

Simone Campanoni
simone.campanoni@northwestern.edu



Outline

- L2
- The L2 compiler

Introduction to L2

Variable is a powerful abstraction
Compilers have to deal with the complexity
of implementing that abstraction

- Like L1, but we have variables in L2!

- Example of L1 programs

```
(@go
(@go 0 1
 mem rsp 0 <- 5
 //code
 rdi <- mem rsp 0
 rdi <<= 1
 mem rsp 0 <- rdi
 return
)
)
```

*To store
a value
on the stack*

Ret addr
Local

rsp

L2 programs

```
(@go
(@go 0
 %myVar1 <- 5
 //code
 %myVar1 <<= 1
 return
)
)
```

```
(@go
(@go 0
 %myVar1 <- 5
 rdi <- %myVar1
 call print 1
 return
)
)
```

Variables in L2

- Variables (on top of registers)
- L2 variables are function local

```
(@myF
  1
  %myVar <- 5
  rdi++
  mem rsp -8 <- :RET
  call @myF2 1
  :RET
  rdi <- %myVar
  call print 1
  return
)
```

```
(@myF2
  1
  %myVar <- rdi
  %myVar++
  return
)
```

```

p ::= (l f+)
f ::= (l N N i+)
i ::= w <- s | w <- mem x M | mem x M <- s |
      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
w ::= a | rax | rbx | rbp | r10 | r11 | r12 | r13 | r14 | r15
a ::= rdi | rsi | rdx | sx | r8 | r9
sx ::= rcx
s ::= t | label | l
t ::= x | N
u ::= w | l
x ::= w | rsp
aop ::= += | -= | *= | &=
sop ::= <<= | >>=
cmp ::= < | <= | =
E ::= 1 | 2 | 4 | 8
F ::= 1 | 3 | 4
M ::= multiplicative of 8 constant (e.g., 0, 8, 16)
N ::= (+|-)? [1-9][0-9]* | 0
l ::= @name
label ::= :name
name ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*

```

L2

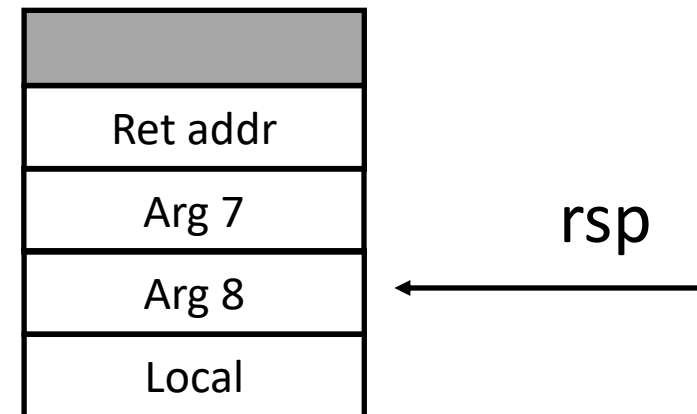
```
p ::= (l f+)
f ::= (l N i+)
i ::= w <- s | w <- mem x M | mem x M <- s | w <- stack-arg M |
     w aop t | w sop sx | w sop N | mem x M += t | mem x M -= t | w += mem x M | w -= mem x M |
     w <- t cmp t | cjump t cmp t label | label | goto label |
     return | call u N | call print 1 | call input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

w ::= a | rax
a ::= rdi | rsi | rdx | sx | r8 | r9
sx ::= rcx | var
s ::= t | label | l
t ::= x | N
u ::= w | l
x ::= w | rsp
aop ::= += | -= | *= | &=
sop ::= <<= | >>=
cmp ::= < | <= | =
E ::= 1 | 2 | 4 | 8
F ::= 1 | 3 | 4
M ::= multiplicative of 8 constant (e.g., 0, 8, 16)
N ::= (+|-)? [1-9][0-9]* | 0
l ::= @name
label ::= :name
name ::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
var ::= %name
```

A problem for L1 developers

- We want to print the last argument of @myF plus 1
- Is there any bug in our L1 program?
- We need to save r12
- Is there any bug in our L1 program?

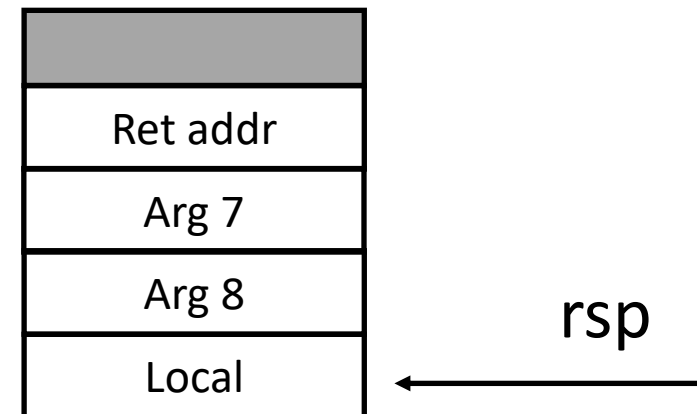
```
(@myF
  8 1 mem rsp 0 <- r12
  r12 <- mem rsp 0
  r12 += 2
  rdi <- r12
  call print 1
  return r12 <- mem rsp 0
)
```



A problem for L1 developers

- We want to print the last argument of @myF plus 1
- Is there any bug in our L1 program?
- We need to save r12
- Is there any bug in our L1 program?

```
(@myF
 8 1 ← mem rsp 0 <- r12
r12 <- mem rsp 8
r12 += 2
rdi <- r12
call print 1
return ← r12 <- mem rsp 0
)
```

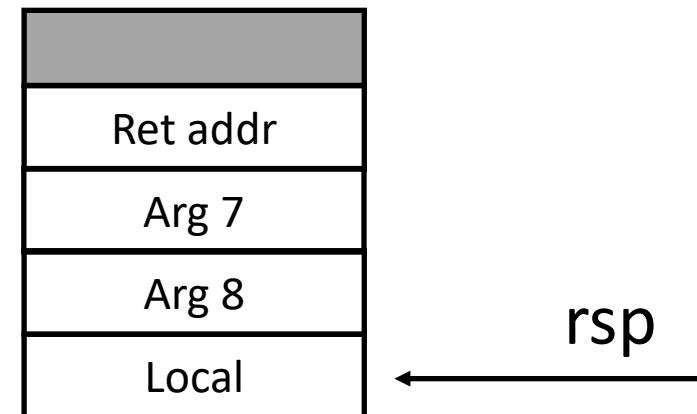


The new L2 instruction

- It accesses stack-based arguments
`w <- stack-arg M`

- `stack-arg 0` is always the last stack argument
- `stack-arg 8` is always the second to last argument

```
(@myF
  8
  r12 <- stack-arg 0
  r12 += 2
  rdi <- r12
  call print 1
  return
)
```



Stack locations for L2 programs

- No locals
- Stack locations can be used only
 - To store stack arguments at the caller site
 - To store the returning label

```
mem rsp -8 <- :MYL
```

```
mem rsp -16 <- 5
```

```
call @F 7
```

```
:MYL
```

- Hence: mem rsp X

 It must be negative

Final notes on L2

As for L1:

- The scope of labels is the program
- Values are encoded following the same rules of L1
- Same calling convention of L1
- Same rules for memory heap allocation
- Same undefined behaviors
- Same scopes for labels
- Same constraints for branches

Tests for homework 1

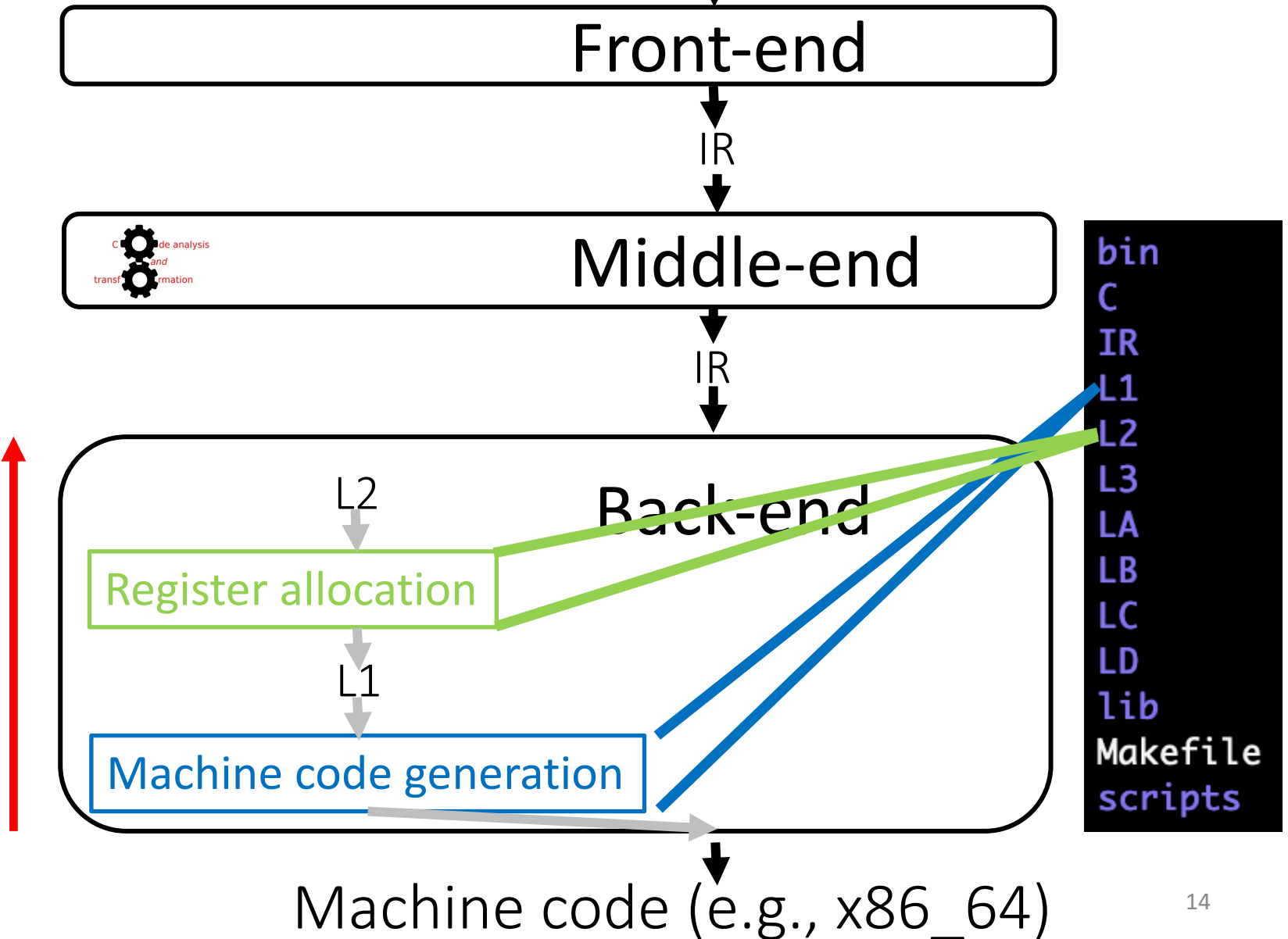
- Rewrite your L1 program using the L2 language
 - To compile an L2 program:
 - Use the original framework, which is still available on Canvas
 - Not the one you have modified to implement your L1 compiler
 - `cd L2`
 - Interpreter:
 - `./L2i L2_program.L2`
 - Compiler:
 - `./L2c L2_PROGRAM.L2 ; ./a.out`
- Write another L2 program that implements any algorithm that you want that generates an output from an input
 - Upload the input (.L2.in), the expected output (.L2.out), and the L2 program (.L2)
- Deadline: check Canvas

Outline

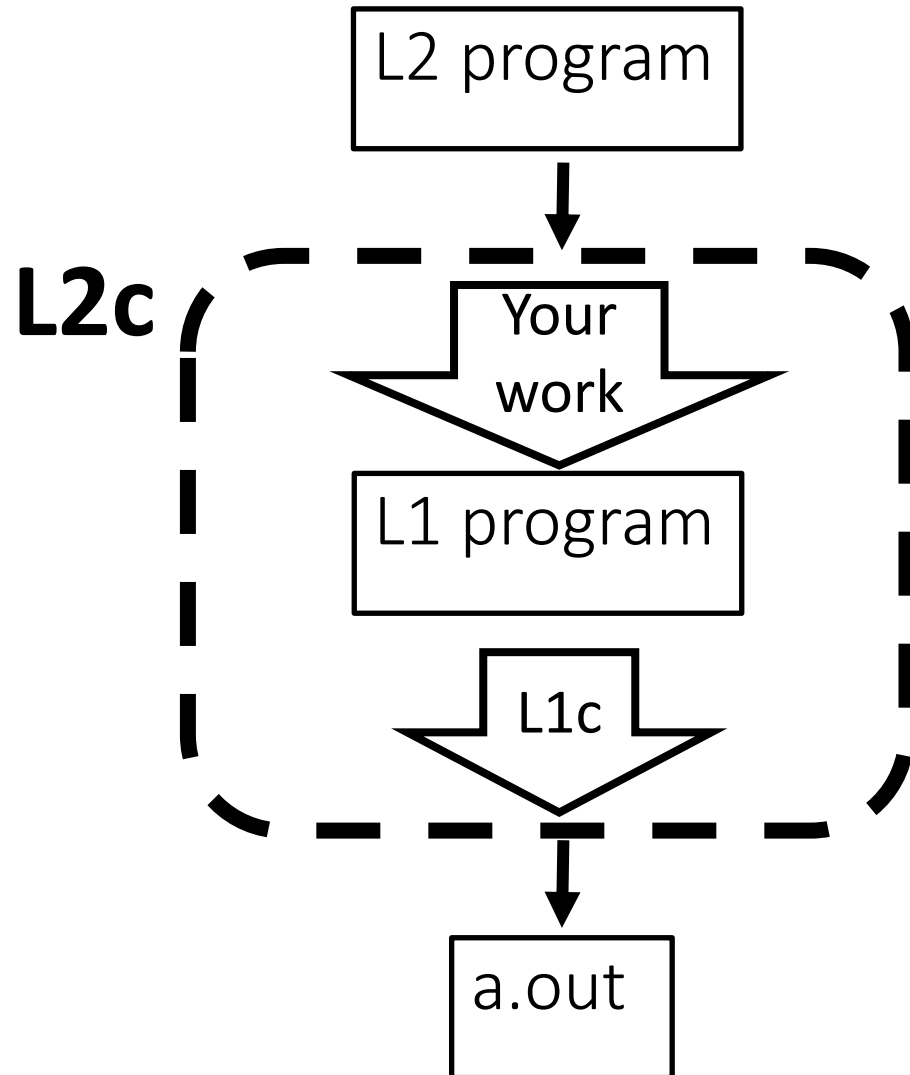
- L2
- The L2 compiler

A compiler

Character stream --- Source code (e.g., C)



The L2 compiler (L2c)



- To build L2c:
translate an L2 program
to an equivalent L1
- We need to map L2 variables to registers
 - Register allocation
- We need to translate the L2 instruction
`w <- stack-arg M`

Debugging Suggestion: testing your L1 compiler with my L2 compiler

- If your L2c does not pass a test, then the bug can be either
 - in your L1 compiler or
 - In your L2 compiler
- To understand where is the bug, you can mix your and mine compilers
- To compile an L2 program:
 - Compile your own L1 compiler: L1/bin/L1
 - Make sure you have my L2 compiler: L2/bin/L2
 - cd L2
 - ./L2c L2_PROGRAM.L2
 - ./a.out

Register allocation task

- Intra-procedural approach (most used)

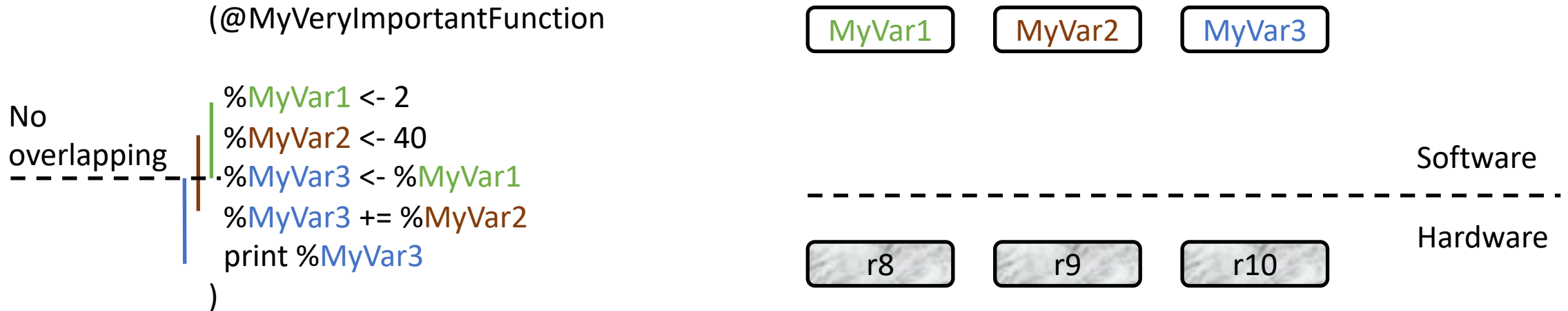
For each function f

Map each variable of f to either a register or to a stack location
(within locals in the L1 stack)

- Inter-procedural approach

Map variables of functions in registers
exploiting their caller-callee links

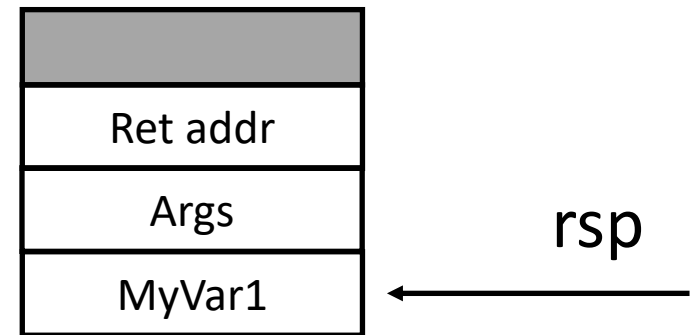
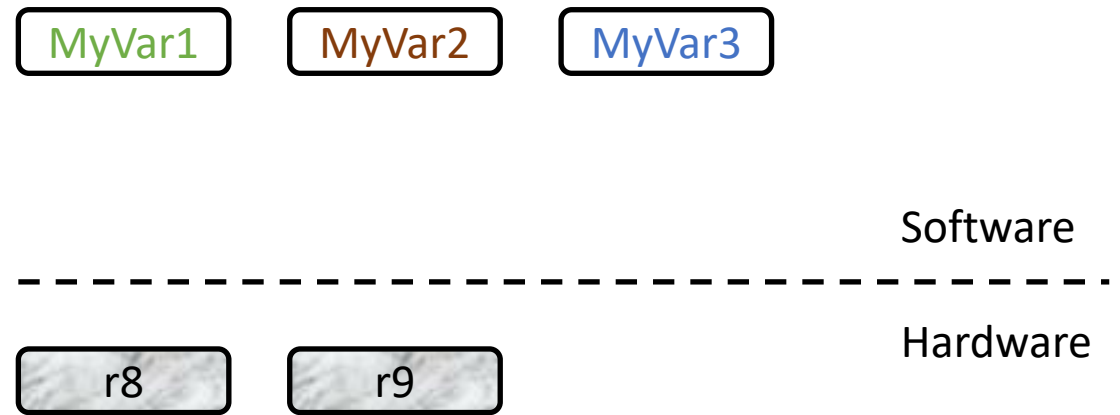
Task: From Variables to Registers



To register allocators: what are you doing?

```
(@MyVeryImportantFunction
```

```
%MyVar1 <- 2  
%MyVar2 <- 40  
%MyVar3 <- 0  
%MyVar3 += %MyVar1  
%MyVar3 += %MyVar2  
print %MyVar3  
)
```



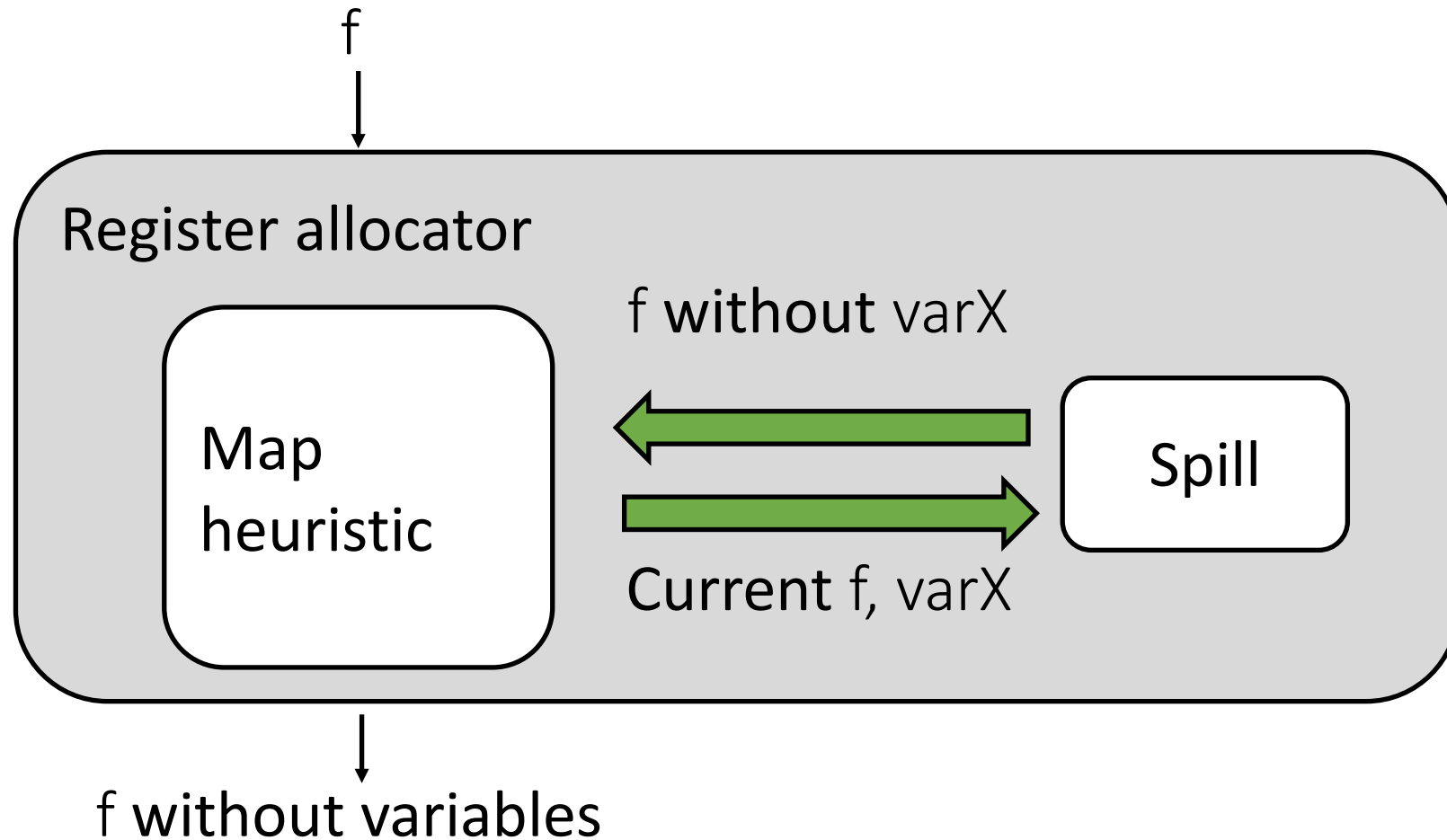
- MyVar1 -> stack (spilled)
- MyVar2 -> r8
- MyVar3 -> r9

Two naïve solutions for register allocation:

1. Spill all variables
2. Increase the #registers

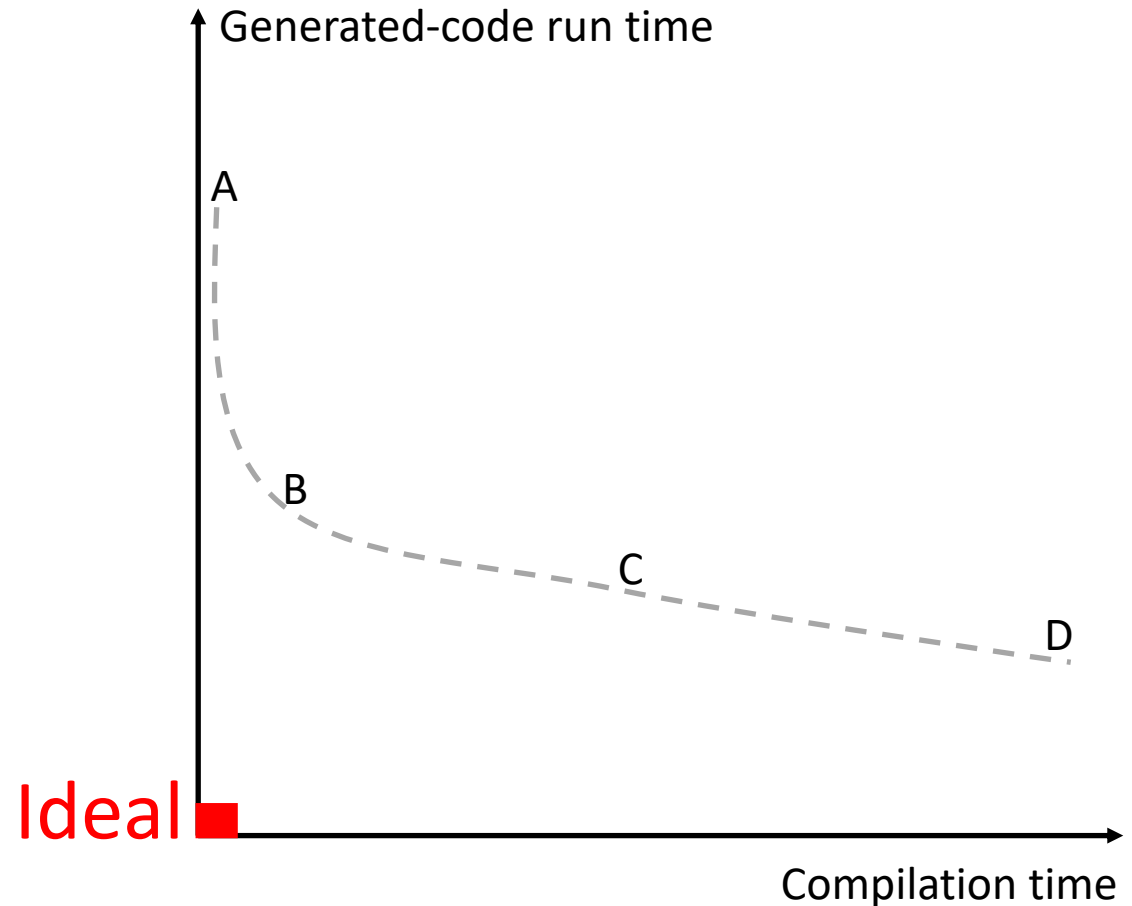


A register allocator structure



Register Allocation

- A. Spill all variables
- B. Linear scan
- C. Graph coloring
- D. Integer linear programming



Always have faith in your ability

Success will come your way eventually

Best of luck!