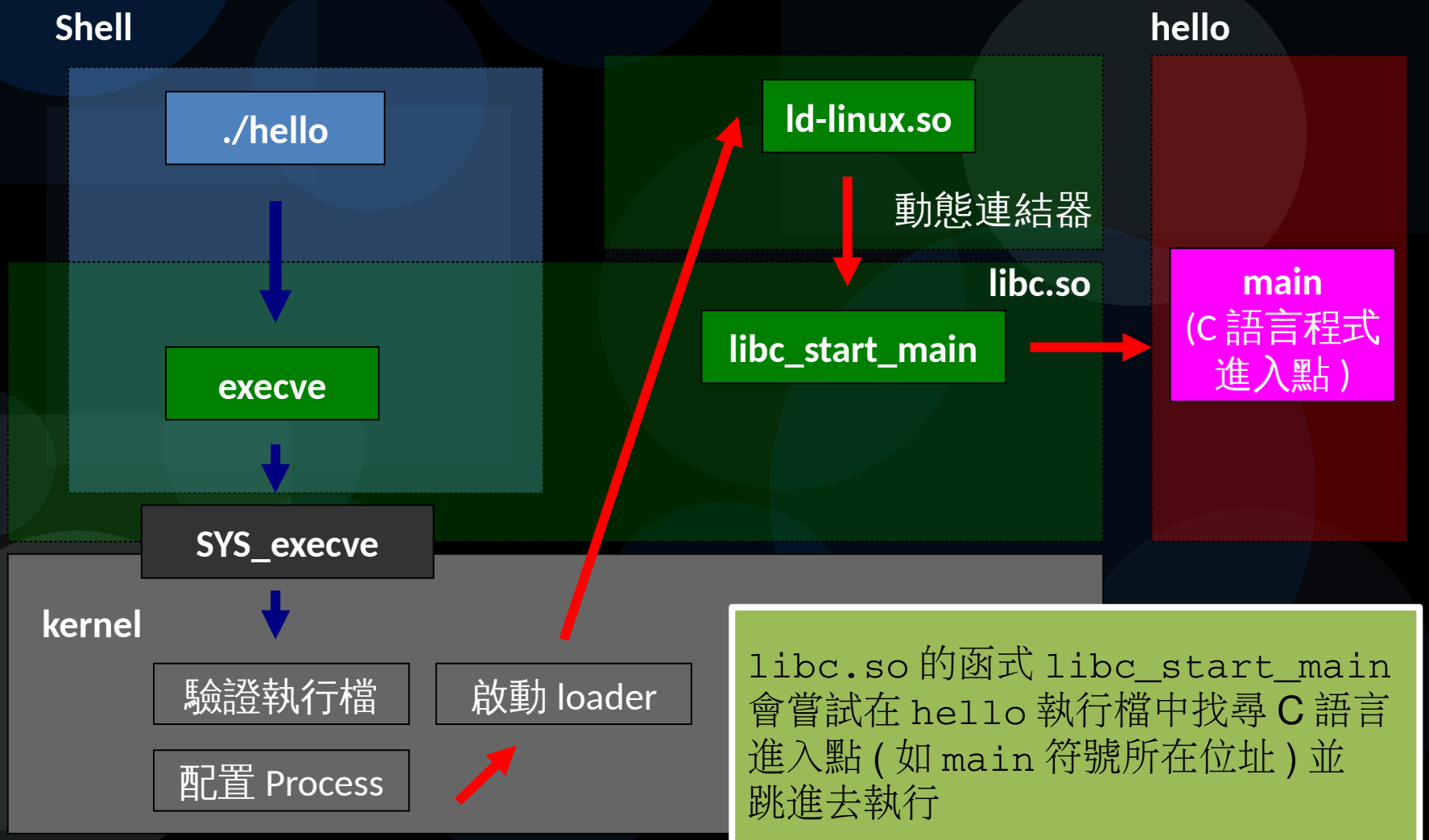
Tip: 請先參考 **UNIX** 作業系統文件，以便理解 **UNIX process** 運作原理以及 **fork, execve, exit** 等等系統呼叫

驗證執行檔並呼叫動態連結器

一旦執行檔被核心驗證完畢並且准許配置 process 空間後，會依據執行檔的資訊，載入對應的動態連結器 (dynamic linker，圖例為 `ld-linux.so`)，並載入對應的動態函式庫，如 `libc.so` (包含 C 語言執行時期支援 ;C Runtime)

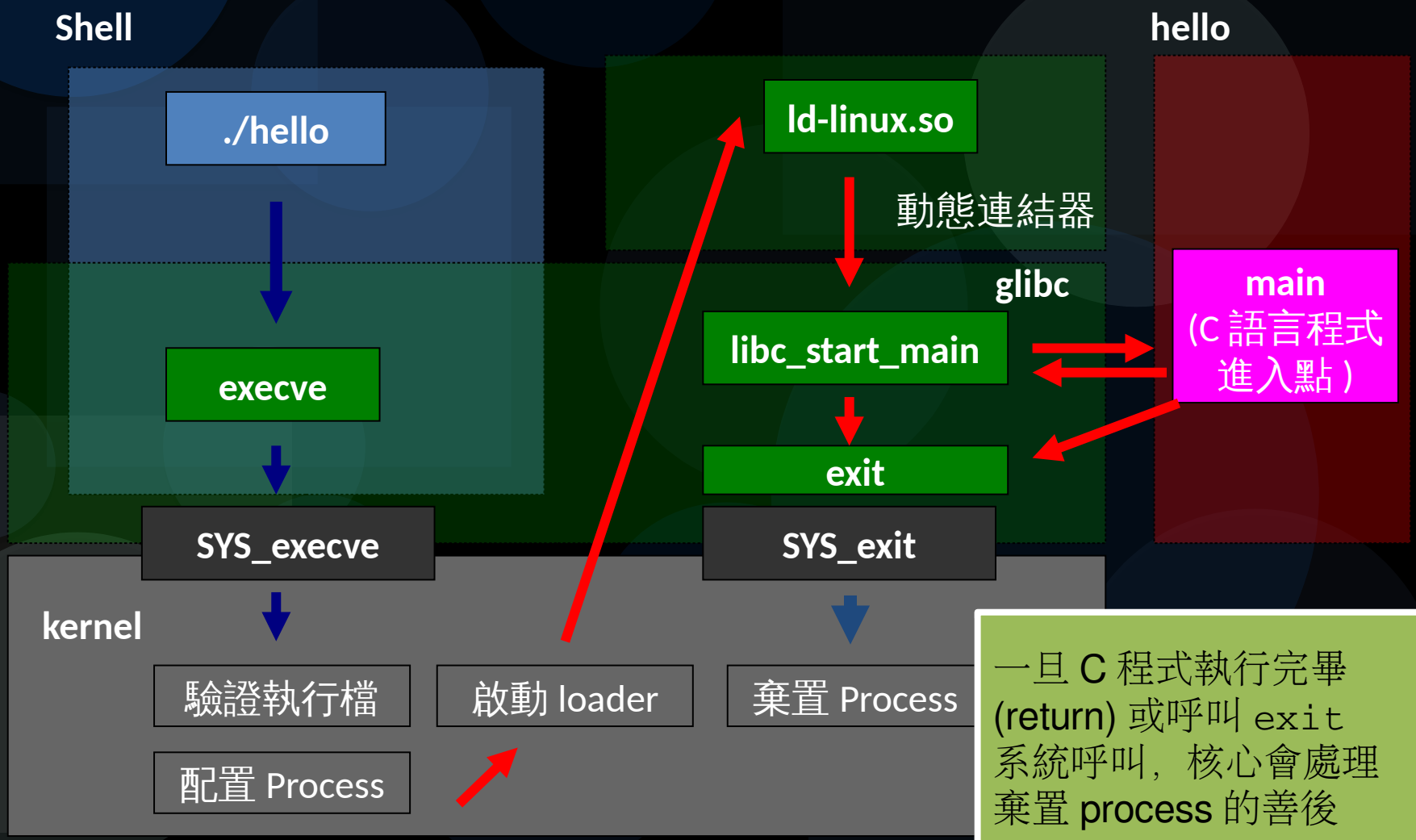
`ld-linux.so` 和 `libc.so` 在 GNU/Linux 上，一般由 glibc 提供

由 C 執行時期跳入真正的程式進入點



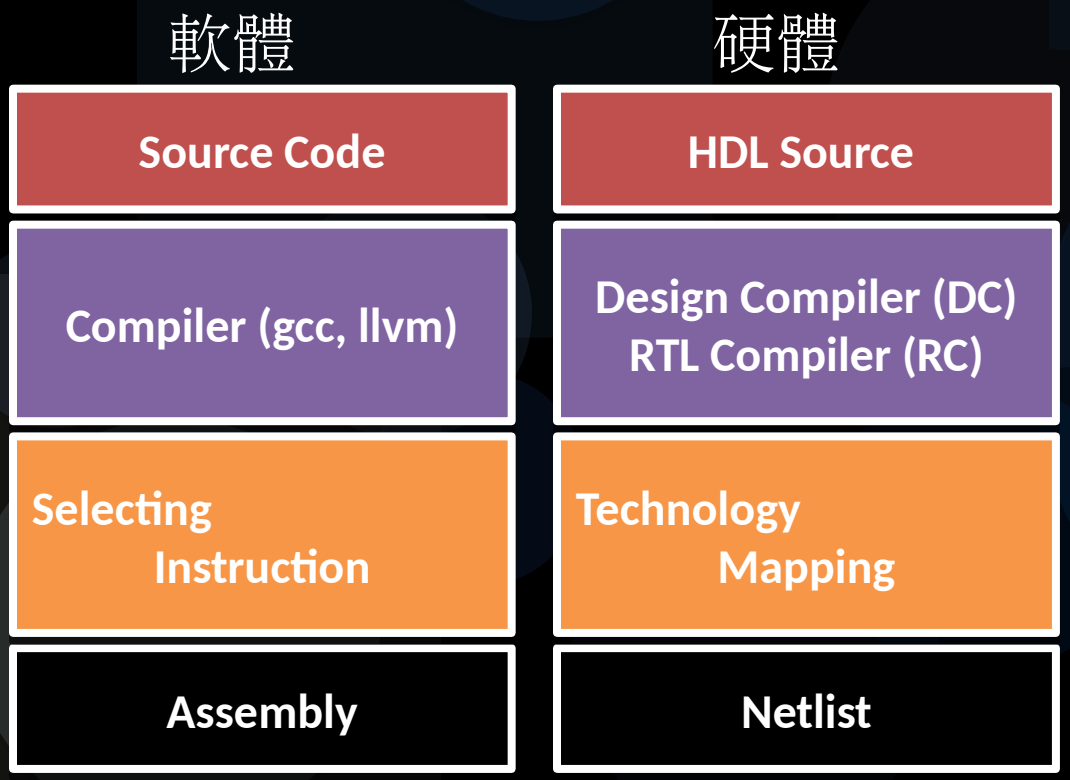
libc.so 的函式 libc_start_main 會嘗試在 hello 執行檔中找尋 C 語言進入點 (如 main 符號所在位址) 並跳進去執行

C 語言程式結束並呼叫 exit 系統呼叫



從原始碼到二進制：常見的誤解

- 從原始碼到二進制的過程，不僅需要 Compiler
- 還要有很多工具介入才可完成： Toolchain
- 真正的 Compiler 其實只做以下：



A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code)

source : http://en.wikipedia.org/wiki/Compiler

編譯 Hello world 程式的流程

三大步驟

編譯

組譯

連結

圖解編譯流程 (1/5)

```
#include <stdio.h>
int main() {
  printf (“Hello World”);
  return 0;
}
```

- 給定一個普通的輸出 Hello World 的程式
- 編譯方式：

  gcc -o hello hello.c（或 ）**clang -o hello hello.c**
- 檢驗正確輸出 ELF 格式的執行檔
  - **file hello**
  - 預期輸出：

    hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs)
- 執行方式：
  - ./hello

圖解編譯流程 (2/5)

```
#include <stdio.h>
int main() {
  printf ("Hello World");
  return 0;
}
```

stdio.h

```
int printf (const char* fmt, …);
```

Pre-process

```
int main() {
  printf("Hello World");
  return 0;
}
```

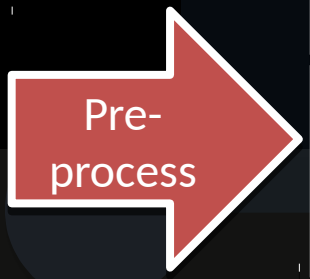
- C preprocessor 參照標頭檔 `stdio.h` 的內容，展開 macro 和驗證 prototype，因此輸出的結果就不會再見到 `#include` 字樣

圖解編譯流程 (3/5)

```
#include <stdio.h>
int main() {
  printf ("Hello World");
  return 0;
}
```

stdio.h

```
int printf (const char* fmt, …);
```




```
int main() {
  printf ("Hello World");
  return 0;
}
```


```
main:
LDR R1, [R2 + 12]
BAL printf
```

- C compiler 輸出對應的組合語言，上圖針對 ARM 架構
- BAL 是 ARM 的一種跳躍指令，後方緊接著 printf 的位址
- 編譯階段不知道 printf 的位址，所以暫時填入 printf 符號名稱

圖解編譯流程 (4/5)

```
#include <stdio.h>
int main() {
  printf ("Hello World");
  return 0;
}
```

stdio.h

```
int printf (const char* fmt, …);
```

Pre-process

```
int main() {
  printf ("Hello World");
  return 0;
}
```

Compile

```
main:
LDR R1, [R2 + 12]
BAL printf
```

Assemble

```
main:
    EC 00 00 12
    F0 ?? ?? ??
```

- ARM assembler 依據編譯器輸出的組合語言，組譯為對應的 ARM 機械碼
- 此時仍不知道 printf 的位址，所以暫時不填入

圖解編譯流程 (5/5)

```
#include <stdio.h>
int main() {
  printf ("Hello World");
  return 0;
}
```

stdio.h

```
int printf (const char* fmt, …);
```

Pre-process

```
int main() {
  printf ("Hello World");
  return 0;
}
```

Compile

```
main:
LDR R1, [R2 + 12]
BAL printf
```

Assemble

```
main:
   EC 00 00 12
   F0 ?? ?? ??
```

libc.a

Linking

```
0x1000 <printf>:
   EC 00 00 12
   …
```

```
0x2000 <main>:
   EC 00 00 12
   F0 00 10 00
```

- printf 實做於 libc.a(C 語言標準函式庫的靜態版本)，其位址為 0x1000， Linker 重新配置 (relocate)

常見的誤解： GCC 是個 C 編譯器
事實： GCC 是個 Compiler Driver

gcc
binutils
glibc

gcc

Source (`.c` and `.h`)

cpp → Preprocessed source (`.i` or `.ii`)

cc1 → Assembly (`.s`)

as → Relocatable (`.o`)

collect2

library
crt*
link script

ld → Binary (Executable)

GCC 這個執行檔，可當 Compiler，也能當 Preprocessor，也能當 Assembler，更能當 Linker

編譯器簡史

（程式語言不是年代較早，就比較低階。 1970 年代才出現的 C 語言遠比十幾年前就出現的 Fortran 與 COBOL 低階）

- First compiler (1952)
  - 第一個**編譯器**
  - A-0, **Grace Murray Hopper**
- First complete compiler (1957)
  - 第一個**完整**的編譯器
  - Fortran, **John Backus** (18-man years)
- First multi-arch compiler (1960)
  - 第一個**多平台支援**的編譯器
  - COBOL, **Grace Murray Hopper** el al.
- The first self-hosting compiler (1962)
  - 第一個**能自己編譯自己**的編譯器
  - LISP, Tim Hart and Mike Levin

Grace Hopper, inventor of A-0, COBOL, and the term “compiler.”

Image source:
http://www-history.mcs.st-and.ac.uk/PictDisplay/Hopper.html
http://upload.wikimedia.org/wikipedia/commons/thumb/5/55/Grace_Hopper.jpg/300px-Grace_Hopper.jpg

先有雞還是先有蛋?

- 先有 C 語言還是先有 C 編譯器?
- 故事： Ken Thompson – Reflections on Trusting Trust
  - (Compiling a compiler)
    在 C compiler(事實上是 pre-processor) 中放了一個後門，若 compiler 發現它所處理的程式是 login 的原始程式，它就會在裡面加入一段程式碼，讓 Ken Thomposon 可以任意 login 進去該主機。由於 UNIX 原始是碼得用 C compiler 來編譯，這意味著 Thompson 將可登入任何安裝 UNIX 的電腦
  - 那換個乾淨的 compiler 總可以吧？但新 compiler 的原始碼也得由舊 compiler 來編譯， 而為了讓這個特洛伊木馬能流傳久遠， 如果舊 compiler 發現自己正處理的的是另一個 (正常的) C compiler， 它就會把前述那段「如果是 login ，就加入一段…」的程式碼，注入到這個新的 compiler 裡面

Bootstrapping & Cross-compilation

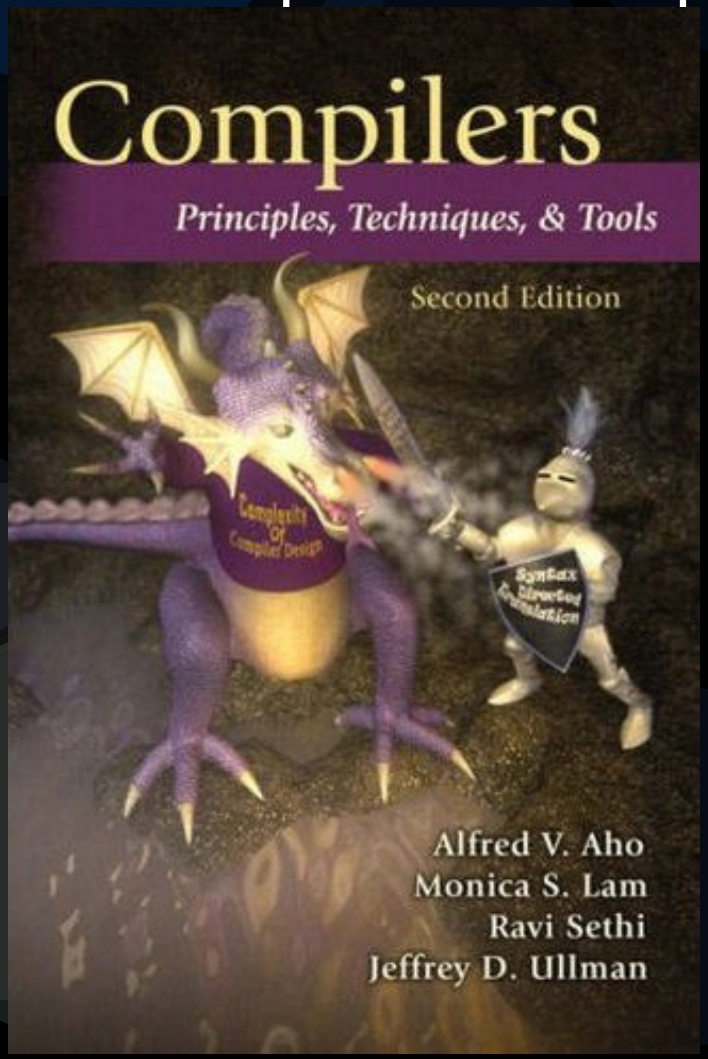
- 原有系統的內建 C compiler 可能較簡陋或老舊，因此我們想把新的 compiler 用舊的 compiler 編譯，然後當成系統內建的 compiler 用
- 先有雞還是先有蛋?
  - 用 X 語言（像是既有的 Fortran，或者直接用組合語言）寫一個 C-（比 C 語言更簡化的語言）的編譯器
  - 用 C- 寫一個 C 的編譯器
  - 用 C 寫一個 C 的編譯器

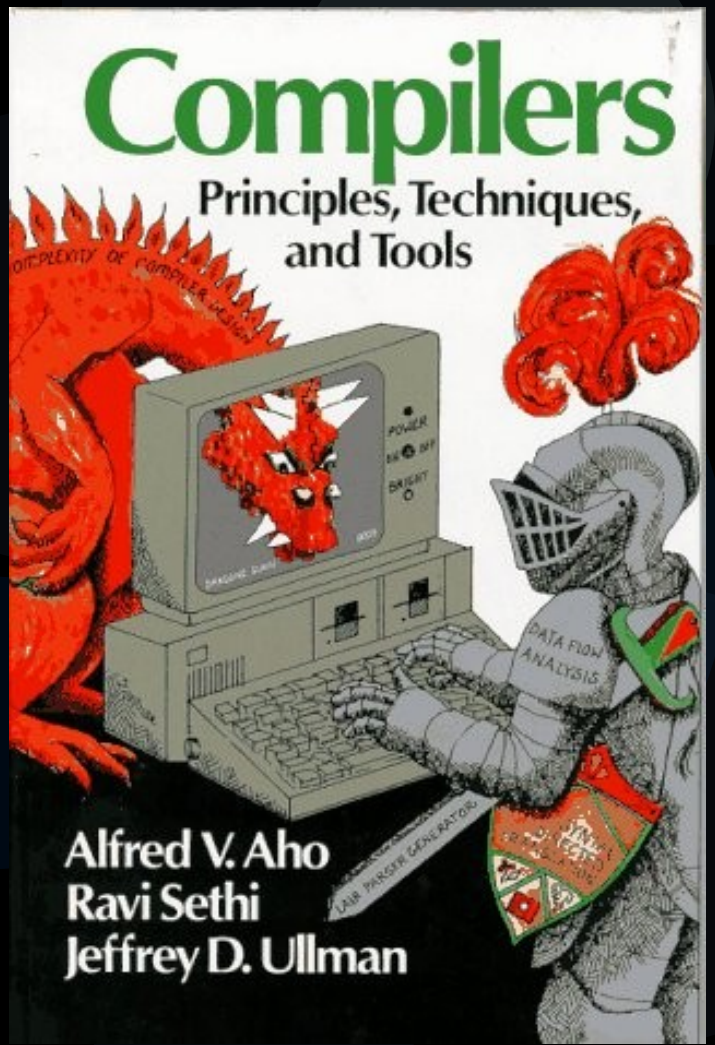
Oracle JDK 中，Java 編譯器 javac 就是用 Java 語言撰寫，甚至有 Maxine Research VM 這樣主要用 Java 撰寫的 Java 虛擬機器

相傳在十八世紀，德國 Baron Münchhausen 男爵常誇大吹噓自己的英勇事蹟，其中一項是「拉著自己的頭髮，將自己從受陷的沼澤中提起」，此事後來收錄於《吹牛大王歷險記》，改寫為「用拔靴帶把自己從海中拉起來」，其中「拔靴帶」（bootstrap）指的是長統靴靴筒頂端後方的小環帶，是用以輔助穿長統靴。這種有違物理原理的誇大動作，卻讓不同領域的人們獲得靈感，商業上，bootstrapping 引申為一種創業模式，即初期投入少量的啓動資本，然後在創業過程中主要依靠從客戶得來的銷售收入，形成一個良好的正現金流。在資訊領域，因開機過程環環相扣，得先透過簡單的程式讀入記憶體，執行後又載入更多磁區、程式碼來執行，直到作業系統完全載入為止，因此開機過程也稱 bootstrapping，簡稱 "boot"

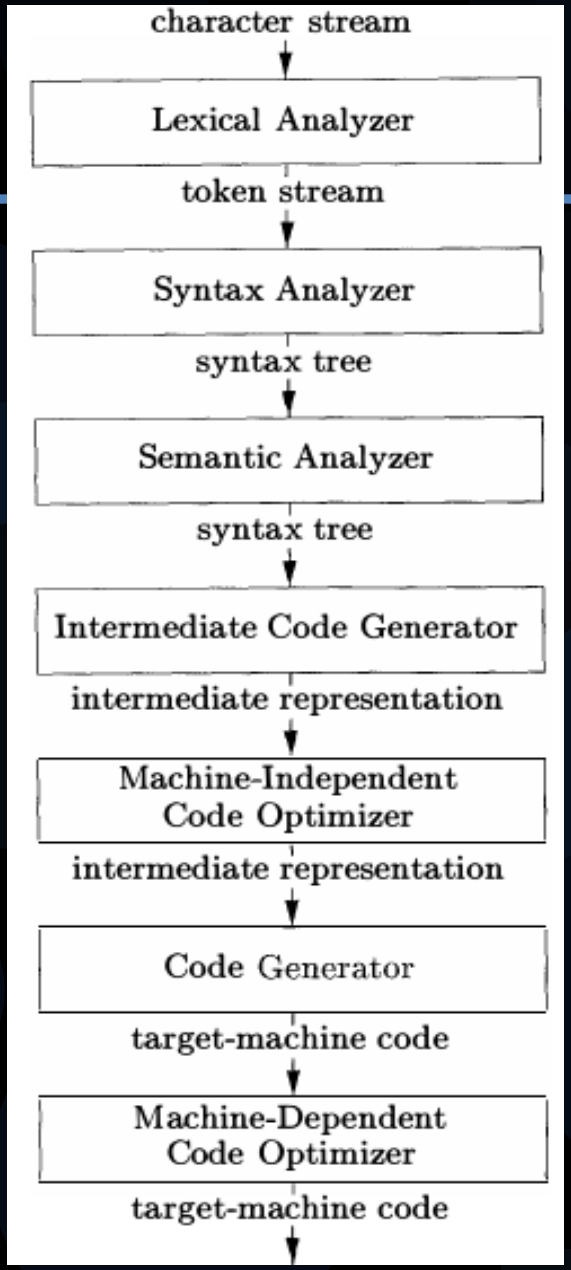
編譯器教科書

《Compilers: Principles, Techniques, & Tools》



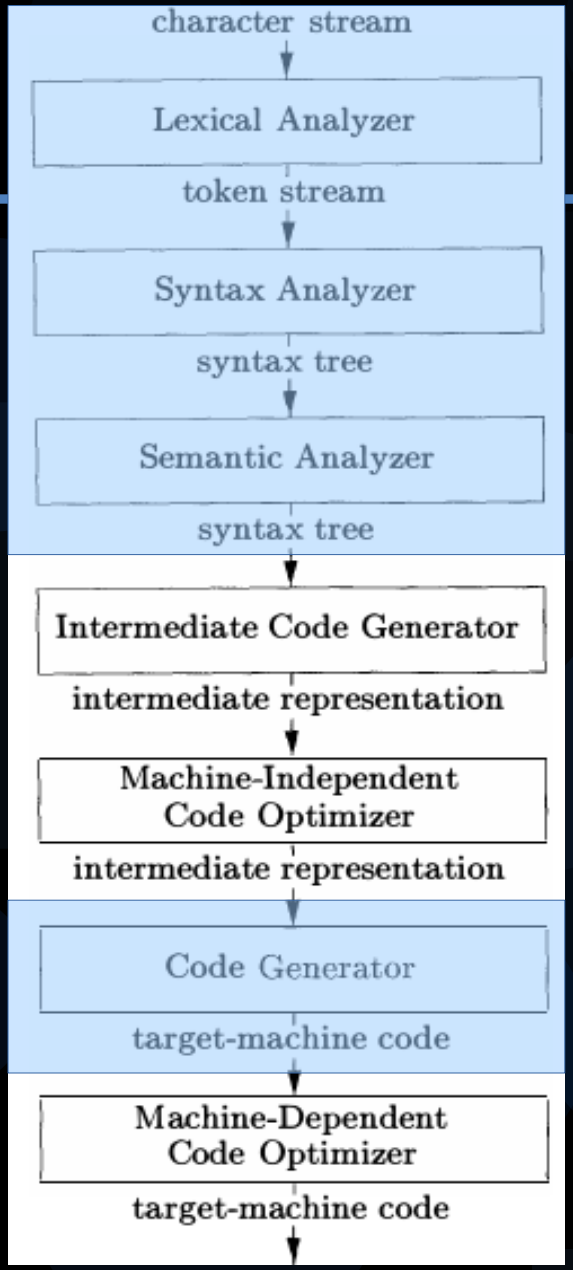


編譯流程



Source: Compilers: Principles, Techniques, and Tools (2nd Edition) p.5

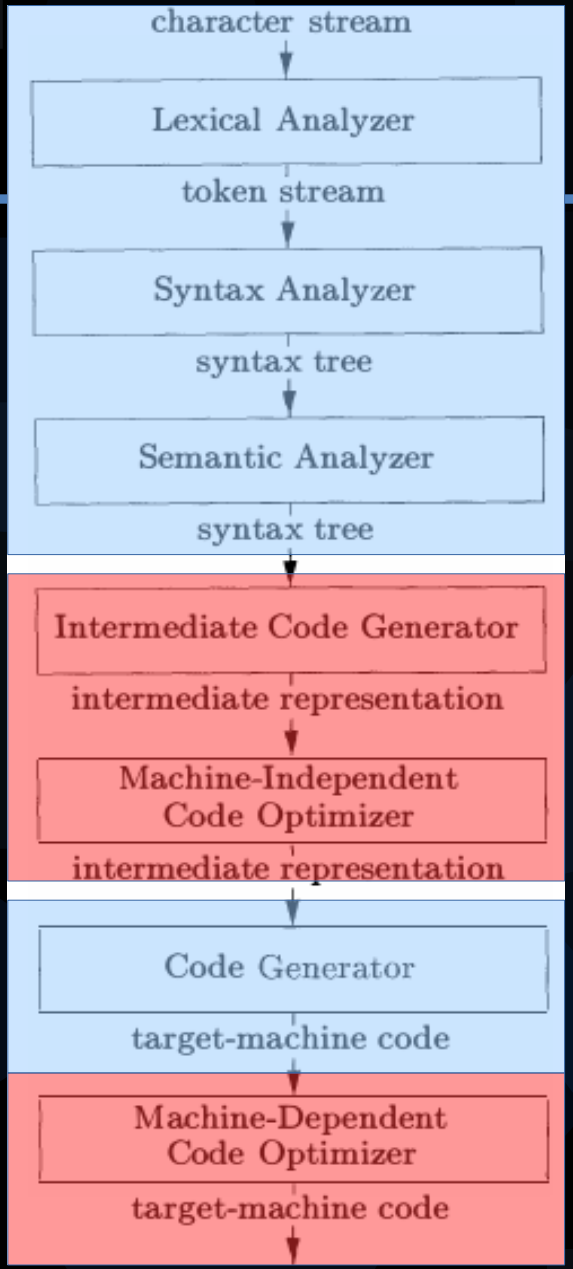
編譯流程



通常大學部編譯器課程僅涵蓋 parser 的部份

以及陽春的 code generator

編譯流程

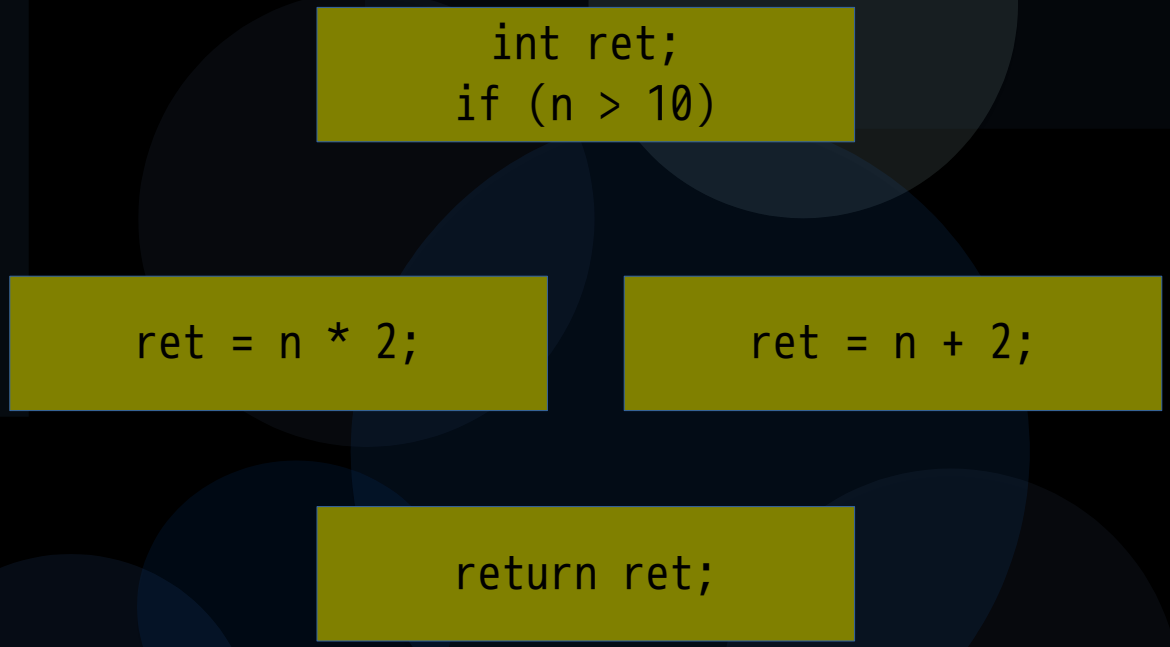


但 Compiler 引人入勝之處，
其實都在最佳化的地方

編譯器術語： Basic Block

- 單一進入點、單一出口點的程式區段

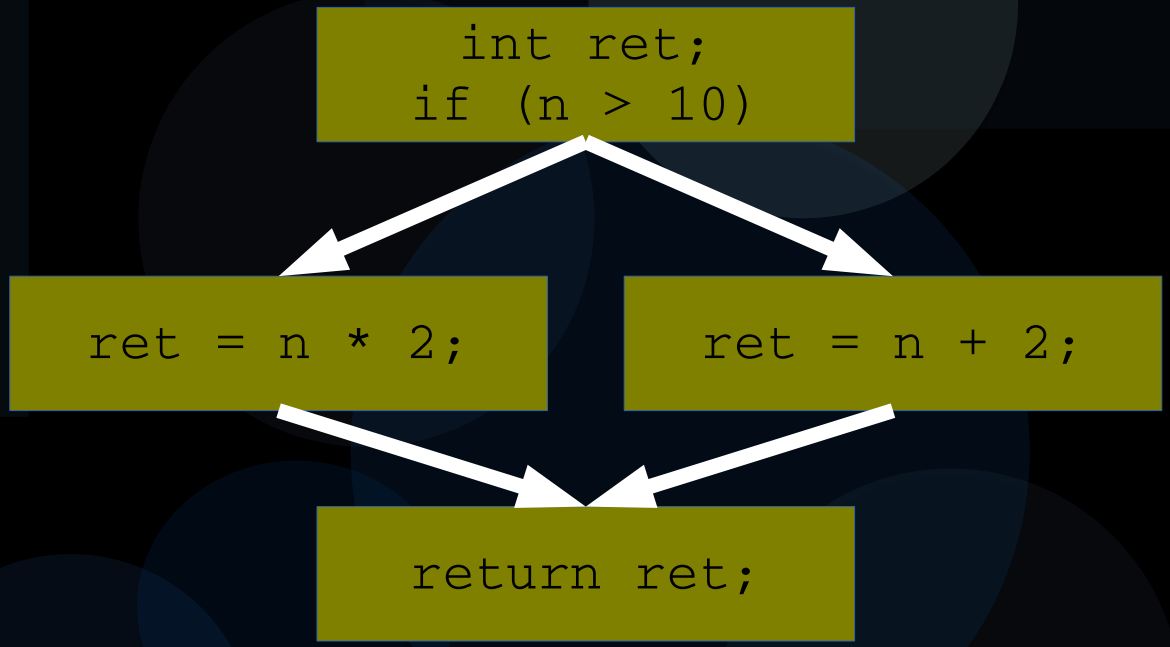
```
int foo (ini n)
{
  int ret;
  if (n > 10)
    ret = n * 2;
  else
    ret = n + 2;
  return ret;
}
```



編譯器術語： CFG (Control Flow Graph)

- 簡單來說就是程式的流程圖

```
int foo (ini n)
{
  int ret;
  if (n > 10)
    ret = n * 2;
  else
    ret = n + 2;
  return ret;
}
```



Basic Block vs. CFG

Basic Block

```
int ret = 0;
int i;
```

```
i = 0;
```

```
i < n;
```

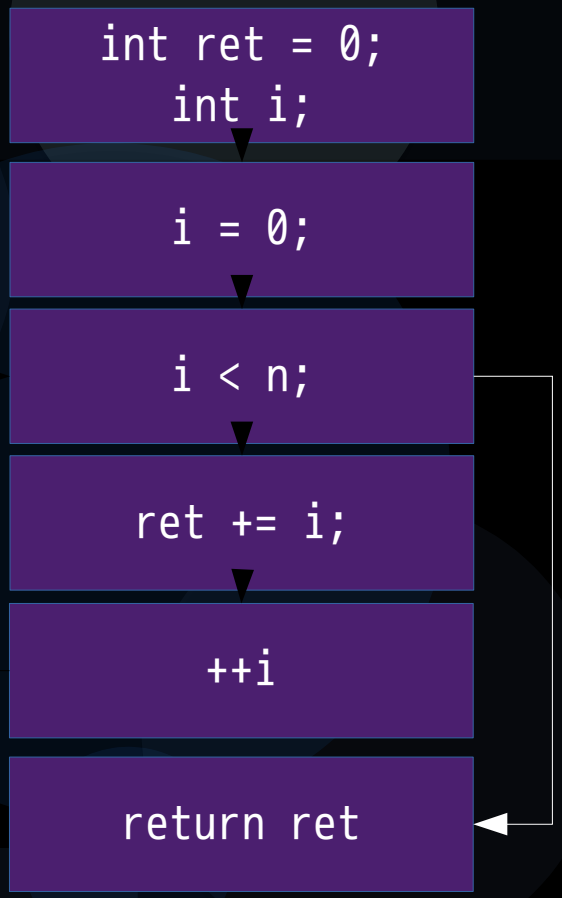
```
ret += i;
```

```
++i
```

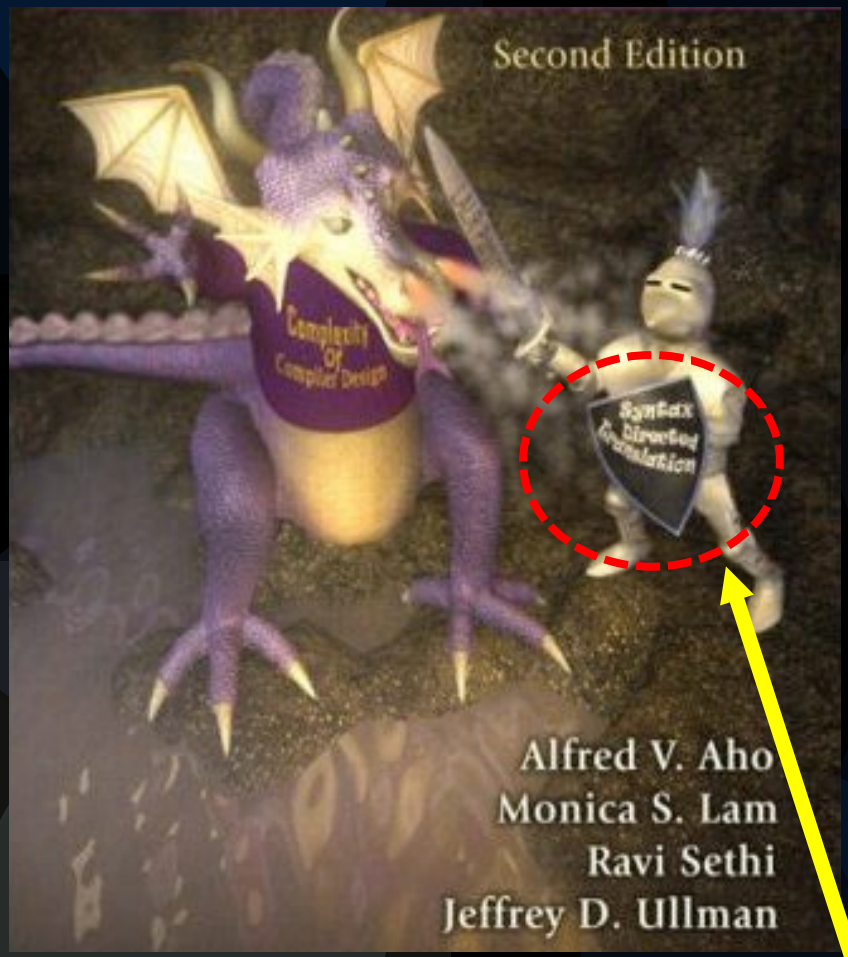
```
return ret
```

```
int sum (int n)
{
  int ret = 0;
  int i;
  for (i = 0; i < n; ++i)
    ret += i;
  return ret;
}
```

CFG



教科書回顧



- 原始碼
- 字彙分析 (正規語言)
- 語法分析 (Context-Free Grammar)
- 語法樹
- 語意分析 (Type Checking ..etc)
- 中間語言
- 最佳化
- 目標語言

屠龍刀： Syntax Directed Translator

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

- T_While
- T_LeftParen
- T_Identifier y
- T_Less
- T_Identifier z
- T_RightParen
- T_OpenBrace
- T_Int
- T_Identifier x
- T_Assign
- T_Identifier a
- T_Plus
- T_Identifier b
- T_Semicolon
- T_Identifier y
- T_PlusAssign
- T_Identifier x
- T_Semicolon
- T_CloseBrace

T_ 開頭表示 token

原始碼

字彙分析 (正規語言)

語法分析 (Context-Free Grammar)

語法樹

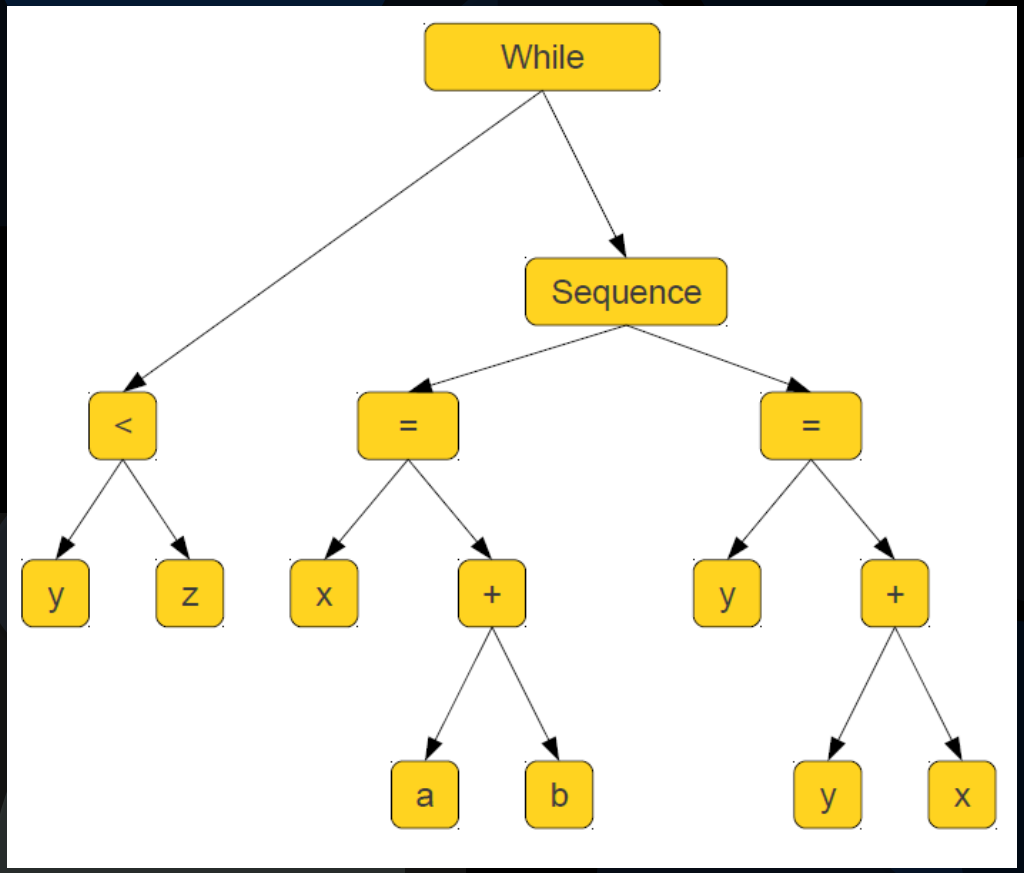
語意分析 (Type Checking ..etc)

中間語言

最佳化

目標語言

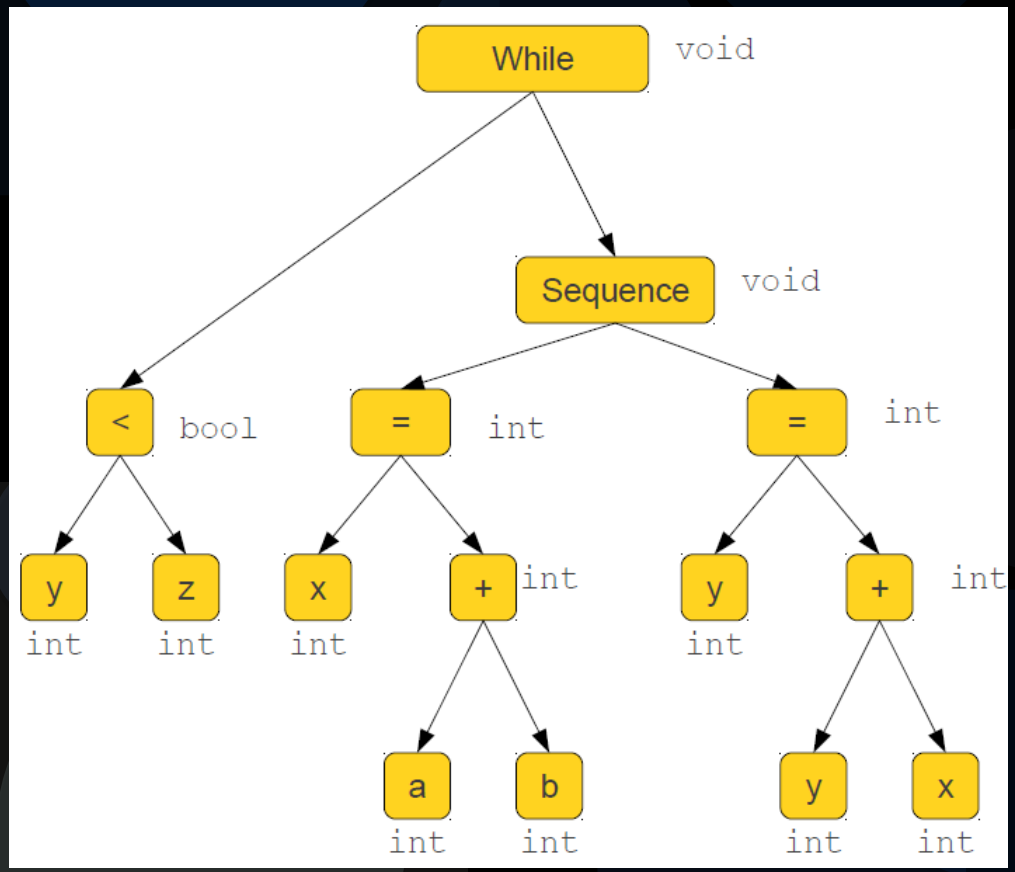
```
while (y < z) {
    int x = a + b;
    y += x;
}
```



- 原始碼
- 字彙分析 (正規語言)
- 語法分析 (Context-Free Grammar)
- 語法樹
- 語意分析 (Type Checking ..etc)
- 中間語言
- 最佳化
- 目標語言

Source: http://www.stanford.edu/class/cs143/

```
while (y < z) {
    int x = a + b;
    y += x;
}
```



原始碼

字彙分析 (正規語言)

語法分析 (Context-Free Grammar)

語法樹

語意分析 (Type Checking ..etc)

中間語言

最佳化

目標語言

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
Loop: x = a + b
      y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

```
      x = a + b
Loop: y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

```
      add $1, $2, $3
Loop: add $4, $1, $4
      slt $6, $1, $5
      beq $6, loop
```

原始碼

字彙分析 (正規語言)

語法分析 (Context-Free Grammar)

語法樹

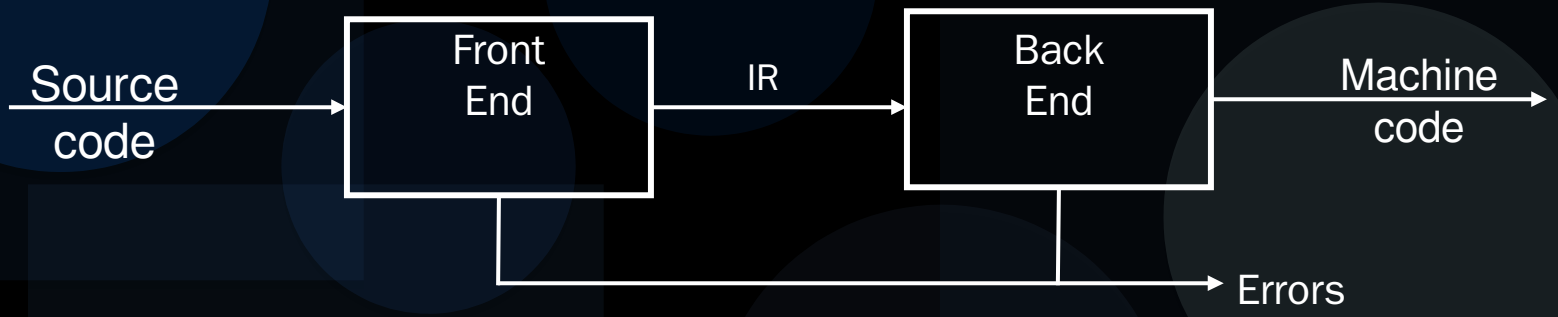
語意分析 (Type Checking ..etc)

中間語言

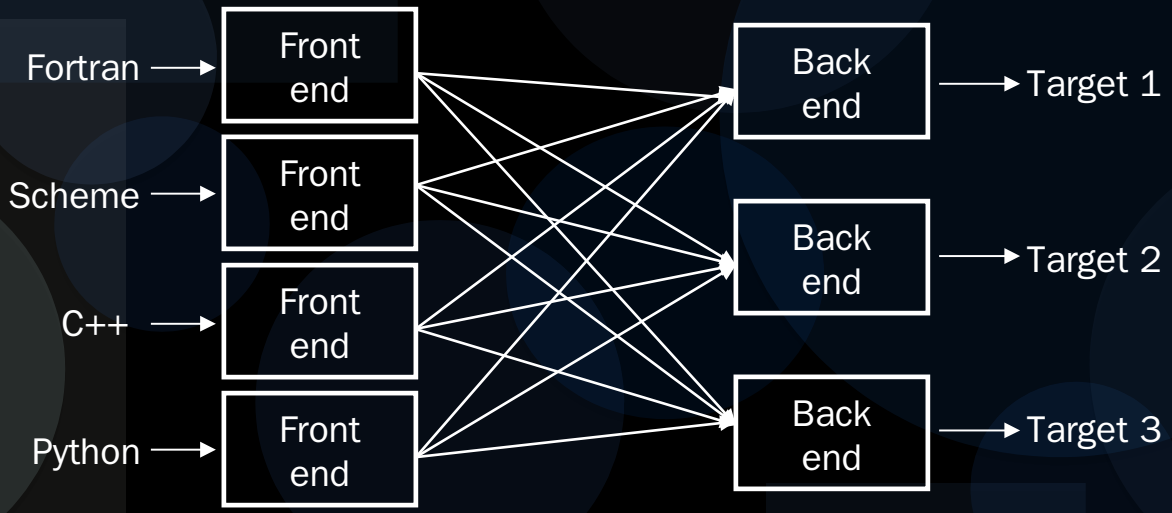
最佳化

目標語言

IR (Intermediate Representation)



引入 IR 後，可以更容易支援各式前端、後端，以及重用最佳化技術



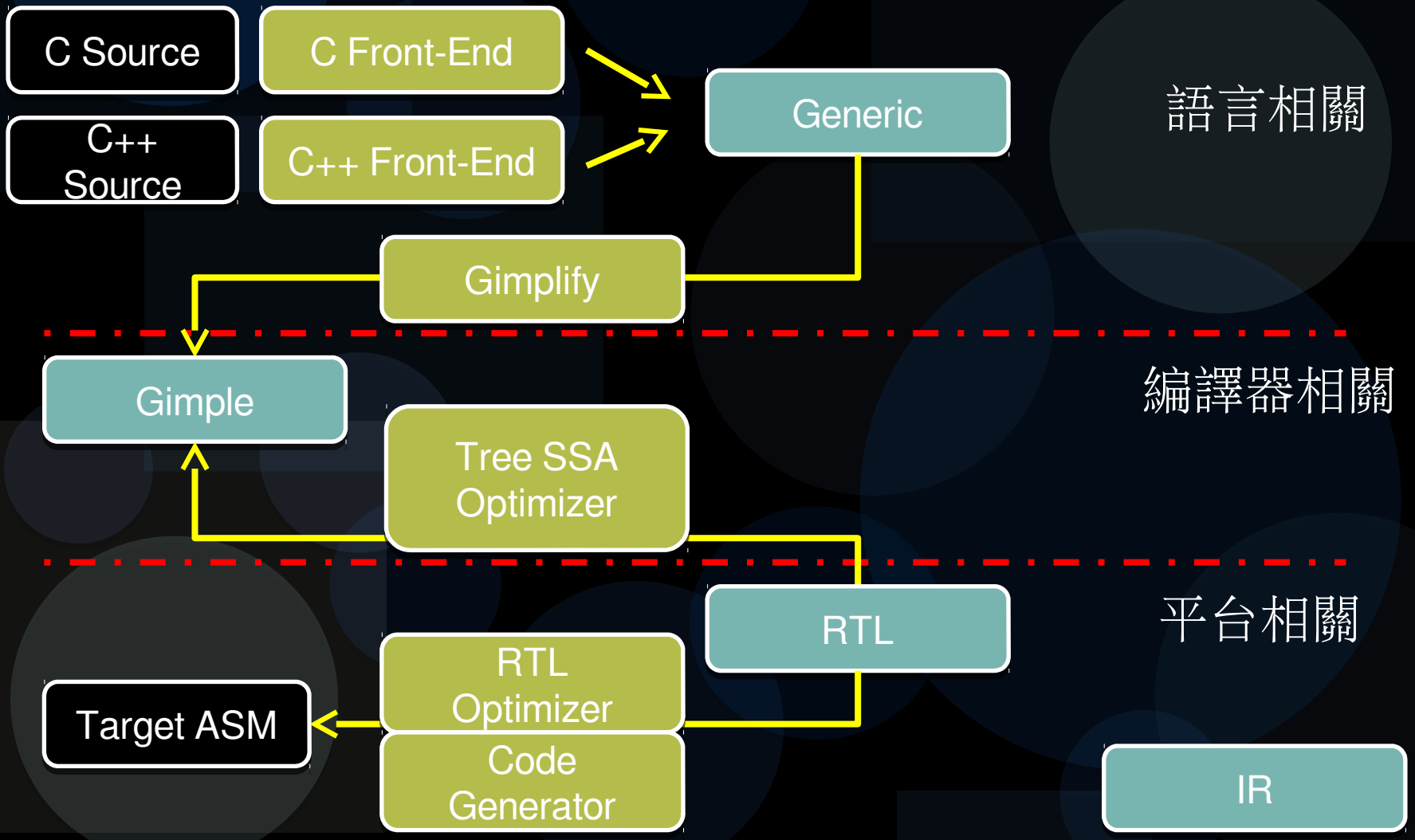
從教科書到真實世界

- 「龍書」的關鍵概念：Syntax Directed Translator
  - 課本終究還是課本
  - 程式的語法樹結構將無法和處理器架構脫鉤
  - 函式參數的傳遞順序、堆疊處理
- GCC 的特性與挑戰：
  - 支援多種語言 (前端 ; front-end)
  - 支援多種處理器 (後端 ; back-end)

```
int main()
{
    return 0;
    IamDeadCode();
}
```

- 思考：有些最佳化和語言跟硬體平台無關
  - 如：消除沒有用到的程式碼 (Dead code elimination)
- GCC 引入中間層最佳化 (編譯器相關最佳化)

GCC 的解決之道： 自 4.x 以來的架構



Code Motion (程式碼搬移)

- 有些運算提前執行，可能會比較快，例如 Loop Invariant
- Loop Invariant: 迴圈內有些運算，不論跑幾次都是相同結果，這類的運算拉出迴圈外則只要算一次即可

資料流程分析 (Data-flow analysis)

- 為什麼需要資料流程分析?

- 兩個指令在什麼狀況可交換位置 (code motion) ?
  - 如果兩個指令彼此沒有相依性的話

- 資料流程分析 = 偵測指令相依性
- **Reaching definition** for a given instruction is another instruction, the target variable of which may reach the given instruction without an intervening assignment
  - Compute use-def chains and def-use chains

Code Motion & Pointer Aliasing

- C 語言中阻擋編譯器最佳化的一大原因
  - Pointer Aliasing: 兩個不同的指標指到同一個記憶體位置

考慮以下最佳化技巧

```
while (y < z) {
  int x = a + b;
  y += x;
}
```

```
int t1 = a + b;
while (y < z) {
  y += t1;
}
```

既然 x=a+b 涉及重複的加法和變數指定，而 a 和 b 在 while 陳述中未曾改變過內含值，於是 x 必為不會變動，所以最佳化的策略即為在 while 陳述前預先計算 a+b 的值，並在 while 陳述中帶入

那麼以下程式碼是否適用前述最佳化?

```
void foo(int *a, int *b, int *y, int z) {
    while (*y < z) {
        int x = *a + *b;
        *y += x;
    }
}
```

Code Motion & Pointer Aliasing

- 貿然施加最佳化，會導致什麼後果?

```
void foo(int *a, int *b, int *y, int z) {
    while (*y < z) {
        int x = *a + *b;
        *y += x;
    }
}
```

```
void foo(int *a, int *b, int *y, int z) {
    int t1 = *a + *b;
    while (*y < z) {
        *y += t1;
    }
}
```

- 編譯器**不能**把 (*a)+(*b) 移到迴圈外
  - 因為不確定開發者是否這樣寫：

    `foo(&num, &num, &num, 5566);`
  - 這樣 a, b, y 互為別名 (alias)，*y 的值改變會使 *a 和 *b 連帶改變，所以不能提出到迴圈外

Strict Aliasing & Restrict Pointer Aliasing

- 如果 Pointer Aliasing 無限上綱
  - 一旦遇到 C pointer，編譯器所有的最佳化都無用武之地
- Strict Aliasing Rule
  - 現代編譯器假設不同 type 之間的 pointer 沒有 aliasing
  - 但有很多舊的程式碼並非這樣認定，造成了很多問題
  - GCC 中可用 `-fno-strict-aliasing` 編譯參數，來抑制 Strict Aliasing rule
- C99: Restrict Pointer Aliasing
  - 使用 `restrict` 關鍵字（GCC 需要一併使用 `--std=c99` 編譯參數讓編譯器採用 C99 規格）告知編譯器指定的 pointer 不會有 Pointer Aliasing 疑慮

Static Single Assignment, SSA Form

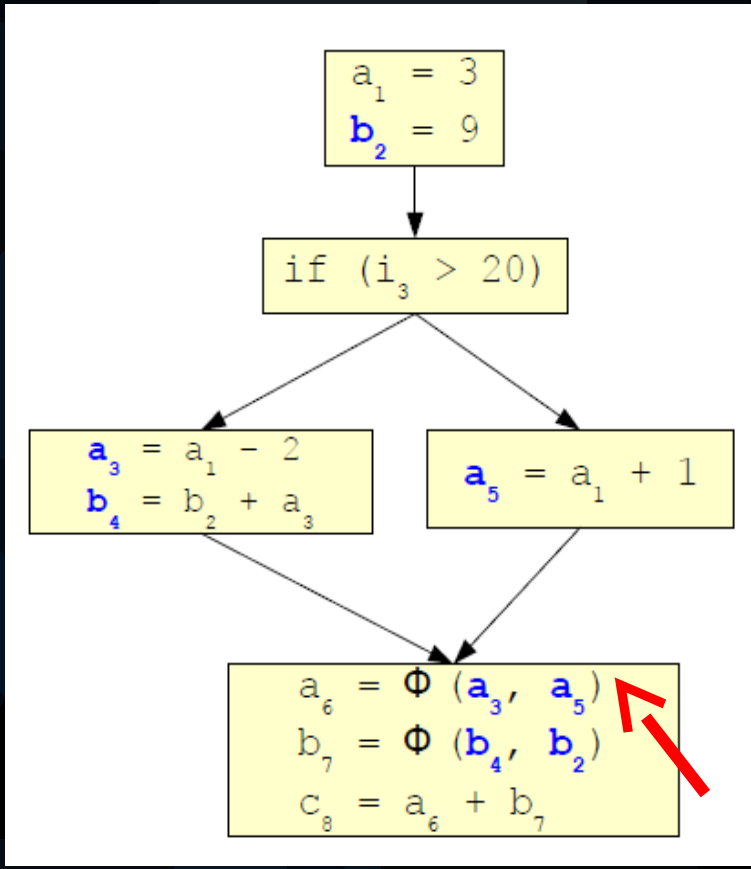
- SSA Form(靜態單賦值形式)

用白話講，在該表示法當中

- 每個被 Assign 的變數只會出現一次
- 作法
  - 每個被 Assign 的變數給一個 Version number
  - 使用 Φ functions

- 範例：
  - INPUT: `y = 1, y =2, x = y`
  - SSA: `y1 = 1, y2 = 2, x1 = y2`

Diego Novillo, GCC Internals - IRs



Static Single Assignment, SSA Form

```
int foo (ini n)
{
  int ret;
  if (n > 10)
    ret1 = n * 2;
  else
    ret2 = n + 2;
  return ret?;
}
```

程式中有分歧點會合時
無法判定是從何而來

```
int foo (ini n)
{
  int ret;
  if (n > 10)
    ret1 = n * 2;
  else
    ret2 = n + 2;
  ret3 = ? (ret1, ret2)
  return ret3;
}
```

此時需要使用 **Φ** function
來處理這種情況，表示值的
定義需由程式流程決定
並給予新的版本號

使用 SSA 可加强 / 加快以下的最佳化

- Constant propagation
- Sparse conditional constant propagation
- Dead code elimination
- Global value numbering
- Partial redundancy elimination
- Strength reduction
- Register allocation

變成 0 的魔法

SSA: Constant Propagation (常數傳遞)

- 有了 SSA，每次 constant propagation 都考 100 分

```
int main()
{
  int a = 11,b = 13, c = 5566;
  int i, result;
  for (i = 0 ; i < 10000 ; i++)
    result = (a*b + i) / (a*c);
  return result;
}
```

GCC-3.x (No-SSA)

```
main:
    …
    mov     r4, #0
.L5:
    mov     r1, #61184
    add     r0, r4, #143
    add     r1, r1, #42
    add     r4, r4, #1
    bl      __divsi3
    cmp     r4, r5
    ble     .L5
    …
```

gcc3 的 ARM 組合語言輸出結果可見 `add` 和 `bl __divsi3` (呼叫 libgcc 提供的除法函式)，顯然的確有計算 `(a*b+i)/(a*c)` 的動作

GCC-4.x (Tree SSA)

```
main:
    mov     r0, #0
    bx      lr
```

gcc4 卻直接輸出 0，ARM 組合語言相當於 C 語言的 `return 0`
為什麼呢？因為整數除法裡頭，被除數永遠小於除數，結果必為零

解說: 切割 Basic Block

```
int main()
{
  int a = 11,b = 13, c = 5566;
  int i, result;
  for (i = 0 ; i < 10000 ; i++)
    result = (a*b + i) / (a*c);
  return result;
}
```

```
int main () {
bb0:
    a = 11
    b = 13 op
    c = 5566
    result = UNDEFINED
bb1:
    i = 0
bb2:
    if (i < 10000) goto bb3
    else goto bb5
bb3:
    result = (a*b + i) / (a*c)
bb4:
    i = i + 1
    goto bb2
bb5:
    return result
}
```

GCC 可輸出包含 Basic Block 的 CFG，使用範例:

```
gcc -c -fdump-tree-cfg=out test.c
```

out 為輸出檔名

解說：合併 Basic Block

```
int main () {
bb0:
    a = 11
    b = 13 op
    c = 5566
    result = UNDEFINED
bb1:
    i = 0
bb2:
    if (i < 10000) goto bb3
    else goto bb5
bb3:
    result = (a*b+i)/(a*c)
bb4:
    i = i + 1
    goto bb2
bb5:
    return result
}
```

```
int main() {
bb0:
    a = 11
    b = 13
    c = 5566
    result = UNDEFINED
    i = 0
bb2:
    if (i < 10000) goto bb3
    else goto bb5
bb3:
    result = (a*b+i)/(a*c)
    i = i + 1
    goto bb2
bb5:
    return result
}
```

解說: 最佳化 CFG

```
int main() {
bb0:
    a = 11
    b = 13
    c = 5566
    result = UNDEFINED
    i = 0
bb2:
    if (i < 10000) goto bb3
    else goto bb5
bb3:
    result = (a*b+i)/(a*c)
    i = i + 1
    goto bb2
bb5:
    return result
}
```

```
int main() {
bb0:
  a = 11
  b = 13
  c = 5566
  result = UNDEFINED
  i = 0
  goto bb2
bb3:
  result = (a*b + i) / (a*c)
  i = i + 1
bb2:
  if (i < 10000) goto bb3
  else goto bb5
bb5:
  return result
}
```

變成 0 的魔法

解說: 轉化為 3-address code

```
int main() {
bb0:
  a = 11
  b = 13
  c = 5566
  result = UNDEFINED
  i = 0
  goto bb2
bb3:
  result = (a*b+i)/(a*c)
  i = i + 1
bb2:
  if (i < 10000) goto bb3
  else goto bb5
bb5:
  return result
}
```

```
int main() {
bb0:
  a = 11
  b = 13
  c = 5566
  result = UNDEFINED
  i = 0
  goto bb2
bb3:
  t0 = a * b
  t1 = t0 + i
  t2 = a * c
  result = t1 / t2
  i = i + 1
bb2:
  if (i < 10000) goto bb3
  else goto bb5
bb5:
  return result
}
```

變成 0 的魔法

解說：轉成 SSA form

```
int main() {
bb0:
  a = 11
  b = 13
  c = 5566
  result = UNDEFINED
  i = 0
  goto bb2
bb3:
  t0 = a * b
  t1 = t0 + i
  t2 = a * c
  result = t1 / t2
  i = i + 1
bb2:
  if (i < 10000) goto bb3
  else goto bb5
bb5:
  return result
}
```

每個被 assign 的變數只會出現一次
- 每次被 assign 的變數加上 version number

```
int main() {
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = a0*b0, t1 = t0+i1, t2 =a0*c0
  result1 = t1/t2, i2 = i1+1
bb2:
  i1 = Φ(i0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = Φ(result0, result1)
  return result2
}
```

此時需要使用 Φ function 來處理這種情況，表示值的定義需由程式流程決定，並給予新的版本號

變成 0 的魔法

解說: Constant Propagation

```
int main(){
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = a0*b0, t1 = t0+i1
  t2 = a0*c0
  result1 = t1/t2
  i2 = i1+1
bb2:
  i1 = Φ(i0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = Φ(result0, result1)
  return result2
}
```

傳遞 11,13,5566 等常數到 t0, t2

傳遞常數 0 到 i0

```
int main(){
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = 11*13, t1 = t0+i1
  t2 = 11*5566
  result1 = t1/t2
  i2 = i1+1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = Φ(UNDEFINED,
              result1)
  return result2
}
```

變成 0 的魔法

解說: Constant Folding

```
int main(){
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = 11*13, t1 = t0+i1
  t2 = 11*5566
  result1 = t1/t2
  i2 = i1+1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = Φ(UNDEFINED,
              result1)
  return result2
}
```

展開常數，也就是計算出 11*13, 11*5566 等表示式的常數值

```
int main(){
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = 143, t1 = t0+i1
  t2 = 61126
  result1 = t1/t2, i2 = i1+1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = Φ(UNDEFINED,
              result1)
  return result2
}
```

變成 0 的魔法

解說: Constant Propagation

```
int main(){
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = 143, t1 = t0+i1
  t2 = 61126
  result1 = t1/t2, i2 = i1+1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = Φ(UNDEFINED,
                result1)
  return result2
}
```

```
int main(){
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = 143, t1 = 143 + i1
  t2 = 61126
  result1 = t1/61126, i2 = i1+1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = Φ(UNDEFINED,
                result1)
  return result2
}
```

變成 0 的魔法

解說: Dead Code Elimination

```
int main(){
bb0:
  a0 = 11, b0 = 13, c0 = 5566
  result0 = UNDEFINED, i0 = 0
  goto bb2
bb3:
  t0 = 143, t1 = 143 + i1
  t2 = 61126
  result2 = t1/61126, i2 = i1+1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = Φ(UNDEFINED,
                result1)
  return result2
}
```

```
int main(){
bb0:
  goto bb2
bb3:
  t1 = 143 + i1
  result1 = t1 / 61126
  i2 = i1 + 1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = Φ(UNDEFINED,
                result1)
  return result2
}
```

變成 0 的魔法

解說: Value Range Propagation

```
int main(){
bb0:
  goto bb2
bb3:
  t1 = 143 + i1
  result1 = t1 / 61126
  i2 = i1 + 1
bb2:
  i1 = Φ(0, i2)
  if (i1
  else go
bb5:
  result2 = Φ(UNDEFINED,
                result1)
  return result2
}
```

result0 = 0 (Treat UNDEFINE as 0)
i1 = [0, 10000], i2 = [1, 10000]
t1 = [143, 10143]
result1 = t1 / 61126
result2 = 0 or t1 / 61126

result1 = [143, 10143] / 61126
result2 = 0 or [143, 10143] / 61126

result1 = [143/61126, 10143/61126]
result2 = 0 or [143/61126, 10143/61126]

result1 = [0, 0]
result2 = 0 or [0, 0]

result1 = 0
result2 = 0 or 0

result2 = 0

```
int main(){
bb0:
  goto bb2
bb3:
  t1 = 143 + i1
  result1 = 0
  i2 = i1 + 1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = 0
  return result2
}
```

變成 0 的魔法

解說: Constant Propagation

```
int main(){
bb0:
  goto bb2
bb3:
  t1 = 143 + i1
  result1 = 0
  i2 = i1 + 1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = 0
  return result2
}
```

```
int main(){
bb0:
  goto bb2
bb3:
  t1 = 143 + i1
  result1 = 0
  i2 = i1 + 1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = 0
  return 0
}
```

變成 0 的魔法

解說: Dead Code Elimination

```
int main(){
bb0:
  goto bb2
bb3:
  t1 = 143 + i1
  result1 = 0
  i2 = i1 + 1
bb2:
  i1 = Φ(0, i2)
  if (i1 < 10000) goto bb3
  else goto bb5
bb5:
  result2 = 0
  return 0
}
```

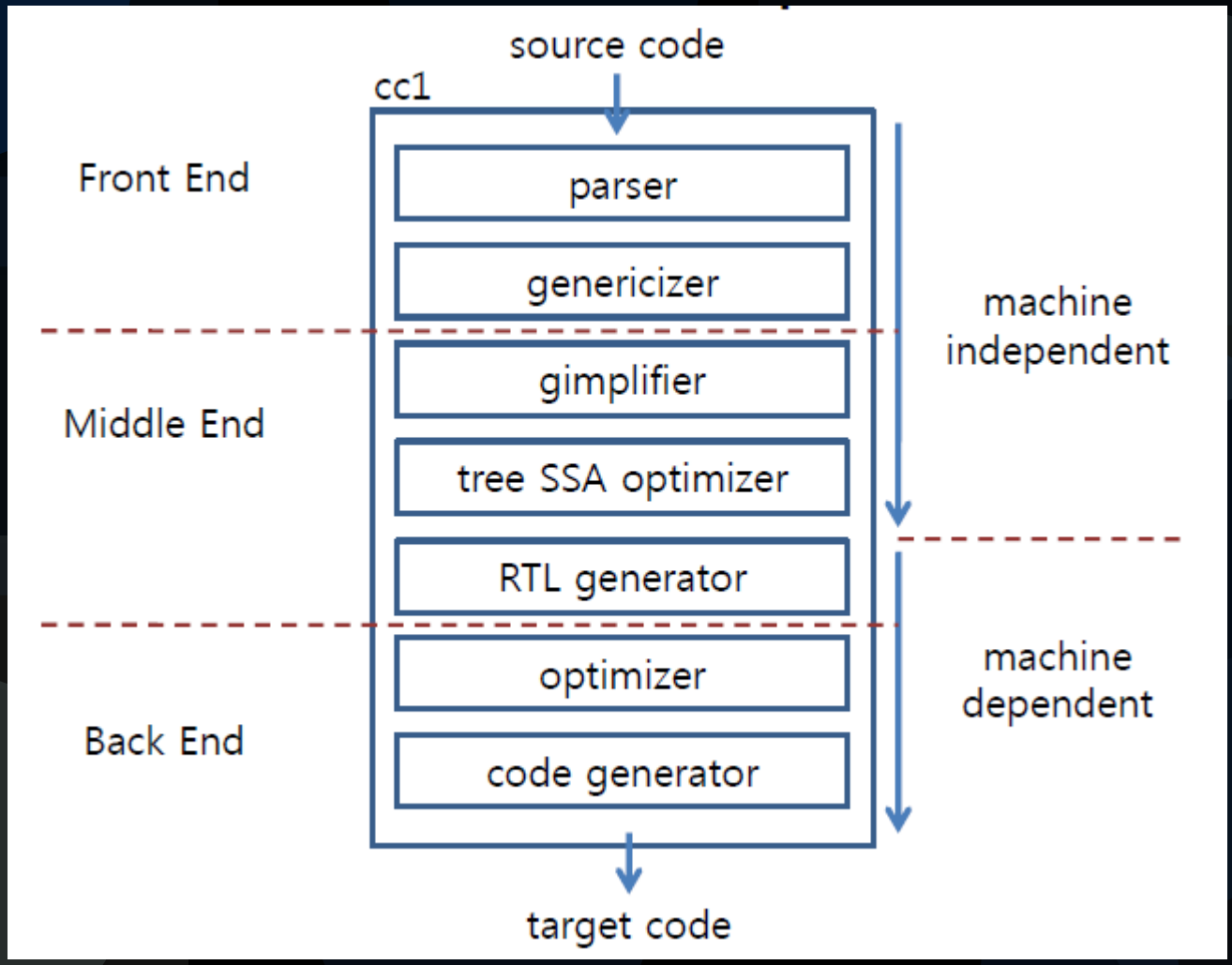
```
int main(){
  return 0
}
```

GCC 可輸出最佳化的 Basic Block，使用範例:

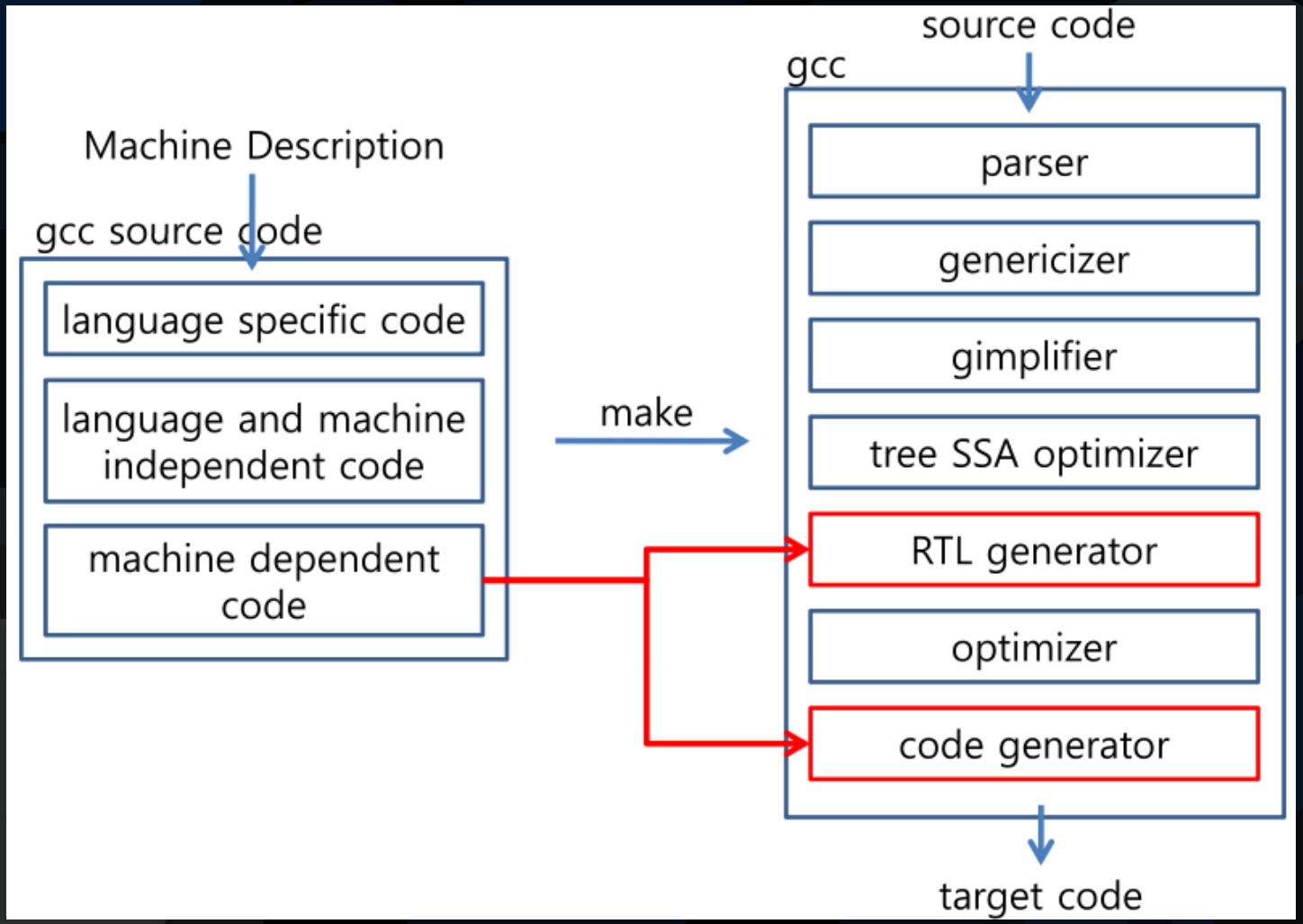
```
gcc -Os -c -fdump-tree-optimized=out test.c
```

out 為輸出檔名

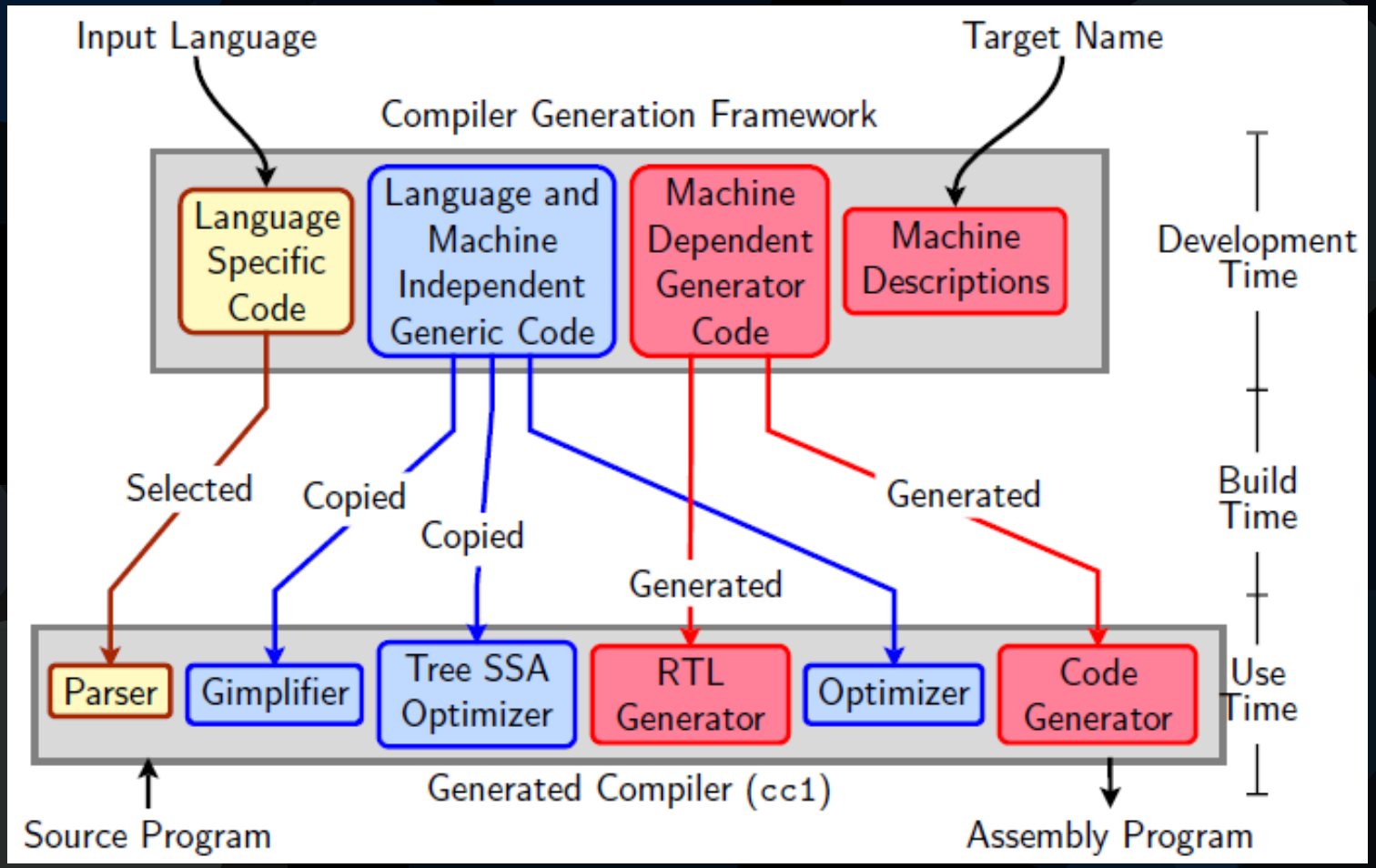
GCC 結構


source code
cc1
Front End
parser
genericizer
machine
independent
Middle End
gimplifier
tree SSA optimizer
RTL generator
optimizer
machine
dependent
Back End
code generator
target code

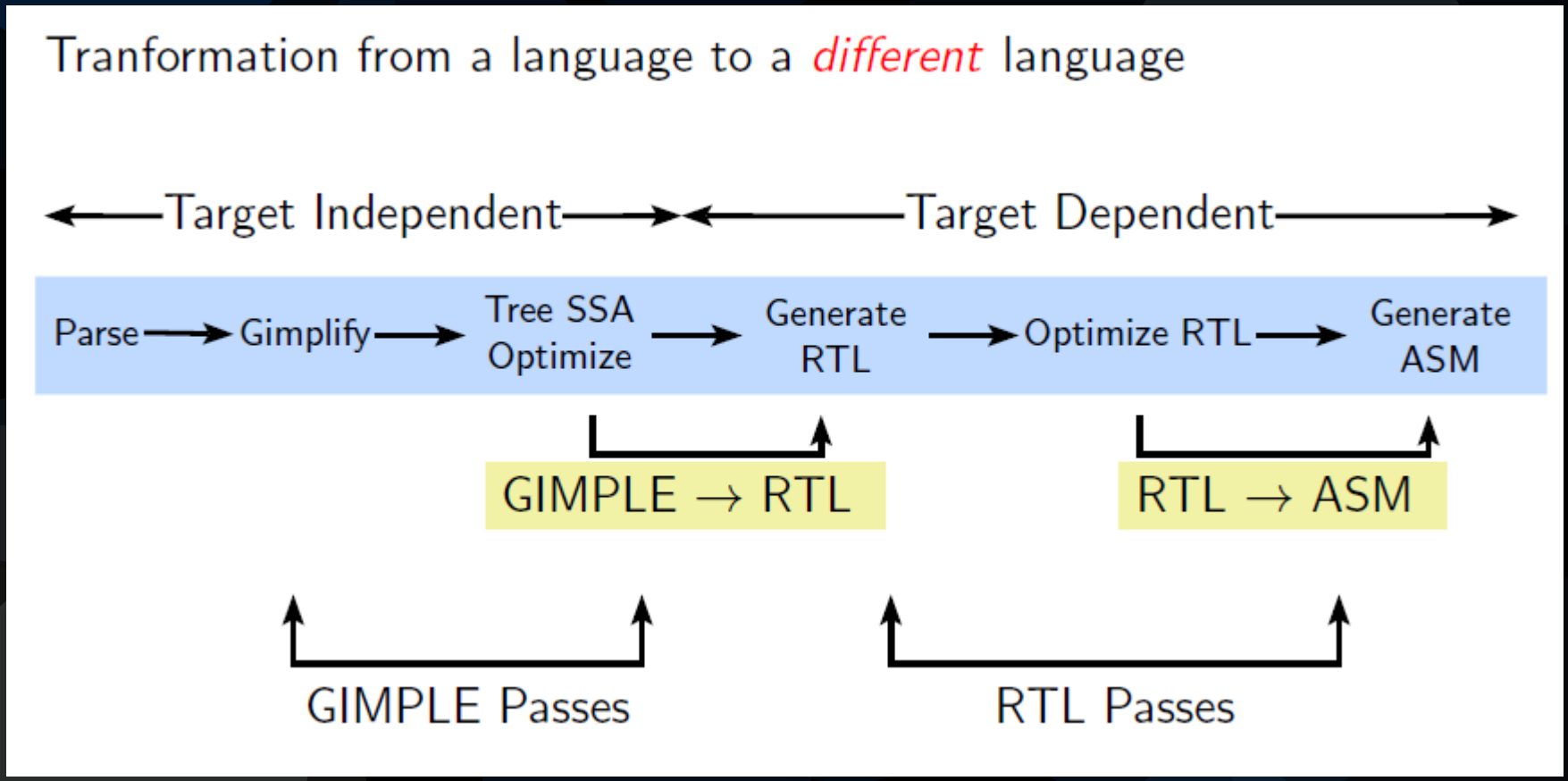
GCC 結構


source code
gcc
Machine Description
gcc source code
language specific code
language and machine independent code
machine dependent code
make
parser
genericizer
gimplifier
tree SSA optimizer
RTL generator
optimizer
code generator
target code

GCC 結構


Input Language
Target Name
Compiler Generation Framework
Language Specific Code
Language and Machine Independent Generic Code
Machine Dependent Generator Code
Machine Descriptions
Development Time
Selected
Copied
Copied
Generated
Generated
Build Time
Parser
Gimplifier
Tree SSA Optimizer
RTL Generator
Optimizer
Code Generator
Use Time
Generated Compiler (cc1)
Source Program
Assembly Program

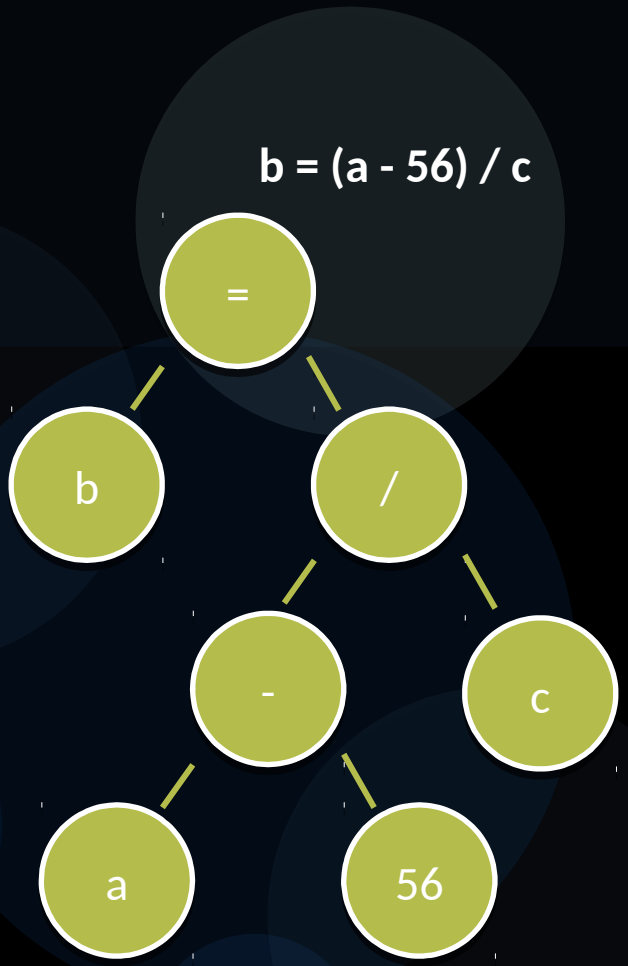
GCC 的 Transformation： 語言間轉換



GCC 前端: Source → AST → Generic

- GCC C/C++ frontend
  - bison (3.x)
  - Handwritten recursive descent parser (4.x)
- 辨識 C/C++ 原始程式碼，並轉換為剖析樹 (parsing tree)
- AST (Abstract Syntax Tree)
  - 剖析樹 + 語意
  - 右圖範例：

```
b = (a - 56) / c
```



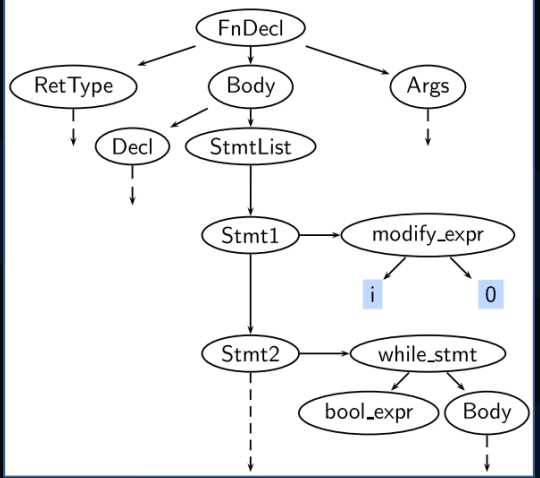
cc1：真正的 C 編譯器

1 The Source

```
int f(char *a)
{
  int n = 10; int i, g;

  i = 0;
  while (i < n) {
    a[i] = g * i + 3;
    i = i + 1;
  }
  return i;
}
```

2 Simplified AST



3 Gimple IR

```
f (a)
{
  unsigned int i.0;   char * i.1;
  char * D.1140;      int D.1141;
  ...
  goto <D1136>;
  <D1135>:;
  ...
  D.1140 = a + i.1;   D.1141 = g * i;
  ...
  <D1136>:;
  if (i < n) {goto <D1135>;}
  ...
}
```

4 Tree SSA form

```
f (a)
{
  ... int D.1144; ...
<bb 0>:
  n_2 = 10;    i_3 = 0;
  goto <bb 2> (<L1>);
<L0>:;
  ...
  D.1140_9 = a_8 + i.1_7;
  D.1141_11 = g_10 * i_1;
  ...
<L1>:;
  if (i_1 < n_2) goto <L0>; else goto <L2>
  ...
}
```

5 RTL IR (fragment)

```
(insn 21 20 22 2 (parallel [
  (set (reg:SI 61 [ D.1141 ])
    (mult:SI (reg:SI 66)
      (mem/i:SI
        (plus:SI
          (reg/f:SI 54 ...)
          (const_int -8 ...)))))
  (clobber (reg:CC 17 flags))
 ]) -1 (nil)
 (nil))
```

6 Final ASM (partial)

```
.file "sample.c"
        ...
f:
pushl %ebp
        ...
movl -4(%ebp), %eax
imull -8(%ebp), %eax
addb $3, %al
        ...
leave
ret
```

GCC 中端:
Gimple & Tree SSA Optimizer

- Gimple 衍生自 Generic Gimplify
  - 被限制每個運算只能有兩個運算元 (3-Address IR)
    `t1 = A op B` （op 即 operator ，像是 +-*/ 等等）
  - 被限制語法只能有某些控制流程
  - Gimple 可以被化簡成 SSA Form
  - 可使用 `-fdump-tree-<type>-<option>` 來觀看其結構

- Tree SSA Optimizer
  - 100+ Passes
  - Loop, Scalar optimization, alias analysis ...etc
  - Inter-procedural analysis, Inter-procedural optimization

GCC 後端：
Register Transfer Language (RTL)

```
b = a -56
```

```
(set (reg:SI 60 [b])
     (plus:SI (reg:SI 61 [a])
              (const_int -56 [0xffffffc8])))
```

- 類似 LISP 的表示法 (S-expression)
- RTL 使用 virtual Register (具有無限多個 register)
- Register Allocation (Virtual Register -> Hard Register)
- Instruction scheduling (Pipeline scheduling)
- Peephole optimization
- GCC built-in operation and architecture-defined operation

GCC 後端：
RTL Patten Match Engine

MIPS.md

```
(set (reg:SI 60 [b])
     (plus:SI (reg:SI 61 [a])
              (const_int -56 [0xffffffc8])))
```

```
(define_insn "*addsi3"
  [(set (match_operand:GPR 0 "register_operand" "=d,d")
        (plus:GPR (match_operand:GPR 1 "register_operand" "d,d")
                  (match_operand:GPR 2 "arith_operand" "d,Q")))]
  "!TARGET_MIPS16"
  "@
   addu\t%0,%1,%2
   addiu\t%0,%1,%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "si")])
```

d 代表是 register
Q 代表是整數

指令的屬性（用於管線排程）

指令限制（非 MIPS16 才可使用）

b = a - 56

addiu $2, $3, -56

GCC 後端: 暫存器分配 (Register Allocation)

- 暫存器分配其實是個著色問題 (NP-Complete)
  - 給一個無向圖，相鄰的兩個 vertex 不能著同一種顏色
  - 每個 vertex 代表的是一個變數， 每個 edge 代表的是兩個變數的生存時間有重疊
- GCC 的暫存器分配，其實比著色問題更複雜
  - 在有些平台，某些指令其實只能搭配某組的暫存器
  - 其實有些變數是常數，或是 memory form， 或是他是參數或回傳值，所以只能分配到某些特定暫存器
- Old GCC RA: Reload Pass
  - GCC 把 pseudo-registers 根據指令的限制對應到硬體暫存器的過程
  - 其實是個複雜到極點的程式，所以到最後就沒人看得懂了
- 於是後來又重寫
  - IRA (**Integrated** Register Allocator)： 整合 Coalescing, Live range splitting 以及挑選較好的 Hard Register 的做法

GCC 後端 ：管線排程

- 以經典課本的 5-stage pipeline 為例

```
lw $8, a
lw $9, b
mul $10,$8,$8
add $11,$9,$10
lw $12,c
add $13,$12,$0
```

```
lw $8, a
lw $9, b
lw $12,c
mul $10,$8,$8
add $13,$12,$0
```

Clock 0
Clock 1
Clock 2
Clock 3
Clock 4
Clock 5
Clock 6
Clock 7
Clock 8

GCC 後端：Peephole optimization

- 以管窺天最佳化法
- 掃過整個 IR，看附近 2 ~ n 個 IR 有沒有最佳化的機會
  - 感覺起來就像是用奧步
  - 可是如果奧步有用，那就得要好好使用
  - 在某些時候，這個最佳化很好用

```
;;  sub  rd, rn, #1
;;  cmn  rd, #1 (equivalent to cmp rd, #-1)
;;  bne  dest
```

```
;;  subs rd, rn, #1
;;  bcs  dest   ((unsigned)rn >= 1)

;; This is a common looping idiom (while (n--))
```

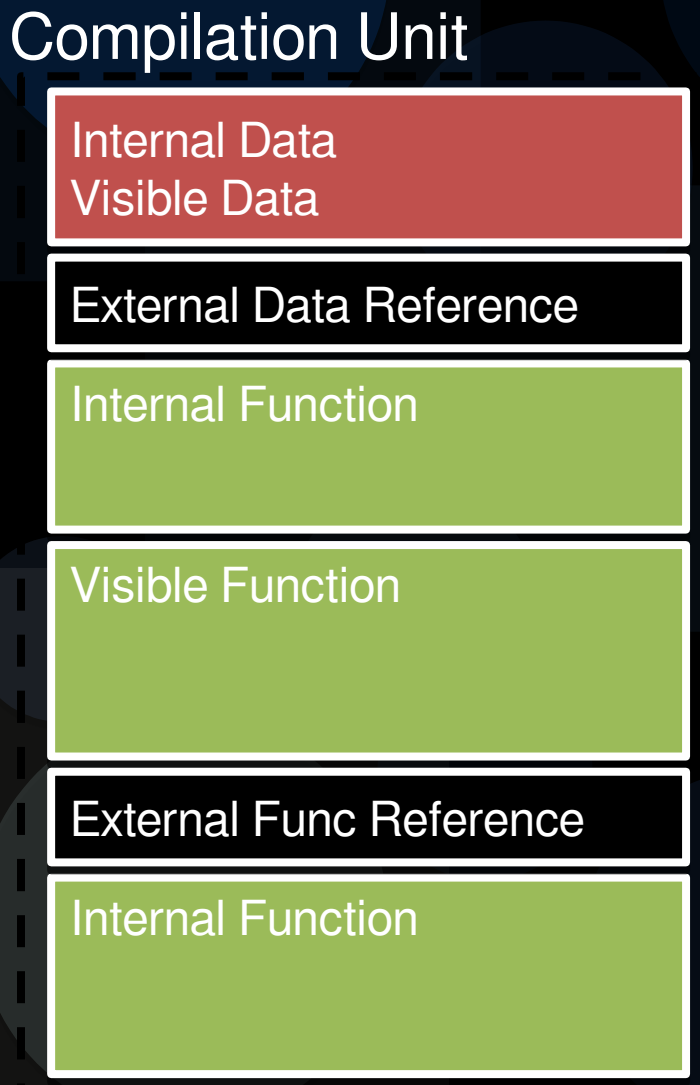
進階編譯器最佳化技術

- Loop Optimization
- Inter Procedure Optimization
- Auto Vectorization
- Auto Parallelization

由若干個 C 程式原始檔構成專案

- 為什麼要分很多檔案？
  - 因為要讓事情變得簡單
- 為什麼編譯程式時可下 `make -j24` ？
  - 因為編譯器在編譯每個 `.c` 檔案時視為獨立個體，彼此之間沒有相依性
  - 用編譯器的術語稱為 Compilation Unit
- Static Global Variable vs Non-Static Global Variable
  - static: 只有這個 Compilation Unit 可以看到這個變數
  - non-static: 所有 Compilation Unit 可以看到這個變數
    - Q: 怎麼使用別的別的 Compilation Unit 內的變數
    - A: 使用 extern 關鍵字，如 `extern int i;`

深入淺出 Compilation Unit



- Compilation Unit 產生的限制
  - 關於沒有人使用的 static global variable
    - Compiler 可以砍掉它來節省空間
  - 關於沒有人使用的 non-static global variable
    - 編譯器不能砍掉它
    - 因為不能確定別的 Compilation Unit 會不會用到它
- 注意：若確定沒有別的檔案會使用到這個變數或函式，請宣告成 static

Compilation Unit 盡量使用區域變數

```
static int i;
int main()
{
  for (i = 0 ; i < 10; i++)
    printf ("Hello %d\n", i);
}
```

```
main:
    mov   r3,#0
    stmfd  sp!,{r4,lr}
    movw   r4,#:lower16:.LANCHOR0
    movt   r4,#:upper16:.LANCHOR0
    mov   r1,r3
    str   r3,[r4,#0]
.L2:
    ldr   r0,.L4
    bl    printf
    ldr   r1,[r4,#0]
    add   r1,r1,#1
    str   r1,[r4,#0]
    cmp   r1,#9
    ble   .L2
    ldmfd  sp!,{r4,pc}
.L4:
    .word  .LC0
```

```
int main()
{
  int i;
  for (i = 0 ; i < 10; i++)
    printf ("Hello %d\n", i);
}
```

```
main:
    stmfd  sp!,{r4,lr}
    mov   r4,#0
.L2:
    mov   r1,r4
    ldr   r0,.L4
    add   r4,r4,#1
    bl    printf
    cmp   r4,#10
    bne   .L2
    ldmfd  sp!,{r4,pc}
.L4:
    .word  .LC0
```

GCC 內建函式 (Built-in Function)

- GCC 為了最佳化，會辨認標準函式（因為語意有標準規範），若符合條件，就會以處理器最適合的方式來處理
  - `strcpy` → x86 字串處理指令
  - `printf` → `puts`, `putc`
  - `memcpy` → Block Transfer Instruction
- 這對開發嵌入式系統者常造成困擾
  - 因為往往需要客製化的 `printf`
  - 但 `gcc` 會自作聰明，把特定的 `printf` 悄悄換成 `puts`，而系統中可能剛好缺乏該函式實做
    - 注意：使用 `-fno-builtin-XXX` 或 `-fno-builtin` 來關閉最佳化

IPO (Inter-Procedural Optimization)

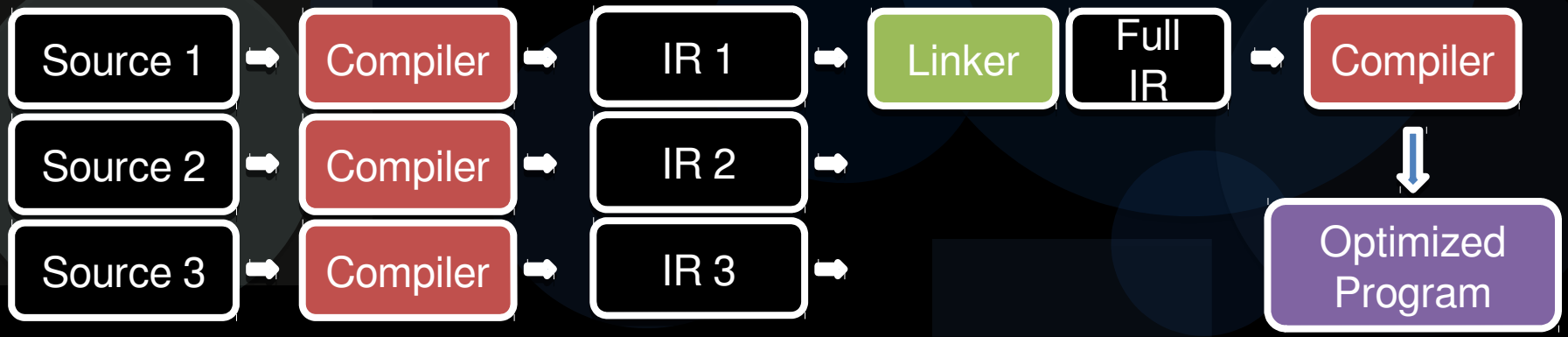
- Procedure / Function： 結構化程式的主軸，增加可用性，減少維護成本
  - 副程式呼叫的執行成本
    - 準備參數：把變數移到到特定 register 或 push 到 stack
    - 呼叫副程式，儲存和回復 register context
  - 現代化程式：有許多很小的副程式

```
int IsOdd(int num) {
   return (num & 0x1);
}
```

- 最簡單的 IPO 就是 function inlining
  - 簡單來說，把整個 function body 複製一份到 caller 的程式碼去

IPO (Inter-Procedural Optimization)

- 如何 inline 一個外部函式？
  - 基本上編譯器無法做到
    - 因為 compiler 在編譯程式的時候不會去看別的 Compilation Unit 的內容，不知道內容自然就沒辦法 inline 一個看不見的函式
- 解法：Link Time Optimization (LTO)
  - 或叫：Whole-Program Optimization (WPO)
  - 關鍵 Link Time Code Generation



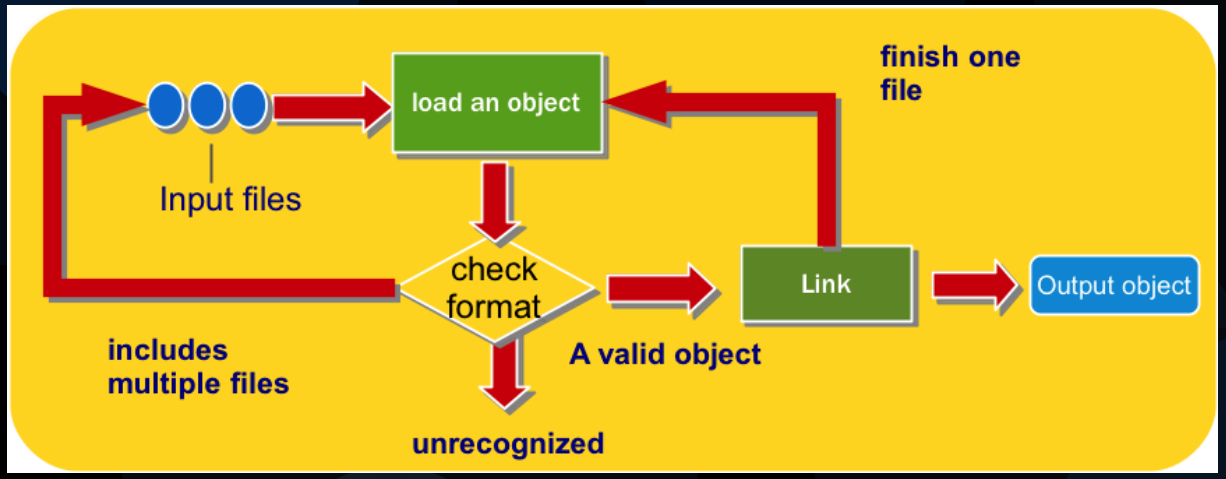
Linker 也大有學問，特別是配合 LTO

- Linker 的命令列本身就是一種語言
- 個別選項的意義取決於 [位置] 和 [其他部份]
- 甚至還有複雜的語法

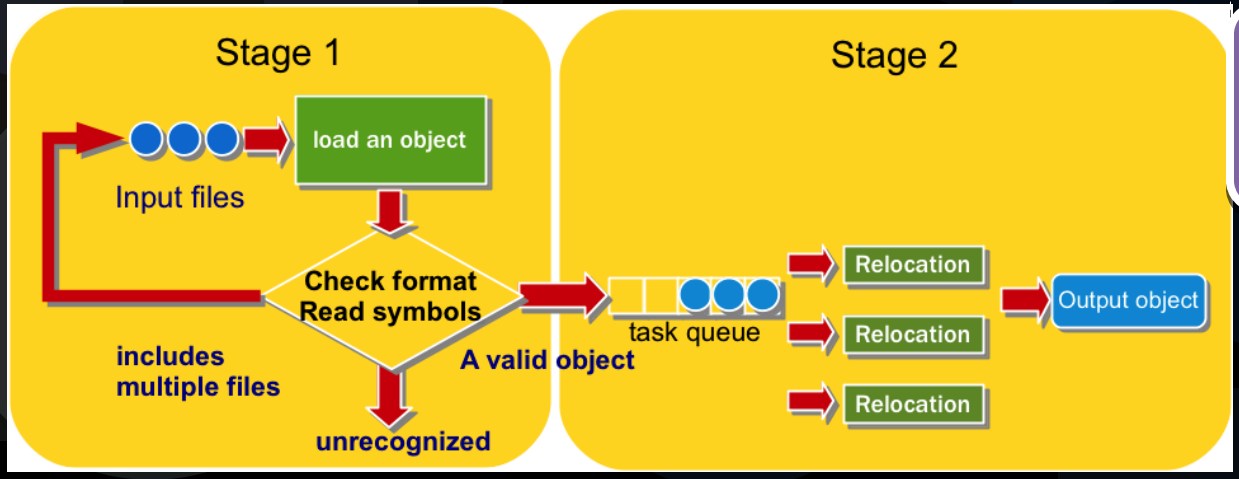
- Four categories of the options
  - Input files
  - Attributes of the input files
  - Linker script options
  - General options
- Examples

```
ld /tmp/xxx.o –lpthread
ld –as-needed ./yyy.so
ld –defsym=cgo13=0x224
ld –L/opt/lib –T ./my.x
```

GNU ld linker vs. Google gold linker



GNU ld 仰賴 BFD (Binary File Descriptor)，慢且難以維護



Google gold 效率比 GNU ld 快 2 倍。僅支援 ELF

21 世紀的另外一種選擇：LLVM

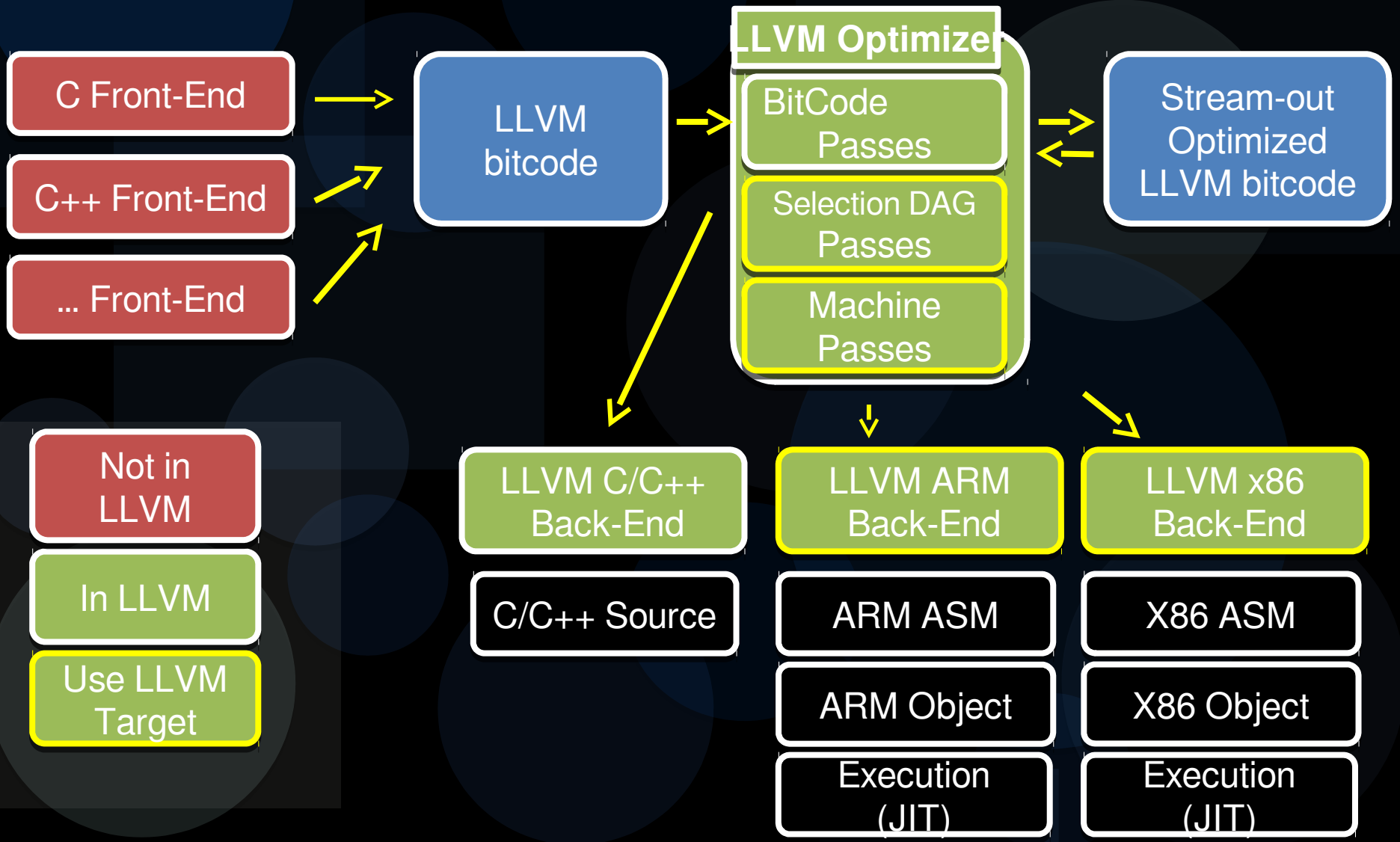
- LLVM (以前稱為 Low Level Virtual Machine)
  - 用 現代 C++ 寫成的 Compiler infrastructure
  - 設計來進行全時最佳化 (compile-time, link-time, run-time, and idle-time optimizations)

- LLVM 起源
  - 2000 年 UIUC Vikram Adve 與 Chris Lattner 的研究發展而成 (碩士論文)
- Apple Inc. 採用而大放異彩
- FreeBSD 10 使用 Clang/LLVM 為預設的編譯器

LLVM 核心觀念：LLVM is a Compiler IR

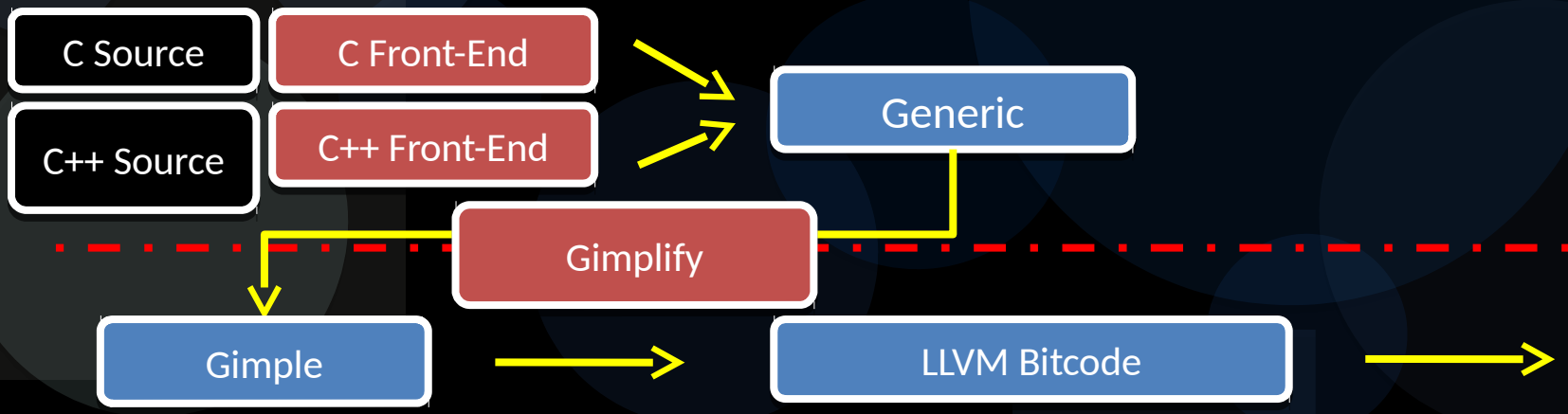
- 既然 Compiler framework 要支援多種 front-end 和多種 back-end
  - 使用類似 RISC 平台無關的指令 (LLVM Instruction) 和資料 (Metadata) 來表達原始碼
- LLVM IR 的三變化 (等價形式)
  - In-Memory Form: JIT
  - Assembly language representation: ASM format (.ll)
  - Bitcode representation: Object code format (.bc)
- 特點
  - Self-contained Representation 且有良好規範的 Compiler IR
  - 可以將編譯器處理到一半的結果序列化

LLVM 的架構：從原始碼到二進制



LLVM Toolchain: llvm-gcc

- 既然 LLVM 只是個 Compiler-IR， 就意味著 LLVM 發展初期無法提供完整個編譯流程
- LLVM-GCC (gcc-4.2)
  - 利用 GCC 已存在而成熟多樣語言支援的 Front-end
  - 後來 Richard Stallman 認為此方式有架空 GPL 授權的 GCC 的嫌疑，而反對此方式 (GCC 4.3, GPLv3)
  - 後來 Apple Inc. 就跳出來發展自己的 front-end (Clang)
  - DragonEgg 利用 gcc plugin， 讓 LLVM 作為 GCC backend



LLVM Toolchain: Clang

- Clang : C, C++, Objective-C 的編譯器前端
  - Apple 為了取代 GCC 重新打造的 Frontend
  - 重寫一個前端代價是非常昂貴的
    - GNU/Linux 世界使用了 GCC 行之有年的非標準語法
  - Clang 設計成模組化 Frontend : Clang C API
    - 可 export AST（如 Android RenderScript 來產生 Java 的 Binding)
    - 可用來實作自動完成、 Refactor Tool 等等
    - 可用來實作靜態程式碼分析工具 (Linting tool 等等)
  - Clang 改善了錯誤訊息 (expressive diagnostics)
  - 快速編譯，低記憶使用量，容易學習和延展的 Frontend

範例： Clang Expressive Diagnostics

```
struct a {
  virtual int bar();
};
struct foo : public virtual a {
};
void test(foo *P) {
  return P->bar() + *P;
}
```

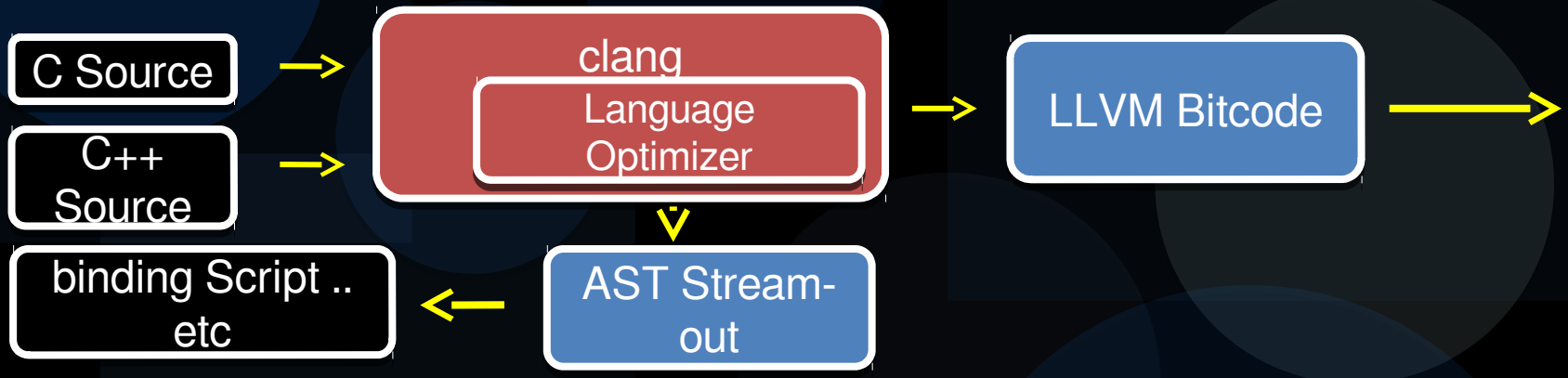
g++-4.7

```
xx.cc: In function 'void test(foo*)':
xx.cc:9:24: error: no match for 'operator+' in '(((a*)P) + ((sizetype)(*(long int*)(P-
>foo::<anonymous>.a::_vptr.a + -32u))))->a::bar() + * P'
xx.cc:9:24: error: return-statement with a value, in function returning 'void' [-fpermissive]
```

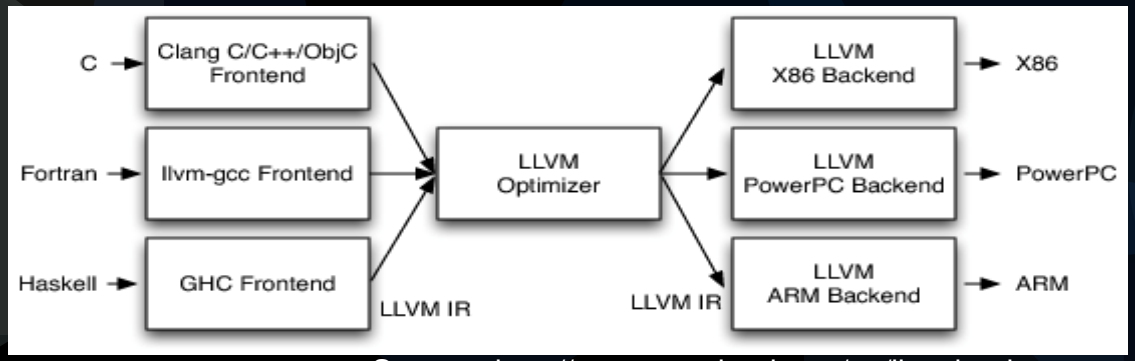
clang-3.1

```
xx.cc:9:21: error: invalid operands to binary expression ('int' and 'foo')
    return P->bar() + *P;
           ~~~~~~~~ ^ ~~
```

Clang/LLVM Toolchain

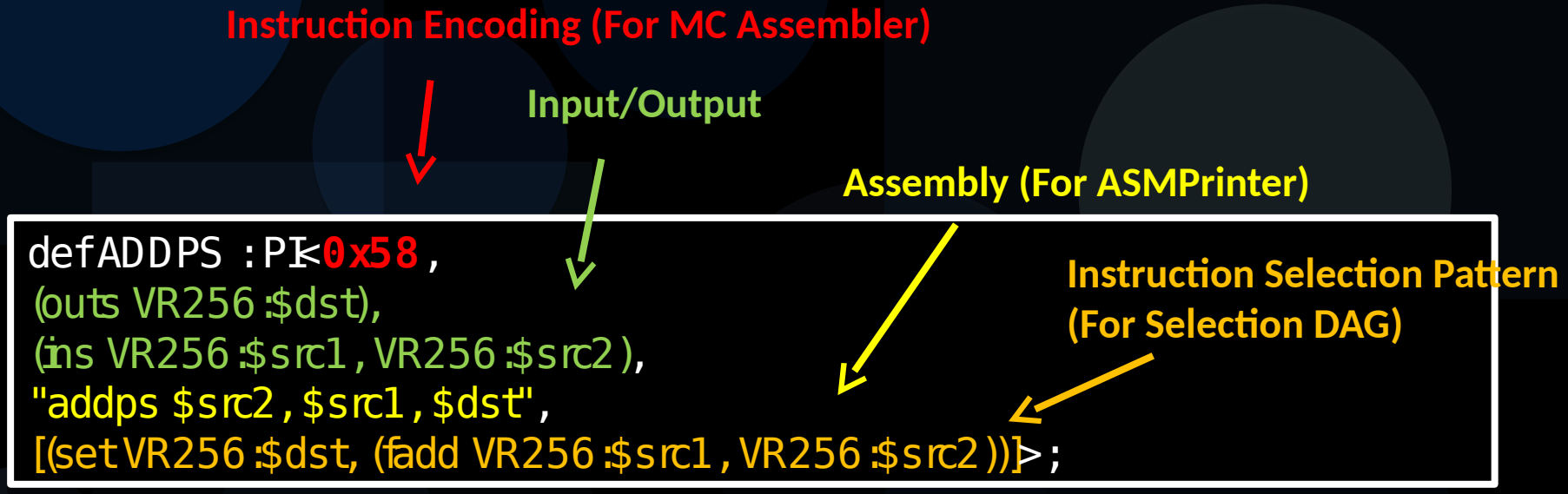


- Clang Compiler Driver
  - 盡量和 GCC 行為相容，遇到不認識的編譯選項，不會停下來
  - 不需要 cross compiler，本身就有 cross compile 的能力
    - Cross Compiler != Cross Compiler Toolchian



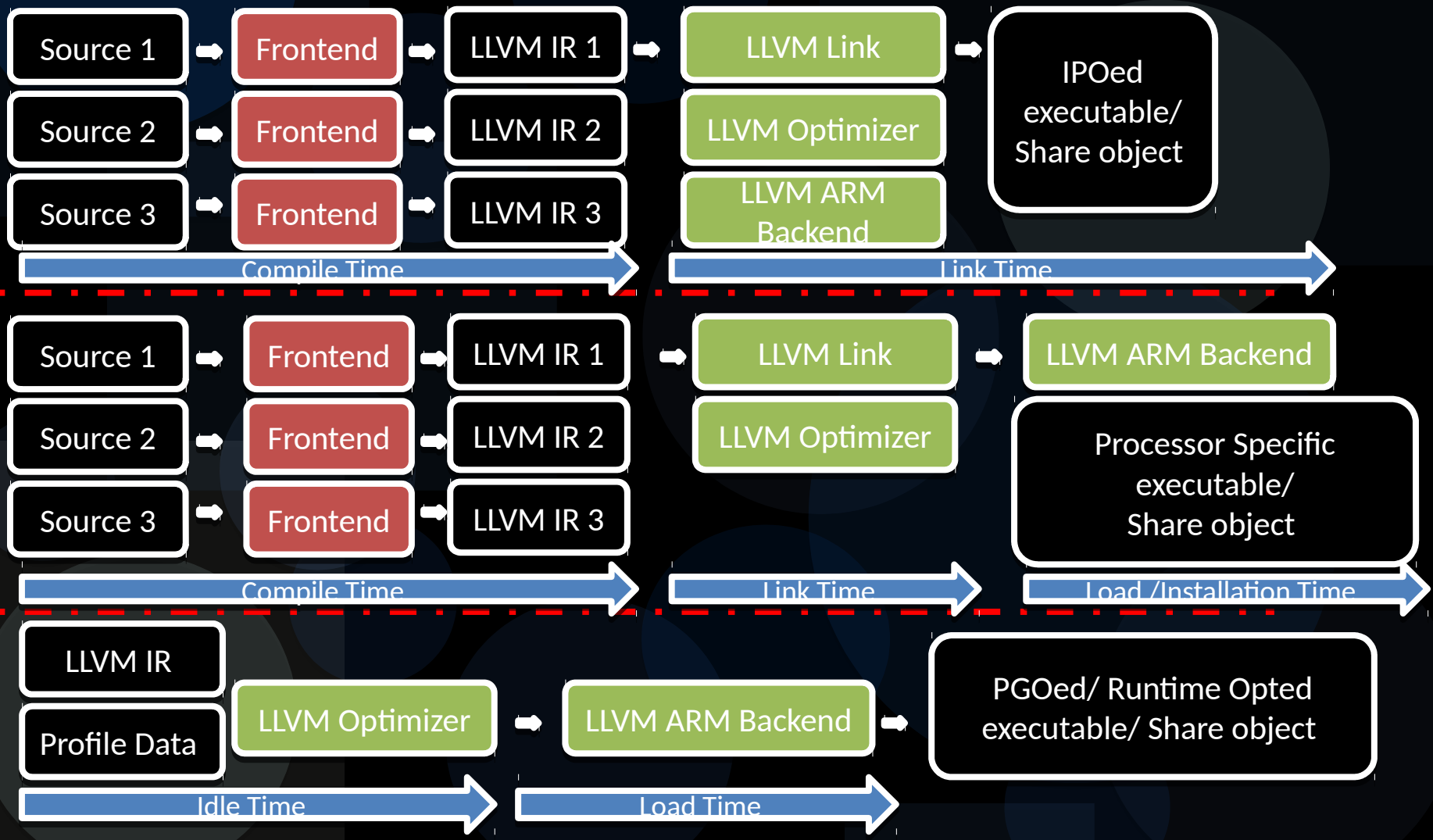
Source: http://www.aosabook.org/en/llvm.html

LLVM TableGen (Target Description)



- 編譯器 Backend 常會引入特別的程式語言來表示後端機械碼生成
  - GCC RTL Pattern : (MD, Machine Description file)
  - LLVM TableGen : (TD, Target Description file)

編譯器進化論：LLVM 的多變化



LLVM 的發展

- 最美好的時代，最壞的時代
  - LLVM Bitcode 用來當傳遞格式還有很多問題
    - Binary Compatibility 最為人所詬病
    - http://lists.cs.uiuc.edu/pipermail/llvm-commits/Week-of-Mon-20120521/143197.html
- CASE STUDY: OpenCL SPIR
  - The OpenCL Khronos group proposes to extend LLVM to be the standard OpenCL IR (called SPIR)
    - http://www.khronos.org/registry/cl/specs/spir_spec-1.0-provisional.pdf
  - Khronos SPIR For OpenCL Brings Binary Compatibility
    - http://www.phoronix.com/scan.php?page=news_item&px=MTE4MzM

參考資料

- http://www.redhat.com/magazine/002dec04/features/gcc/
- http://gcc.gnu.org/onlinedocs/gcc
- http://en.wikipedia.org/wiki/Interprocedural_optimization
- http://en.wikipedia.org/wiki/Link-time_optimization
- http://llvm.org/
- http://www.aosabook.org/en/llvm.html
- Compiling a Compiler, 穆信成
- GNU Compiler Collection, Kyungtae Kim