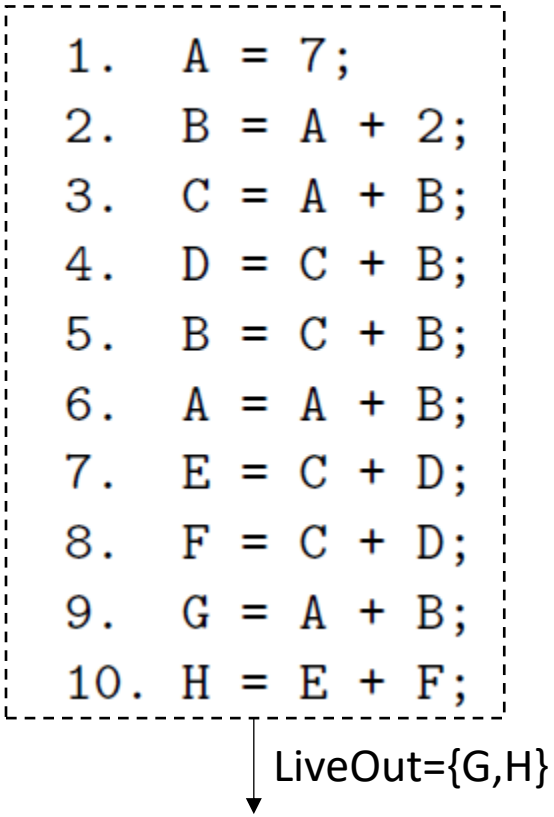
CS620T: Compilers – Principles and Implementation

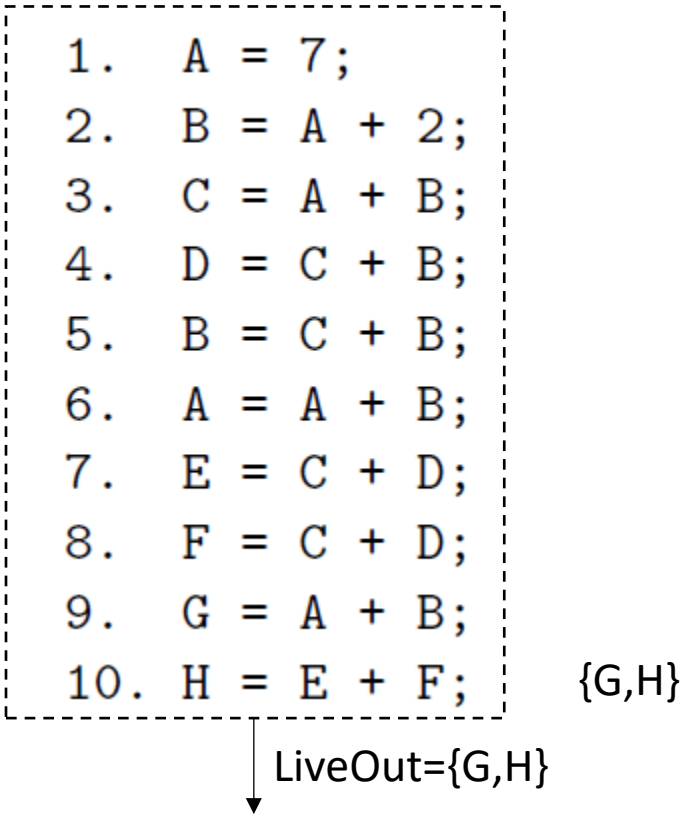
Autumn 2026

Week 14: Register Allocation via Graph Coloring

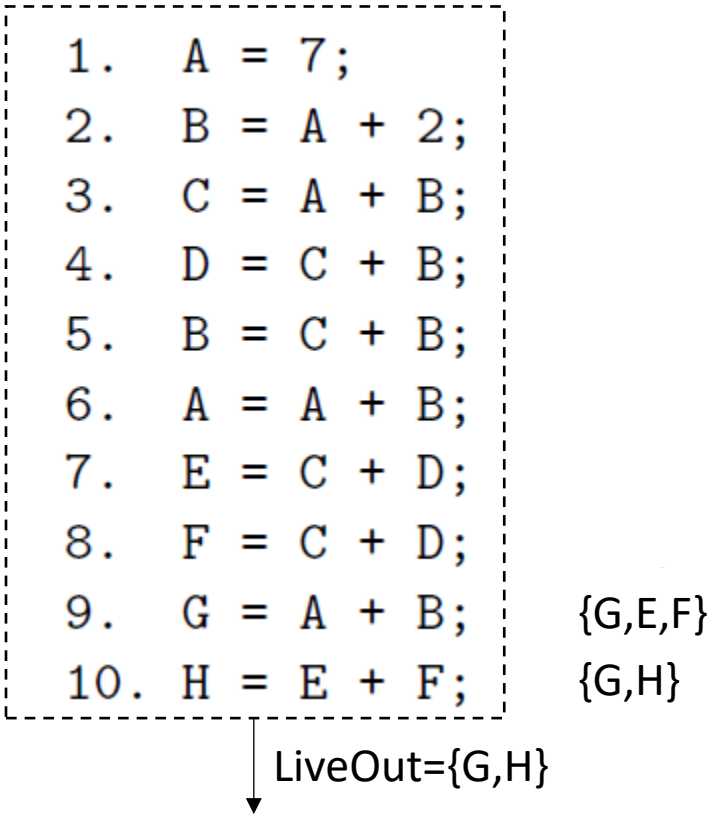
Register Allocation via Graph Coloring - Example



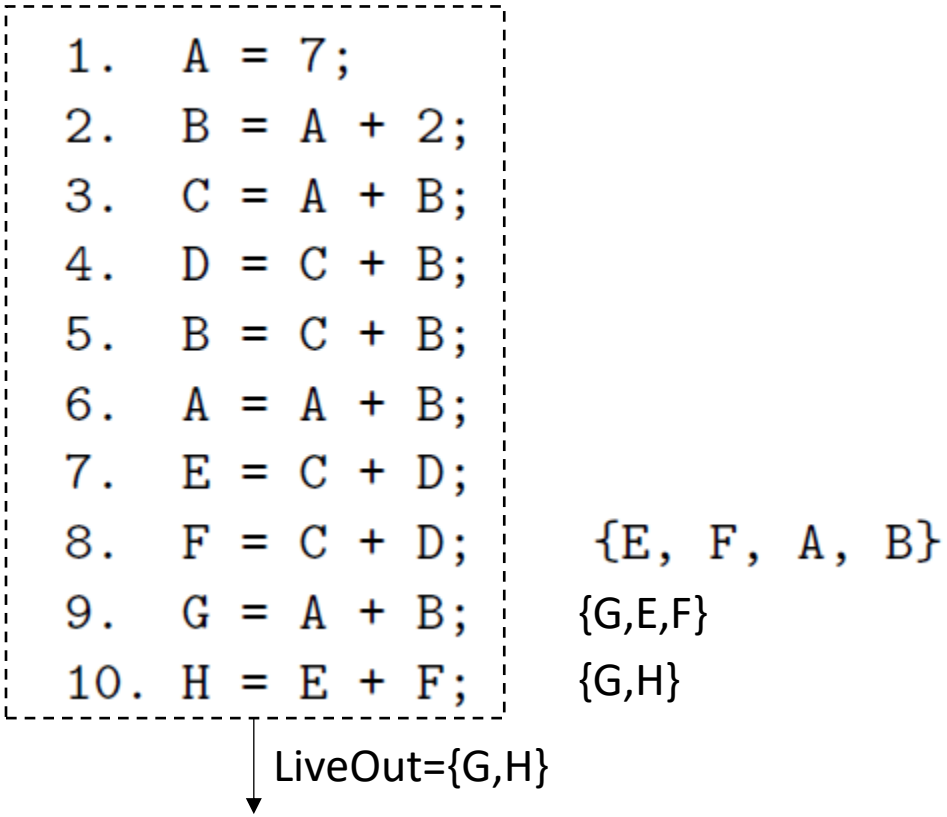
Register Allocation via Graph Coloring - Example



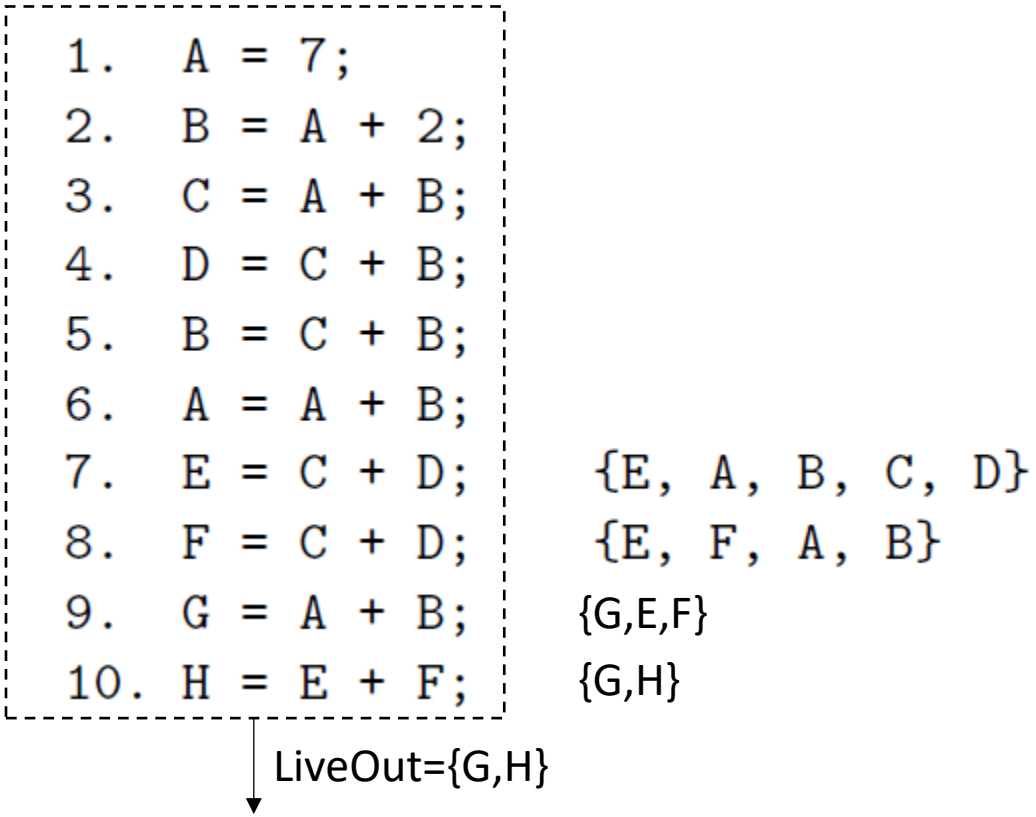
Register Allocation via Graph Coloring - Example



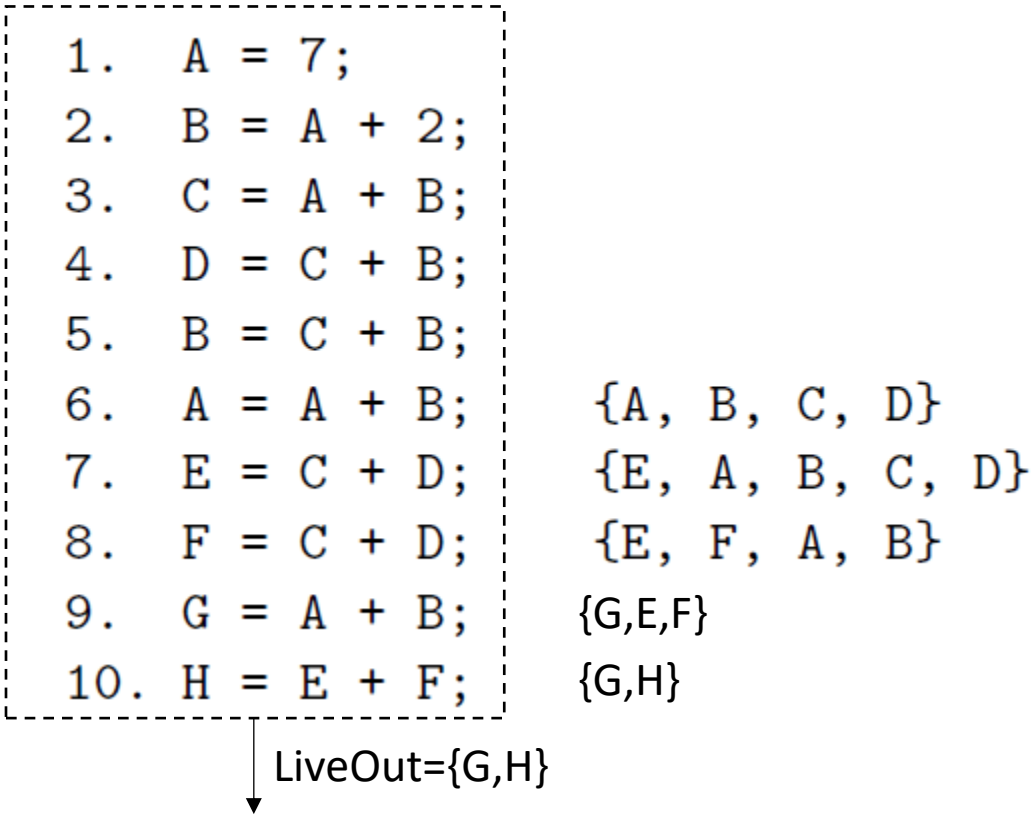
Register Allocation via Graph Coloring - Example



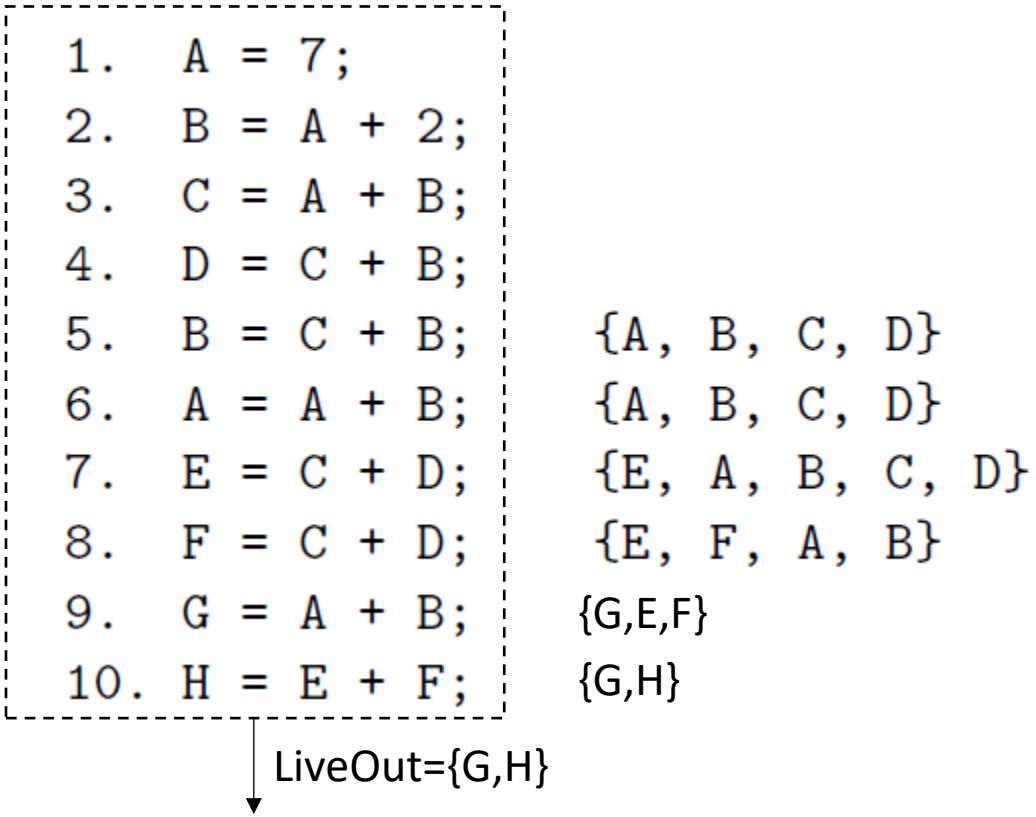
Register Allocation via Graph Coloring - Example



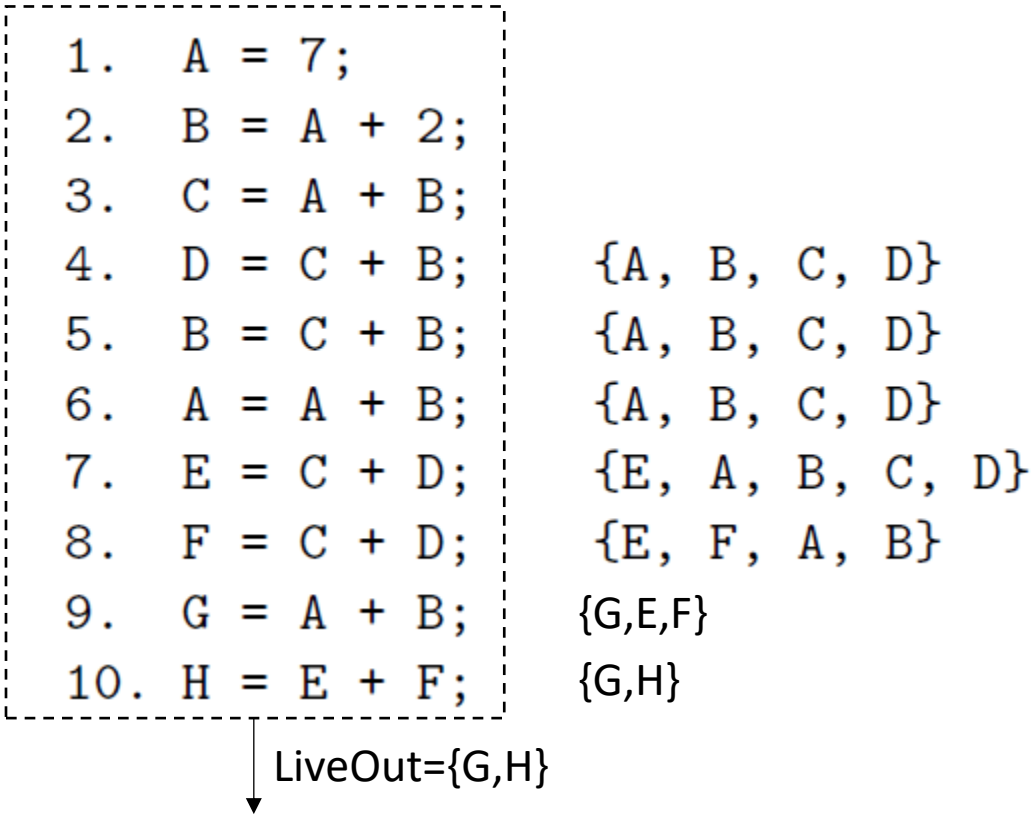
Register Allocation via Graph Coloring - Example



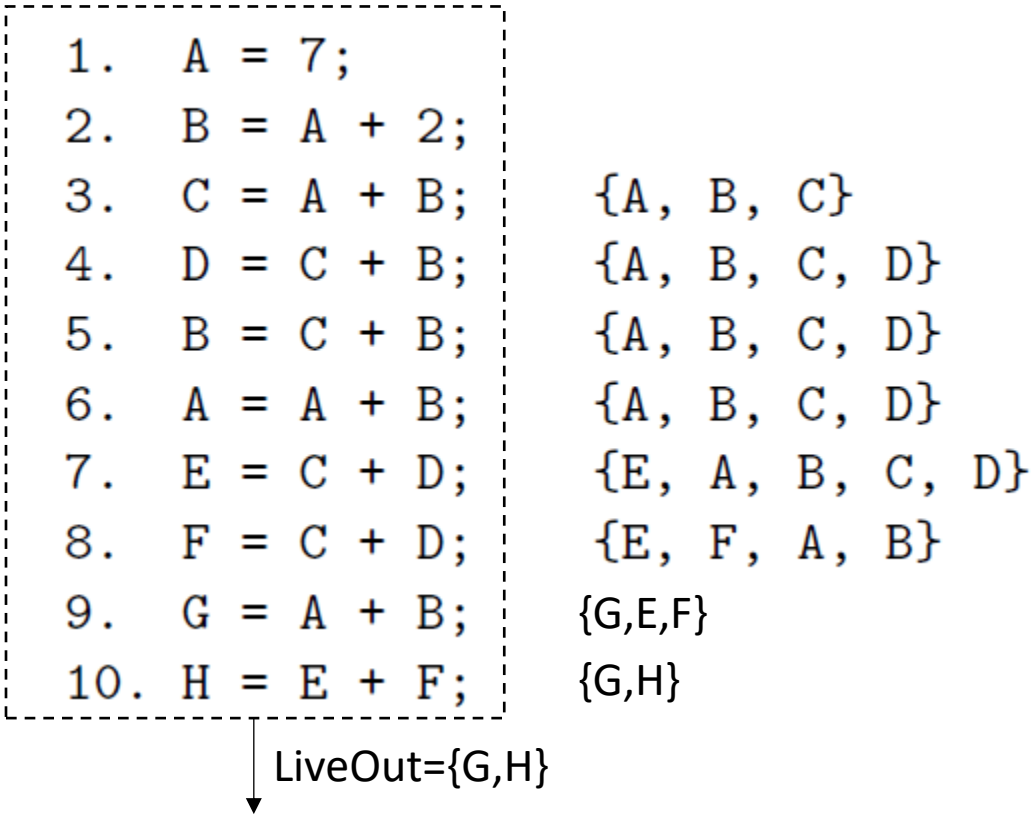
Register Allocation via Graph Coloring - Example



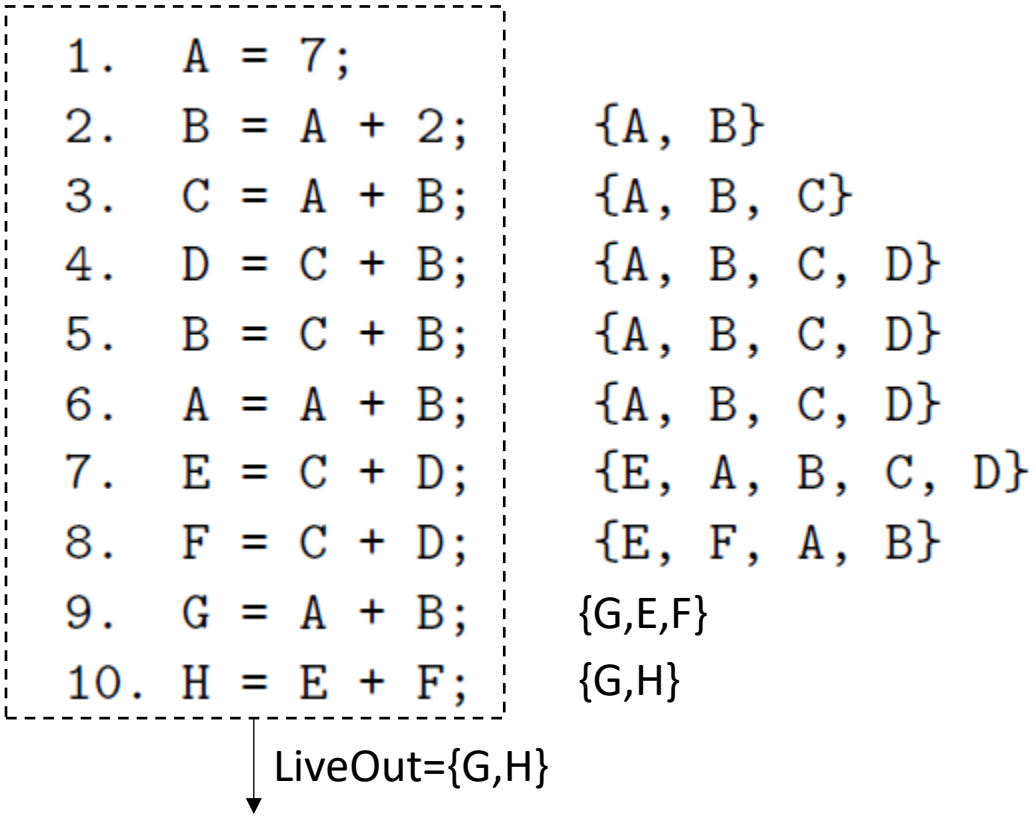
Register Allocation via Graph Coloring - Example



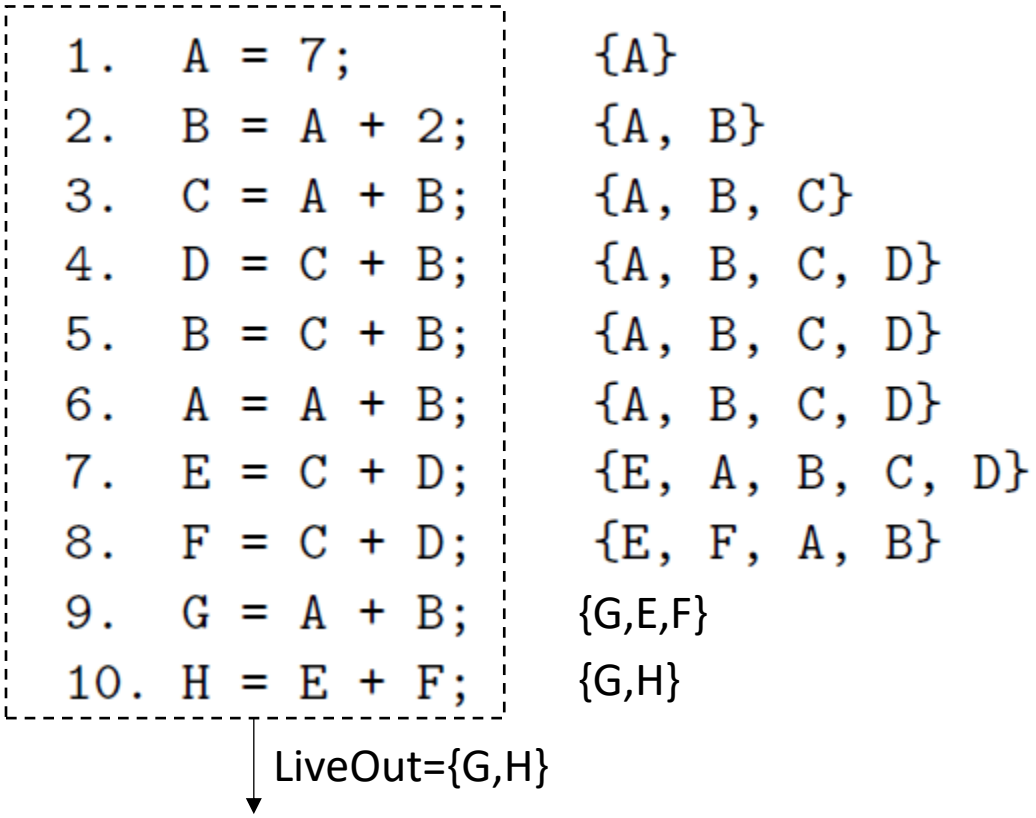
Register Allocation via Graph Coloring - Example



Register Allocation via Graph Coloring - Example

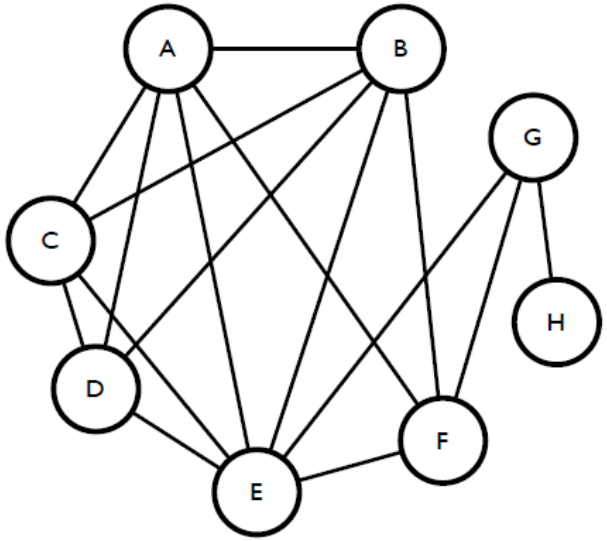


Register Allocation via Graph Coloring - Example



Register Allocation via Graph Coloring - Example

{A}
{A, B}
{A, B, C}
{A, B, C, D}
{A, B, C, D}
{A, B, C, D}
{E, A, B, C, D}
{E, F, A, B}
{G,E,F}
{G,H}



Interference graph

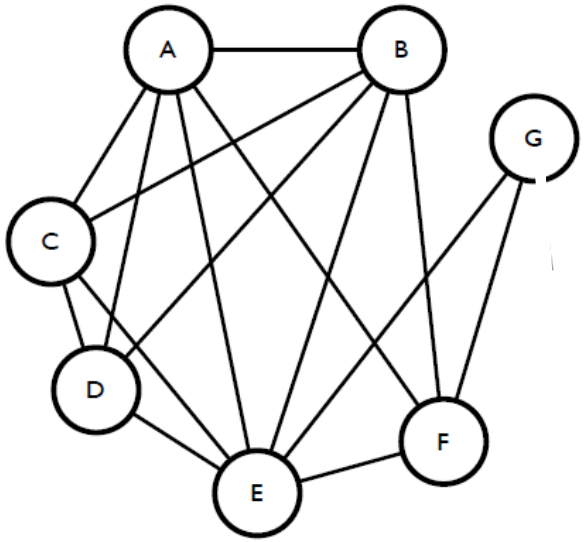
Remove H

Customized rules (3-coloring):

- Remove nodes in reverse alphabetical order
- Spill variables that are used least (spill the variable with most number of edges in case of a tie)

Register Allocation via Graph Coloring - Example

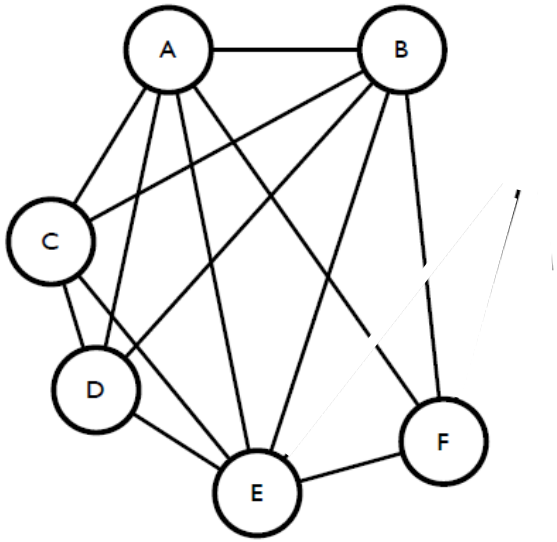
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove G

Register Allocation via Graph Coloring - Example

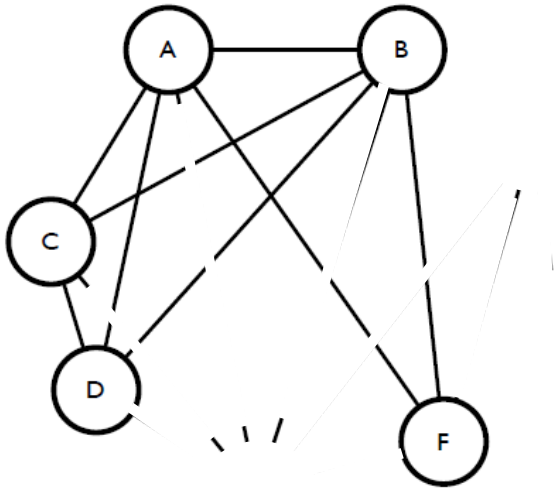
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove E

Register Allocation via Graph Coloring - Example

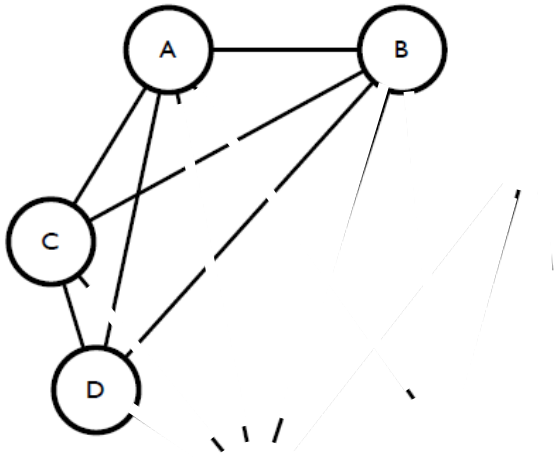
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove F

Register Allocation via Graph Coloring - Example

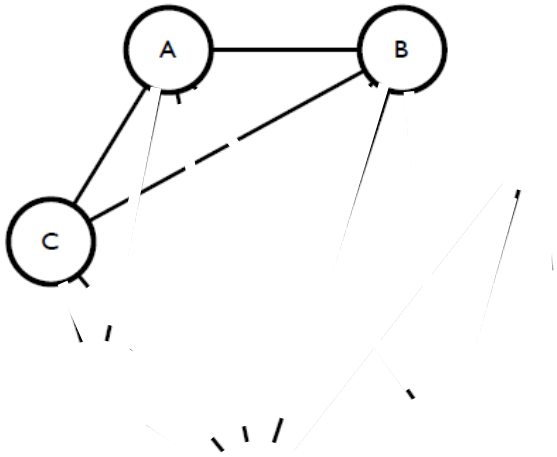
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove D

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove C then B then A

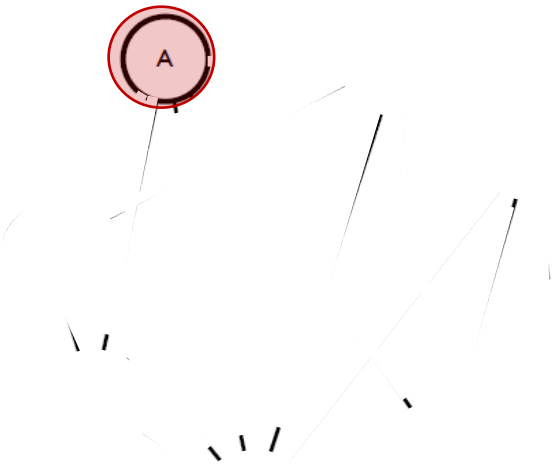
Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```

Stack:
A
B
C
D
F
E
G
H

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```

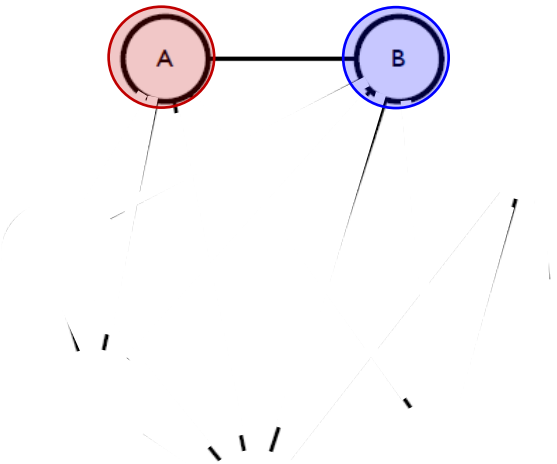


Stack: A - Red
B
C
D
F
E
G
H

3-Color the variables:

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



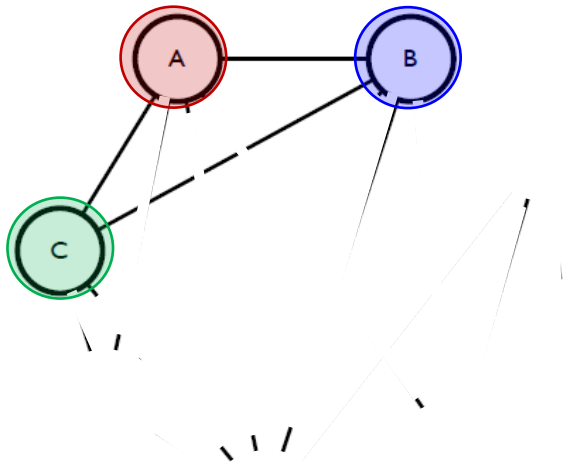
Stack:

B - Blue
C
D
F
E
G
H

3-Color the variables:

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



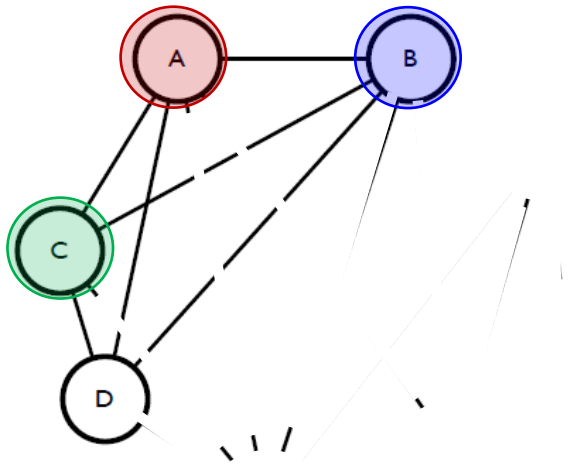
Stack:

C-Green
D
F
E
G
H

3-Color the variables:

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Stack:

D - ??
F
E
G
H

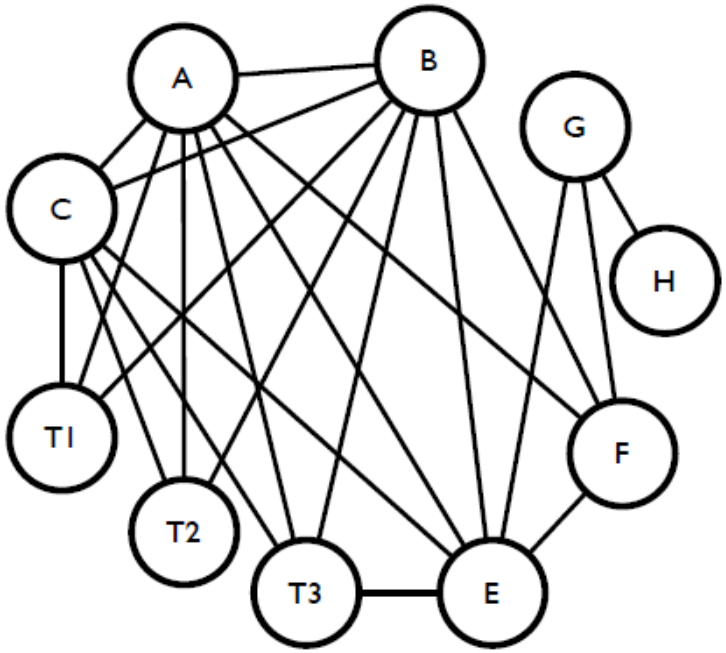
3-Color the variables: Spill D

Register Allocation via Graph Coloring - Example

Earlier code:

```
A = 7;
B = A + 2;
C = A + B;
D = C + B;
B = C + B;
A = A + B;
E = C + D;
F = C + D;
G = A + B;
H = E + F;
```

	Rewritten code:	Liveness info:
1.	A = 7;	{A}
2.	B = A + 2;	{A, B}
3.	C = A + B;	{A, B, C}
4.	T1 = C + B;	{A, B, C, T1}
4'.	ST T1, D	{A, B, C}
5.	B = C + B;	{A, B, C}
6.	A = A + B;	{A, B, C}
6'.	LD D, T2	{A, B, C, T2}
7.	E = C + T2;	{A, B, C, E}
7'.	LD D, T3	{A, B, C, E, T3}
8.	F = C + T3;	{A, B, E, F}
9.	G = A + B;	{G, E, F}
10.	H = E + F;	{G, H}

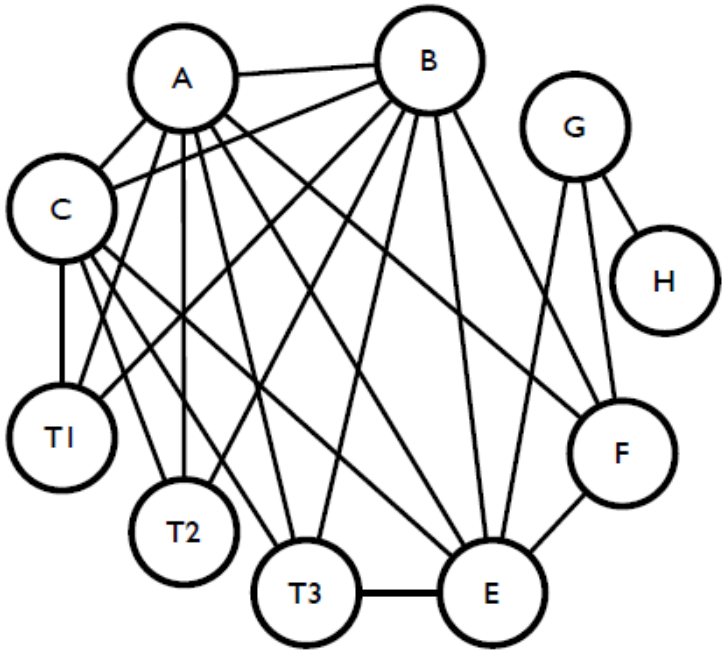


New interference graph

3-Color the variables: Spill D, rewrite code and recalculate liveness

Register Allocation via Graph Coloring – Exercise

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   T1 = C + B;     {A, B, C, T1}
4'.  ST T1, D        {A, B, C}
5.   B = C + B;      {A, B, C}
6.   A = A + B;      {A, B, C}
6'.  LD D, T2        {A, B, C, T2}
7.   E = C + T2;     {A, B, C, E}
7'.  LD D, T3        {A, B, C, E, T3}
8.   F = C + T3;     {A, B, E, F}
9.   G = A + B;      {G, E, F}
10.  H = E + F;      {G, H}
```



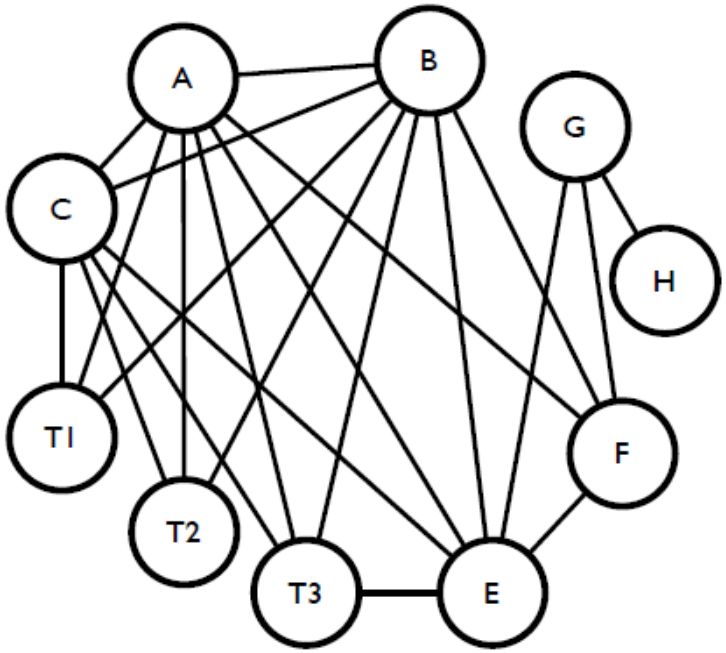
Simplify (step 1)

Stack (left-bottom, right-top): H, G~~, E, F, C, T1, T2, T3, B, A~~

~~?~~

Register Allocation via Graph Coloring - Exercise

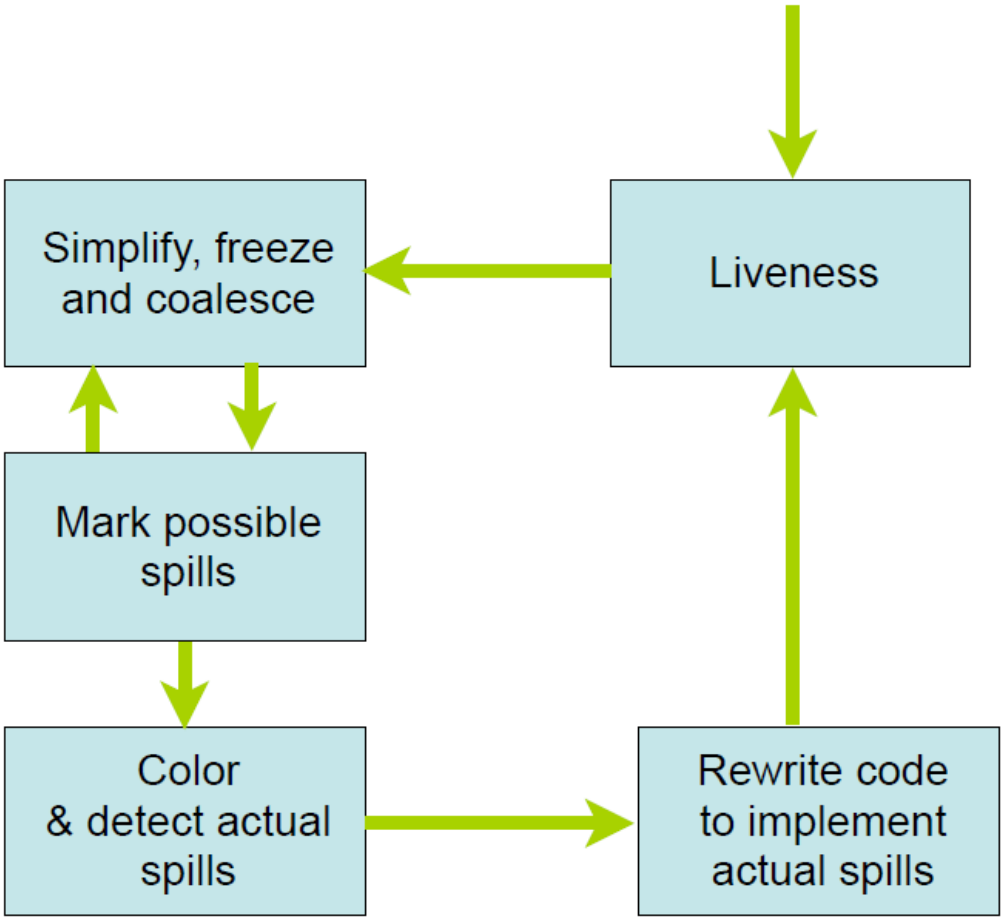
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   T1 = C + B;     {A, B, C, T1}
4'.  ST T1, D        {A, B, C}
5.   B = C + B;      {A, B, C}
6.   A = A + B;      {A, B, C}
6'.  LD D, T2        {A, B, C, T2}
7.   E = C + T2;     {A, B, C, E}
7'.  LD D, T3        {A, B, C, E, T3}
8.   F = C + T3;     {A, B, E, F}
9.   G = A + B;      {G, E, F}
10.  H = E + F;      {G, H}
```



Color (step 2) Stack (left-bottom, right-top): H, G~~, E, F, C, T1, T2, T3, B, A~~

Which node must be Spilled now?

Overall Algorithm



Additional slide on spilling

Rewriting code for spilling

- Assign spilled variables to stack locations
  (can share stack locations across variables that don't interfere)
- Replace each use/def of a spilled variable with a new fresh temporary.
- Add code around use/def to shuttle data from/to stack location.
- Example: t2 is selected for spilling and assigned to stack location `-24(%rbp)` in this code:

```
addq t1, t2
...
movq 8(t2), t3
```

- Invent temporary variables t90, t91 for *just these instructions*, rewrite:

```
movq -24(%rbp), t90
addq t1, t90
movq t90, -24(%rbp)
…
movq -24(%rbp), t91
movq 8(t91), t3
```

(May be able to avoid introducing new temps via fancy addressing modes)

- Adds nodes in general, but new nodes have fewer edges ⇒ easier to color[12]

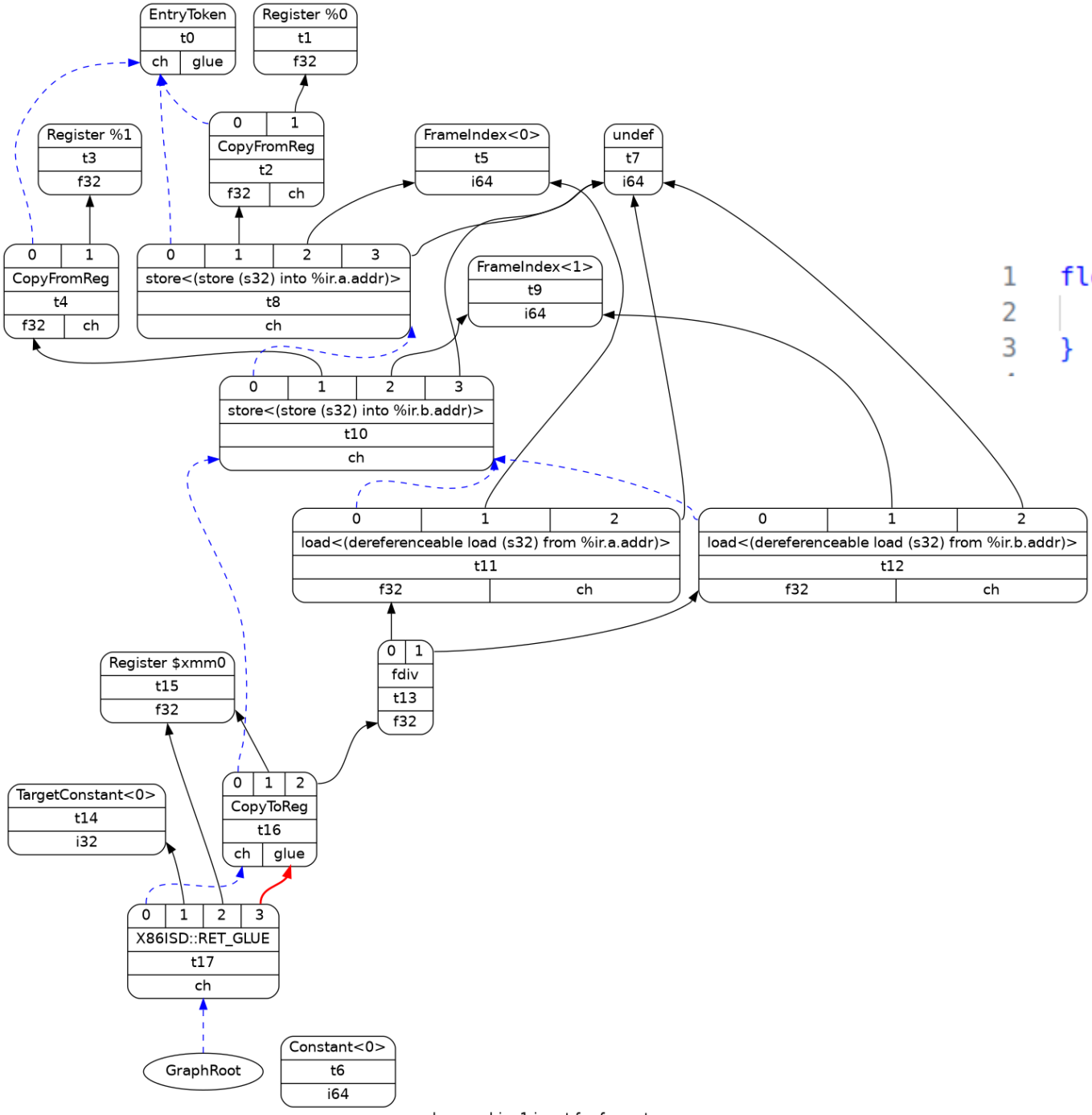
Slide courtesy: https://www.cs.cornell.edu/courses/cs4120/2016sp/lectures/22regalloc/lec22-sp16.pdf

Instruction Selection in LLVM

- **Objective:** LLVM IR to target-specific machine instruction
- LLVM implementations: SelectionDAG, GlobalISel
- Considering LLVM's codegen support for a variety of target platforms, the design of Instruction Selection module has reusable and target specific components.
- SelectionDAG - why DAG representation?
  - After few iterations, the DAG we would have makes it easy to schedule instructions. (schedulingDAG).
  - Enable automation

Selection DAG

```
1  float foo(float a, float b){
2      return a/b;
3  }
```



dag-combine1 input for foo:entry

Selection DAG

- Node
  - Opcode, Operand, Value(s), Machine Value Type,
  - Values are of two kinds: representing data- and control- flow
- Edges
  - Simple: represent dataflow from e.g., type int to another int
  - Chained: define ordering between nodes that have side effect. E.g. stores, loads, call, return.
- Can have legal and illegal DAGs
  - Illegal DAG: a node with value, whose type is not supported on the target machine. E.g. Native support for `i1` type on most modern processors.

Instruction Selection

1. Build Initial DAG
2. Optimize Initial DAG
3. Legalize SelectionDAG Types
4. Optimize Selection DAG
5. Legalize SelectionDAG Operations
6. Optimize SelectionDAG
7. Select Instructions from SelectionDAG (also called DAG to DAG)
8. Selection DAG scheduling and Formation

Demo

Legalize Type

```
1.  Optimized lowered selection DAG: %bb.0 'foo:'
2.  SelectionDAG has 15 nodes:
3.      t0: ch,glue = EntryToken
4.      t2: i32,ch = CopyFromReg t0, Register:i32 %0
5.                  t3: i16 = truncate t2
6.              t5: i16 = add t3, Constant:i16<5>
7.          t8: i1 = setcc t5, Constant:i16<6>, setugt:ch
8.        t10: i32 = select t8, t2, Constant:i32<9>
9.      t13: ch,glue = CopyToReg t0, Register:i32 $eax, t10
10.     t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```

```
1.  Type-legalized selection DAG: %bb.0 'foo:'
2.  SelectionDAG has 17 nodes:
3.      t0: ch,glue = EntryToken
4.      t2: i32,ch = CopyFromReg t0, Register:i32 %0
5.                  t3: i16 = truncate t2
6.              t5: i16 = add t3, Constant:i16<5>
7.            t15: i8 = setcc t5, Constant:i16<6>, setugt:ch
8.          t18: i8 = and t15, Constant:i8<1>
9.        t10: i32 = select t18, t2, Constant:i32<9>
10.     t13: ch,glue = CopyToReg t0, Register:i32 $eax, t10
11.     t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```

Legalize Type

- Promotion
- Expansion
- Soften
- Split vector
- Widen vector

Legalize Operation

```
1. Type-legalized selection DAG: %bb.0 'foo:'
2. SelectionDAG has 17 nodes:
3.     t0: ch,glue = EntryToken
4.     t2: i32,ch = CopyFromReg t0, Register:i32 %0
5.                   t3: i16 = truncate t2
6.                 t5: i16 = add t3, Constant:i16<5>
7.               t15: i8 = setcc t5, Constant:i16<6>, setugt:ch
8.             t18: i8 = and t15, Constant:i8<1>
9.           t10: i32 = select t18, t2, Constant:i32<9>
10.    t13: ch,glue = CopyToReg t0, Register:i32 $eax, t10
11.    t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```

```
1  define i32 @foo(i32 %v) {
2    %lo = trunc i32 %v to i16
3    %p = add i16 %lo, 5
4    %c = icmp ugt i16 %p, 6
5    %r = select i1 %c, i32 %v, i32 9
6    ret i32 %r
7  }
```

```
1. Legalized selection DAG: %bb.0 'foo:'
2. SelectionDAG has 15 nodes:
3.     t0: ch,glue = EntryToken
4.     t2: i32,ch = CopyFromReg t0, Register:i32 %0
5.                   t3: i16 = truncate t2
6.                 t5: i16 = add t3, Constant:i16<5>
7.               t20: i16,i32 = X86ISD::SUB t5, Constant:i16<7>
8.             t23: i32 = X86ISD::CMOV Constant:i32<9>, t2, TargetConstant:i8<3>, t20:1
9.     t13: ch,glue = CopyToReg t0, Register:i32 $eax, t23
10.    t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```

Legalize Operation

- Expansion
- LibCall
- Promotion
- Custom

LLVM Code Generator Steps

- Instruction Selection
  - LLVM IR -> DAG
- Scheduling and Formation
  - determine a schedule for DAG nodes
- SSA-based Machine Code Optimization
  - e.g. peephole optimizations
- Register Allocation
  - Unlimited virtual registers to limited machine registers. Introduce spill code if required.
- Prolog/Epilog Code generation
  - Function call conventions, frame pointer elimination
- Late Machine Code Optimization
  - Spill code scheduling, peephole optimizations
- Code Emission
  - Assembler code or direct machine code

https://llvm.org/docs/CodeGenerator.html

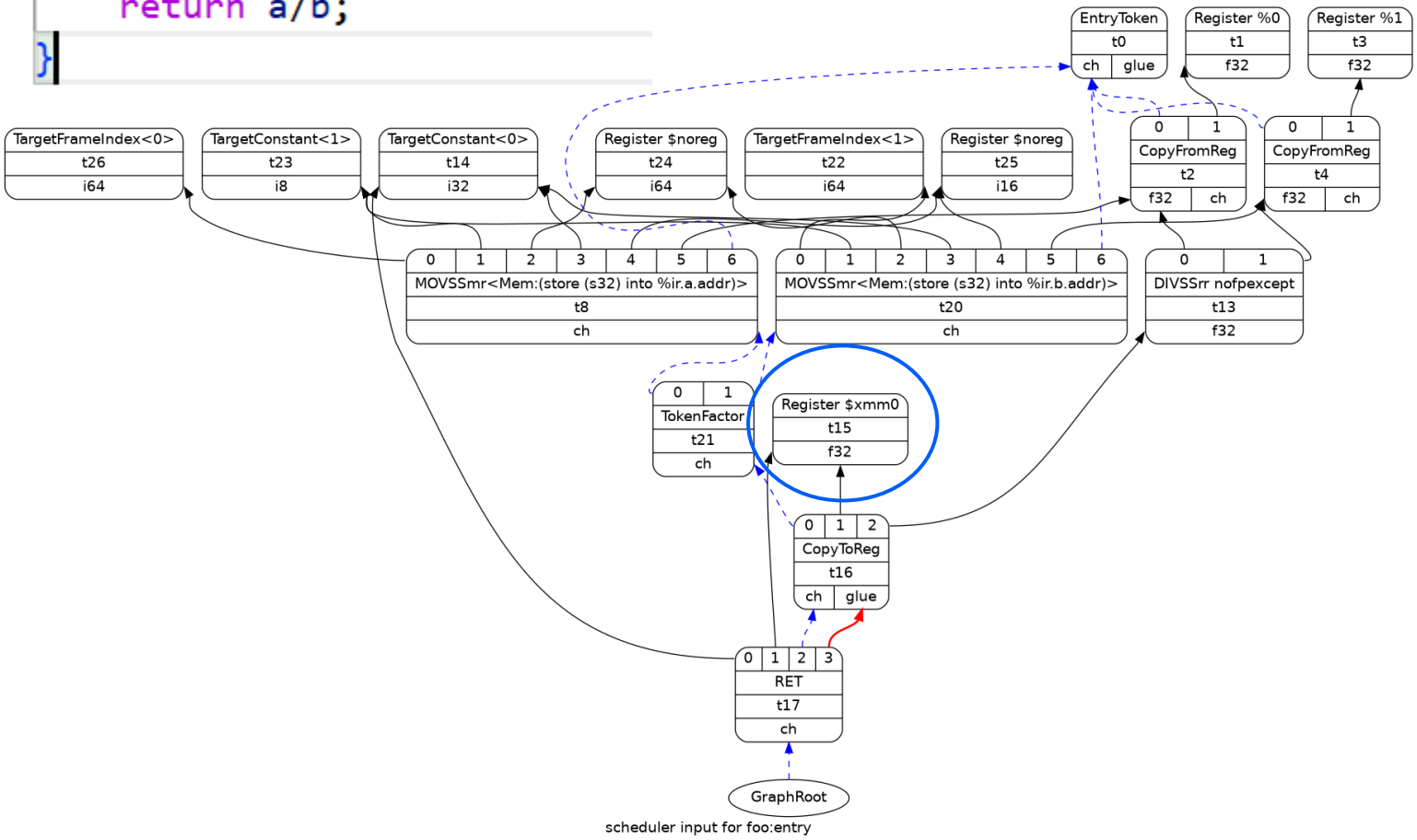
Recall what we got after instruction selection - GlobalISel

```
1  float foo(float a, float b){
2      return a/b;
3  }
```

```
109  %0:fr32 = COPY $xmm0
110  %1:fr32 = COPY $xmm1
111  %7:gr32 = COPY %0
112  MOV32mr %stack.0.a.addr, 1, $noreg, 0,
113  %8:gr32 = COPY %1
114  MOV32mr %stack.1.b.addr, 1, $noreg, 0,
115  %4:fr32 = MOVSSrm_alt %stack.0.a.addr,
116  %5:fr32 = MOVSSrm_alt %stack.1.b.addr,
117  %6:fr32 = nofpexcept DIVSSrr %4, %5, im
118  $xmm0 = COPY %6
119  RET 0, implicit $xmm0
```

Recall what we got after instruction selection - SelectionDAG

```
1  float foo(float a, float b){
2      return a/b;
3  }
```



scheduler input for foo:entry