CS614: Advanced Compilers

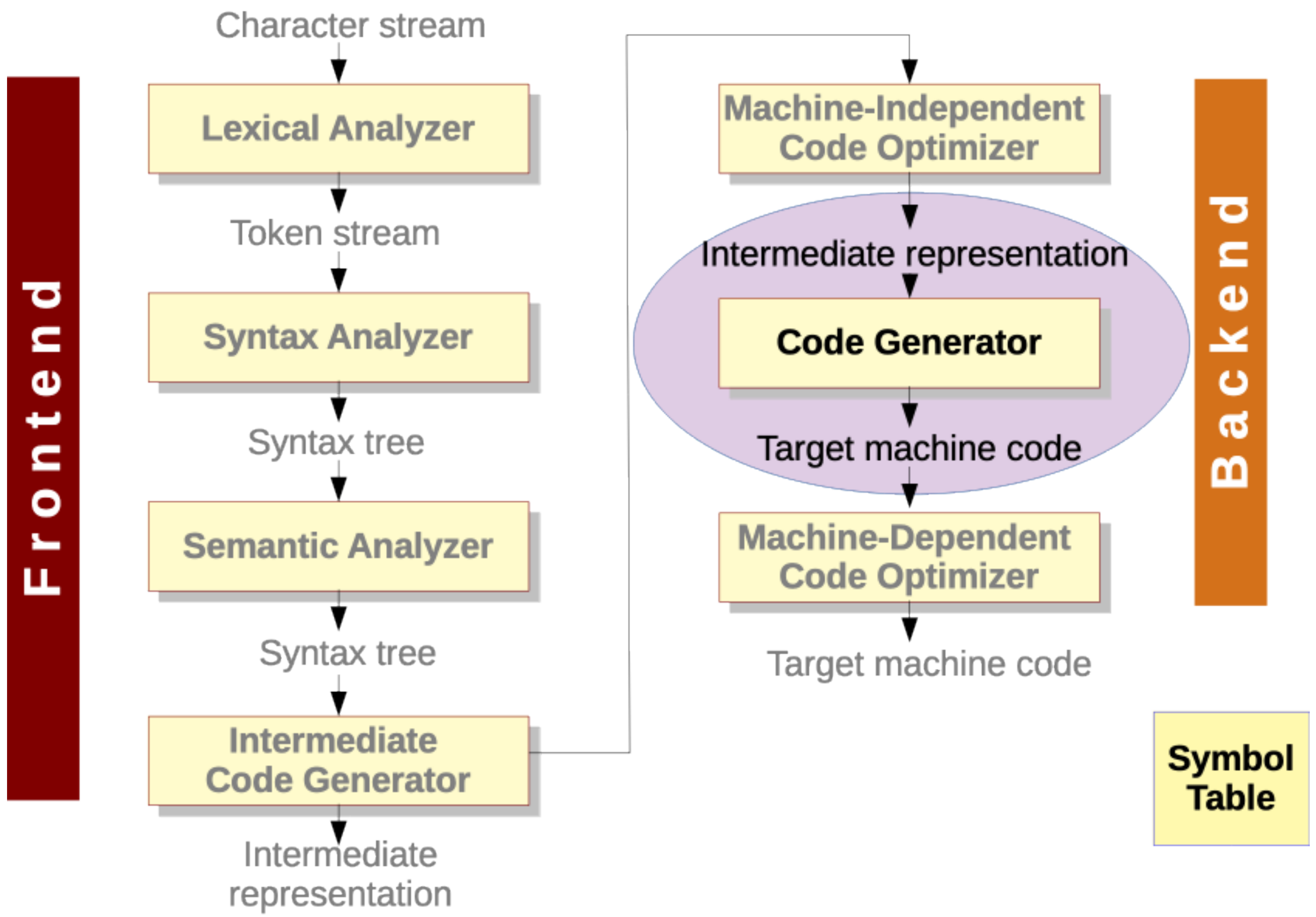
Register Allocation

Manas Thakur

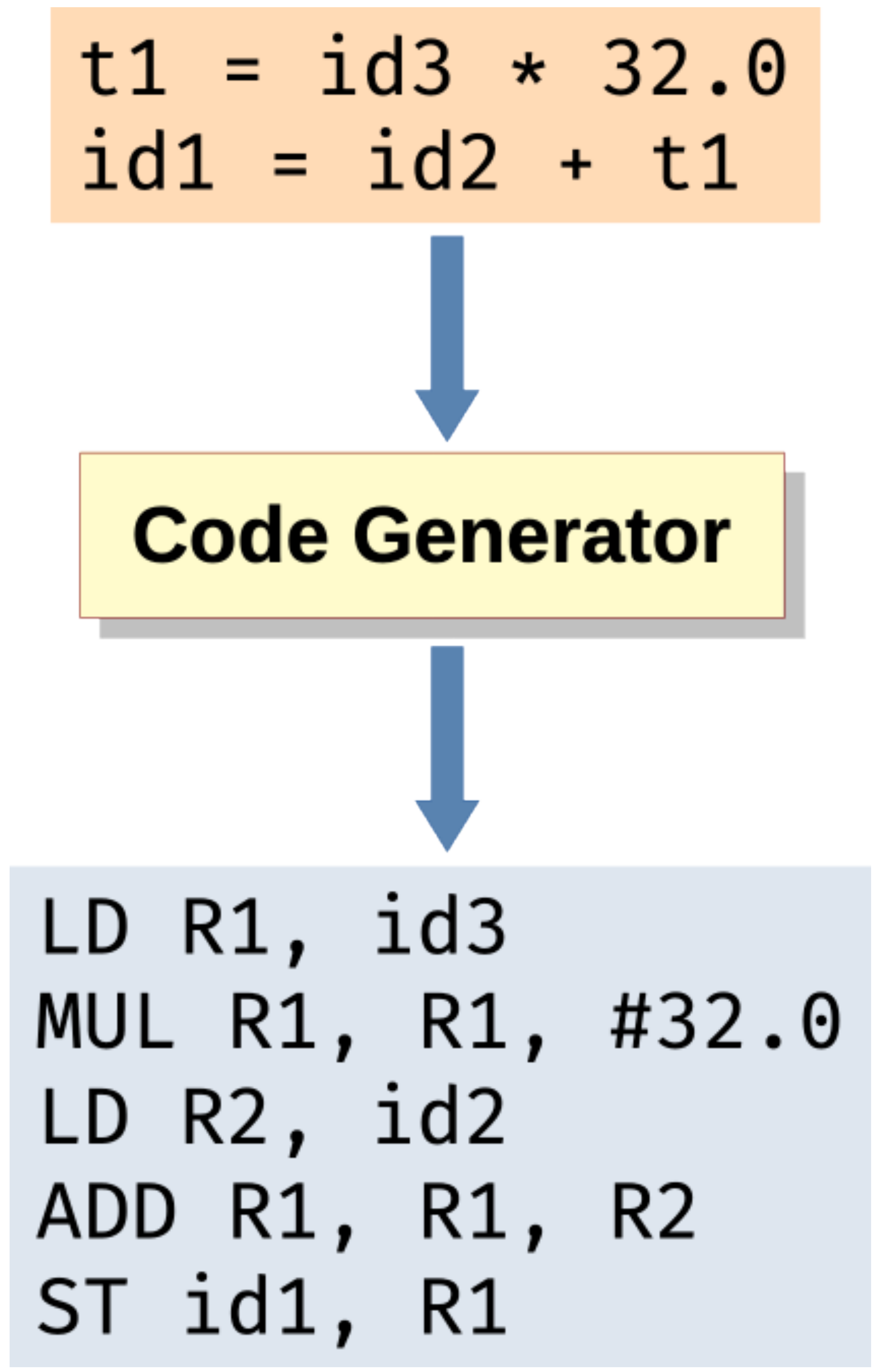
CSE, IIT Bombay

Spring 2025

Where are we?


Character stream
Frontend
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Machine-Independent Code Optimizer
Intermediate representation
Code Generator
Target machine code
Machine-Dependent Code Optimizer
Target machine code
Backend
Symbol Table

Recall from Lecture 1



```
t1 = id3 * 32.0
id1 = id2 + t1
```

```
LD R1, id3
MUL R1, R1, #32.0
LD R2, id2
ADD R1, R1, R2
ST id1, R1
```

Roles of Code Generator

- Convert IR to target program.
  - Bring it down!
- Using the primitives available on the target machine.
  - Usually a form of assembly.
- Requirement: Preserve the semantics of the source program.
  - In terms of the *observable* behaviour.
- Expectation: Target code is of high quality.
  - Execution time, space, energy, and so on.
- Expectation 2: Code generator itself should be efficient.
  - *I would go on summer vacation and hope compilation would finish by the time I am back!*

Code generation in reality

- The problem of generating an optimal target program is undecidable.
    - Recall we had said most problems in the front-end are simple, and most in the back-end are complex?
- Several subproblems are NP-hard or NP-complete.
- Need to depend upon:
    - Approximation algorithms
    - Heuristics
    - Conservative estimates

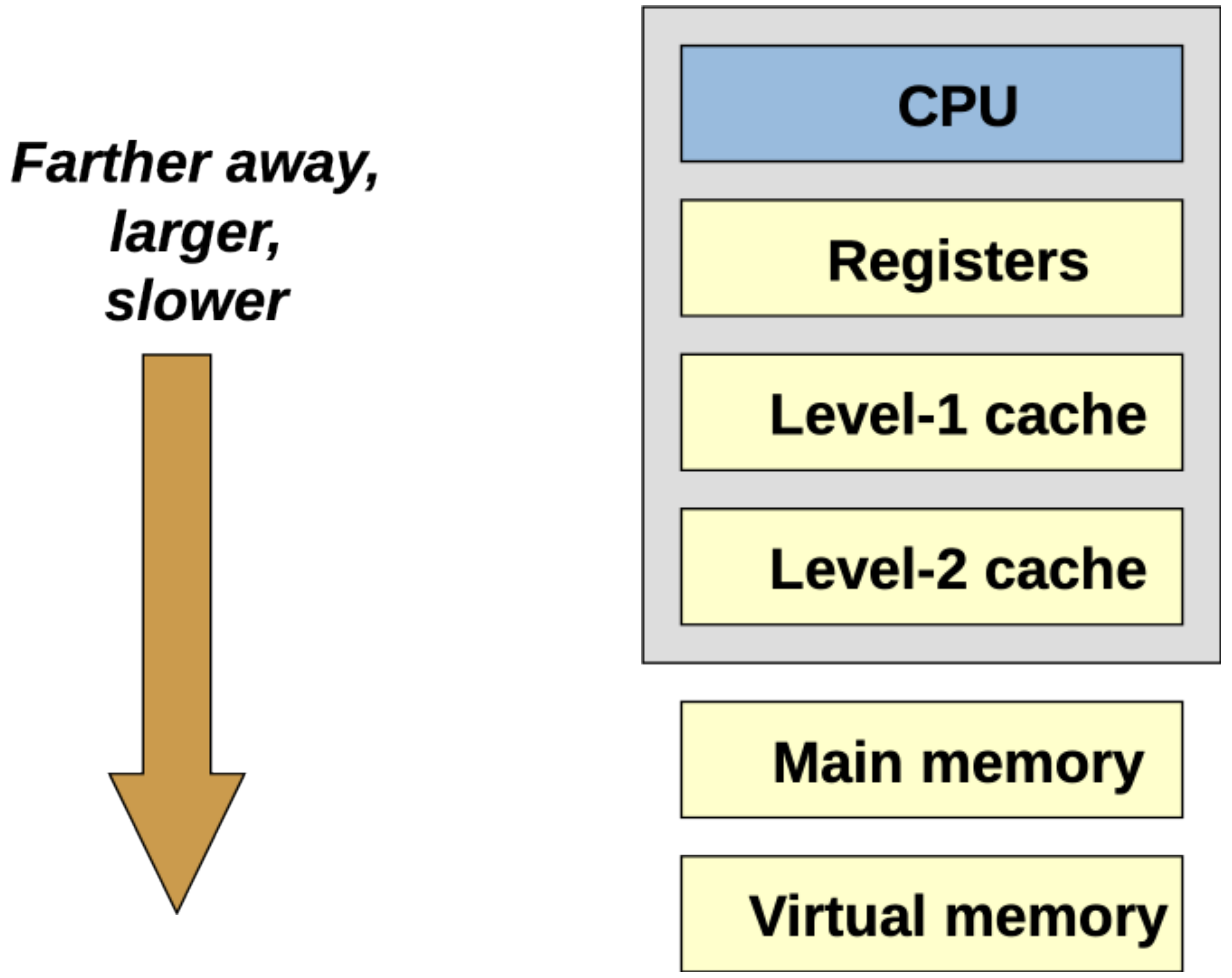
Essential tasks during code generation

- Instruction selection
  - Map low-level IR to actual machine (or assembly) instructions
  - Not necessarily 1-1 mapping
  - Several things vary with the architecture:
    - Instruction set
    - Addressing modes

Also, **Instruction scheduling**.

- Register allocation
  - Low-level IR assumes unlimited registers (temporaries)
  - Map to actual resources provided by the hardware
  - **Goal:** Make the best use of registers

Where are registers?



Pentium 4 3.2 Ghz	*Core 2 Duo*	*Athlon 64*
1 cycle	1 cycle	1 cycle
2 cycles (16 KB)	3 cycles (64 KB)	3 cycles (128 KB)
19 cycles (2 MB)	14 cycles (2 MB)	13 cycles (1 MB)
204 cycles	180 cycles	125 cycles
millions of cycles	millions of cycles	millions of cycles

Architecture and Registers

- **RISC**
  - Many registers, 3AC, simple addressing modes
  - Examples: IBM PowerPC, Oracle SPARC, ARM (mobiles, tablets, apples!)
- **CISC**
  - Few registers, 2AC, Variety of addressing modes, several register classes, variable-length instructions, instructions with side-effects
  - Examples: Intel x86, AMD Athlon
- **Stack machine**
  - Push/Pop, stack-top uses registers
  - Example: JVM

Register allocation

- Involves
  - Allocation: which variables to be put into registers
  - Assignment: which register to use for a variable
- Finding an optimal assignment of registers to variables is an NP-complete problem.
- Architectural conventions complicate matters:
  - Combination of registers used for double-precision arithmetic
  - Result must be stored into accumulator
  - Some registers reserved for special purposes

Live ranges

- A variable is **live** if its current value may be used in future.
  - Two variables that are *live* at the same time cannot use the same register.
  - They *interfere* with each other.
- Conversely, if two variables do not interfere, then they can use the same register.
- Need to determine the durations in which variables are *live*.

```
    [S1] a = 0
L1: [S2] b = a + 1
    [S3] c = c + b
    [S4] a = b * 2
    [S5] N = a + 3
    [S6] return c
```

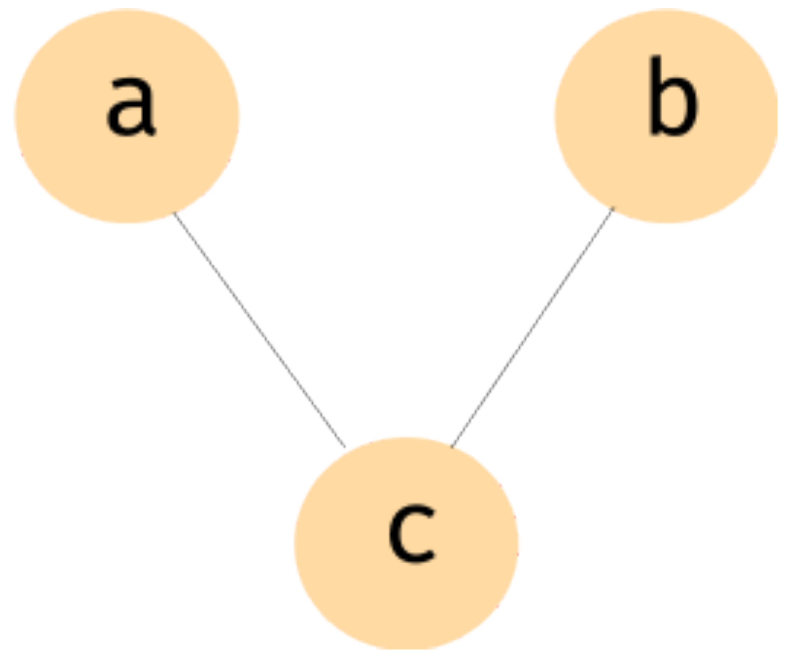
```
a: {S1-S2, S4-S5}

b: {S2-S4}

c: {S3-S6}
```

Interference graphs

- Represent program variables/temporaries as nodes.
- If the live ranges of variables u and v overlap, then draw an edge between u and v.
- An edge `(u,v)` indicates that variables u and v interfere, and hence cannot be mapped to the same register.

```
    [S1] a = 0
L1: [S2] b = a + 1
    [S3] c = c + b
    [S4] a = b * 2
    [S5] N = a + 3
    [S6] return c
```

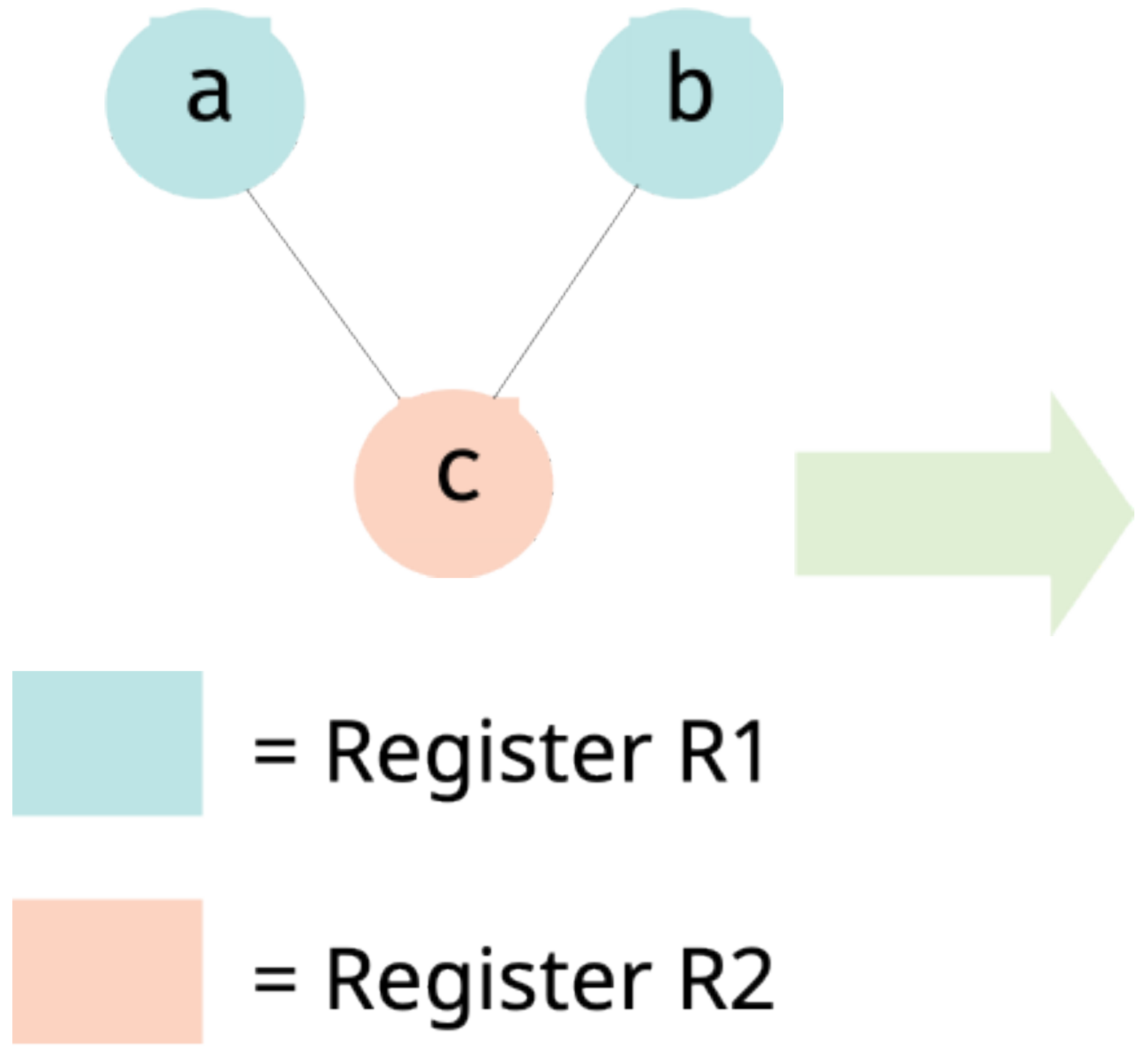
```
a: {S1-S2, S4-S5}

b: {S2-S4}

c: {S3-S6}
```



Register allocation using graph coloring

- **Idea:**
  - If we can color the interference graph using K colors, then we can allocate the variables to K registers.
  - Two nodes that interfere with each other must use different colors.

```
    [S1] a = 0
L1: [S2] b = a + 1
    [S3] c = c + b
    [S4] a = b * 2
    [S5] N = a + 3
    [S6] return c
```



```
    R1 = 0
L1: R1 = R1 + 1
    R2 = R2 + R1
    R1 = R1 * 2
    N = R1 + 3
    return R2
```

Graph coloring

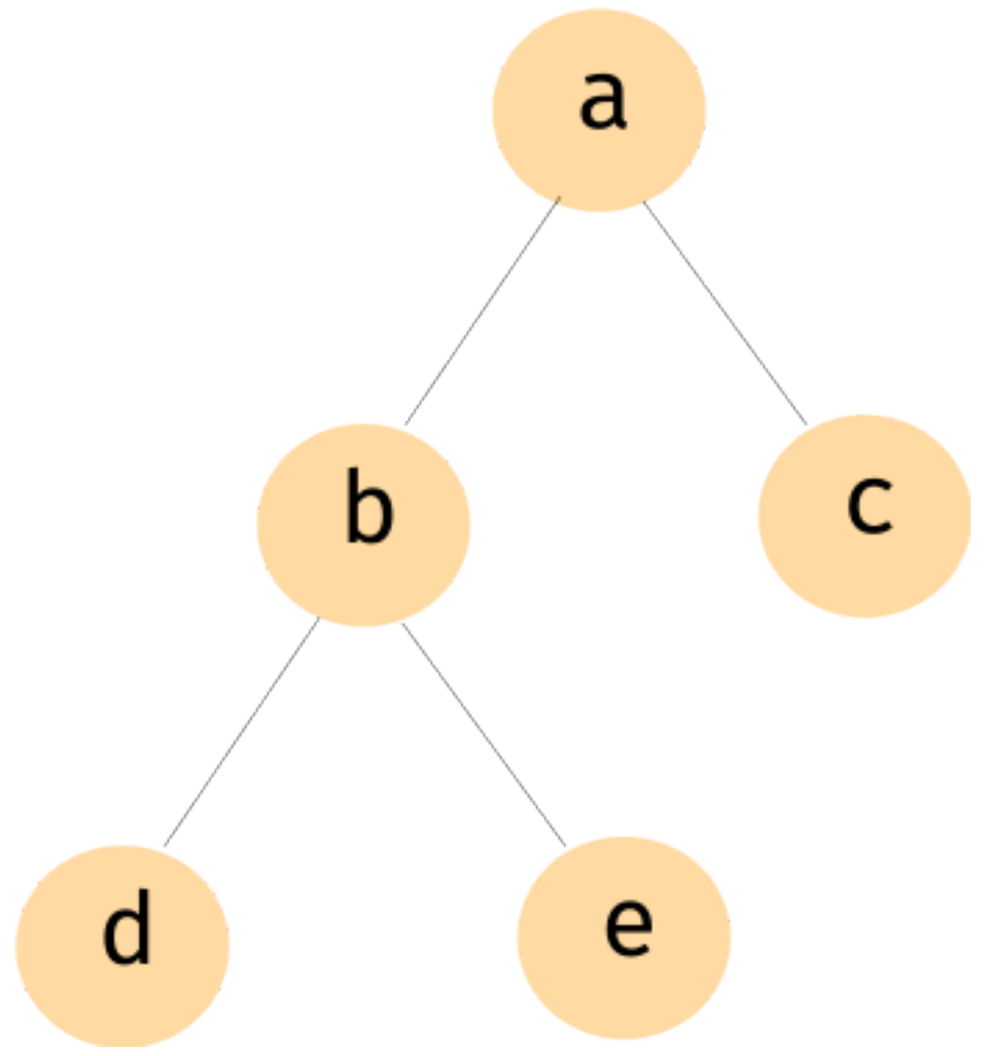
- Can we efficiently find a K-coloring of the graph?
    - Bad news: Graph coloring is an NP-complete problem.
- Can we efficiently find the optimal coloring of the graph (i.e., using the least number of colors)?
    - We don't necessarily need the perfect coloring.
    - Compute an approximation with heuristics.
- What do we do when there aren't enough colors (registers) to color the graph?
    - Temporarily move a variable to memory (slow, but what else!).
    - Called **spilling**.
    - Need to add instructions to "store" and (later) "load" the spilled variable.

Graph coloring: A simplistic approach

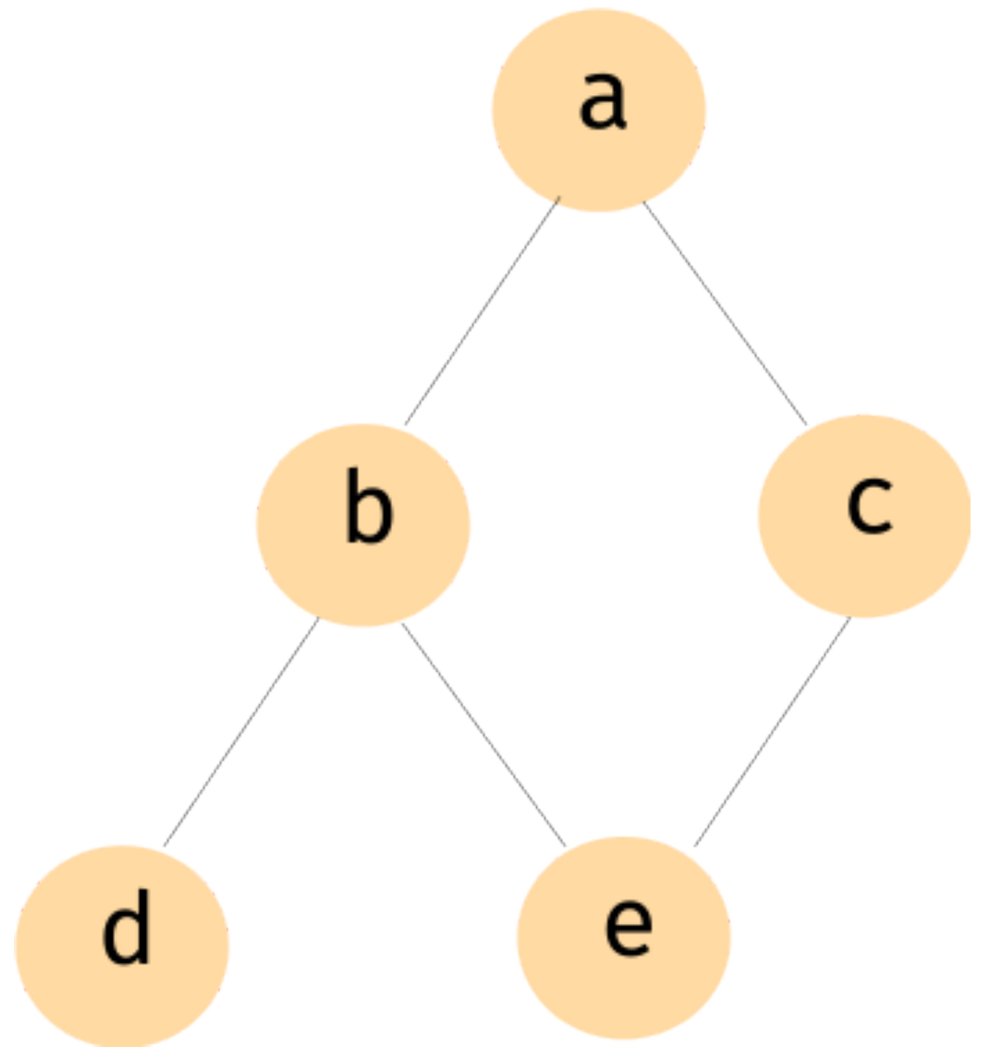
```
repeat
    repeat                                                              Simplification
        Remove a node n and all its edges from G, such that the degree of n is less than K
        Push n onto a stack
    until G has no node with degree less than K
    // G is now either empty or all its nodes have degree ≥K
    if G is not empty then
        Take a node m and all its edges out of G, and mark m for spilling
    endif
until G is empty
Take one node at a time from stack and assign a non-conflicting color
```

Need for spill

Is this graph 2-colorable?



What about this one?

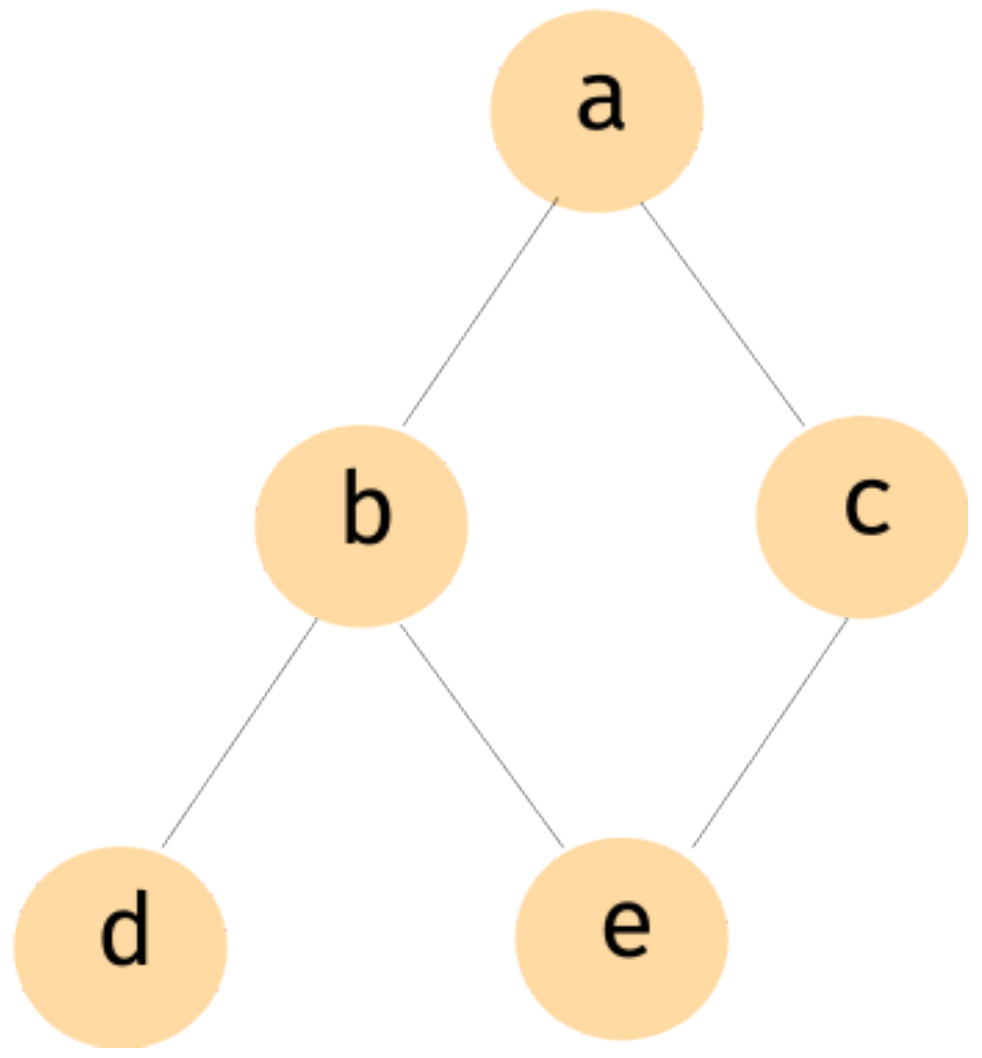


Kempe's heuristic (1879) to reduce spilling (Chaitin, 1981)

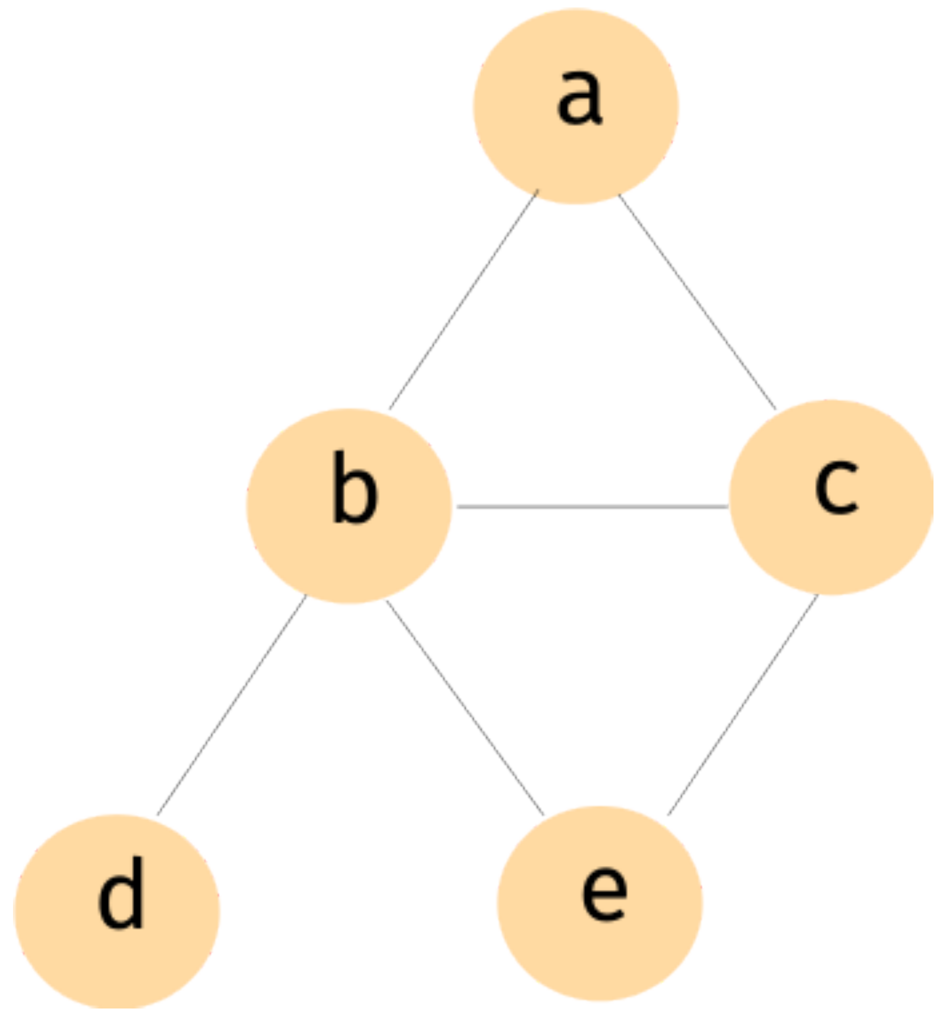
```
repeat
    repeat                                                                     Simplification
        Remove a node n and all its edges from G, such that the degree of n is less than K
        Push n onto a stack
    until G has no node with degree less than K
    // G is now either empty or all its nodes have degree ≥K
    if G is not empty then
        Take a node m out of G
        Push m onto stack
    endif
until G is empty
Take one node at a time from stack and assign a non-conflicting color if possible, else spill
```

NFS revisited!

No need to spill now.



Don't have a choice; need to spill.



Coalescing

- If there is a copy statement `x = y` such that x and y do not interfere:
  - We can use the same register for both x and y
  - Merge the graph nodes for x and y into one node

- Good because:
  - Reduces the number of registers and removes move instructions

- Bad because:
  - Increases the number of neighbors (i.e. the degree) of the merged node, which may lead to more spilling!

Conservative Coalescing

- **Idea:** Apply tests for coalescing that preserve colorability.
- Suppose *a* and *b* are candidates for coalescing into node *ab*.
- Briggs: Coalesce only if *ab* has <K neighbors of **significant** degree (degree >=K)
  - *Simplification* first removes all insignificant-degree neighbors
  - *ab* will then be guaranteed to be adjacent to <K neighbors
  - *Simplification* can then remove *ab*
- George: Coalesce only if all significant-degree neighbors of *a* already interfere with *b*
  - *Simplification* removes all insignificant-degree neighbors of *a*
  - Remaining significant-degree neighbors of *a* already interfere with *b*, hence coalescing does not increase the degree of any node

Register allocation (Cont.)

- Register allocation is expensive
  - Many algorithms use heuristics for graph coloring
  - Still it may take time quadratic in the number of live ranges
- Online/JIT compilers need to generate code quickly
  - Sacrifice efficient register allocation for compilation speed
- Linear scan register allocation
  - Massimiliano Poletto and Vivek Sarkar (ACM TOPLAS 1999)
  - **Idea:** Make one pass over the list of variables
  - Spill variables with longest lifetimes – those that would tie up a register for the longest time

Linear Scan Register Allocation (LSRA)

- Compute live *intervals*
  - A live interval for a variable is a range `[i,j]`, such that
    - The variable is not live before instruction `i`
    - The variable is not live after instruction `j`
  - Overlapping intervals imply interference
- Given `R` registers and `N` overlapping intervals
  - `R` intervals allocated to registers
  - `N-R` intervals spilled
- **Key:** Choosing the right intervals to spill

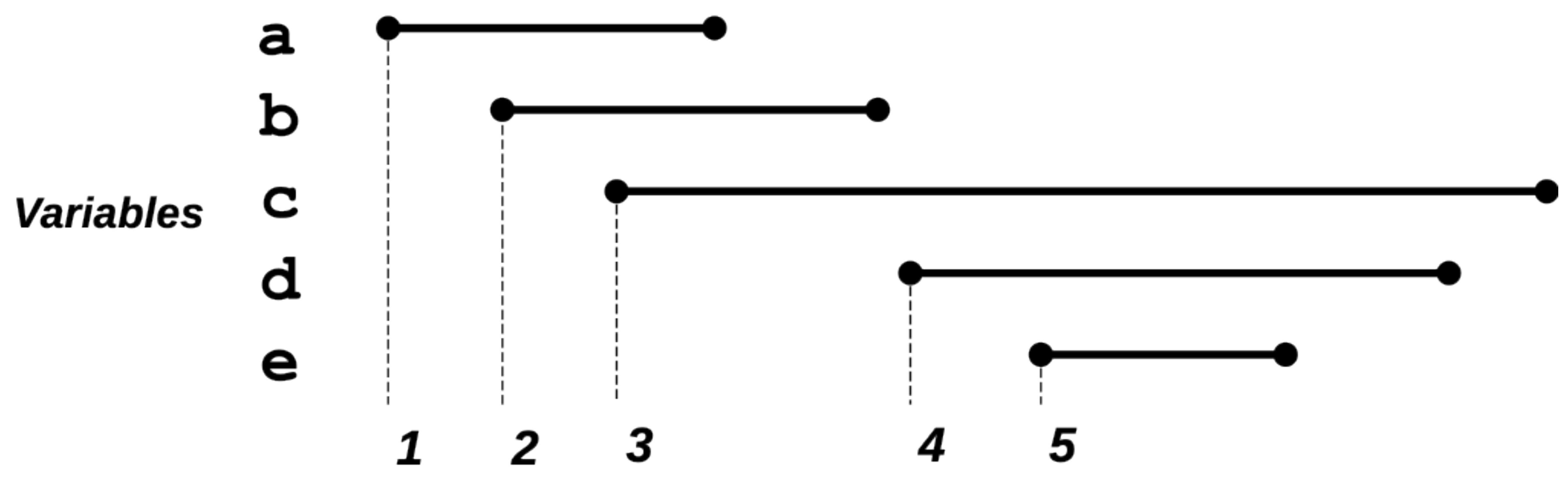
Linear scan algorithm

- Sort live intervals
    - In order of increasing start points
    - Quickly find the next live interval in order
- Maintain a sorted list of `active` intervals
    - In order of increasing end points
    - Quickly find expired intervals
- At each *step*, update `active` as follows:
    - Add the next interval from the sorted list
    - Remove any expired intervals (those whose end points are earlier than the start point of the new interval)

Linear scan algorithm (Cont.)

- Restriction:
  - Never allow `active` to have more than R elements
- Spill scenario:
  - `active` has R elements; new interval doesn't cause any existing intervals to expire
- **Heuristic:**
  - Spill the interval that ends last (furthest from current position)
    - *Greedy algorithm*

LSRA example (2 registers)



Two registers, one spill

- Step 1: `active = {a}`
- Step 2: `active = {a, b}`
- Step 3: `spill c ==> active = {a, b}`
- Step 4: `a and b expire; active = {d}`
- Step 5: `active = {e, d}`

LSRA: A few comments

- Significantly faster RA than graph coloring
- Efficacy of RA not as good:
  - Holes in live ranges not taken into account
    - Graph coloring can take care by maintaining different graphs at different points
  - A variable once spilled remains spilled forever
    - Improvements exist; e.g., Traub et al.'s second-chance binpacking
- The choice in most fast JIT compilers
- **Next:** Can we pack >1 values in one register?