CAP— Compilation (#5) : Intermediate Representations: CFG, Local optimisations
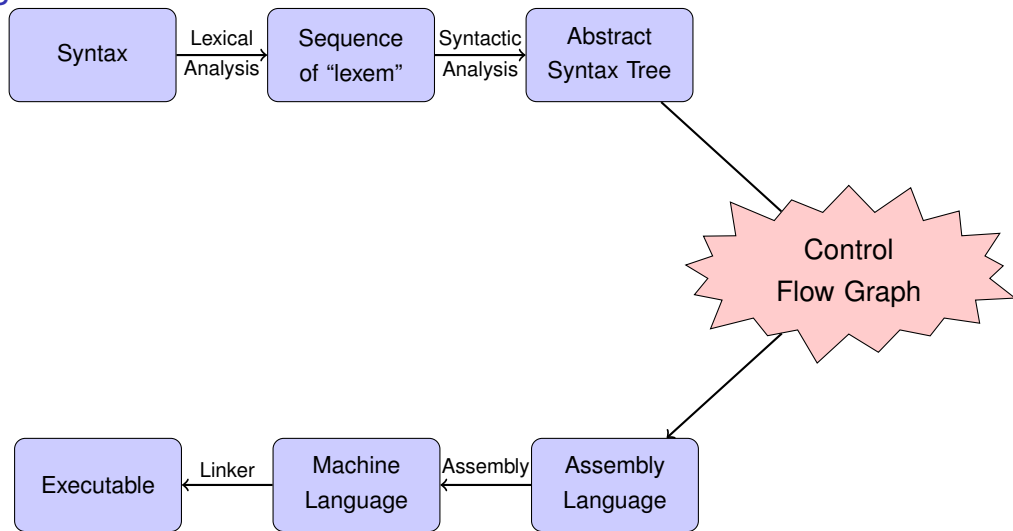
Laure Gonnord & Matthieu Moy & Gabriel Radanne & other

https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2025-2026

Lyon 1

ENS DE LYON

Big Picture


```
┌──────────┐  Lexical    ┌──────────┐  Syntactic  ┌──────────┐
│  Syntax  │───────────▶│ Sequence │───────────▶│ Abstract │
│          │  Analysis   │ of "lexem"│  Analysis  │ Syntax Tree│
└──────────┘             └──────────┘            └──────────┘
                                                       │
                                                       ▼
                                                  ┌─────────┐
                                                  │ Control │
                                                  │Flow Graph│
                                                  └─────────┘
                                                       │
                                                       ▼
┌──────────┐  Linker     ┌──────────┐  Assembly   ┌──────────┐
│Executable│◀───────────│ Machine  │◀───────────│ Assembly │
│          │             │ Language │             │ Language │
└──────────┘             └──────────┘            └──────────┘
```

3 address construction "problems"

Temporary reuse ?

```
li temp3, 4
mv temp0, temp3
;; temp3 is never used again
li temp4, 0
mv temp1, temp4
```

temp3 and temp4 could be mapped
to the same physical location.

```
li temp5, 4
bge ..., else
  ;; temp5 not used. Its physical
   location can be shared.

   j end
else:
   mv temp6, temp5
end
```

▶ **straight-line code** is difficult to reason on.

A first IR

Need a better data structure to propagate and infer information. We need to:

- Reason about the flow of the program.
- Embeds our three address code.

▶ Control-Flow Graph.

Definitions

Definition (BB=Basic Block)

Basic block: largest (3-address RISCV*) instruction sequence without label.*
(except at the first instruction) and without jumps and calls.

Definition (CFG=Control Flow Graph)

It is a directed graph whose vertices are basic blocks, and edge $B_1 \rightarrow B_2$ exists if
B_2 can follow immediately B_1 in an execution.

▶ two optimisation levels: local (BB) and global (CFG)

An example 1/2

Let us consider the program:

```
int x,y;
if (x<4)
    y=7;
else
    y=42;
x=10;
```

We already generated the (linear code) for a large part of it.

An example 2/2

```
    li temp3, 4
    li temp2, 0
    bge temp0, temp3, lbl0
    li temp2, 1
lbl0: # if false, jump (skip the 'then')
    bge temp2, 0, lelse1
    li temp4, 7
    mv temp1, temp4 # y gets 7
    jump lendif1
lelse1:
    li temp4 42
    mv temp1, temp4 # y gets 42
lendif1:
    li temp5, 10
    mv temp0, temp5 # end
```

An example 2/2

```
    li temp3, 4
    li temp2, 0
    bge temp0, temp3, lbl0
    li temp2, 1
lbl0:  # if false, jump (skip the 'then')
  bge temp2, 0, lelse1
  li temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1:
  li temp4 42
  mv temp1, temp4  # y gets 42
lendif1:
  li temp5, 10
  mv temp0, temp5  # end
```

start

```
li temp3 , 4
li temp2 , 0
temp0 >= temp3 ?
```

```
li temp2 , 1
```

lbl0

```
temp2 >= 0 ?
```

lelse1

```
li temp4 , 7
mv temp1 , temp4
```

```
li temp4 , 42
mv temp1 , temp4
```

lendif1

```
li temp5 , 10
mv temp0 , temp5
```
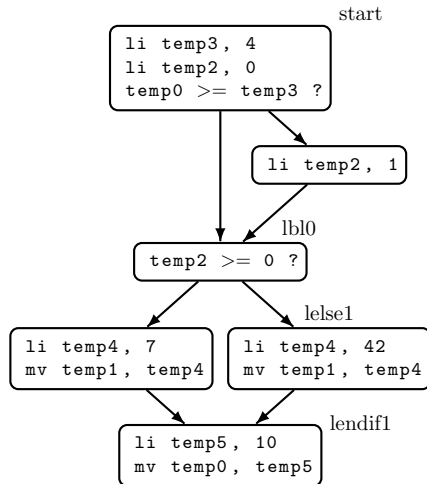
Identifying Basic Blocks (from 3 address code)

- The first instruction of a basic block is called a **leader**.

- We can identify leaders via these three properties:
  1. The first instruction in the intermediate code is a leader.
  2. Any instruction that is the target of a conditional or unconditional jump is a leader.
  3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

- Once we have found the leaders, it is straighforward to find the basic blocks: for each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.

Exercise: CFG by hand

```
  li temp3, 4
  li temp2, 0
  bge temp0, temp3, lbl0
  li temp2, 1
lbl0:  # if false, jump (skip the 'then')
  bge temp2, 0, lelse1
  li temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1:
  li temp4 42
  mv temp1, temp4  # y gets 42
lendif1:
  li temp5, 10
  mv temp0, temp5  # end
```

- Identify the Leaders using the three properties
    1. The first instruction is a leader.
    2. Any instruction that is the target of a conditional or unconditional jump is a leader.
    3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Identify the Blocks
- Draw the Transitions

Big picture (Basic Block Optimisation)

- Front-end $\rightarrow$ a CFG where nodes are basic blocks.

- Basic blocks $\rightarrow$ DAGs that explicit common computations

```
u1 := c - d
u2 := b + u1
u3 := a * u2
u4 := u2 * u1
u5 := u3 + u4
```
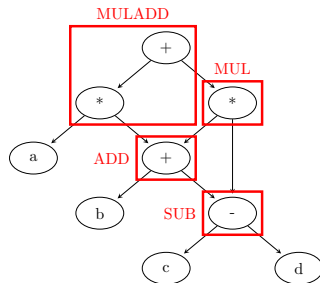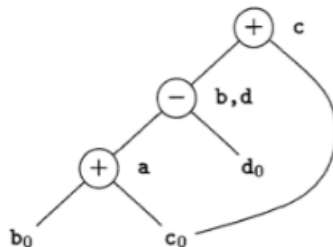

▶ Choose instructions (**selection**) and order them (**scheduling**).

An Example of BB DAG construction

```
a = b + c
b = a - d
c = b + c
d = a - d
```


Useful links : https://www.youtube.com/watch?v=PXTKWvyQUwE and

https://www.cse.iitm.ac.in/~krishna/cs3300/pm-lecture3.pdf for other BB optimisations.

Instruction Selection, in general

The problem:

- a list of instructions/operations that compute one or more expressions.
- map these operations in "real machine instructions".
- at minimum cost.

Instruction Selection

The problem of selecting instructions is a DAG-partitioning problem. But what is the objective ?
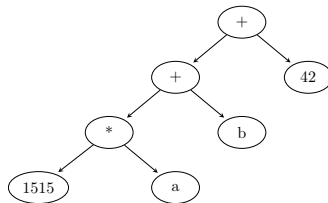
The best instructions:

- cover bigger parts of computation.
- cause few memory accesses.

▶ Assign a cost to each instruction, depending on their addressing mode.

Instruction Selection: an example
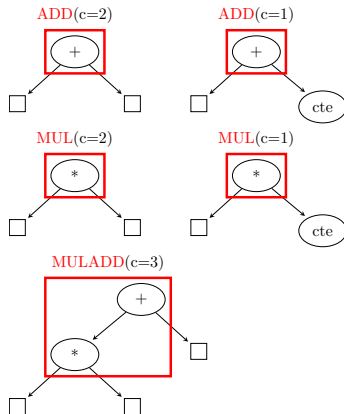
(In RiscV, mulladd is `fmadd` (fuse multiply add) and "add with constants" is `addi`).

What is the optimal instruction selection for:

▶ Finding a tiling of minimal cost: it is **NP-complete** (SAT reduction).

Tiling trees / DAGs, in practice

For tiling:

- There is an optimal algorithm for **trees** based on dynamic programming.
- For DAGs we use heuristics (decomposition into a forest of trees, . . .)

▶ The literature is plethoric on the subject.

Instruction Selection, in our compiler

Mapping one to one. We don't use `fmadd`.

2 Local optimizations

- Basic Blocks DAG Construction
- Instruction Selection
- Instruction Scheduling

Instruction Scheduling, in general

The problem:

- change the order of instructions.
- to "optimise'.
- without "cutting dependencies".

Instruction Scheduling, what for?

We want an evaluation order for the instructions that we choose with **Instruction Scheduling**.

A scheduling is a function θ that associates a **logical date** to each instruction. To be correct, it must respect data dependencies:

```
(S1) u1 := c - d
(S2) u2 := b + u1
```

implies $\theta(S_1) < \theta(S_2)$. We can choose $\theta(S_1) = 0, \theta(S_2) = 1$

▶ How to choose among many correct schedulings? depends on the target architecture.

Architecture-dependant choices

The idea is to exploit the different ressources of the machine at their best:

- instruction parallelism: some machines have parallel units (subinstructions of a given instruction).

- prefetch: some machines have non-blocking load/stores, we can run some instructions between a load and its use (hide latency!)

- pipeline.

- registers: see next slide.

(sometimes these criteria are incompatible)

Register use

Some schedules induce less **register pressure**:
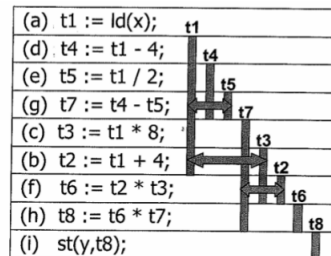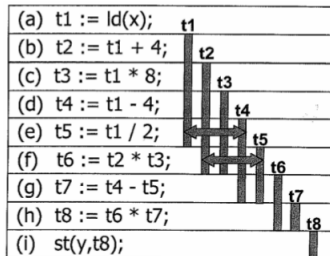
In this picture the dates of the instructions are implicit : line 1 is date 1, line 2 is date 2. . .

▶ How to find a schedule with less register pressure?

Scheduling wrt register pressure

Result: this is a linear problem on trees, but NP-complete on DAGs (Sethi, 1975).

▶ Sethi-Ullman algorithm on trees, heuristics on DAGs

A slight variation of this algorithm can be found on Wikipedia, the leaves values here are chosen equal to 1 since our machine does not have any direct access to constant values.

Sethi-Ullman algorithm on trees

$\rho(node)$ denoting the number of (pseudo)-registers necessary to compute a node:

- $\rho(leaf) = 1$

- $\rho(nodeop(e_1, e_2)) = \begin{cases} max\{\rho(e_1), \rho(e_2)\} & \text{if } \rho(e_1) \neq \rho(e_2) \\ \rho(e_1) + 1 & \text{else} \end{cases}$

(the idea for non "balanced" subtrees is to execute the one with the biggest ρ first, then the other branch, then the op. If the tree is balanced, then we need an extra register)

▶ then the code is produced with postfix tree traversal, the biggest register consumers first.

Sethi-Ullman algorithm on trees - an example

Min number of (additional) registers for $b^2 + 4ac$ with a,b,c already in registers ?
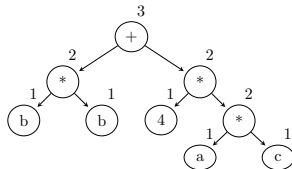
The tree traversal then produces the following code:

	tmp_1	tmp_2	tmp_3	tmp_4
mul tmp1, b, b				
mul tmp2, a, c				
li tmp3, 4				
mul tmp4, tmp2, tmp3				
add tmp5, tmp1, temp4				

cells in black denote for each instruction the set of <u>entry alive</u> temporaries.

Conclusion (instruction selection/scheduling)

Plenty of other algorithms in the literature:

- Scheduling DAGs with heuristics, . . .
- Scheduling loops (M2IF course on advanced compilation)

Practical session:

- we have (nearly) no choice for the instructions in the RISCV ISA.
- evaluating the impact of scheduling is a bit hard.

We won't implement any of the previous algorithms.

1 Control flow Graph

2 Local optimizations

3 Global optimizations
- Introduction to register allocation
- Analysis for optimizations : Liveness

Global optimizations

So far, we have taken advantage of basic blocks to make <u>local</u> optimizations,
where we do not need to take care of control flow.
This is not sufficient for all optimizations !

Global optimizations

So far, we have taken advantage of basic blocks to make <u>local</u> optimizations,
where we do not need to take care of control flow.

This is not sufficient for all optimizations !

- Global Dead Code Elimination

- Constant Folding

- Loop optimizations

- . . .

- **Register allocation**

Global optimization in practice

Let's optimize this function:

```
int f(int a, int b) {
  x=a+b;
  y=a*b;
  while(y*y>a+b) {
    a=a+a;
    x=a+b;
  }
  return x;
}
```

What for?

- Finding storage locations to the values manipulated by the program ▶ registers or memory.
- registers are fast but in small quantity.
- memory is plenty, but slower access time.

▶ A good register allocator should strive to keep in registers the variables used more often.

> "Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important - if not the most important - of the optimizations."

Hennessy and Patterson (2006) - [Appendix B; p. 26]

What for?

Expected behavior of **register allocation**:

- Input: a CFG with basic blocks with 3-address code (and pseudo-registers, aka temporaries)
- Output: same CFG but without pseudo-registers:
    - replace with physical registers as much as possible.
    - if not **spIill**, ie allocate a place in memory.
    - use the same physical register (or memory location) for as many temporaries as possible.
    - all copies assigned to the same physical registers ("moves") can be removed: **coalescing** (**optional**).

The key notion: liveness

Observation

Two variables that are simultaneously **alive** must be assigned different registers.

(formal definition of alive follows)

Register assignment is NP-complete

Theorem

Given P and K general purpose registers, is there an assignment of the variables P in registers, such that (i) every variable gets at least one register along its entire live range, and (ii) simultaneously live variables are given different registers ?

Gregory Chaitin has shown, in the early 80's, that the register assignment problem is NP-Complete (register allocation via coloring, 1981)

Liveness analysis

Previously we called **variable** a pseudo-register or a physical register.

Definition (Alive Variable)

In a given program point, a variable is said to be <u>alive</u> if the value she contains may be used in the rest of the execution.
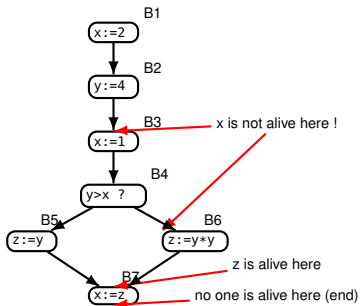
May: non decidable property ▶ overapproximation.

Important remark: here a block = a statement/program point. We have the same kind of analyses with block=basic block.
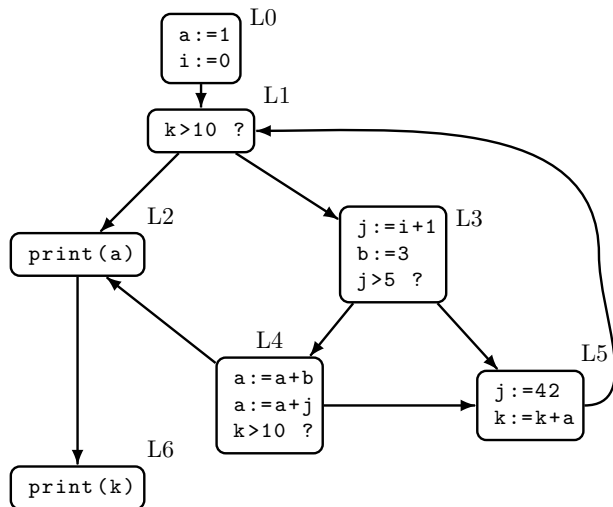
An example for live ranges

Definition

A variable is **live** at the exit of a block if there exists a path from the block to a use of the variable that does not redefine the variable.

```
x:=2;
y:=4;
x:=1;
if (y>x)
    then z:=y
    else z:=y*y ;
x:=z;
```
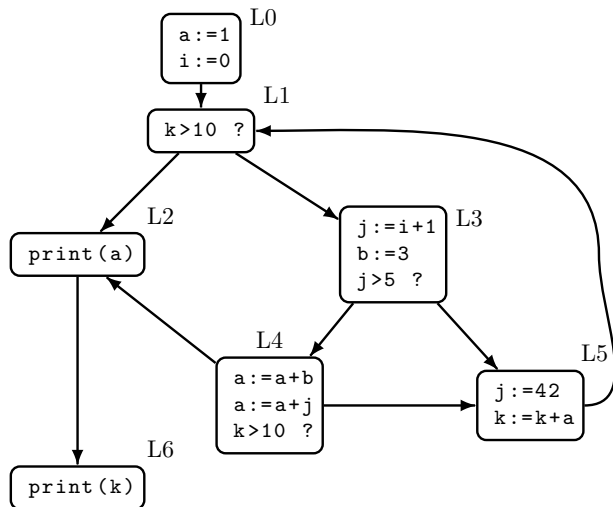

B1 `x:=2`

B2 `y:=4`

B3 `x:=1`   x is not alive here !

B4 `y>x  ?`

B5 `z:=y`   B6 `z:=y*y`

B7 `x:=z`   z is alive here

no one is alive here (end)

▶ The information flow is **backward**: from uses to definitions.

Liveness by hand!

bloc	live variables at bloc exit
L0	
L1	
L2	
L3	
L4	
L5	
L6	$\emptyset$

Liveness by hand!

bloc	live variables at bloc exit
L0	a, i, k
L1	a, i, k
L2	k
L3	i, j, a, k, b
L4	a, i, k
L5	a, i, k
L6	∅

How to compute liveness

Dataflow analysis is a technique to compute many properties.

- Very versatile
- Expensive in general (fix point on the CFG)
- ▶ Next year in the Static Analysis course!

Computing liveness: an alternative approach

Instead, we will use an <u>alternative CFG representation</u> that makes it easy to compute liveness and do program transformations!

▶ Next lesson: The **Single Static Assignment** representation

Summary

1. Control flow Graph

2. Local optimizations
   - Basic Blocks DAG Construction
   - Instruction Selection
   - Instruction Scheduling

3. Global optimizations
   - Introduction to register allocation
   - Analysis for optimizations : Liveness