Compiler Construction

Liveness analysis

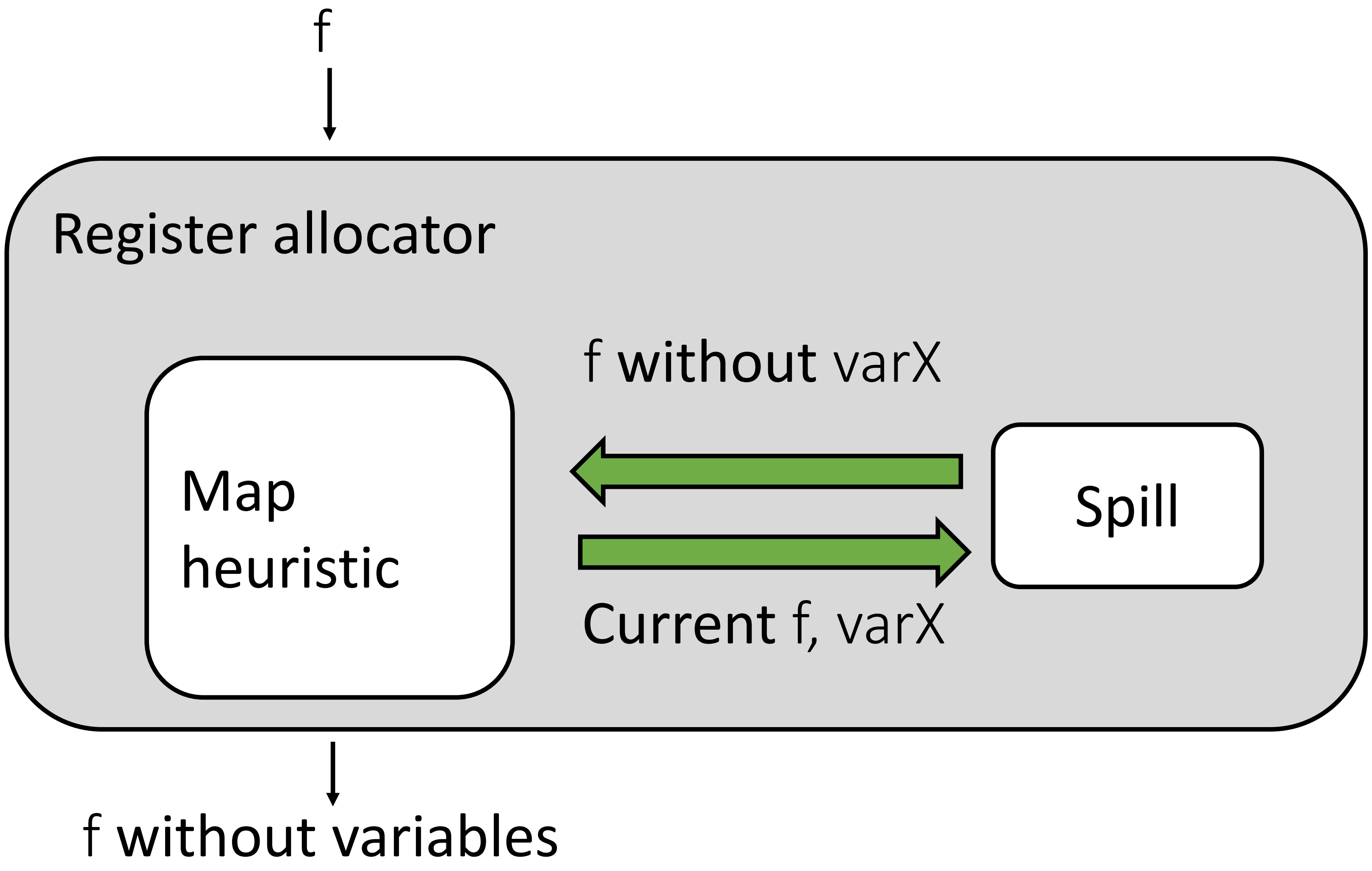
Simone Campanoni
simone.campanoni@northwestern.edu

NORTHWESTERN UNIVERSITY
QUAECUMQUE SUNT VERA
1851

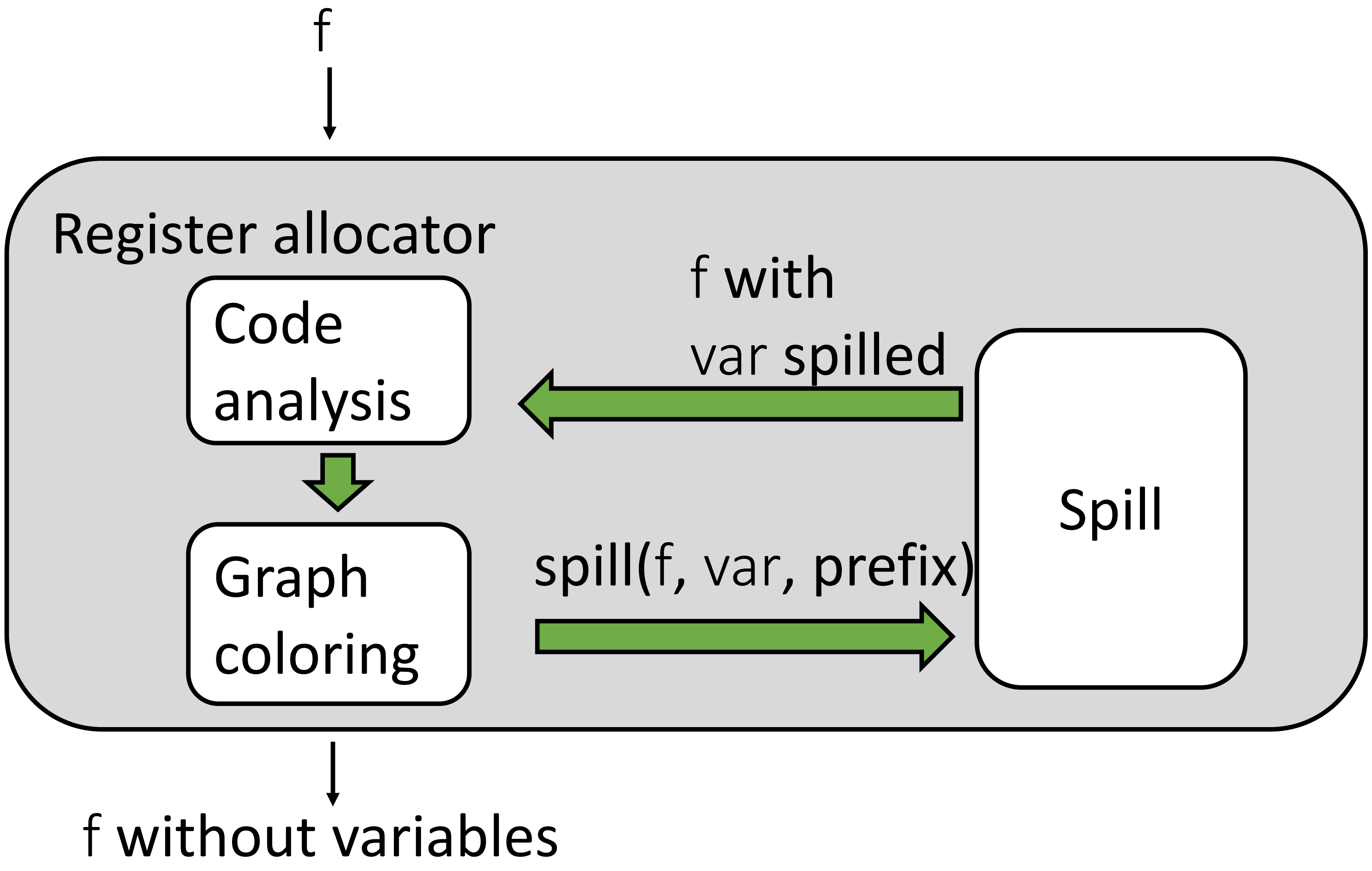
Outline

- Introduction to register allocation
- Liveness analysis
- Calling convention

A register allocator structure



A graph-coloring register allocator structure



Task: From Variables to Registers

```
(@MyVeryImportantFunction 0

  %MyVar1 <- 2
  %MyVar2 <- 40
  %MyVar3 <- %MyVar1
  %MyVar3 += %MyVar2
  print %MyVar3
)
```

MyVar1 MyVar2 MyVar3

- Why can we map MyVar1 and MyVar3 to r8?
- Why can't we map MyVar1 and MyVar2 to r8?

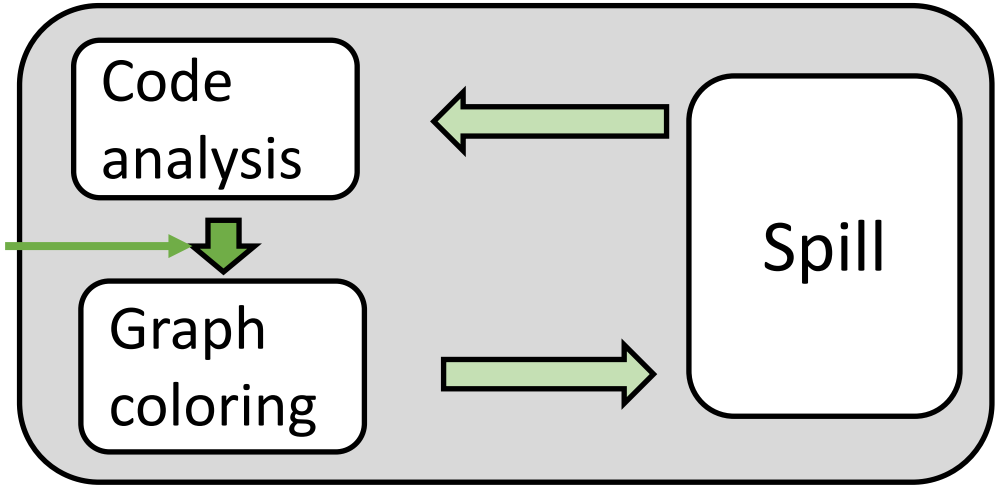
Software

Hardware

r8 r9

We built the interference graph

To compute it automatically, we need the liveness analysis



Liveness analysis

Goal:
Identify the variables whose values might be used in the future just before and just after a given instruction i, for every i in a function f

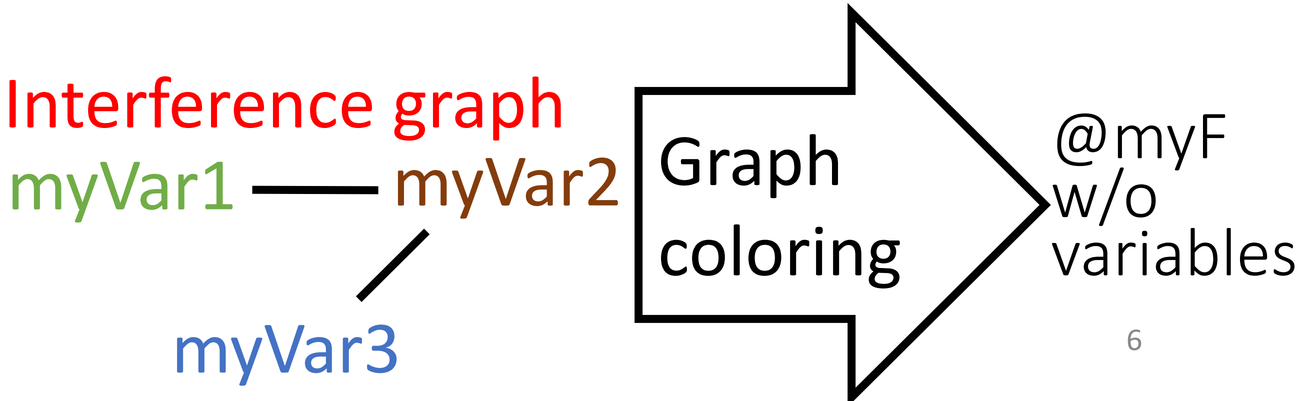
```
(@myF 0
0     %myVar1 <- 2
1     %myVar2 <- 40
2     %myVar3 <- %myVar1
3     %myVar3 += %myVar2
4     rax <- %myVar3
      return
)
```

Live ranges

IN (just before) and OUT (just after) sets

{ }	IN[0]
{myVar1}	OUT[0], IN[1]
{myVar1, myVar2}	OUT[1], IN[2]
{myVar3, myVar2}	OUT[2], IN[3]
{myVar3}	OUT[3], IN[4]

Interference graph



Outline

- Introduction to register allocation
- Liveness analysis
- Calling convention

Variables in the liveness analysis

- General Purpose (GP) 64-bit registers are seen as variables for the liveness analysis
  - rsp is not included

- Every time we say "variable" in the context of liveness analysis, we mean either L2 variables or GP 64-bit registers

- IN and OUT sets of the liveness analysis includes variables
  - Hence, they include L2 variables or GP 64-bit registers
  - IN[i] = {r10}

Execution path

```
(@myF   0
  rdi <- 5
  call print 1
  return
)
```

It is a predecessor of

```
(@myF2   1
  cjump rdi < 4 :true
  :useless_label
  rdi <- 5
  call print 1
  return
  :true
  rdi <- 7
  call print 1
  return
)
```

It is a successor of

Let i be an instruction,
we need to identify the set of variables
with values that will be used
just before and just after i
along all possible execution paths that
include i

Successors of an instruction

```
i     ::= w <- s | w <- mem x M | mem x M <- s | w <- stack-arg M |
          w aop t | w sop sx | w sop N | mem x M += t | mem x M -= t | w += mem x M | w -= mem x M |
          w <- t cmp t | cjump t cmp t label | label | goto label |
          return | call u N | call print 1 | call input 0 | call allocate 2 | call tuple-error 3 | call tensor-error F |
          w++ | w-- | w @ w w E
```

An instruction *i* that has only one successor *s* and *s* is the instruction stored just after *i*

```
rdi <- 5
call print 1
r10 <- rax < 5
```

Successors of an instruction (2)

i ::= w <- s | w <- mem x M | mem x M <- s | w <- stack-arg M |
w aop t | w sop sx | w sop N | mem x M += t | mem x M -= t | w += mem x M | w -= mem x M |
w <- t cmp t | cjump t cmp t label | label | goto label |
return | call u N | call print 1 | call input 0 | call allocate 2 | call tuple-error 3 | call tensor-error F |
w++ | w-- | w @ w w E

An instruction *i* that has only one successor *s* but *s* is not necessarily the instruction stored just after *i*

```
goto :MY_LABEL_0
:MY_LABEL_0
```

```
goto :MY_LABEL_0
rdi <- 5
call print 1
:MY_LABEL_0
```

Successors of an instruction (3)

```
i       ::= w <- s | w <- mem x M | mem x M <- s | w <- stack-arg M |
            w aop t | w sop sx | w sop N | mem x M += t | mem x M -= t | w += mem x M | w -= mem x M |
            w <- t cmp t | cjump t cmp t label | label | goto label |
            return | call u N | call print 1 | call input 0 | call allocate 2 | call tuple-error 3 | call tensor-error F |
            w++ | w-- | w @ w w E
```

An instruction *i* that has no successor

Successors of an instruction (4)

i ::= w <- s | w <- mem x M | mem x M <- s | w <- stack-arg M |
w aop t | w sop sx | w sop N | mem x M += t | mem x M -= t | w += mem x M | w -= mem x M |
w <- t cmp t | cjump t cmp t label | label | goto label |
return | call u N | call print 1 | call input 0 | call allocate 2 | call tuple-error 3 | call tensor-error F |
w++ | w-- | w @ w w E

An instruction *i* that has two successors

```
cjump rax < 5 :L1
 rdi <- 1
 rsi <- 3
 :L1
```

Now with knowledge about paths and successors
we can compute IN and OUT sets
of each instruction of a function
automatically

Liveness analysis

A variable is alive at a particular point in the program
if its value at that point will be used in a path that starts from there (the future).

A variable is dead if it is not alive.

- To compute liveness at a given point, we need to look into the future
- A variable v is alive at a given point of a program p if
  - Exist a directed path from p to an use of v and
  - that path does not contain any definition of v

```
%v1 <- 5
%v3 <- 3
%v2 <- %v1
return
```

%v1 is alive here because it is read here and it is not written here

```
%v1 <- 5
%v1 <- 3
%v2 <- %v1
return
```

%v1 is not alive here

Liveness analysis algorithm

1. Identify which variables are define
   and which ones are read (used) by an instruction
   - GEN and KILL sets
     (local information)

2. Specify how instructions transmit live values around the program
   - How to compute IN and OUT sets from GEN and KILL sets
     (global information)

3. Iterate (2) until nothing (i.e., IN and OUT set) changes
   - Notice that (1) is performed only once!
   - GEN and KILL sets are constants and, therefore, path independent!

GEN and KILL sets

- GEN[i] = {all variables read (used) by instruction i}

```
%myVar3 <- %myVar1      // GEN[i] = {%myVar1}
```

- KILL[i] = {all variables defined by instruction i}

```
%myVar3 <- %myVar1      // KILL[i] = {%myVar3}
```

%myVar3 += %myVar1

KILL[i] = {%myVar3}   GEN[i] = {%myVar1, %myVar3}

GEN and KILL sets: more examples

- GEN[i] = {all variables read (used) by instruction i}
- KILL[i] = {all variables defined by instruction i}

rdi++

KILL[i] = {rdi}

GEN[i] = {rdi}

GEN and KILL sets: more examples

- GEN[i] = {all variables read (used) by instruction i}
- KILL[i] = {all variables defined by instruction i}

cjump rdi <= %v2 :true

KILL[i] = { }

GEN[i] = {rdi, %v2}

Liveness analysis algorithm

1. Define which variables are define
   and which ones are read (used) for each instruction
   - GEN and KILL sets

2. Specify how instructions transmit live values around the program
   - How to compute IN and OUT sets from GEN and KILL sets

3. Iterate (2) until nothing changes

IN and OUT sets

- IN[i] = {all variables live right before instruction i}

  $IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$

```
i:  %v2 <- %v1
If OUT[i] = {} then
IN[i] = {%v1}
```

```
i:  %v2 <- %v1
If OUT[i]={%v2} then
IN[i] = {%v1}
```

- OUT[i] = {all variables live right after instruction i}

  $OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$

```
i    : %v3 <- 5
i+1: %v2 <- %v1
If IN[i+1] = {%v1}, then
OUT[i] = {%v1}
```

```
i    : cjump %v = 1 :s2
i+1:  :s1
i+j:   :s2
If IN[i+1] = {%v1} and
IN[i+j]={%v2},
Then OUT[i] = {%v1,%v2}
```

Algorithm

```
for (each instruction i)  {
        GEN[i] = ...
        KILL[i] = ...
}
for (each instruction i)  IN[i] = OUT[i] = { };
do{
        for (each instruction i){
                IN[i]     = GEN[i] ∪(OUT[i] – KILL[i])
                OUT[i] = ∪_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Outline

- Introduction to register allocation

- Liveness analysis

- Calling convention

Calling convention in GEN/KILL

The call reads the arguments

It models the callee,
which might change all caller save registers

	GEN	KILL
call u N	{ u, args used}	{ caller save registers}
call RUNTIME N	{ args used}	{ caller save registers}
return	{ rax, callee save registers}	{ }

It models the caller,
which might read all
callee save registers

The next lecture will show how the analysis can go wrong
if the above special rules are not enforced

Let’s run an example to show the computation of the liveness analysis

Code example

	GEN	KILL
(@myF		
0		
%a <- 2 // 1	{?}	{?}
rax <- %a // 2	{?}	{?}
→ return // 3	{?}	{?}
)		

Calling convention in GEN/KILL

	GEN	KILL
call u N	{ u, args used}	{ caller save registers}
call RUNTIME N	{ args used}	{ caller save registers}
return	{ rax, callee save registers}	{ }

Registers

Arguments	Result	Caller save	Callee save
rdi	rax	r10	r12
rsi		r11	r13
rdx		r8	r14
rcx		r9	r15
r8		rax	rbp
r9		rcx	rbx
		rdi	
		rdx	
		rsi	

Registers

Arguments	Result	Caller save	Callee save
rdi	rax	r10	r12
rsi		r11	
rdx		r8	
rcx		r9	
r8		rax	
r9		rcx	
		rdi	
		rdx	
		rsi	

Let's assume
we only have 1 callee save register
(for keeping the example
as simple as possible)

Algorithm

```
for (each instruction i)  {
        GEN[i] = ...
        KILL[i] = ...
}
for (each instruction i)  IN[i] = OUT[i] = { };
do{
        for (each instruction i){
                IN[i]     = GEN[i] ∪(OUT[i] – KILL[i])
                OUT[i] = ∪_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}		
rax <- %a // 2	{%a}	{rax}		
→ return // 3	{rax, r12}	{ }		
)				

GEN[i] = {all variables read (used) by instruction i}
KILL[i] = {all variables defined by instruction i}

Algorithm

```
for (each instruction i)  {
        GEN[i] = ...
        KILL[i] = ...
}
for (each instruction i)  IN[i] = OUT[i] = { };
do{
        for (each instruction i){
                IN[i]     = GEN[i] ∪(OUT[i] – KILL[i])
                OUT[i] = ∪_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Code example

```
(@myF
  0
  %a <- 2      // 1
  rax <- %a   // 2
  return     // 3
)
```

Instruction	GEN	KILL	IN	OUT
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{ }	{ }
return // 3	{rax, r12}	{ }	{ }	{ }

Algorithm

```
for (each instruction i)  {
        GEN[i] = ...
        KILL[i] = ...
}
for (each instruction i)  IN[i] = OUT[i] = { };
→ do{
        for (each instruction i){
                IN[i]    = GEN[i] ∪(OUT[i] – KILL[i])
                OUT[i] = ∪_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Code example

Code	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{ }	{ }
return // 3	{rax, r12}	{ }	{ }	{ }
)				

$$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$$

$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{ }	{ }
return // 3	{rax, r12}	{ }	{rax, r12}	{ }
)				

$$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$$
$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
→ rax <- %a // 2	{%a}	{rax}	{ }	{ }
return // 3	{rax, r12}	{ }	{rax, r12}	{ }
)				

$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$

$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{%a, r12}	{rax, r12}
return // 3	{rax, r12}	{ }	{rax, r12}	{ }
)				

$$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$$

$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{ }	{ }
rax <- %a // 2	{%a}	{rax}	{%a, r12}	{rax, r12}
return // 3	{rax, r12}	{ }	{rax, r12}	{ }
)				

$$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$$
$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Code example

Code	GEN	KILL	IN	OUT
(@myF				
0				
→ %a <- 2 // 1	{ }	{%a}	{r12}	{%a, r12}
rax <- %a // 2	{%a}	{rax}	{%a, r12}	{rax, r12}
return // 3	{rax, r12}	{ }	{rax, r12}	{ }
)				

$$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$$
$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Algorithm

```
for (each instruction i)  {
        GEN[i] = ...
        KILL[i] = ...
}
for (each instruction i)  IN[i] = OUT[i] = { };
do{
        for (each instruction i){
                IN[i]     = GEN[i] ∪(OUT[i] – KILL[i])
                OUT[i] = ∪_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Code example

	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{r12}	{%a, r12}
rax <- %a // 2	{%a}	{rax}	{%a, r12}	{rax, r12}
→ return // 3	{rax, r12}	{ }	{rax, r12}	{ }
)				

$$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$$

$$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$$

Algorithm

```
for (each instruction i)  {
        GEN[i] = ...
        KILL[i] = ...
}
for (each instruction i)  IN[i] = OUT[i] = { };
do{
        for (each instruction i){
                IN[i]    = GEN[i] ∪(OUT[i] – KILL[i])
                OUT[i] = ∪_{s a successor of i} IN[s]
        }
} while (changes to any IN or OUT occur);
```

Code example

Code	GEN	KILL	IN	OUT
(@myF				
0				
%a <- 2 // 1	{ }	{%a}	{r12}	{%a, r12}
rax <- %a // 2	{%a}	{rax}	{%a, r12}	{rax, r12}
return // 3	{rax, r12}	{ }	{rax, r12}	{ }
)				

- Variables within the same set are alive at the same time at that point in the code
- Hence, they cannot be placed in the same register

Homework #1

- Compute the IN and OUT sets
  of all instructions of an L2 function given as input

```
(@myF
  0
  %myVar1 <- 5
  %myVar2 <- 0
  %myVar2 += %myVar1
  return
)
```

Your work →

```
( (in
IN[0]  → (r13 r15 rax r14 rbp r12 rbx )
IN[1]  → (r13 r15 rax r14 rbp r12 %myVar1 rbx )
IN[2]  → (r13 r15 rax r14 rbp %myVar2 r12 %myVar1 rbx )
IN[3]  → (r13 r15 rax r14 rbp r12 rbx )
) (out
OUT[0] → (r13 r15 rax r14 rbp r12 %myVar1 rbx )
OUT[1] → (r13 r15 rax r14 rbp %myVar2 r12 %myVar1 rbx )
OUT[2] → (r13 r15 rax r14 rbp r12 rbx )
OUT[3] → ( )
) )
```

Testing your homework #1

- Under L2/tests/liveness there are the tests you have to pass
- A new compiler argument: -l
  - Check L2compiler.cpp on Canvas
- To test:
  - To check all tests: make test_liveness
  - To check one test: ./liveness test/liveness/test1.L2f
- Check out each input/output for each test if you have doubts
  - For example, the correct output for the test
    test/liveness/test1.L2f
    is
    test/liveness/test1.L2f.out

Debugging suggestion

- Don’t forget you have our L2 compiler binary

- So, to help you debug your work:
  - you can write your own test (a new MyTest.L2f)
  - Generate the output of our L2 compiler by invoking
    ./liveness MyTest.L2f > MyTest.L2f.out
  - Compare our output with the output generated by your L2 compiler

Always have faith in your ability

Success will come your way eventually

Best of luck!