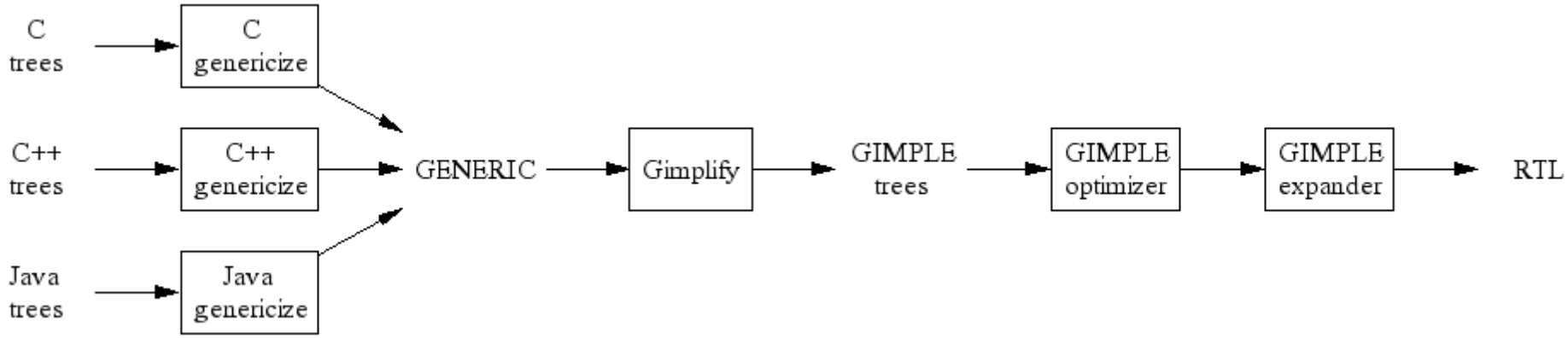
IR, DFA, CFA

Igor Gorban 2.03.19

Intermediate representation

- Структура данных или код, используемый компилятором или виртуальной машиной для работы с низкоуровневыми инструкциями и данными
- «Хороший» IR должен быть точным, т.е. способным представить исходный код без потери информации и не должен зависеть от какого-либо конкретного языка (исходного или целевого).
- Однако он всегда низкоуровневый и должен иметь возможность пользоваться новыми возможностями процессоров (?).

GCC IR



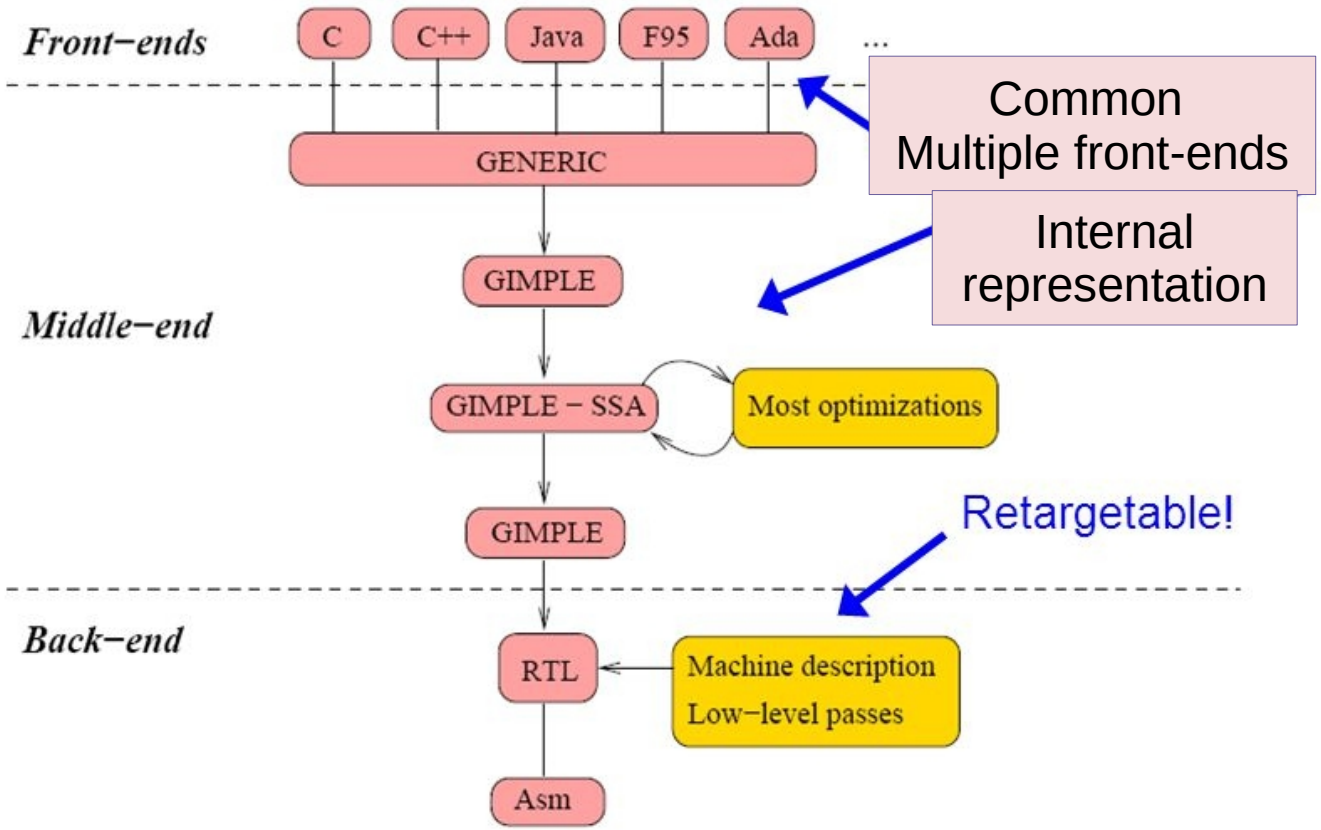
- В GCC используется минимум два представления:
  - GIMPLE — трехадресный код
    - -fdump-tree-gimple (descriprion)
  - RTL - register transfer language
    - -fdump-rtl-all (описание структуры)

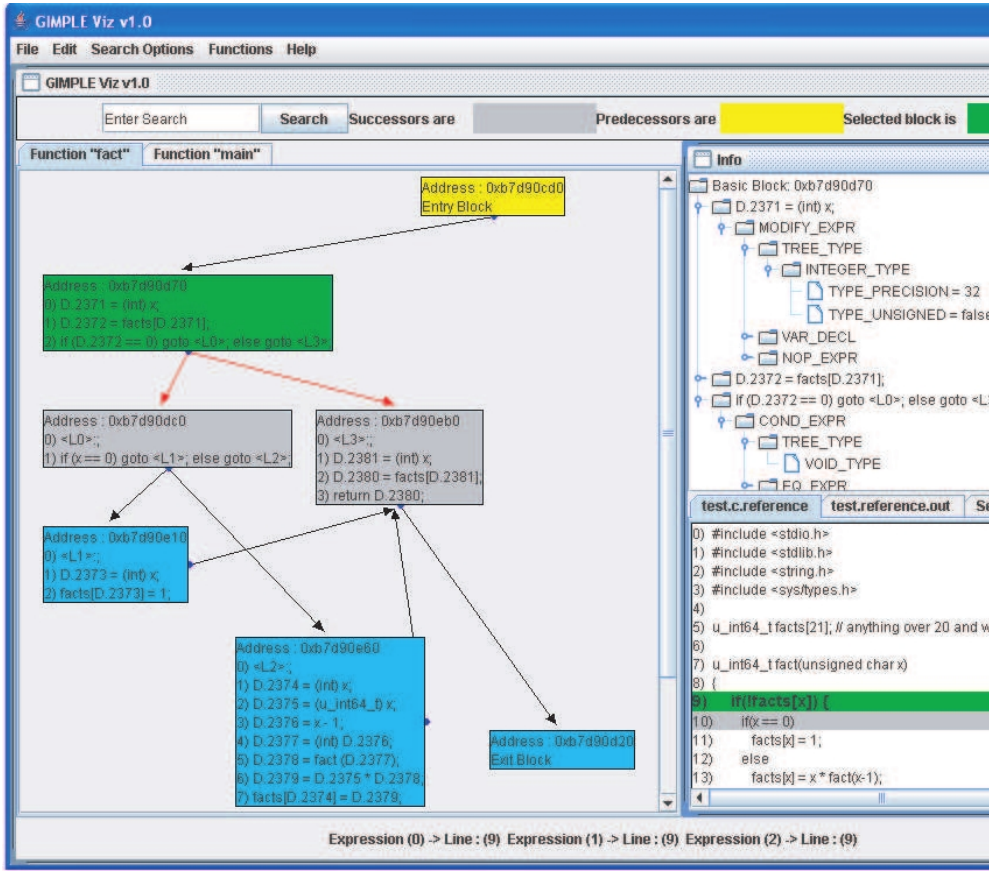
GCC справка

perhaps the most used compiler : your phone, camera, dish washer, printer, car, house, train, airplane, web server, data center, Internet have Gcc compiled code

- [cross-] compiles many languages (C, C++, Ada, Fortran, Go, Objective C, Java, …)
- on many systems (GNU/Linux, Hurd, Windows, AIX, …)
- for dozens of target processors (x86, ARM, Sparc, PowerPC, MIPS, C6, SH, VAX, MMIX, …)
- free software (GPLv3+ licensed, FSF copyrighted)
- still alive and growing (+6% in 2 years)
- big contributing community (≈ 400 “maintainers”, mostly full-time professionals)
- peer-reviewed development process, but no main architect
- ⇒ (IMHO) “sloppy” software architecture, not fully modular yet
- various coding styles (mostly C & C++ code, with some generated C code)
- industrial-quality compiler with powerful optimizations and diagnostics (lots of tuning parameters and options…)

GCC Internals



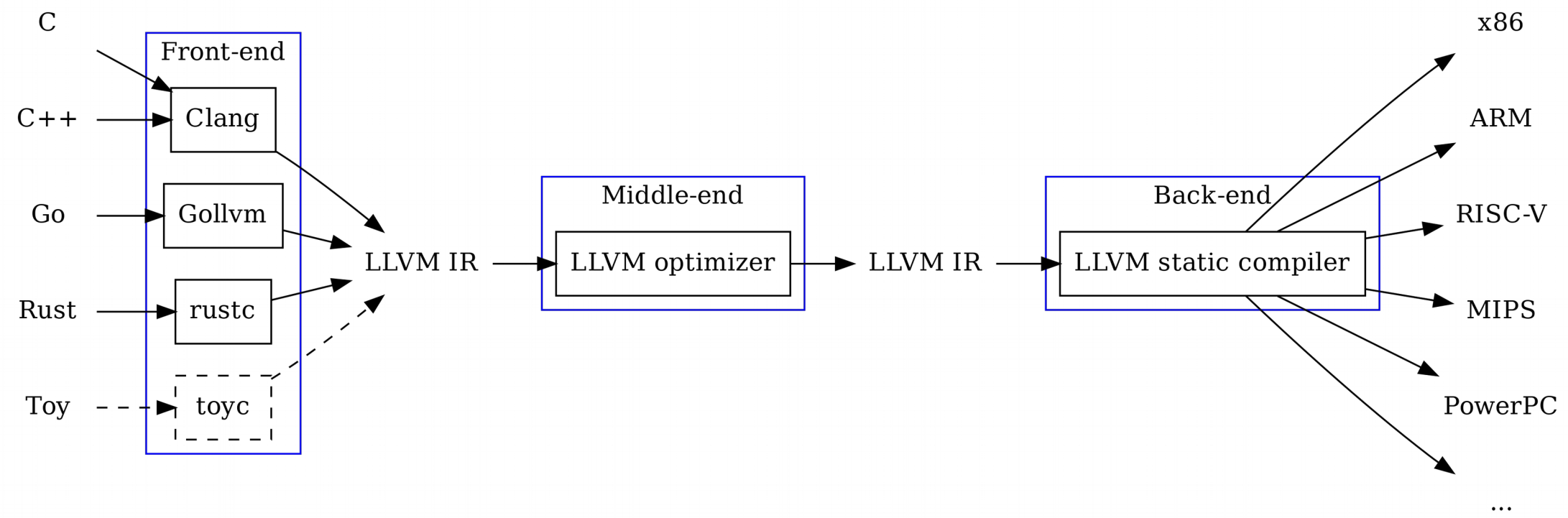


Gimple Viz

SSA

- Static single assignment form - промежуточное представление, используемое компиляторами, в котором каждой переменной значение присваивается лишь единожды. Single — означает, что каждый регистр присваевается ровно один раз (это свойство упрощает анализ потока данных)
- Для обработки переменных, которым присвоено значение более одного раза в исходном коде используется понятие PHI-функции. Инструкция phi по существу возвращает одно значение из набора входящих значений, основываясь на пути потока управления (runtime).

LLVM IR


C
C++
Go
Rust
Toy
Front-end
Clang
Gollvm
rustc
toyc
LLVM IR
Middle-end
LLVM optimizer
LLVM IR
Back-end
LLVM static compiler
x86
ARM
RISC-V
MIPS
PowerPC
...

LLVM - Clang

llc - LLVM static compiler.

- ccc-print-phases — печать используемых оптимизаций

$ clang -emit-llvm test.cpp -S -O0 test.cpp

- На выходе получим ir-файл test.ll

$ llc test.ll

- На выходе получим test.s

$ clang test.s

- На выходе получится бинарный файл a.out

```
clang -help
clang -help-hidden
clang -cc1 -help
clang -cc1 -help-hidden
```

Basic block

- Это последовательность инструкций, сопровождаемых инструкцией ветвления(branch). Основная идея basic block — если выполняется одна инструкция из него, то выполняются и все последующие.
- PHI функции, при условии использования базовых блоков, на каждое входное значение имеют базовый блок предшественника.

```
; <label>:0
    switch i32 %a, label %default [ i32 42, label %case1]
case1:
    %x.1 = mul i32 %a, 2
    br label %ret
...
ret:
    %x.0 = phi i32 [ %x.2, %default ], [ %x.1, %case1 ]
    ret i32 %x.0
```

```
; Global variable initialized to the 32-bit integer value 21.
@foo = global i32 21

; f returns 42 if the condition cond is true, and 0 otherwise.
define i32 @f(i1 %cond) {

; Entry basic block of function containing zero non-branching instructions and a
; conditional branching terminator instruction.
entry:

    ; The conditional br terminator transfers control flow to block_1 if %cond
    ; is true, and to block_2 otherwise.
    br i1 %cond, label %block_1, label %block_2

; Basic block containing two non-branching instructions and a return terminator.
block_1:
    %tmp = load i32, i32* @foo
    %result = mul i32 %tmp, 2
    ret i32 %result

; Basic block with zero non-branching instructions and a return terminator.
block_2:
    ret i32 0
}
```

Fast IR

```
#region 43: PM64; eva=0x4eff49; ginsts=26; LI=26; LS=32; G2
```

```
bb 1: entry; preds b12;
   7. ld.32              rzero,r14,i:2,i:851a80 -> r15
   8. move.8             rzero,r15 -> r0
  13. ld.32              r4,rzero,i:0,i:a8 -> r17
  24. sub.64.f           r0,r17 -> r10,ef
  15. brcrel.ne.1.p      ef,i:0,i:0
T[b1 -> b9], F[b1 -> b10]
```

```
bb 10: preds b1;
  18. br            i:2,i:0
T[b10 -> b0]
```

```
bb 9: preds b1;
T[b9 -> b3]
```

```
bb 3: preds b9;
  19. and.32.f          r0,r0 -> rzero,ef
  20. brcrel.ne.p       ef,i:0,i:0
T[b3 -> b5], F[b3 -> b4]
```

Control Flow Graph

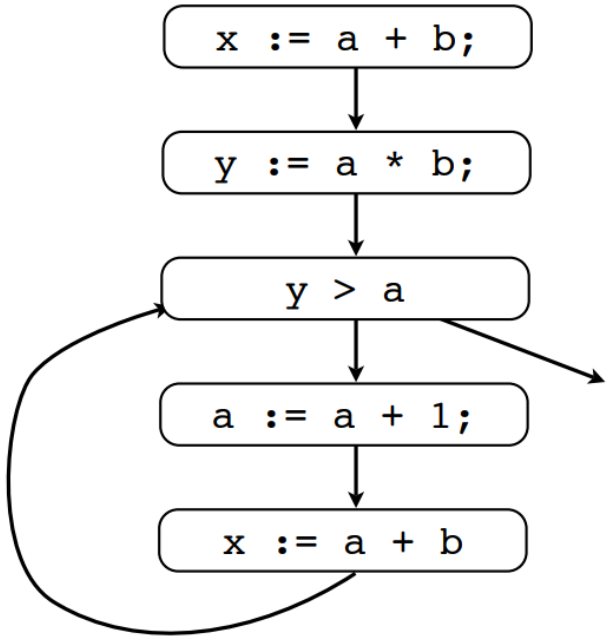
- CFG — это вспомогательный к программе граф, в котором находится информация о изменении потока управления, то есть по-сути является графом построенным на Basic Blocks.
- Является ориентированным графом (орграф), в котором:
  - Каждым узлом является оператор или Basic Block
  - Дуги представляют собой поток управления
- Под операторами подразумеваются:
  - Присваивания («x := y» или «x := y op z» или «x := op y»)
  - Операторы ветвления («goto L» или «if b then goto L»)

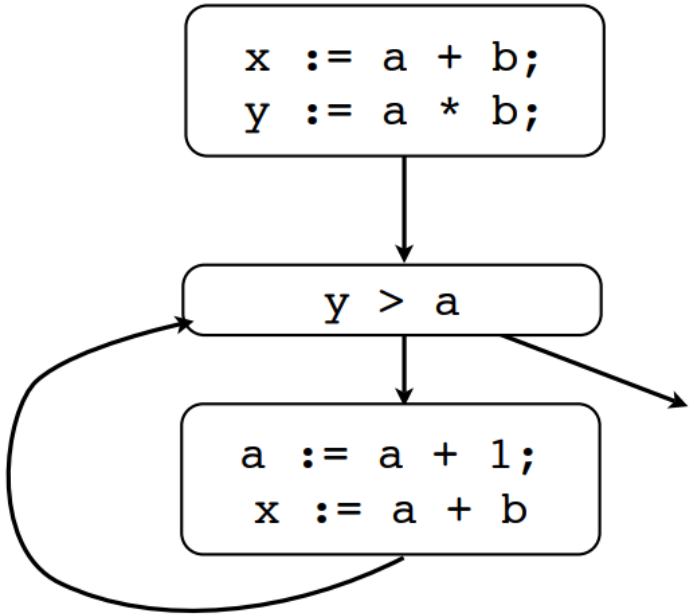
Data Flow Graph

- DFG — так же является вспомогательным графом для внутреннего представления (ir), но содержит в себе информацию об изменении потока данных
- Так же является ориентированным графом (орграф):
  - Каждым узлом является регистр или память
  - Дуги представляют собой следующее использование регистра или памяти
- В случае, если мы работаем с ssa-формой, то phi-функции собирают множественные входные дуги в один.

Control Flow Graph

```
x := a + b;
y := a * b;
while (y > a) {
  a := a + 1;
  x := a + b
}
```





CFG на Basi Blocks

LTO

- Единица трансляции — один обьектный файл
- LTO включает оптимизации между единицами трансляций, например появляется возможность сделать inline для маленьких функций, определенных в разных файлах.
- Для работы lto недостаточно бинарного обьектника — в *.o файл необходимо добавить простое ir-представлени.
- LTO работает на этапе линковки, однако для нее необходимо включение информации при сборке обьектников.