50.051 Programming Language Concepts

W13-S1 Backend and End

Matthieu De Mari

At this point

At this point, we have optimized IR code that needs to be converted into the target language (e.g. assembly, machine code).

Goal of this stage:

- Choose the appropriate machine instructions for each IR instruction.
- Allocate the processor resources (registers, caches, etc.)
- Implement low-level details of the runtime environment.
- Machine-specific optimizations are often done here, though some are often treated as part of a final optimization phase.

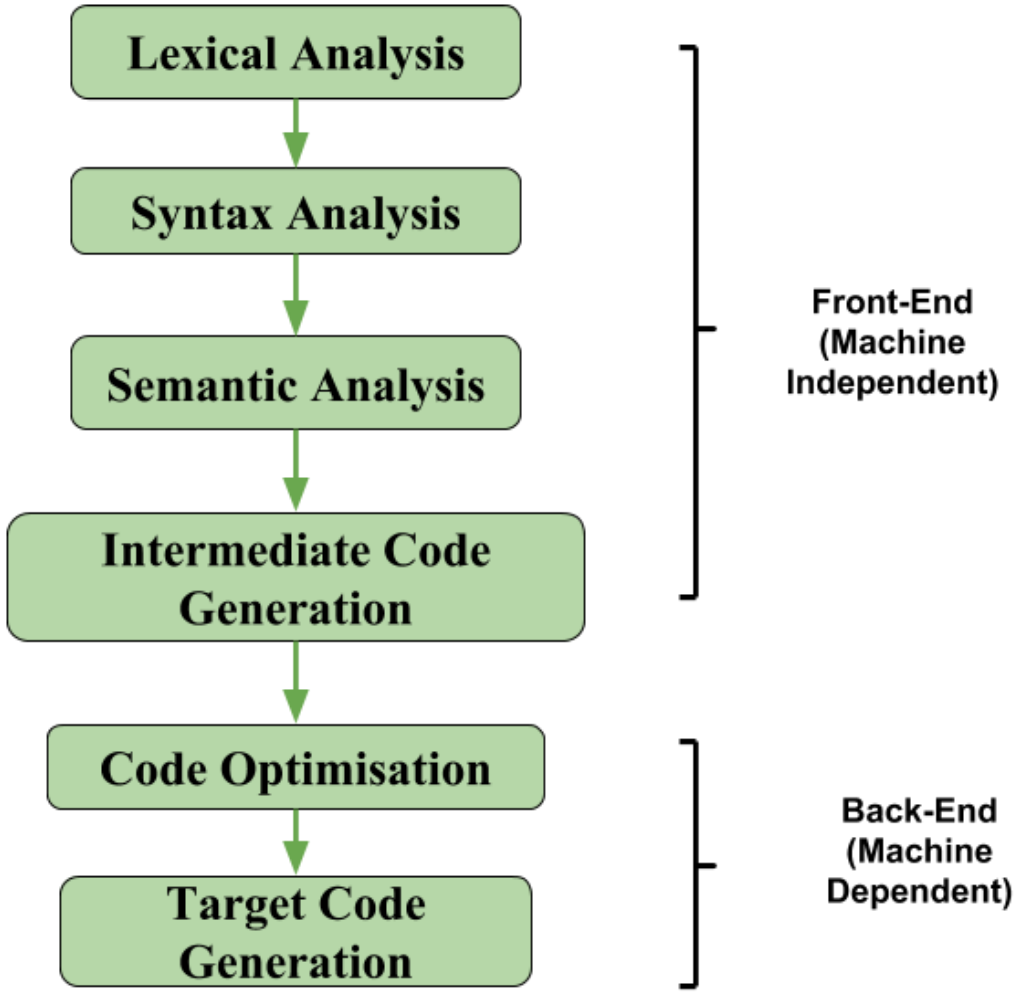
The back-end of a compiler

Definition (The back-end part of a compiler):

The **back-end of a compiler** takes the optimized and transformed code generated by the middle-end and translates it into machine code that can be executed.

It involves tasks, such as:

- **Instruction selection/ordering**,
- **Register allocation**,
- and **Code generation**.



Source code (C)

```
int main() {
    int x = 10;
    int y = 5;
    int z = x + y;
    return z;
}
```

Front-end

Lexical Analysis
Syntax Analysis
Semantic Analysis

After front-end, intermediate code (Three-address) and optimizations

```
t1 = 10
t2 = 5
t3 = t1 + t2
z = t3
return z
```

Machine instructions after code generation and execution

```
movl 10, Reg1
movl 5, Reg2
add Reg3, Reg1, Reg2
ret  Reg3
```

Instruction Selection

Definition (Instruction Selection):

During **instruction selection**, the compiler chooses the specific machine code instructions (mov, add, jmp, etc) to use to implement each operation in the program.

During this step, it might perform some final transformations on the generated code to improve performance and reduce code size.

Typically, there might be multiple ways to choose and order machine code instructions. Some might be equivalent, others might allow to save memory/register space, etc.

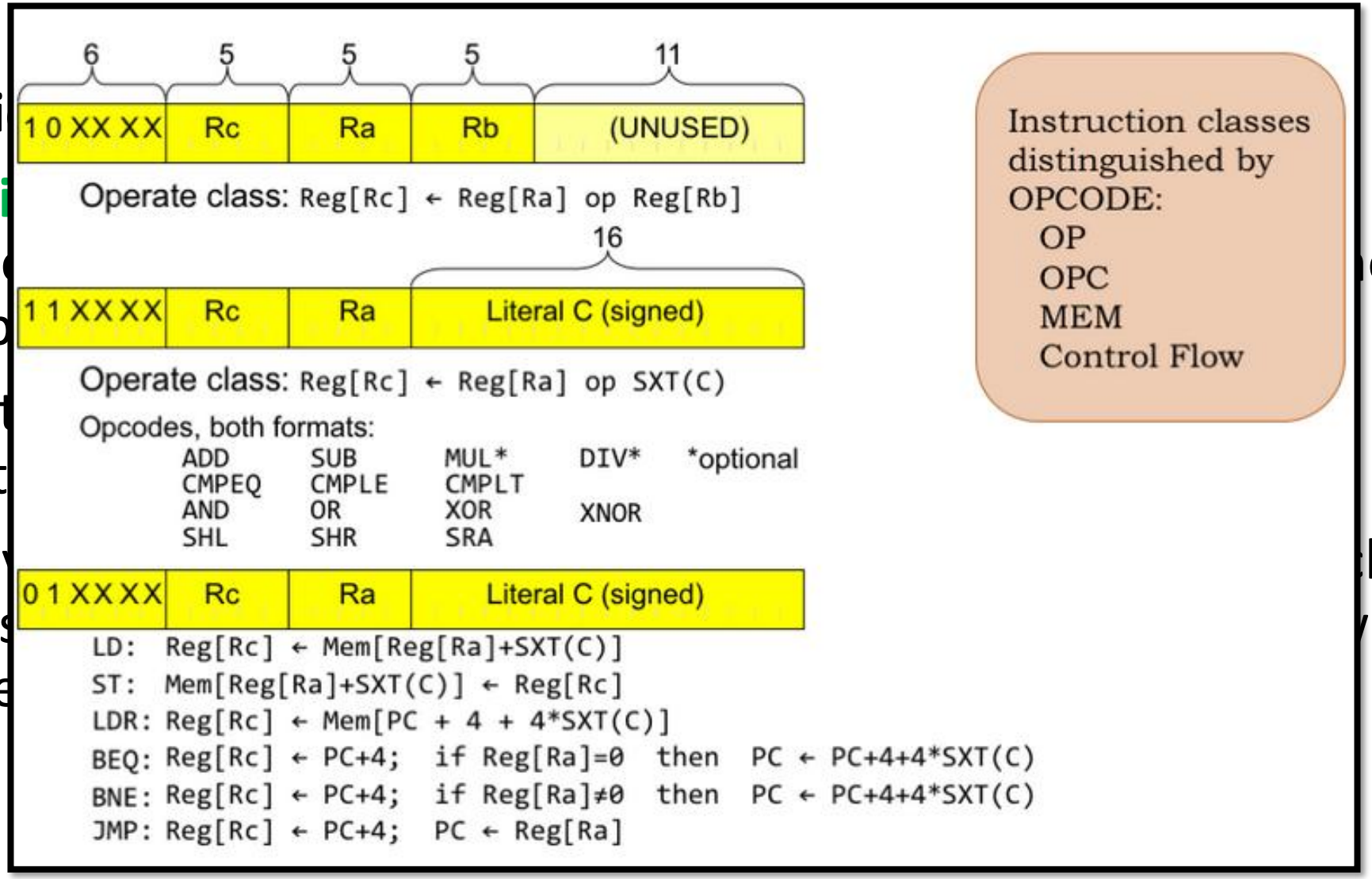
Instruction Selection

Definiti

During i
machin
each op

During t
generat

Typicall
code ins
save me



Instruction Ordering/Scheduling

Definition (Instruction Ordering/Scheduling):

The **instruction ordering** (or **scheduling**) phase is responsible for rearranging the order of instructions in the program to improve performance.

This typically involves reordering instructions to take advantage of the parallelism and pipelining features of modern processors (with many cores running in parallel, for instance).

The instruction scheduler takes into account the dependencies between instructions and the specific features of the processor.

(Too advanced, out-of-scope).

Intuition for code generation

Consider the **TAC code** below,

t1 = a + b

*t2 = t1 * c*

Intuition: This TAC code could be easily **mapped to a sequence of machine code instructions**, as shown on the right.

https://natalieagus.github.io/50002/notes/instructionset#opcode

Machine Code?

LOAD R1, a

LOAD R2, b

ADD R1, R1, R2

STORE t1, R1

LOAD R3, t1

LOAD R4, c

MUL R3, R3, R4

STORE t2, R3

Breakdown of the Machine Code Translation

This final translation is typically broken down in a few steps.

- **Basic block identification:** The compiler identifies the basic blocks in the TAC code, i.e. sequences of instructions that are always executed together and can be used as units for generating machine code.
- **Instruction selection:** For each operation in the TAC, the compiler selects an appropriate instruction or sequence of instructions in the machine assembly language. The choice may depend on the available instruction set, addressing modes, and architectural constraints.

For simplicity, we will stick to an instruction set similar to the one you used in 50.002 Computation Structures!
(https://natalieagus.github.io/50002/notes/instructionset)

Instruction Selection (Simple Version?)

In our simpler version: the instruction selection, as a mapping could be easily done, by using a Syntax-Directed Translation (another one!) on our three-address code CFG!

$R_i \rightarrow \mathbf{c}_a$		{ LD Ri, #a }
$R_i \rightarrow M_x$		{ LD Ri, x }
$M \rightarrow = M_x\ R_i$		{ ST x, Ri }
$M \rightarrow = \mathbf{ind}\ R_i\ R_j$		{ ST *Ri, Rj }
$R_i \rightarrow \mathbf{ind} + \mathbf{c}_a\ R_j$		{ LD Ri, a(Rj) }
$R_i \rightarrow + R_i\ \mathbf{ind} + \mathbf{c}_a\ R_j$		{ ADD Ri, Ri, a(Rj) }
$R_i \rightarrow + R_i\ R_j$		{ ADD Ri, Ri, Rj }
$R_i \rightarrow + R_i\ \mathbf{c}_1$		{ INC Ri }
$R \rightarrow \mathbf{sp}$		
$M \rightarrow \mathbf{m}$		

Instruction Selection (Hard Version?)

Advanced compilers, however, might often rely on more advanced techniques, like tree matching and dynamic programming.

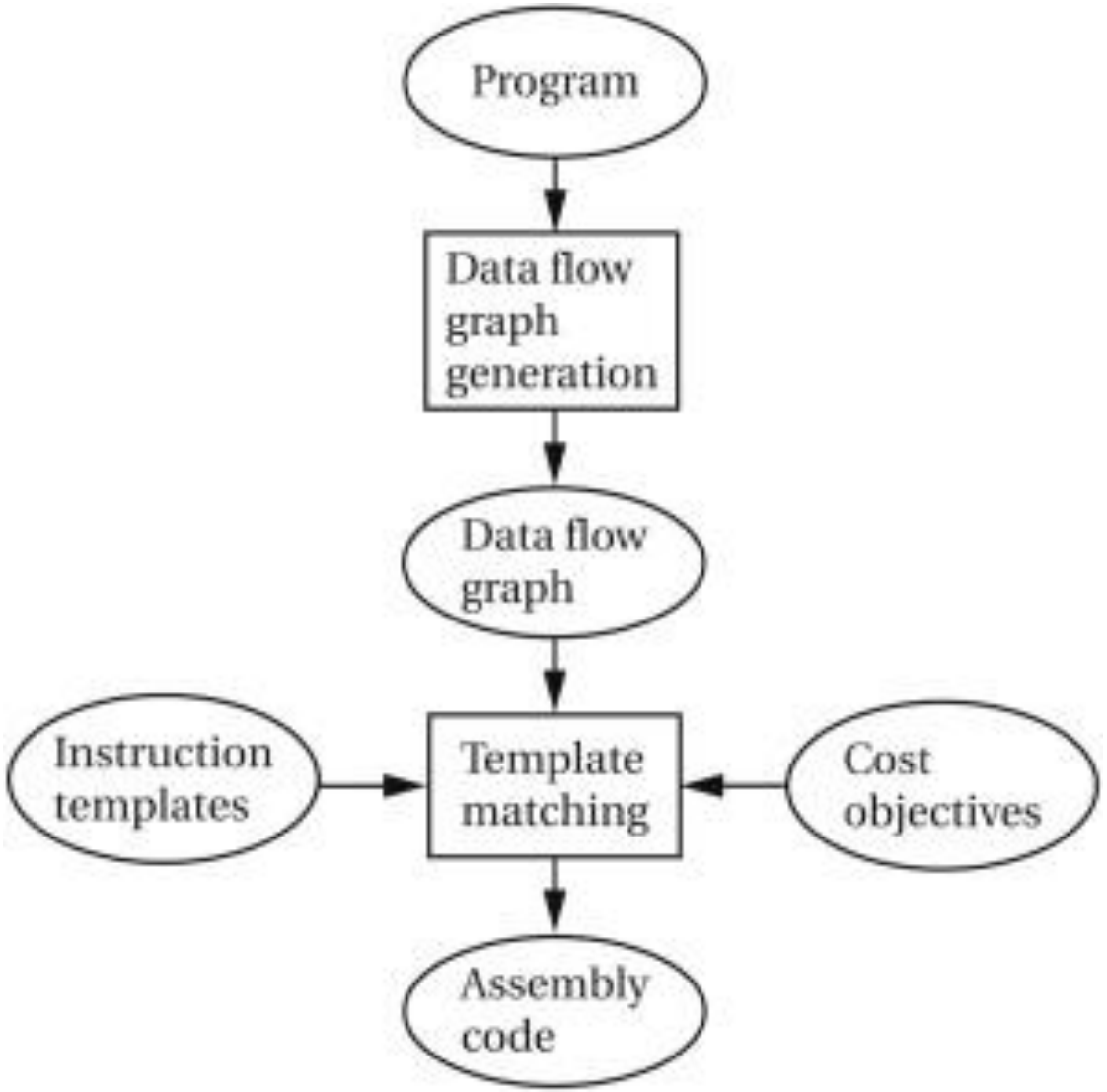
- Tree pattern matching involves representing TAC operations as trees and searching for matching patterns in a database of target machine instructions.
- Other techniques, like optimal code generation using dynamic programming, find the optimal sequence of target instructions by considering all possible combinations and selecting the one with the lowest cost (memory, computation, time, etc.).

This last set of techniques is much more advanced and out-of-scope.

Instruction Selection (mapping vs. advanced)

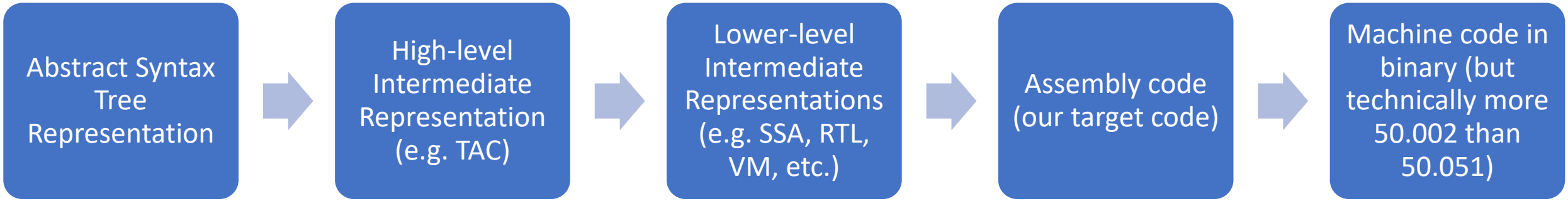
Note: These advanced techniques will often require solving optimization problems on graphs that are NP-Hard!

Definitely out-of-scope!
(It would require an advanced course on Graph Theory to simply explain some of them!)



Instruction Selection (Harder Version?)

In addition, advanced compilers, might also **use several intermediate representations in a row**, instead of jumping straight from TAC to machine code.



Intuition for register allocation

Let us assume that we have successfully transformed our TAC code into a machine/target code.

This code has labeled registers to use, as R_n with n being integers, but it assumes we have an infinite number of registers.

In practice, that is never the case.

Machine Code?

LOAD R1, a

LOAD R2, b

ADD R1, R1, R2

STORE t1, R1

LOAD R3, t1

LOAD R4, c

MUL R3, R3, R4

STORE t2, R3

Memory Hierarchy

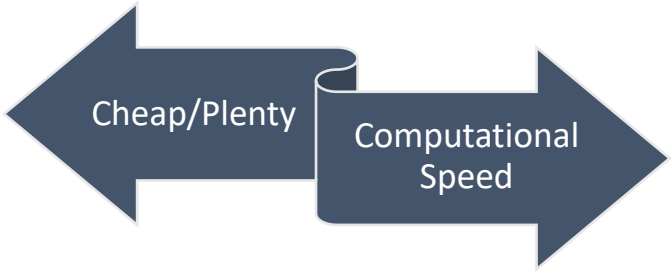
Definition (Memory Hierarchy):

In practice, not all memory locations are equal, there is a memory hierarchy.

- **Registers:** very few of them, fast and expensive.
- **SRAM:** few of them, slightly slower, but still quite fast.
- **DRAM:** more of them, slower.
- **Disk:** Potentially unlimited space, but very slow.

Need a refresher from 50.002?

https://natalieagus.github.io/50002/notes/memoryhierarchy



Types	Space	Latency	Cost
Register	100's of bytes	$20ps$	$$$$
SRAM	100's of Kbytes	$1ns$	$$$
DRAM	100's of Mbytes	$40ns$	$
Disk	100's of Gbytes	$10ms$	c

A resource allocation problem

How would you decide which variables to prioritize and assign to registers, and which variables to keep in other memory locations?

- **Objective:** Position objects in memory in a way that takes maximum advantage of the memory hierarchy and speeds up computation.
- **Additional challenge:** Do so without hints from the programmer.
- **Why it is important:** A good register allocator can generate code orders of magnitude better than a bad register allocator.

Possible algorithms to investigate

The register allocation problem is not an easy one to solve! (NP-hard).

We propose to briefly explore three algorithms for register allocation:

- **Naïve (“no”) register allocation.**
- **Linear scan register allocation.**
- **Graph-coloring register allocation.**

Their implementations might require tools from graph theory (especially the last one), and are therefore out-of-scope.

A starting idea

Starting idea: Store every value in main memory, loading values only when they are needed.

To generate a code that performs a computation:

- Generate load instructions to pull the values from main memory into registers.
- Only use three registers at any given time, because that is the maximal number of operands in TAC/assembly instruction anyway.
- Generate code to perform the computation on these registers.
- Generate store instructions to store the result back into main memory.

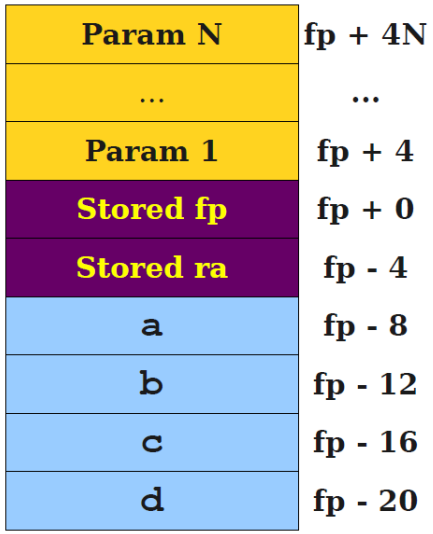
A starting idea

Our TAC code (for a function of some sort)

a = b + c;

d = a;

c = a + d;




```
lw $t0, -12(fp)
lw $t1, -16(fp)
add $t2, $t0, $t1
sw $t2, -8(fp)
lw $t0, -8(fp)
sw $t0, -20(fp)
lw $t0, -8(fp)
lw $t1, -20(fp)
add $t2, $t0, $t1
sw $t2, -16(fp)
```

Analysis of this first naïve register allocation

Disadvantage: Gross inefficiency.

- Issues unnecessary loads and stores by the dozen.
- Wastes space on values that could be stored purely in registers.
- Easily an order of magnitude or two slower than necessary.
- Unacceptable in any production compiler.

Advantage: Simplicity.

- Can translate each piece of IR directly to assembly as we go.
- Never need to worry about running out of registers.
- Never need to worry about function calls or special-purpose registers.
- Good if you just needed to get a prototype compiler up and running.

A slightly better approach

New idea: In practice, we have very few registers, but definitely more than just three! Try to hold as many variables in registers as possible.

- Hopefully, this will reduce memory reads/writes and reduces total memory usage.

However, we will need to address these questions:

- Which registers do we put variables in?
- How do we recognize that we have run out of registers?
- What do we do when we run out of registers?

A slightly better approach

Fact #1: At each program point, each register holds at most one live variable (otherwise, there is a memory conflict).

- This means that we could assign several variables the same register, as long as no two of them ever will be read together.
- The **liveness analysis** we used earlier in the IR optimization phase could therefore hold the key as to which variables need to be live (and therefore in a register somewhere) at any given point?
- At any point of the program, **liveness analysis lets us know when variables are live and for how long.**

Live range and live interval

Definition (live range):

The **live range** for a variable is the set of program points at which that variable is live.

Definition (live interval):

The **live interval** for a variable is the smallest subrange of the IR code containing all a variable's live ranges.

- A property of the IR code, not the CFG.
- Less precise than live ranges, but simpler to work with.

Live range and live intervals: demo

Consider the TAC code below.

L0:

e = d + a

f = b + c

f = f + b

if e goto L1

d = e + f

jump L2;

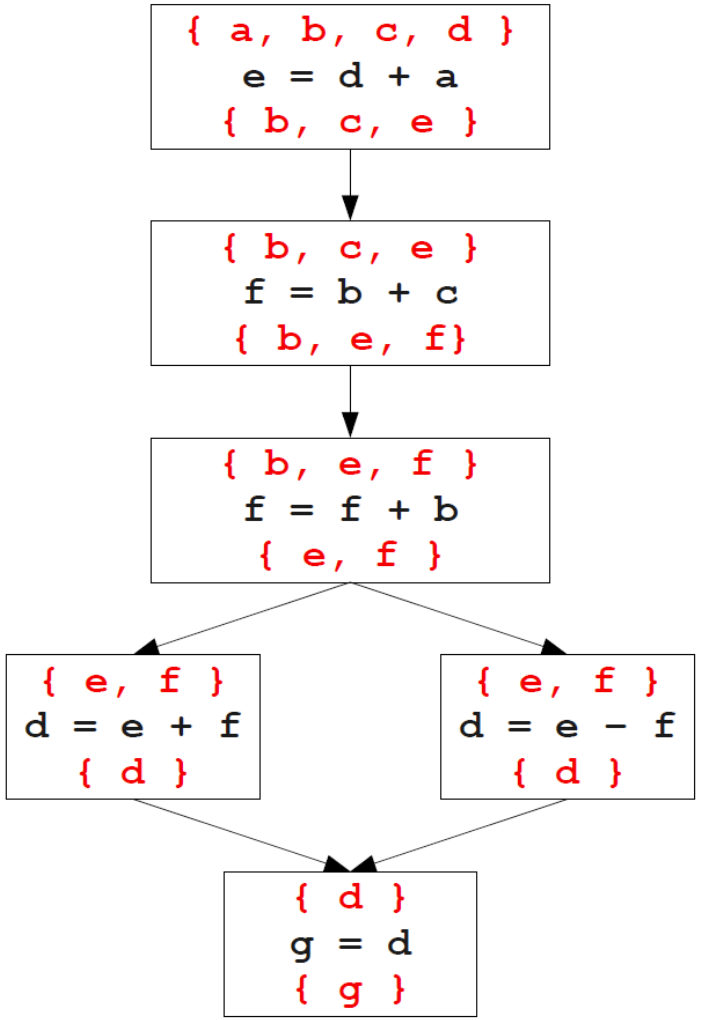
L1:

d = e - f

L2:

g = d

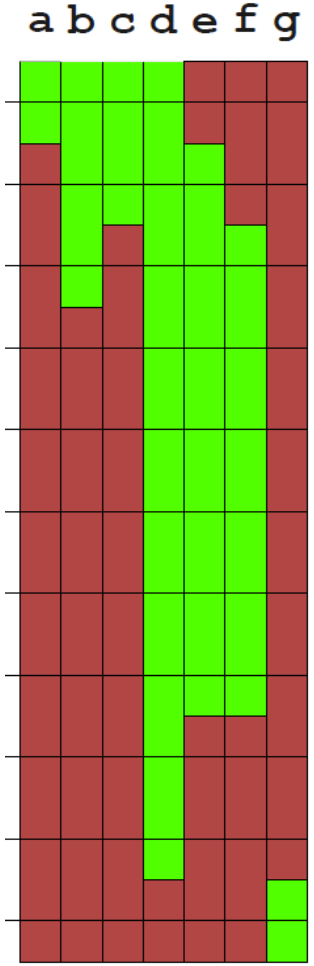
After control-flow and liveness analysis



Live range and live intervals: demo

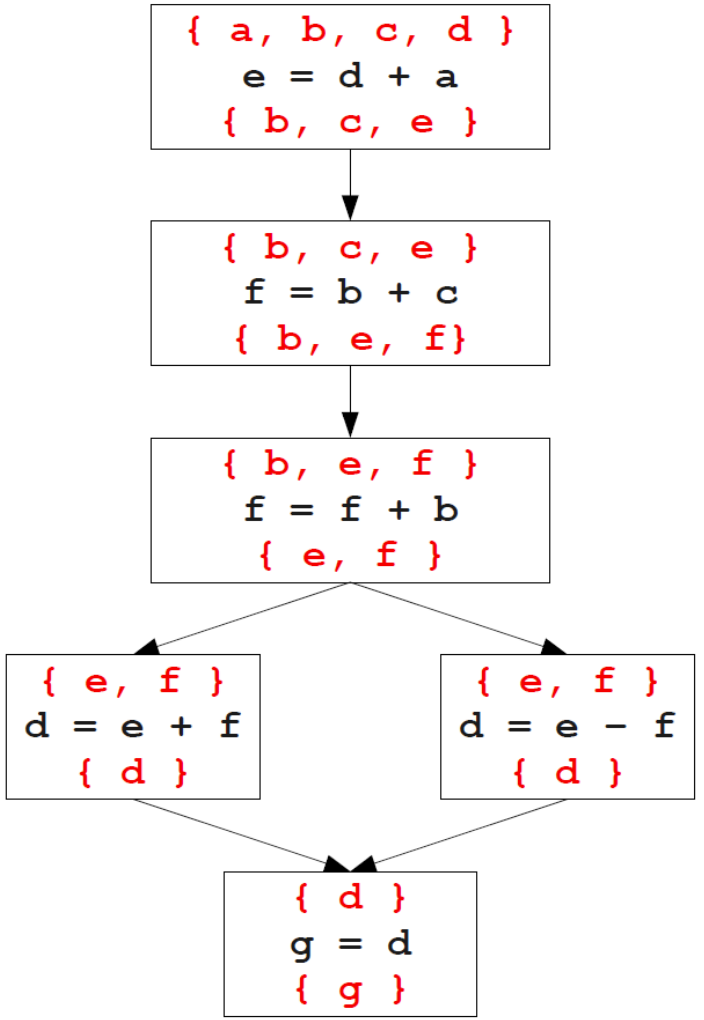
Consider the TAC code below.

Live intervals




```
L0:
e = d + a
f = b + c
f = f + b
if e goto L1
d = e + f
jump L2;
L1:
d = e - f
L2:
g = d
```

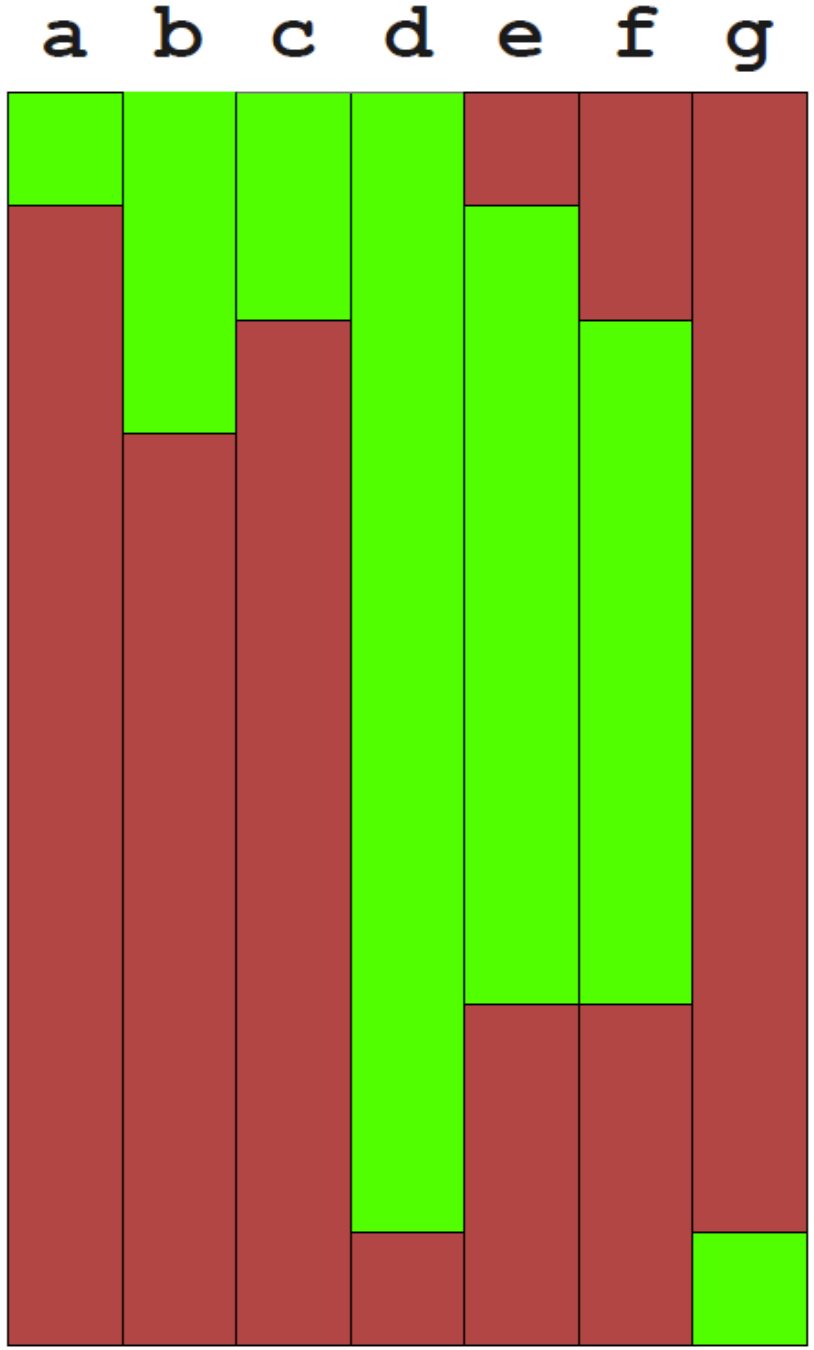
After control-flow and liveness analysis



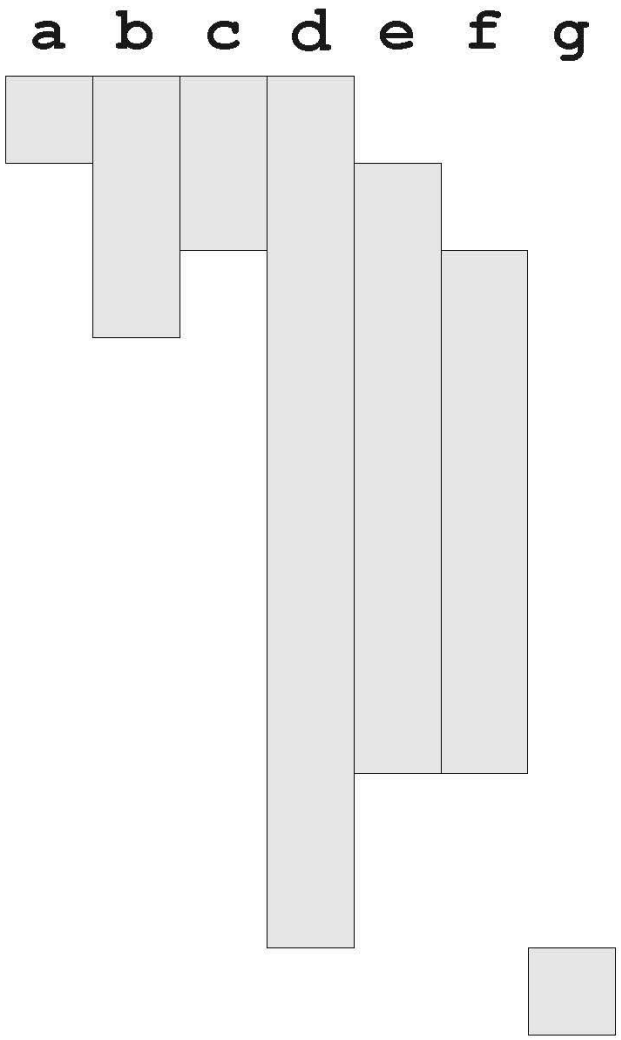
Using live ranges

Idea (continued): Given the live intervals for all the variables/IR registers in the TAC program, we can allocate registers using a **simple greedy algorithm**.

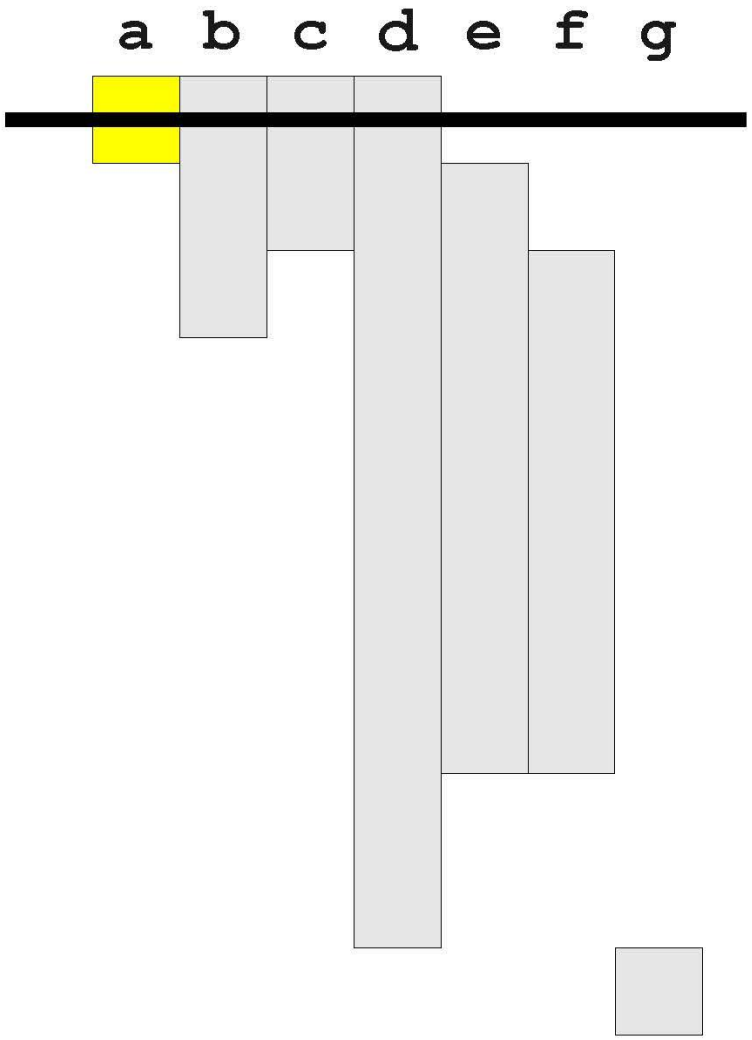
- Track which registers are free.
- When a live interval begins, give that variable a free register.
- When a live interval ends, the register could be freed without consequences.



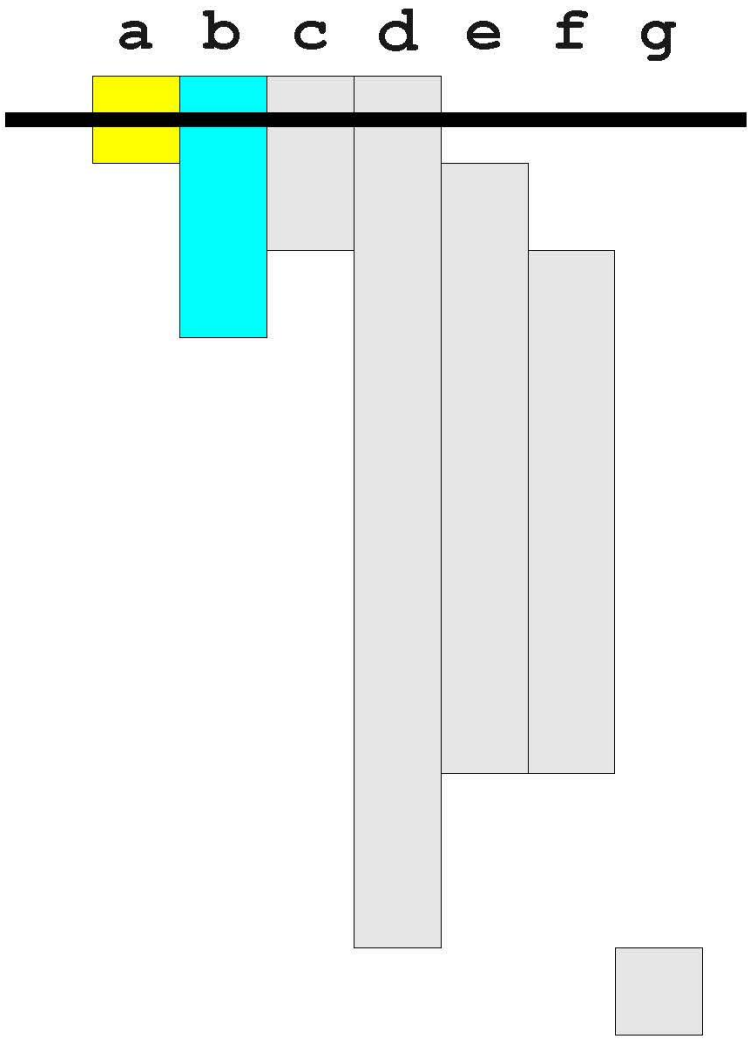
A first demo of greedy allocation



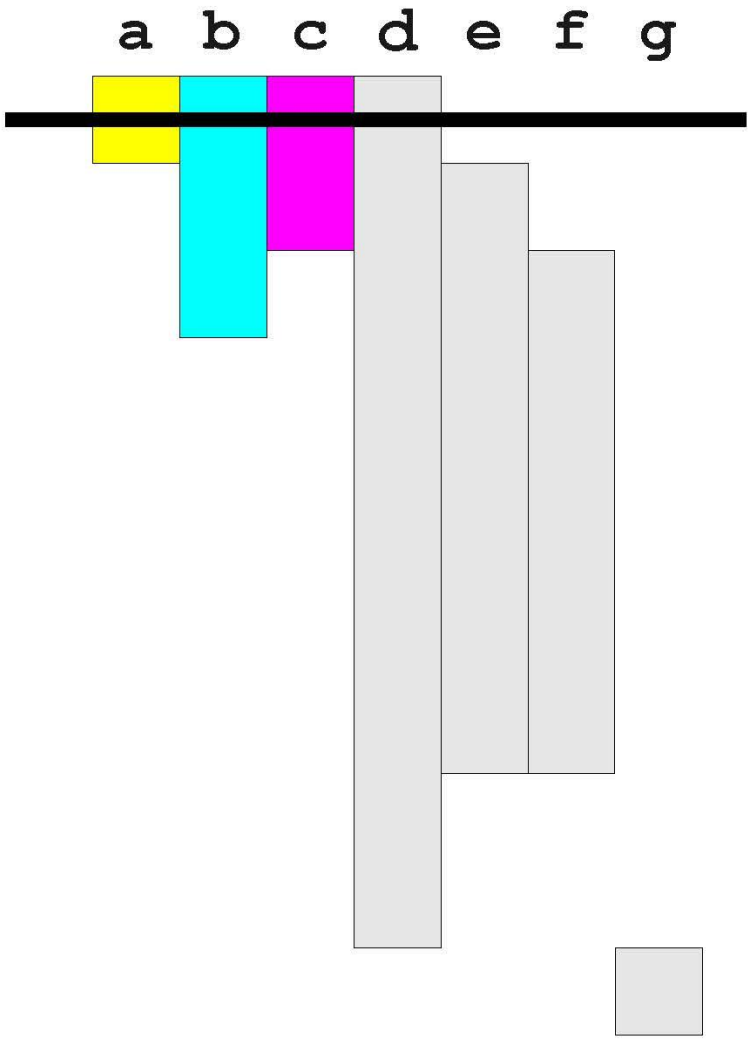
A first demo of greedy allocation



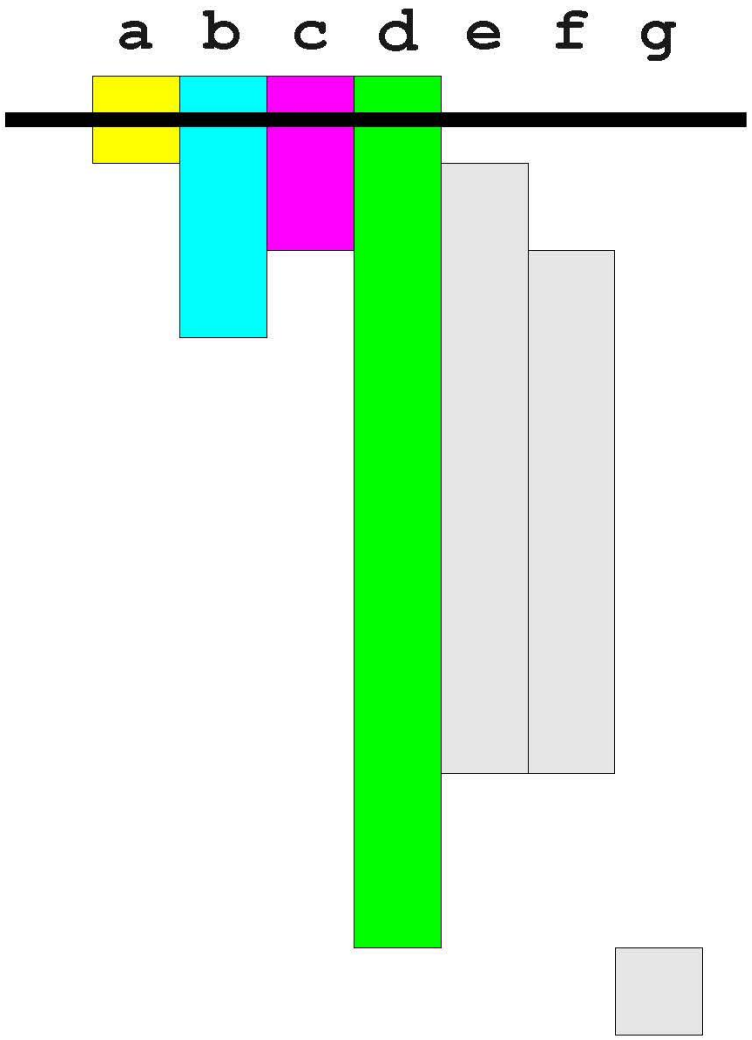
A first demo of greedy allocation



A first demo of greedy allocation



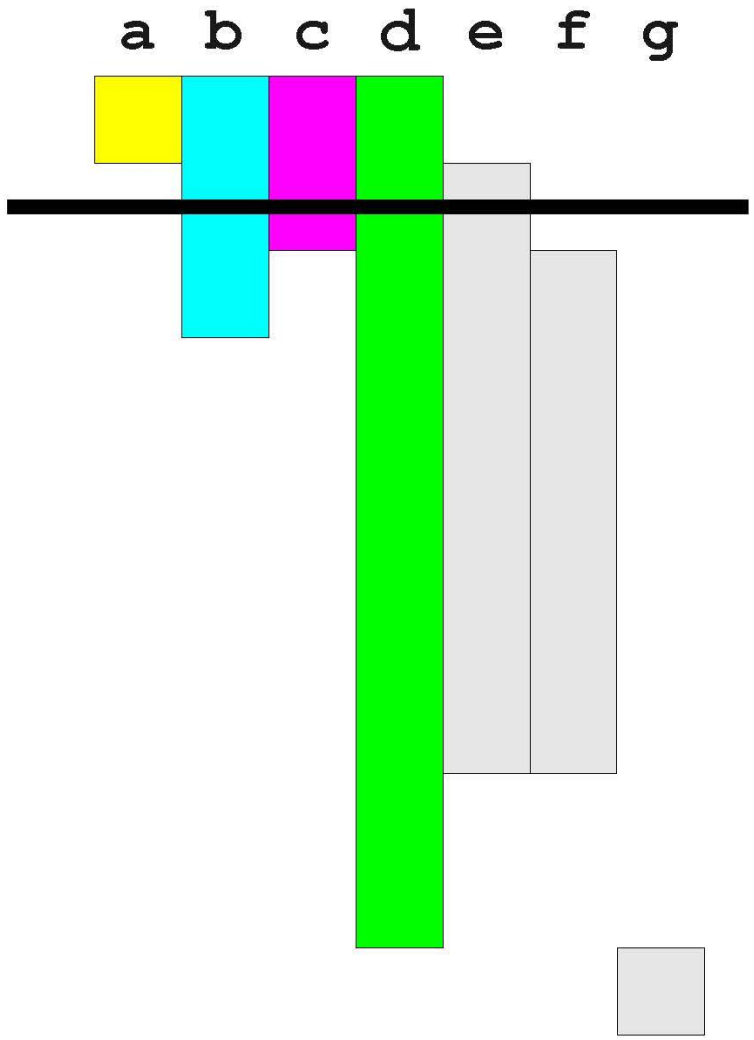
A first demo of greedy allocation



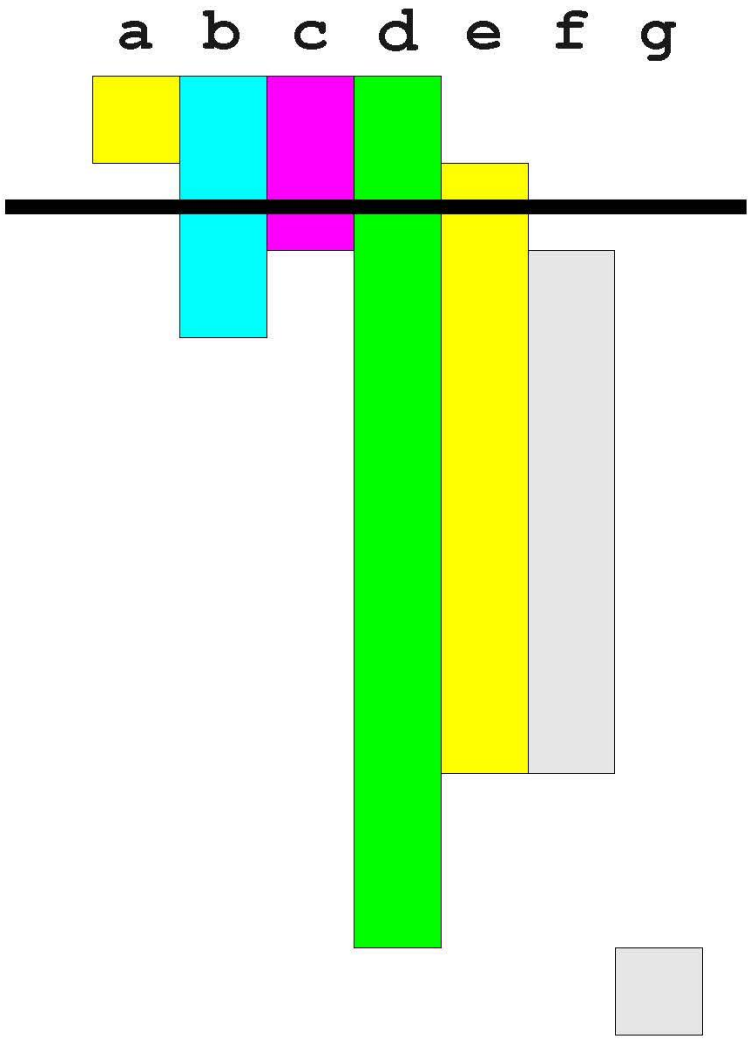
Free Registers

R_0	R_1	R_2	R_2

A first demo of greedy allocation



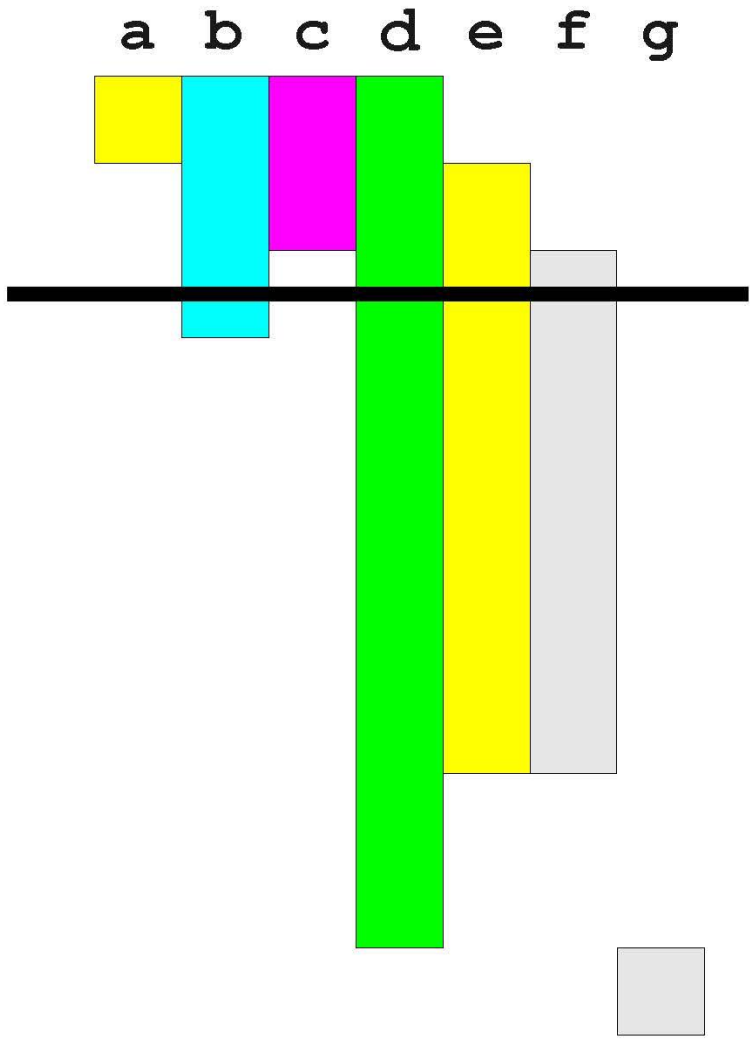
A first demo of greedy allocation



Free Registers

R_0	R_1	R_2	R_2

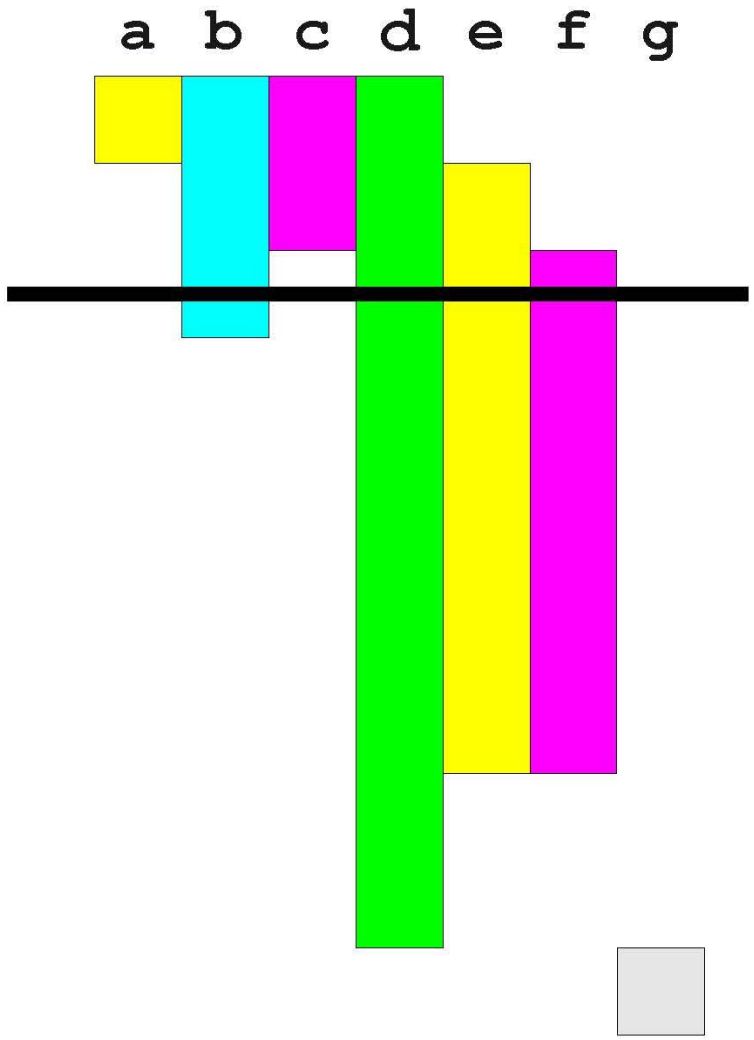
Register Allocation with Live Intervals



Free Registers

R_0	R_1	R_2	R_2

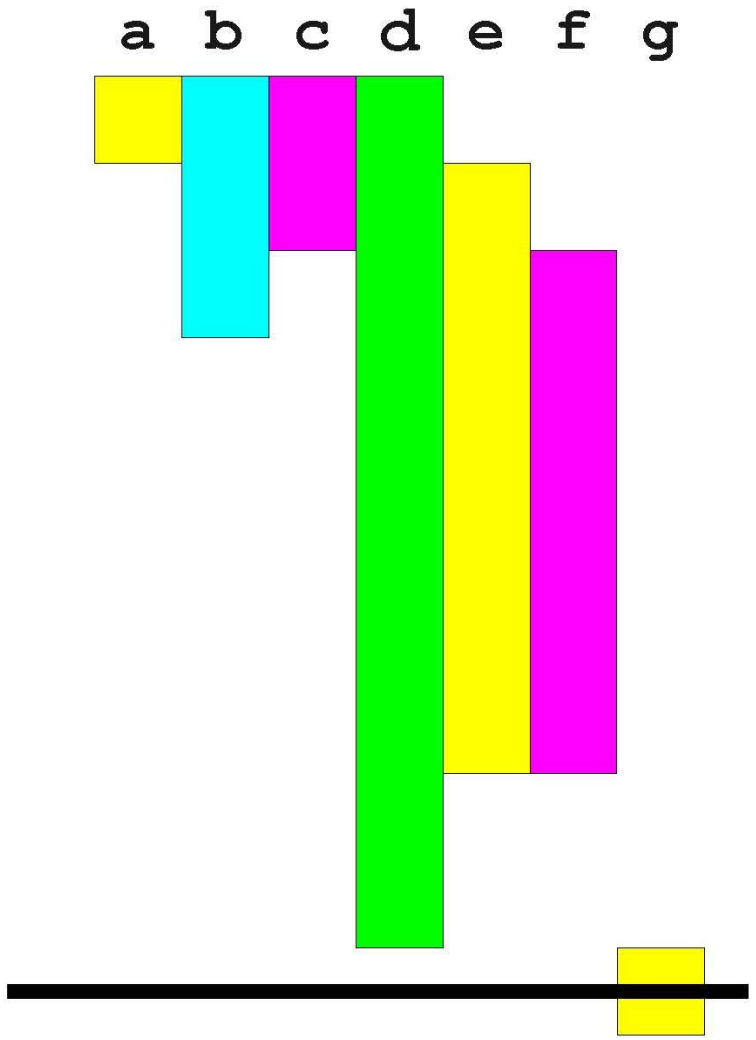
A first demo of greedy allocation



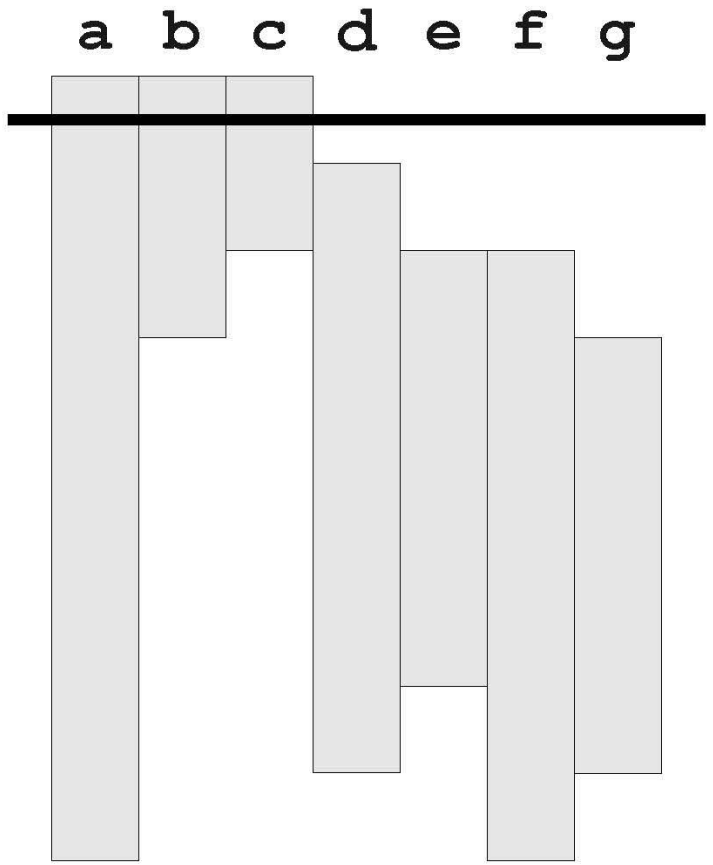
Free Registers

R_0	R_1	R_2	R_2

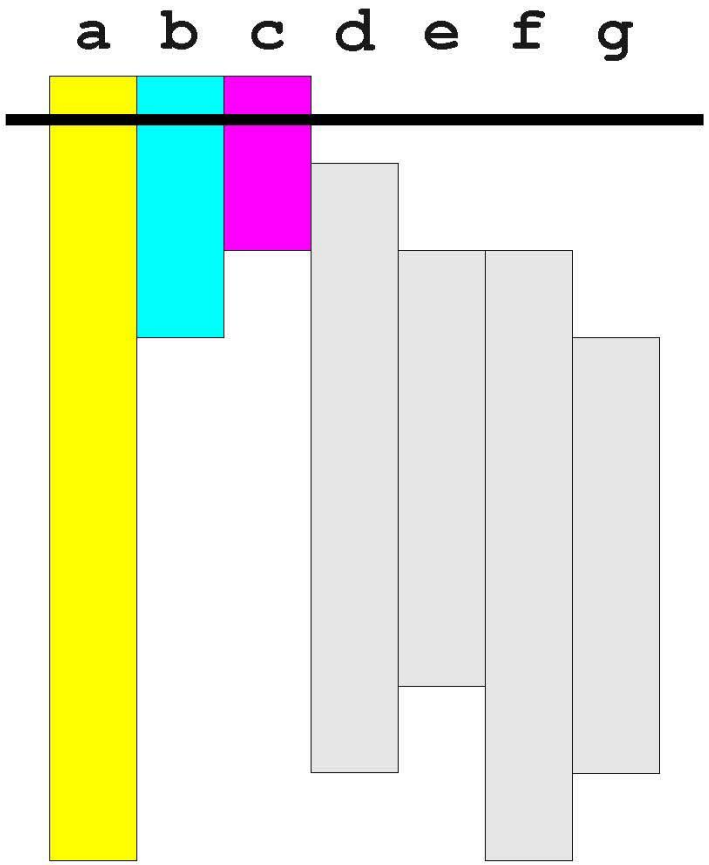
A first demo of greedy allocation



A second demo of greedy allocation



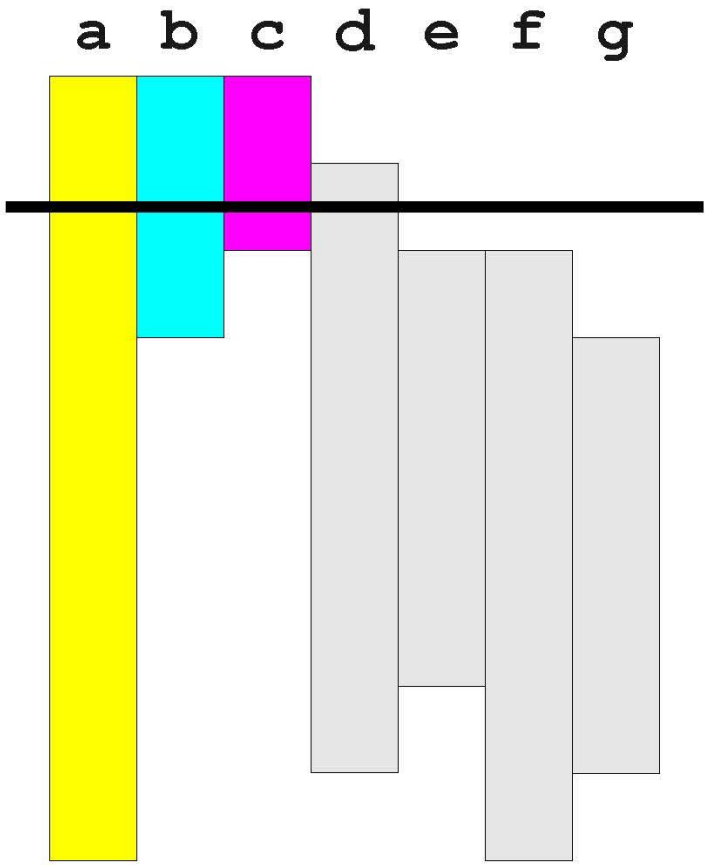
A second demo of greedy allocation

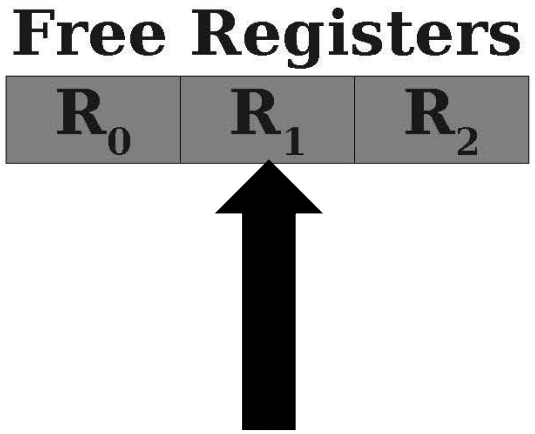


Free Registers

R_0	R_1	R_2

A second demo of greedy allocation

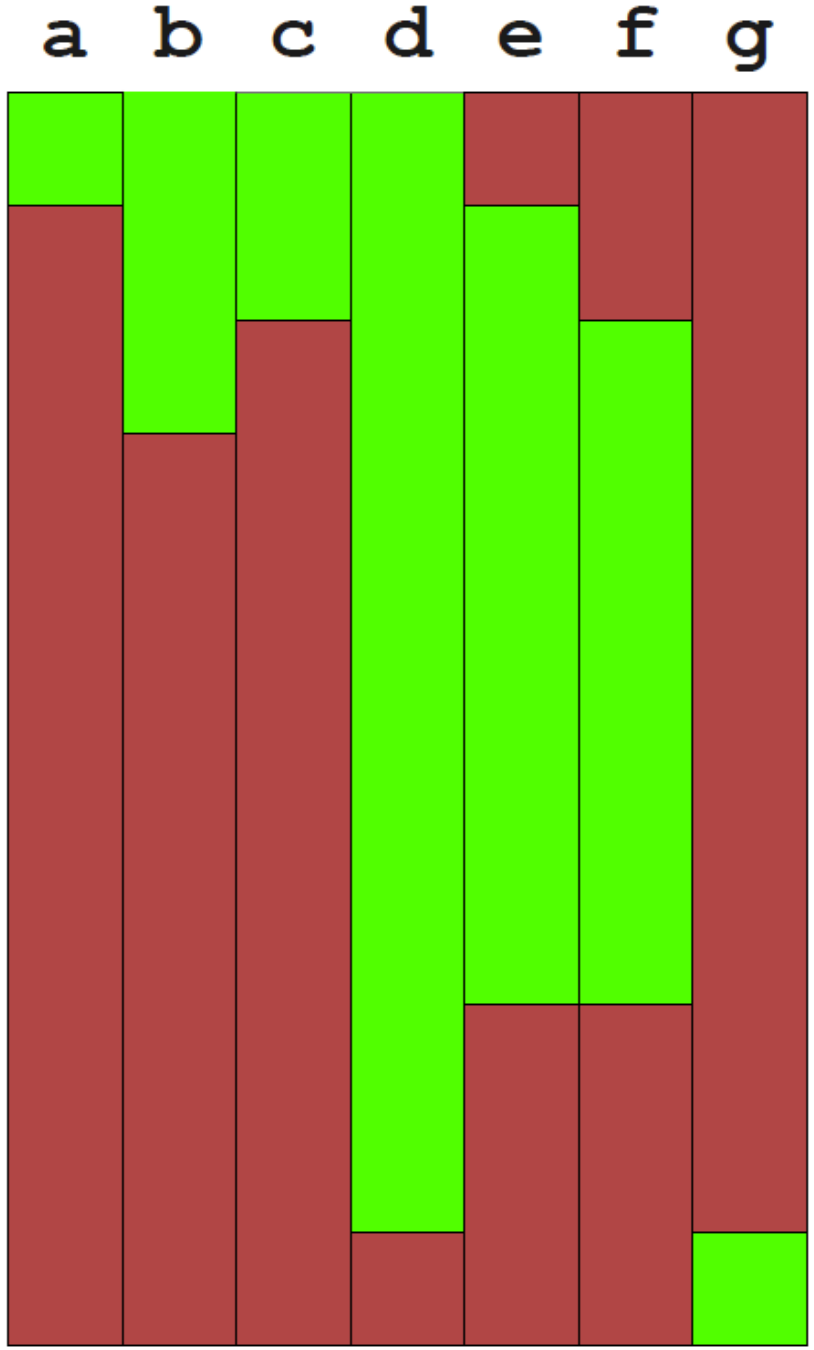




We need a register for d, but we are out of free registers at the moment!

What to do?

Using live ranges



Idea (continued): Given the live intervals for all the variables/IR registers in the TAC program, we can allocate registers using a **simple greedy algorithm**.

- Track which registers are free.
- When a live interval begins, give that variable a free register.
- When a live interval ends, the register could be freed without consequences.

Works if live variables sets is smaller than number of registers. What to do if not?

Register spilling

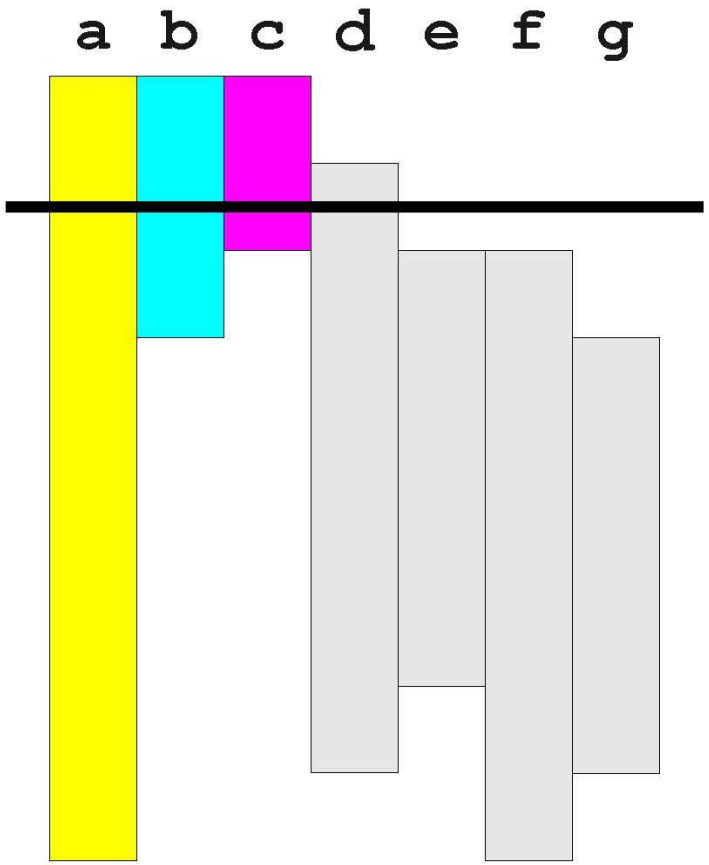
Definition (Register Spilling):

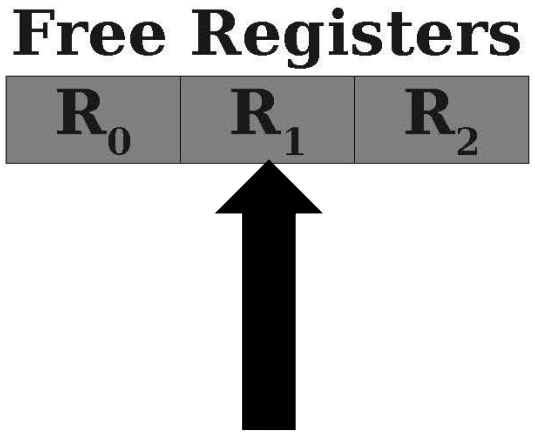
If, at a given time, a free register cannot be found for a live variable v, we will need to **spill a variable**. When a variable is spilled, it is stored in a secondary/less efficient memory location rather than a register.

When we need a register for the spilled variable:

- Evict some existing register to a secondary memory location.
- Load the variable into the register.
- When done, write the register back to memory and reload the register with its original value.
- Spilling is slow, but sometimes necessary.

A second demo of greedy allocation

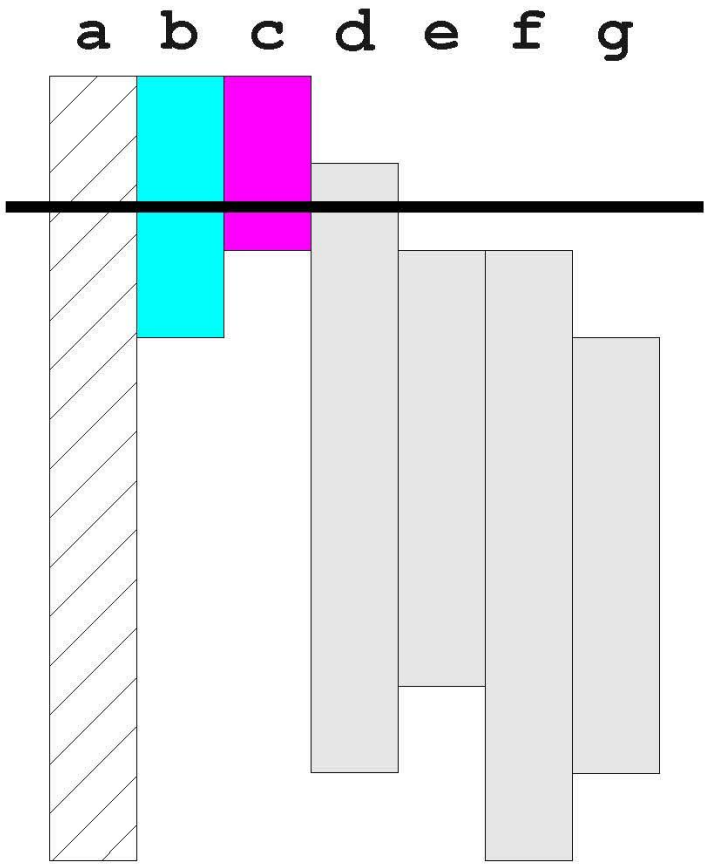


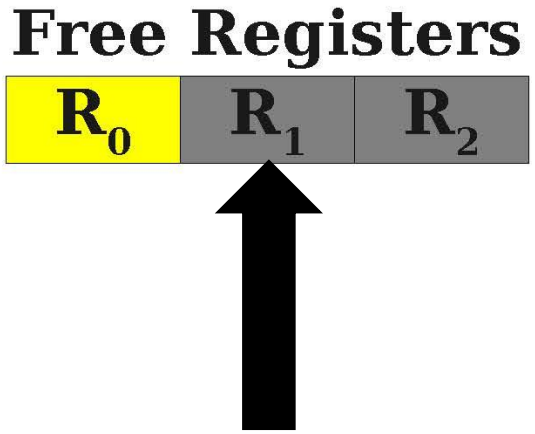


We need a register for d, but we are out of free registers at the moment!

What to do?

A second demo of greedy allocation

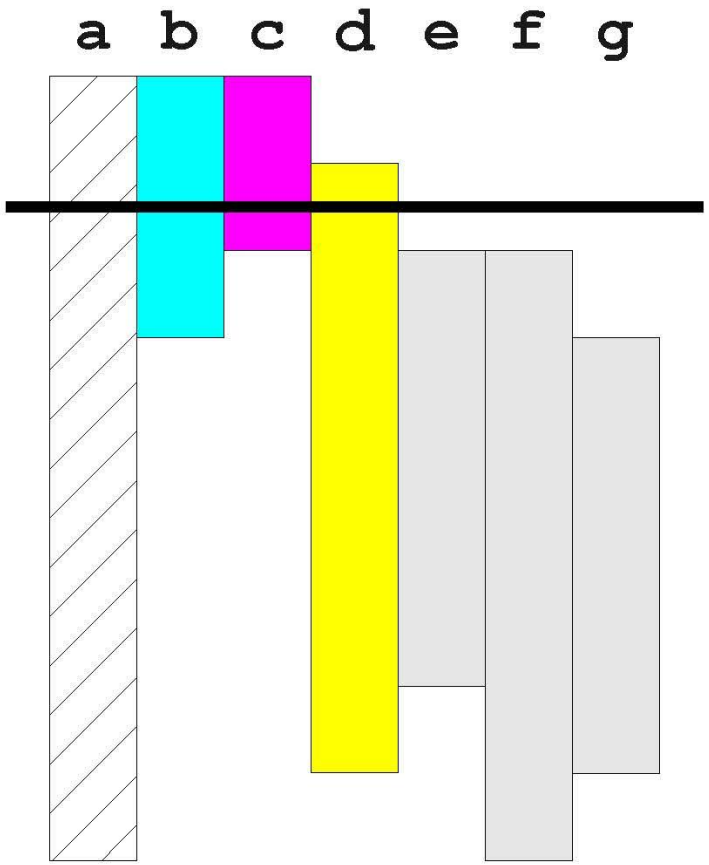


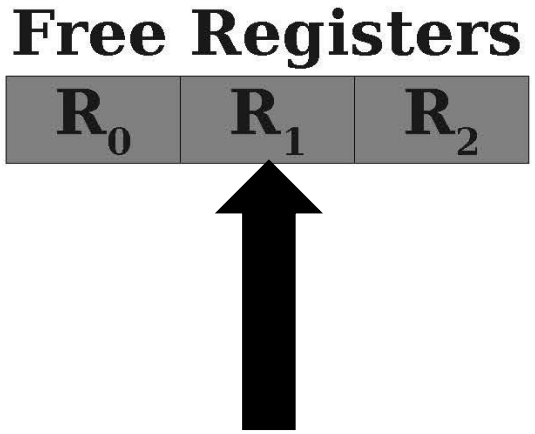


We need a register for d, but we are out of free registers at the moment!

What to do?
Spill variable a for now.
Give R1 to d

A second demo of greedy allocation

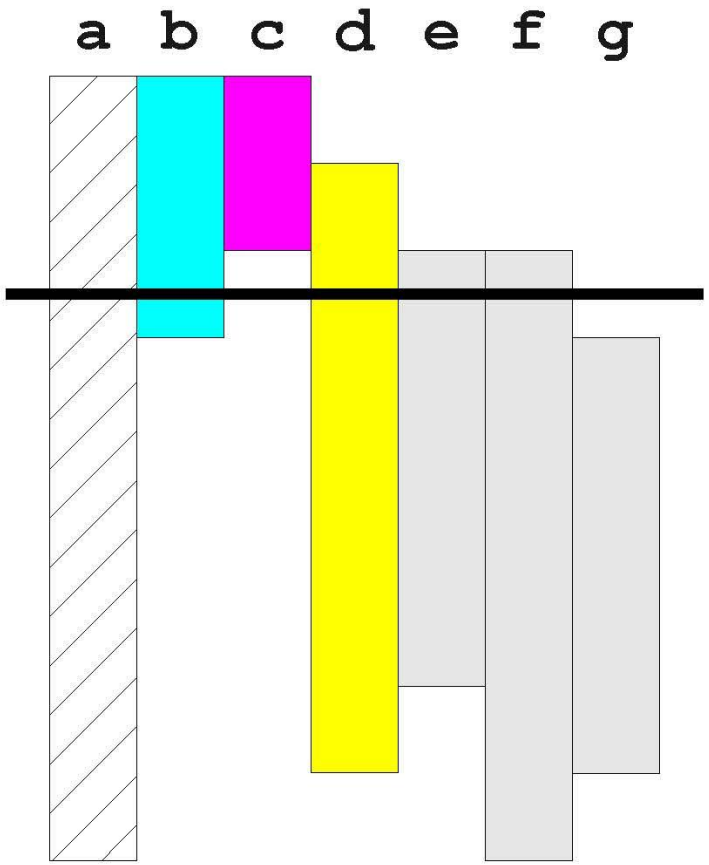




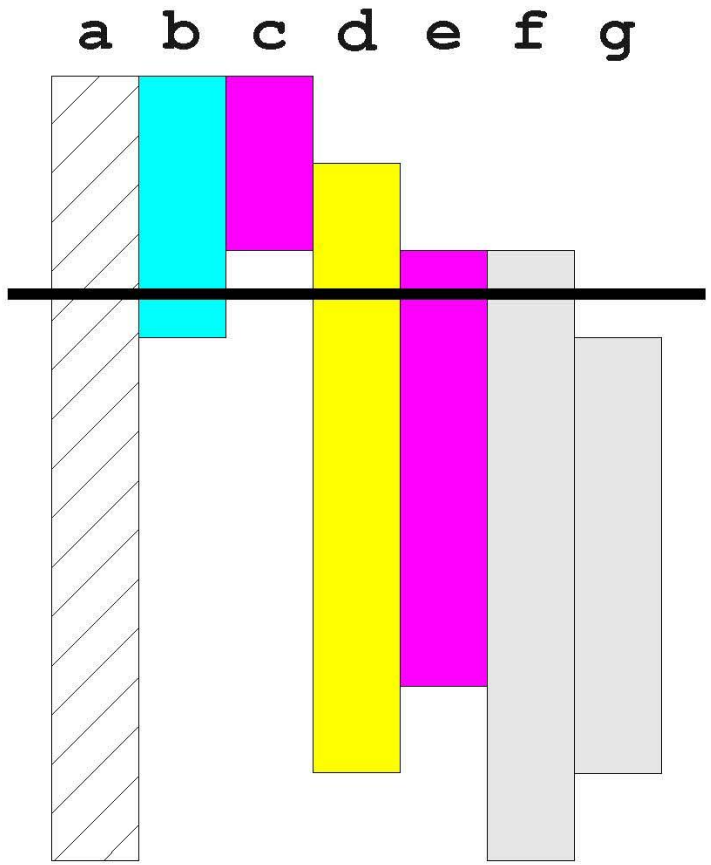
We need a register for d, but we are out of free registers at the moment!

What to do?
Spill variable a for now.
Give R1 to d

A second demo of greedy allocation



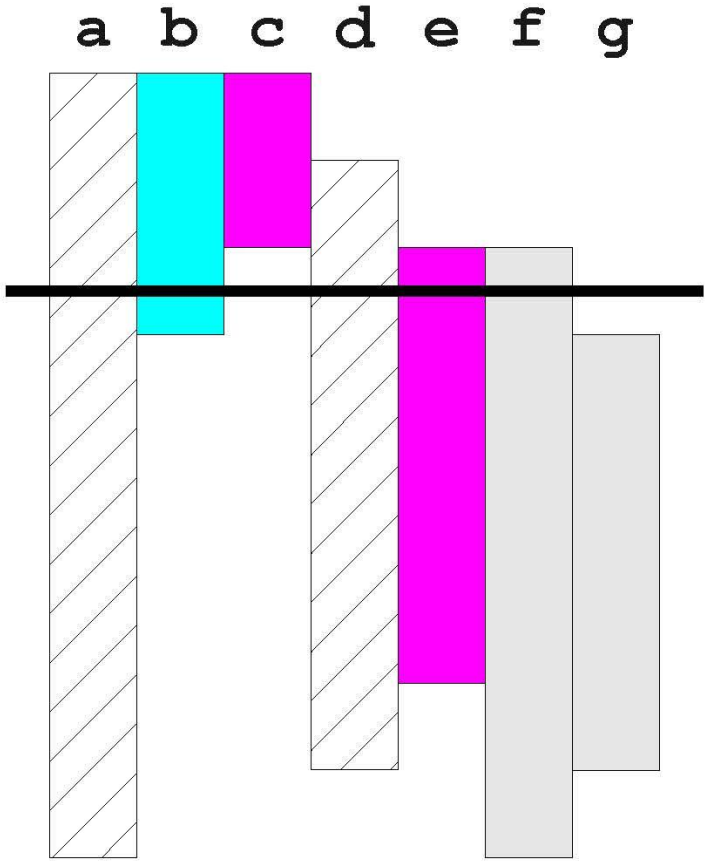
A second demo of greedy allocation



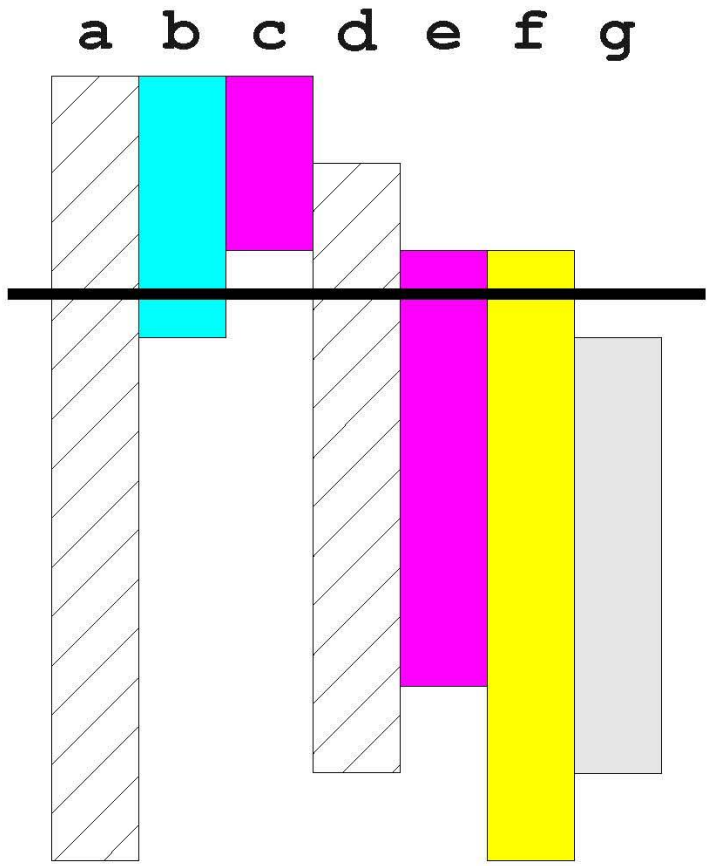
Free Registers

R_0	R_1	R_2

A second demo of greedy allocation



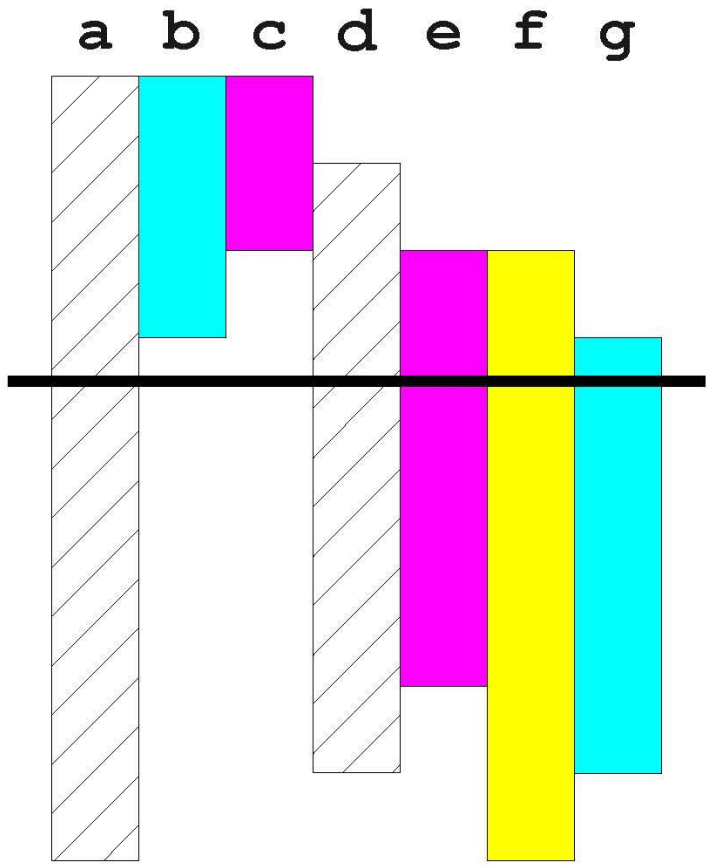
A second demo of greedy allocation



Free Registers

R_0	R_1	R_2

A second demo of greedy allocation



Free Registers

R_0	R_1	R_2

Linear scan register allocation

This greedy algorithm with register spilling is called the **linear scan register allocation** algorithm.

Advantages:

- Very efficient (after computing live intervals, runs in linear time)
- Produces good code in many instances.
- Allocation step works in one pass; can generate code during iteration.

Disadvantages:

- Imprecise due to use of live intervals rather than live ranges.
- Other techniques known to be superior in many cases.

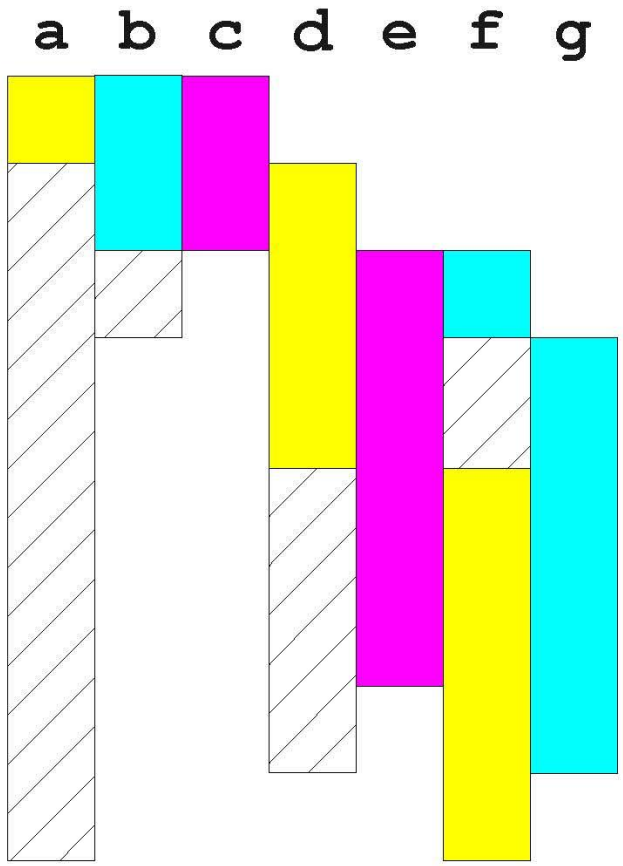
Linear scan register allocation

This greedy algorithm with register spilling is called the **linear scan register allocation** algorithm.

More advanced versions of this algorithm exist, for instance the **linear scan register allocation with second-chance bin packing** algorithm.

- If a variable must be spilled, do not spill all uses of it.
- A later live range might still fit into a register.
- Requires a final data-flow analysis to confirm variables are assigned consistent locations.

A more advanced greedy allocation



Free Registers

R_0	R_1	R_2

Live interval dependency graph

Definition (Live Interval Dependency Graph):

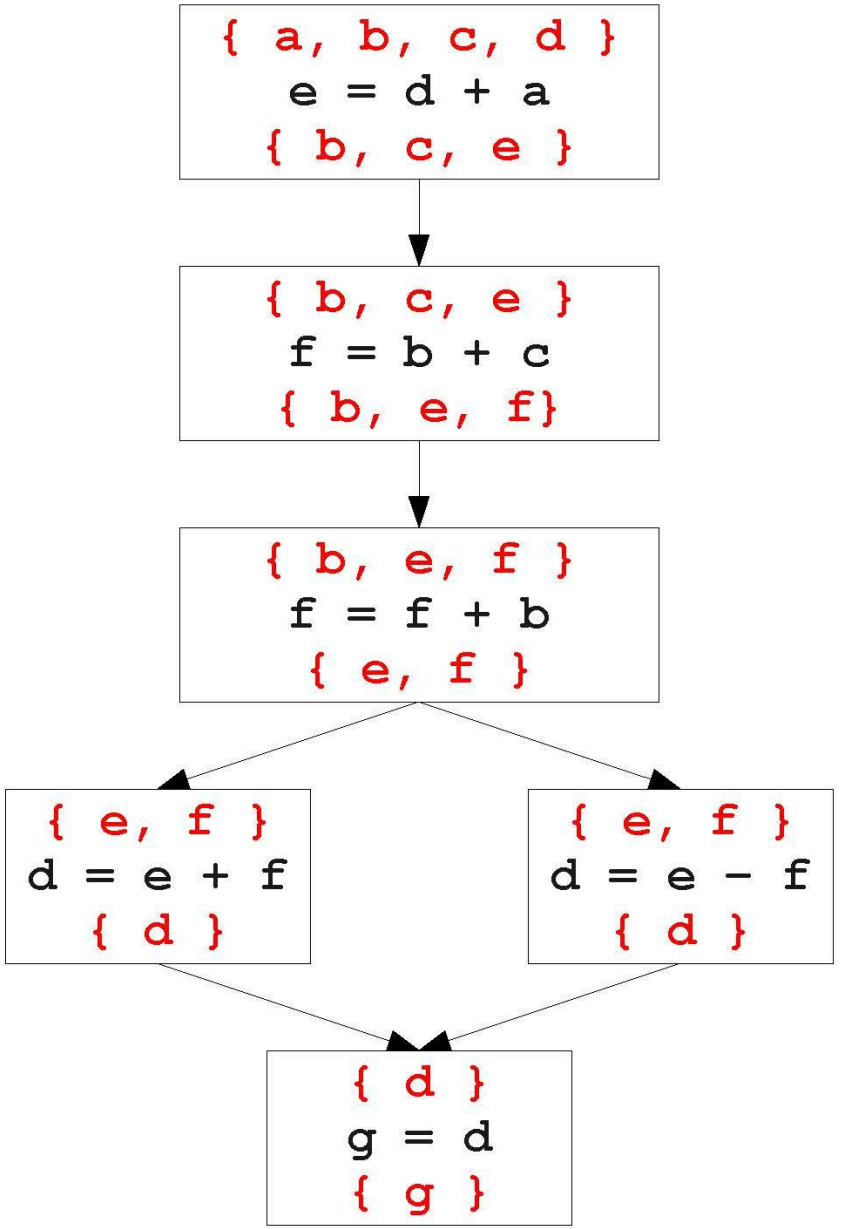
The more advanced resource allocation algorithms require to understand which variables might happen to be live at the same time.

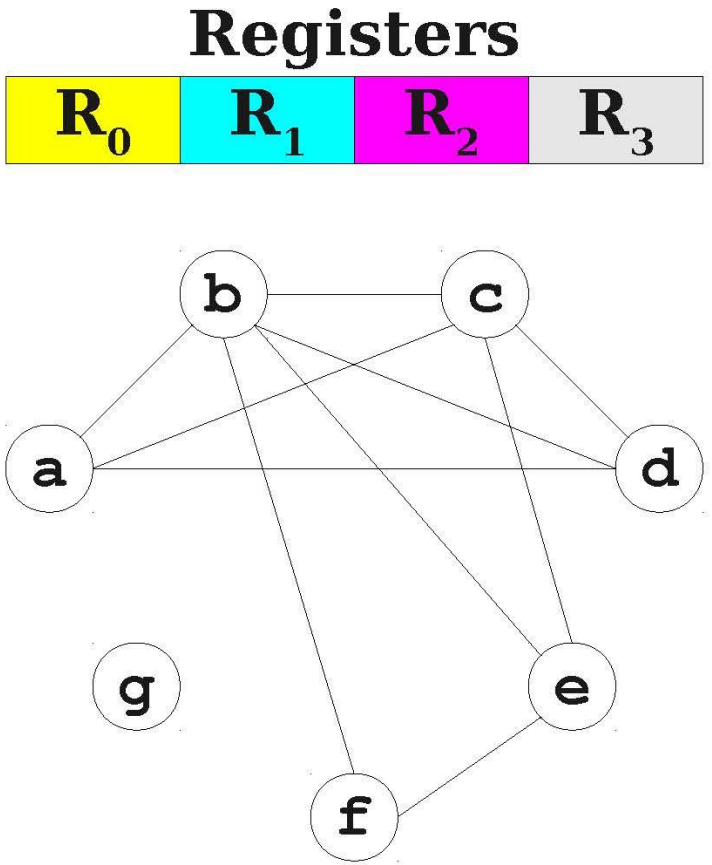
Represent these dependencies using a graph, where:

- Nodes are variables,
- Edges describe if two variables will have to be live at the same time at a given point of the program.

This defines a **live interval dependency graph** for live variables.
Also sometimes called **register interference graph**.

A live dependency graph example



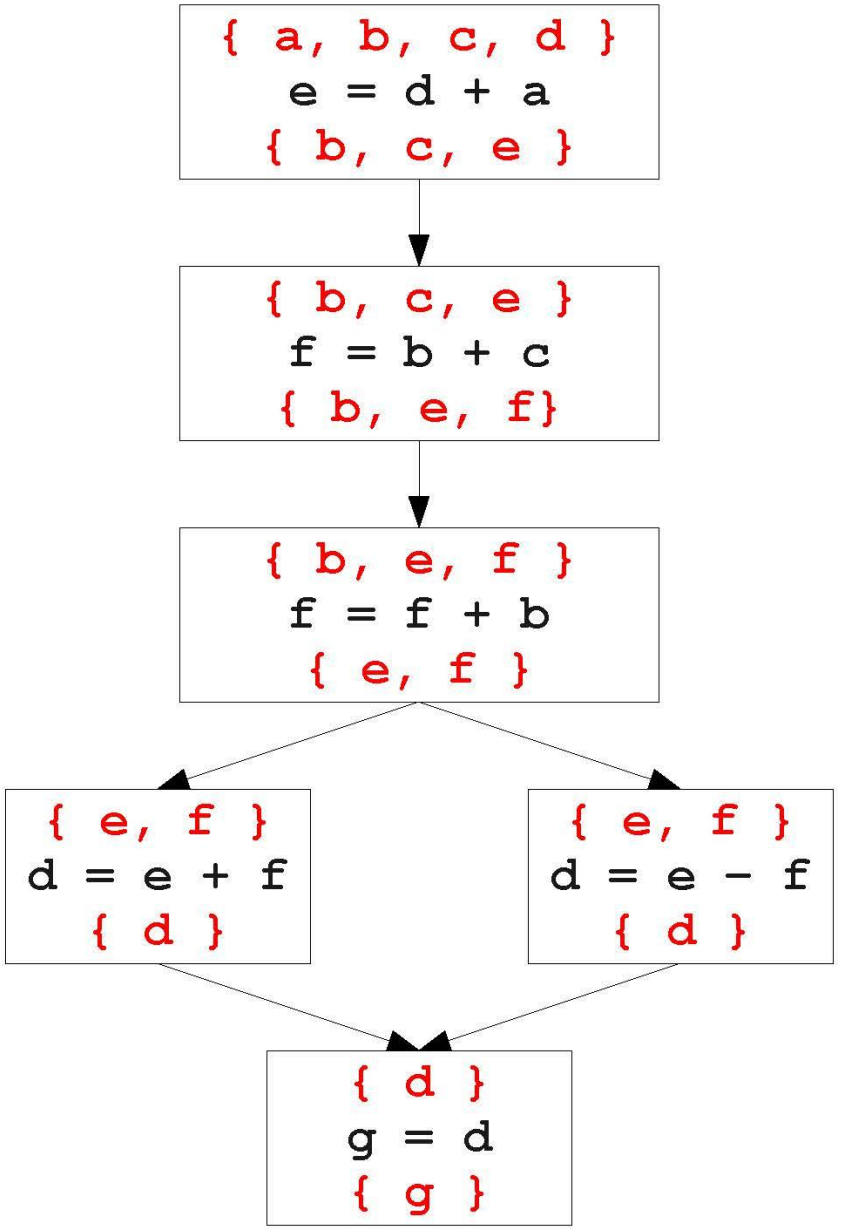


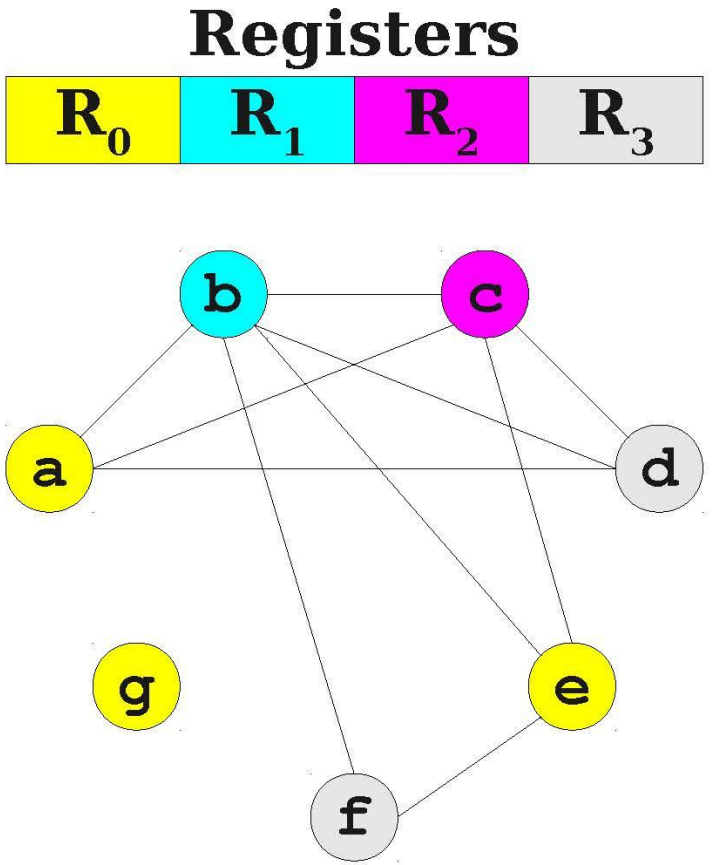
Resolving assignments via the graph

Idea: If we know which variables are supposed to be live together at any given point, then

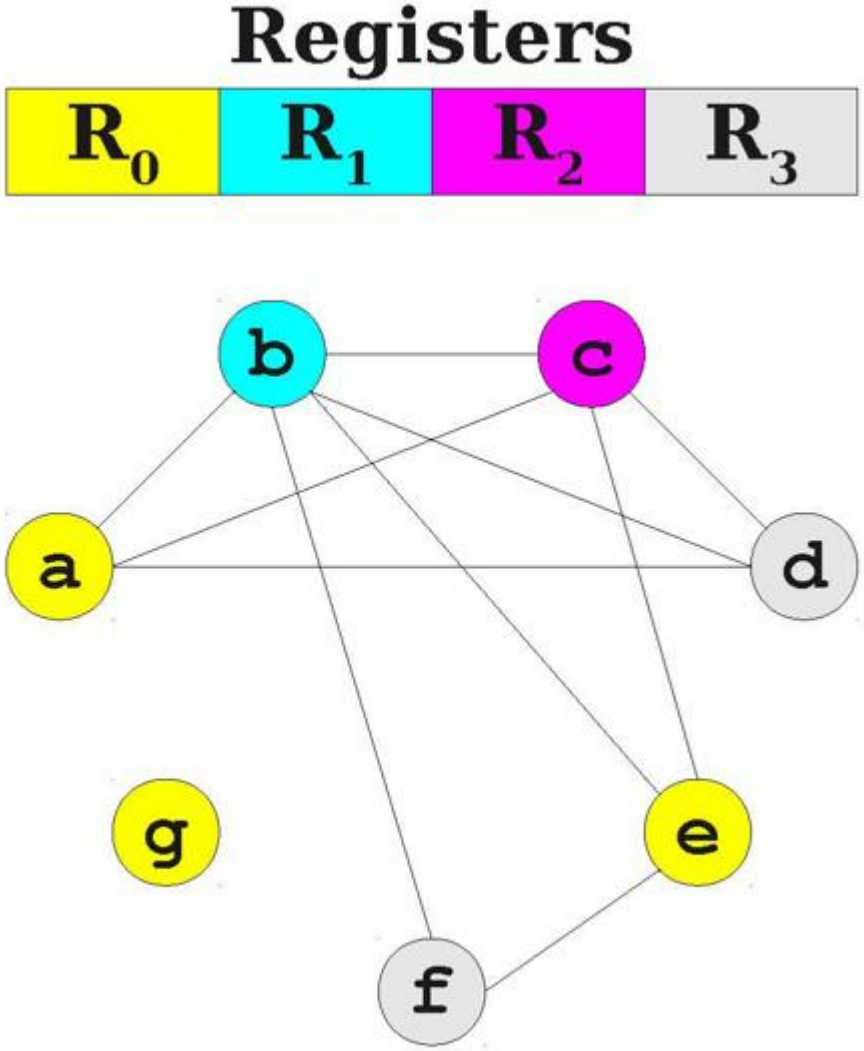
- We can decide, optimally, on the minimum number of memory locations required at any given time to store live variables.
- We can decide which memory location to assign to any given variable.
- All we need is to assign registers to variables
- And make sure no two variables connected by an edge in our live dependency graph are assigned to the same memory location!

A live dependency graph example





Graph coloring problem



Definition (graph coloring problem):

This problem is a well-known problem in graph theory, called the **graph coloring problem**.

Typically, try to colorize each node, with no two nodes with the same color connected via an edge.

Small problem, it is NP-hard (again!).

Some **heuristics** exist (e.g. Chaitin's algorithm), but still pretty much an open problem in graph theory! (out-of-scope)

Possible algorithms to investigate

Lesson: The register allocation problem, like many other problems in compilers, is not an easy one to solve! (NP-hard).

- Naïve (“no”) register allocation.
- Linear scan register allocation.
- Graph-coloring register allocation.

In addition: Advanced versions of these algorithms might need to discuss with the processor and the OS as to what resources are available (e.g. other programs running, multi-core CPUs, parallelised computing, number of registers, GPU computing, etc.).

These are, again, too advanced and out-of-scope.

Quiz time!

What is the main purpose of translating three-address code to assembly code?

A. To create an executable binary
B. To optimize the code to its maximal level
C. To make the code human-readable
D. To simplify the code and ensure it runs fast

Quiz time!

What is the main purpose of translating three-address code to assembly code?

A. **To create an executable binary**
B. To optimize the code to its maximal level
C. To make the code human-readable
D. To simplify the code and ensure it runs fast

Quiz time!

Which of the following is NOT a characteristic of assembly code?

A. It is a low-level programming language
B. It is human-readable
C. It contains variables and loops
D. It is specific to a particular computer architecture

Quiz time!

Which of the following is NOT a characteristic of assembly code?

A. It is a low-level programming language

B. It is human-readable

C. It contains variables and loops

D. It is specific to a particular computer architecture

Quiz time!

In the context of memory hierarchy, which of the following memory types has the fastest access time?

A. Registers
B. SRAM
C. DRAM
D. SSD drive

Quiz time!

In the context of memory hierarchy, which of the following memory types has the fastest access time?

A. **Registers**
B. SRAM
C. DRAM
D. SSD drive

Quiz time!

What is the primary goal of register allocation in a compiler?

A. To improve code readability
B. To reduce the number of registers required
C. To minimize the number of memory accesses
D. To ensure no memory information is lost upon computer shutdown

Quiz time!

What is the primary goal of register allocation in a compiler?

A. To improve code readability

B. To reduce the number of registers required

C. To minimize the number of memory accesses

D. To ensure no memory information is lost upon computer shutdown

Quiz time!

Which register allocation strategy attempts to minimize the number of registers needed for a specific computation?

A. Graph coloring algorithms
B. Linear scan algorithm
C. Naive algorithm
D. Greedy algorithm

Quiz time!

Which register allocation strategy attempts to minimize the number of registers needed for a specific computation?

A. **Graph coloring algorithms**
B. Linear scan algorithm
C. Naive algorithm
D. Greedy algorithm

What we have seen this term

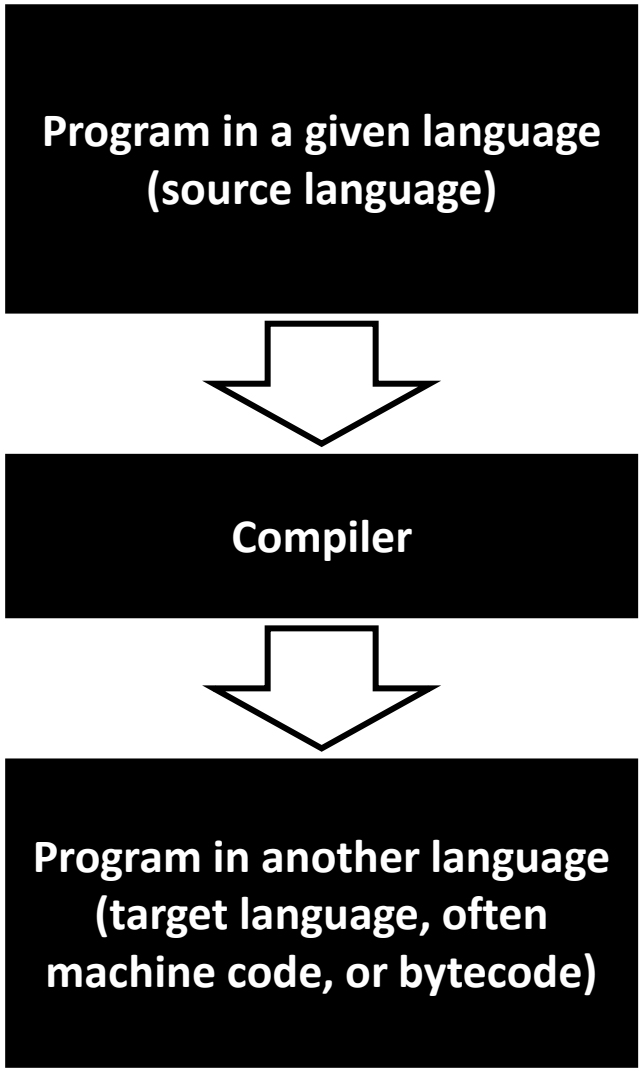
About compilers

Compiler: a definition

Definition (Compilers):
Compilers are **computer programs** whose purpose is to

- **Translate** a **program** written in **one language** (called **source language**),
- Into a **program** written in **another language** (called **target language**).

Target language is often machine code or bytecode.

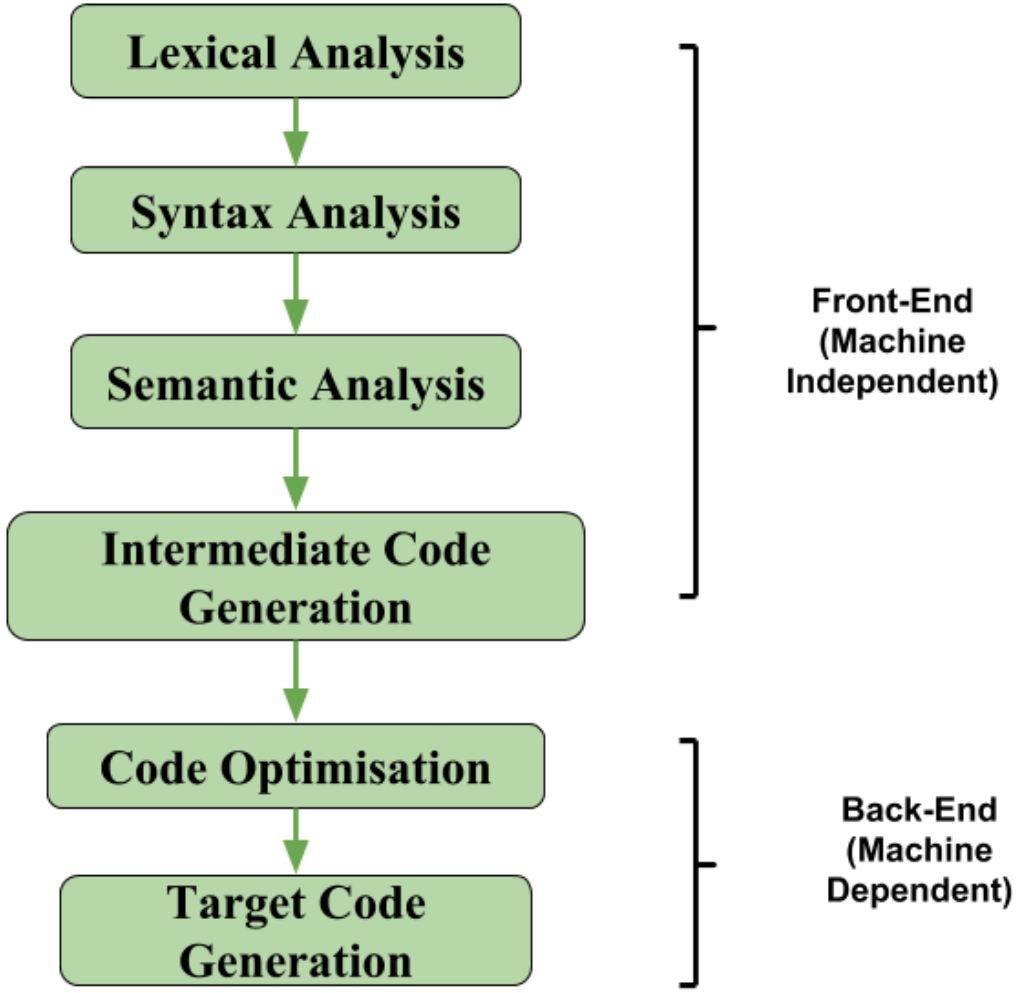


The architecture of a compiler

Definition (the three parts of a typical compiler architecture):

A typical compiler architecture consists of three main components: the **front-end**, **middle-end**, and **back-end**.

- **Front-end:** checks source code is legal (90% of the compiler job!).
- **Middle-end (optional?):** IR code generation and optimization.
- **Back-end:** Final translation and handover to CPU!



How to continue your learning

Because you should!

Continue your learning

W4-5: FSMs and their implementation

- Continue with more advanced types of FSMs (non-deterministic, push automatons, etc.).
- Learn how to convert any NDFSM into its equivalent FSM (tedious, but possible all the time).

Good online courses:

- MIT Theory of Computation
- Stanford Automata Theory

Continue your learning

W4-5: RegEx and their implementation

- Not much else in terms of implementing FSMs and RegEx.
- Maybe try making your own RegEx engine that takes any RegEx expression (basic operations only) and generates the correct FSM.
- Then use said FSM to check validity of a given string.
- Learn to recognize the RegEx behind any given FSM.

Good online courses:

- Few good courses will teach you about implementation in C…
- Maybe this one?

Continue your learning

W8: Tokenization

- Not much else on this topic.
- Possible improvement #1: We generate/compile the RegEx FSM every time we call said RegEx function.
  Would be better to only generate it once?
- How to handle the C pre-processing instructions before tokenization?

Continue your learning

W9: CFGs

- How to implement a full CFG engine?
- For a given set of production rules to use, write an algorithm that would build the parse tree matching a given derivation.

Good online courses:

- Very few, most of the time, better to check Github for implementations.
- Maybe this? (not C)

Continue your learning

W9: SDTs

- How to implement an SDT, and following the idea of the parse tree generating algorithm, how would you build an abstract syntax tree?

Good online courses:

- As with CFGs, very few courses online, most of the time, better to check Github for implementations.

Continue your learning

W10: Parsing

- Try implementing LR(0) and LR(1) algorithms as in Lab 4!
- Learn about more classes of LR parsers like SLR and LALR.

Good course:

- SUTD Term 7 course? **50.054 Compiler Design and Program Analysis, by Prof. Kenny Lu (in the case of functional programming).**

Continue your learning

W11: Semantics

- Try implementing a symbol table, as a simple stack, or better, a spaghetti stack! Later on, implement your own scope checking for the semantics analysis.
- Try implementing your own type checking rules and operation tables. Later on, implement your own type checking for the semantics analysis.

Good online courses:

- As with CFGs/SDTs, very few courses online, most of the time, better to check Github for implementations.

Continue your learning

W11: TAC code generation

- Write your own CFG and SDT for the TAC language.
- Write your own translator function that reads an AST and produces the TAC accordingly.
- Work out more cgen() functions for while loops, for loops, break, functions calls, class definitions and methods, etc.

Good online courses:

- As with CFGs/SDTs, very few courses online, most of the time, better to check Github for implementations.

Continue your learning

W12: TAC code optimization

- Try implementing the Copy Propagation optimization procedure.
- Try implementing a liveness analysis (challenging!).
- Try implementing a Dead Code Elimination optimization procedure.
- Chain them, rinse and repeat local optimization function for all basic blocks in your code!
- Try implementing more local optimization techniques (arithmetic simplifications, short-circuit evaluation, constant folding, etc.)
- Eventually, try writing a control-flow graph representation for your TAC code and study/implement more advanced (global) optimization procedures.

Continue your learning

W13: Backend

- Try writing a simple translator that reads TAC code (either as a string of text of a control-flow graph), and produces assembly code (following the Beta CPU instruction set from 50.002).
- Try implementing a few register allocation algorithm (naïve, linear scan, etc.)
- Try your hands on the graph coloring problem and the Chaitin algorithm implementation!
- More backend stuff (runtime, garbage collection, code optimization and clock cycles optimization on your CPU by performing instruction ordering/section, parallelism, etc.)

And interpreters in all of this?

Definition (Interpreters):

Many of you have probably heard about the **compilers vs. interpreters paradigm**, but what is it about?

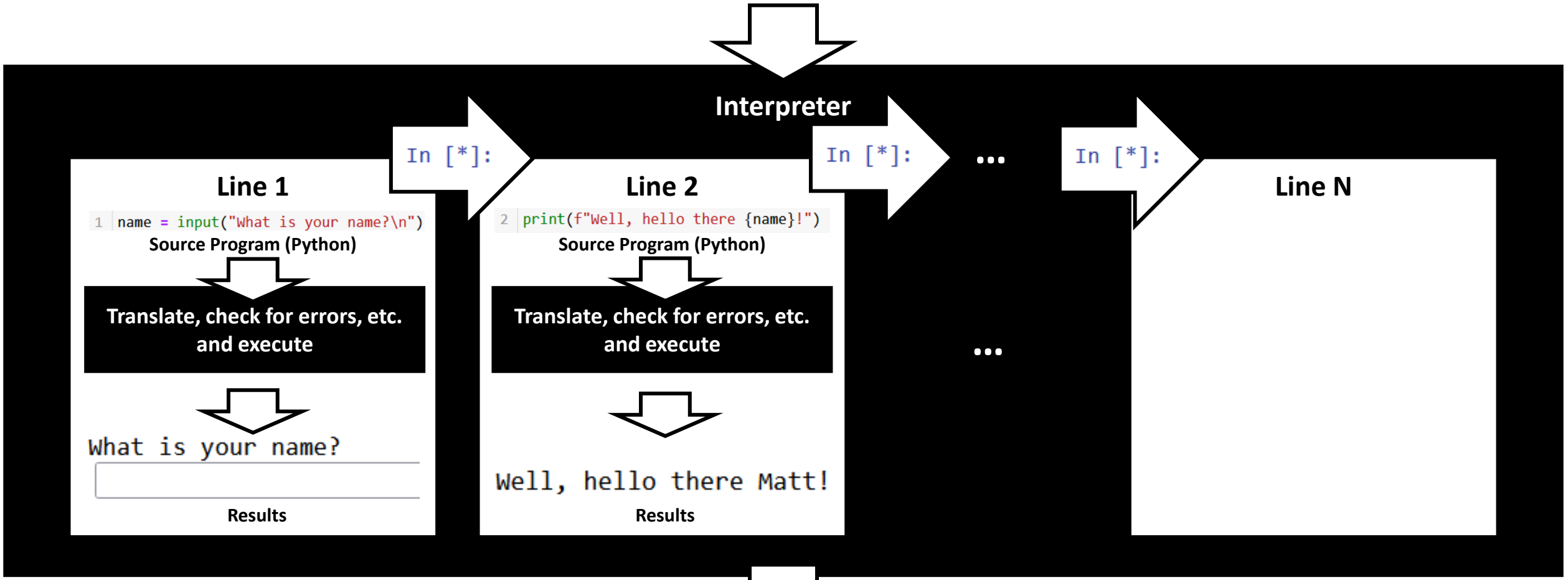
Interpreters have the **same objective as compilers**, i.e. **translate a source program into target code which can be executed by the CPU**.

What changes is the translation and execution procedure:

- **Compilers** will translate the source code into target code **in its entirety first**, and THEN will execute the target code.
- **Interpreters**, on the other hand, will translate each line of the source code **one line at a time**, and execute each one of them in succession.

```
1 name = input("What is your name?\n")
2 print(f"Well, hello there {name}!")
```

Source Program (Python)




```
What is your name?
Matt
Well, hello there Matt!
```

Results

Reference courses, in short

- If interested to learn more about parsers, the reference course is the Compiler course from Stanford

https://web.stanford.edu/class/cs143/

- Available for free online and comes with video recordings

https://www.edx.org/course/compilers

- SUTD Term 7 course? **50.054 Compiler Design and Program Analysis, by Prof. Kenny Lu (in the case of functional programming).**

(Not yet added to ISTD course catalog as of 20/04/2023.)

(Tentative syllabus to be shown in class!)

Your feedback matters!

It is our second time trying this course, and as instructors, we learned a lot by trying it, but also realized a few things did not work out…

- Typically, 6 weeks might be a bit too short to teach all the concepts of compilers correctly (maybe we should shorten the C/C++ part to make room for compilers?).
- A **lab session** or two to have you **implement some semantics checks, in a guided manner**, would have been nice.
- Will improve for next run.

Let us know if you have more feedback for us!

The End

Also, remember: if the code is legal according to the compiler but does not produce the behavior you expect, then it is not the compiler's fault…The problem is…

The End

Also, remember: if the code is legal according to the compiler but does not produce the behavior you expect, then it is not the compiler's fault…The problem is… **You!**

It is not the compiler's job to figure out the logic you want your code to have!

And debug it for you if it fails!

(How would the compiler do it anyway?)