Compiler Construction

Graph coloring

Simone Campanoni
simone.campanoni@northwestern.edu

Outline

- Graph coloring
- Heuristics
- L2c

Graph coloring task

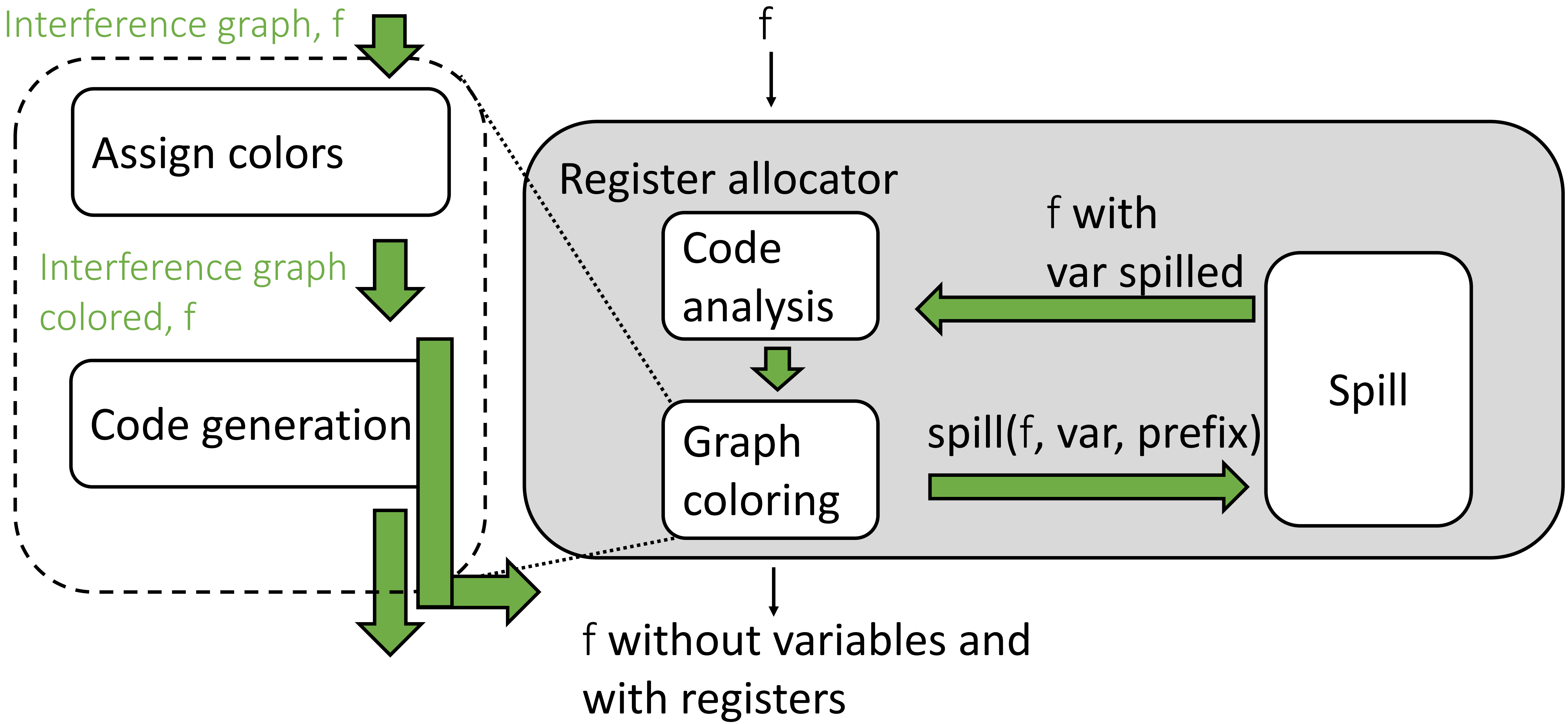
- Input : the interference graph
- Output: the interference graph where each node has a color (or fail)

- Task: Color the nodes in the graph
  such that connected nodes have different colors

- Abstraction: colors are registers

- After performing the graph coloring task:
  Replace L2 variables with the registers specified by the colors

A graph-coloring register allocator structure



Colors

- At design time of the register allocator:
  Map general purpose (GP) registers to colors

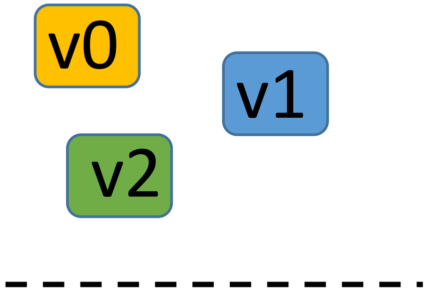
- The L1 (15) GP registers:
  rdi, rsi, rdx, rcx, r8, r9, rax, r10, r11, r12, r13, r14, r15, rbp, rbx

- Each register has one node in the interference graph
  - Pre-colored nodes

- Before starting coloring the nodes related to variables:
  Color register nodes with their own colors

A coloring algorithm

HEURISTICS

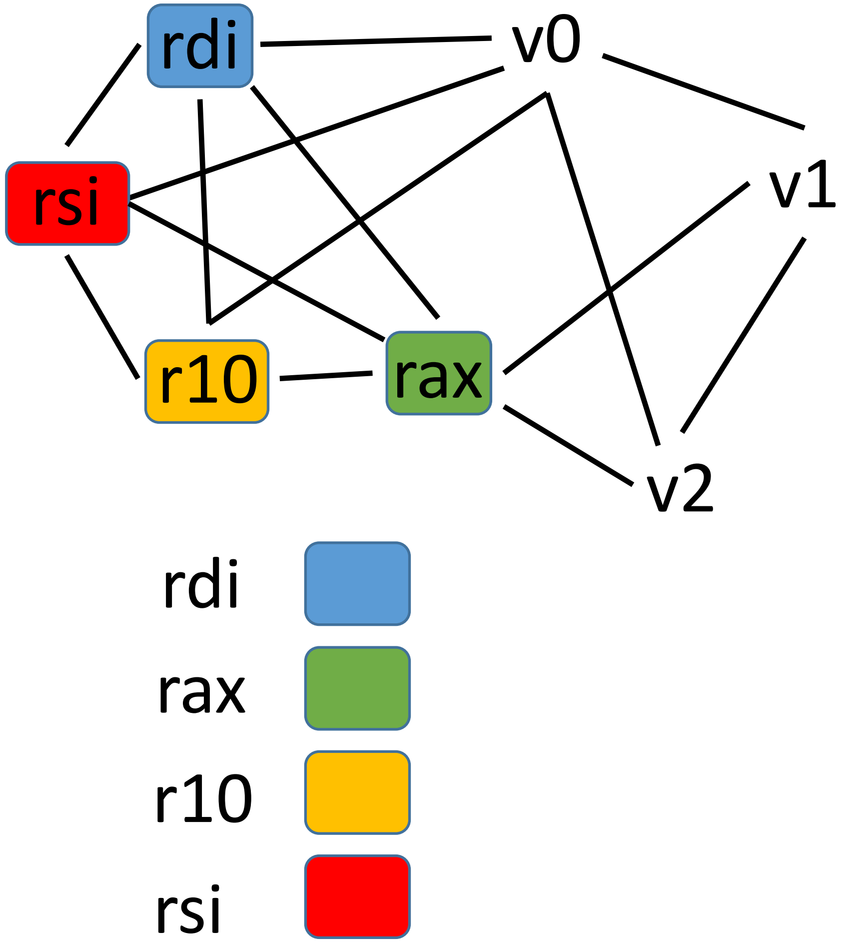


Algorithm:

1. Repeatedly select a node and remove it from the graph, putting it on top of a stack

2. When the graph is empty, rebuild it
   - Select a color on each node as it comes back into the graph, making sure no adjacent nodes have the same color
   - If there are not enough colors, the algorithm fails
     - Spilling happens in this case
     - Select the nodes you want to spill

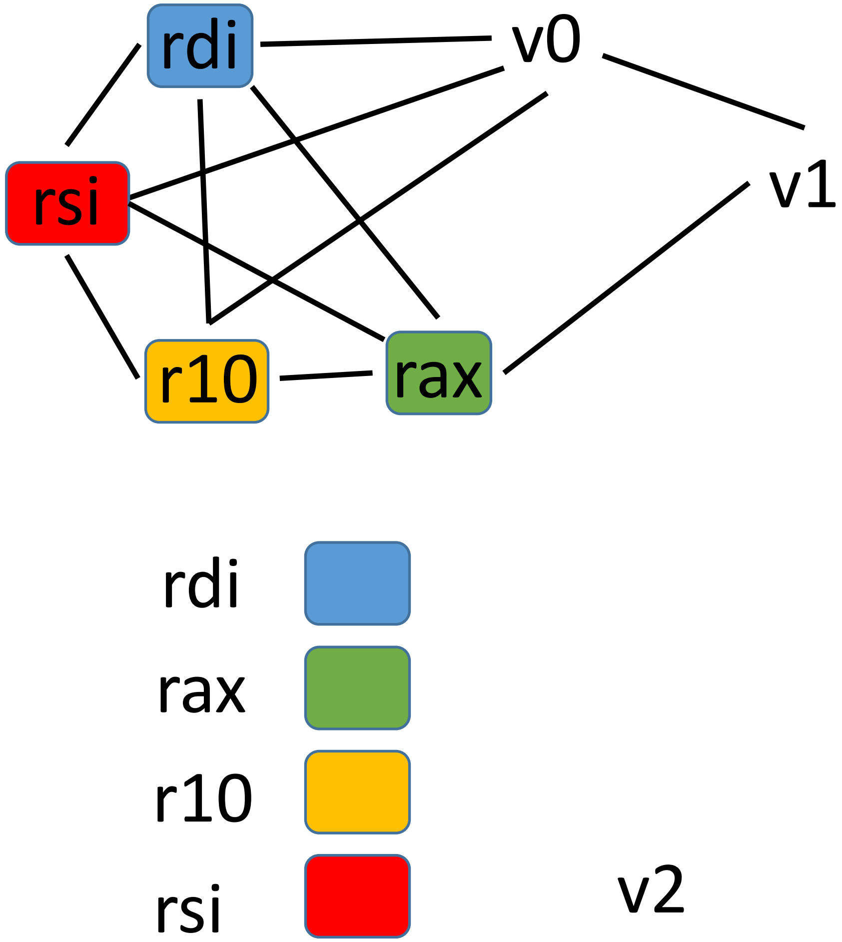
```
(@myF 3
  %v0 <- rdi
  %v0 += rdi
  %v0 += rsi
  %v0 += r10
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)
```




```
@myf(%p0, %p1, %p2){
   return (%p0 *2 + %p1 + %p2) * 3
}
```

We just need 1 register

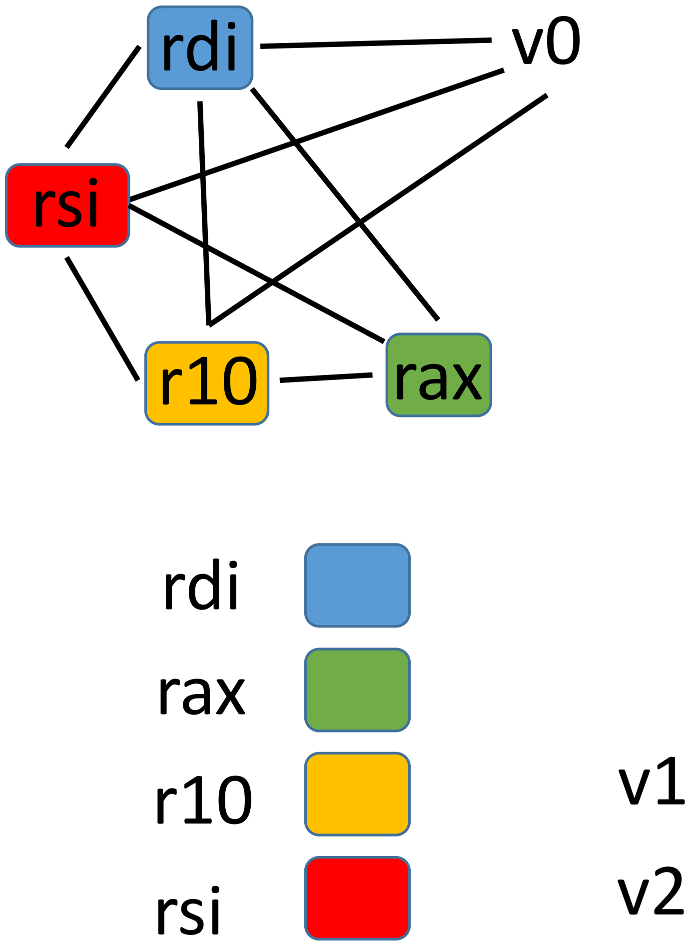
```
(@myF 3
  %v0 <- rdi
  %v0 += rdi
  %v0 += rsi
  %v0 += r10
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)
```




```
@myf(%p0, %p1, %p2){
   return (%p0 *2 + %p1 + %p2) * 3
}
```

We just need 1 register

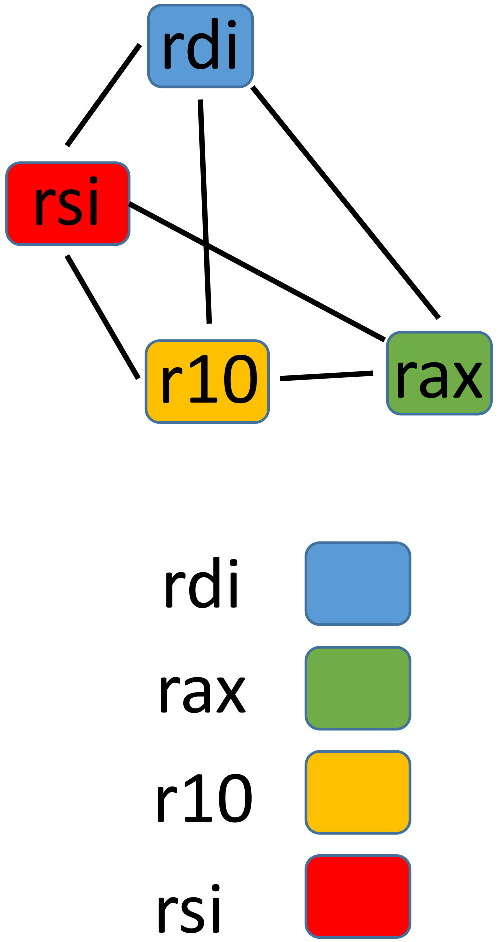
```
(@myF 3
  %v0 <- rdi
  %v0 += rdi
  %v0 += rsi
  %v0 += r10
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)
```




```
@myf(%p0, %p1, %p2){
   return (%p0 *2 + %p1 + %p2) * 3
}
```

We just need 1 register

```
(@myF 3
  %v0 <- rdi
  %v0 += rdi
  %v0 += rsi
  %v0 += r10
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)
```



v0

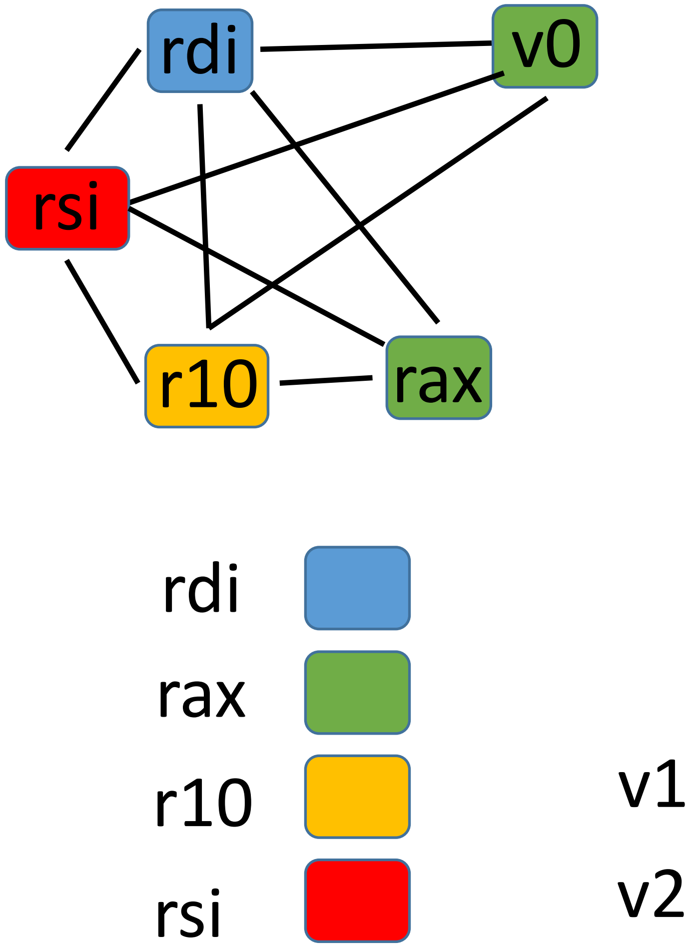
v1

v2

```
@myf(%p0, %p1, %p2){
   return (%p0 *2 + %p1 + %p2) * 3
}
```

We just need 1 register

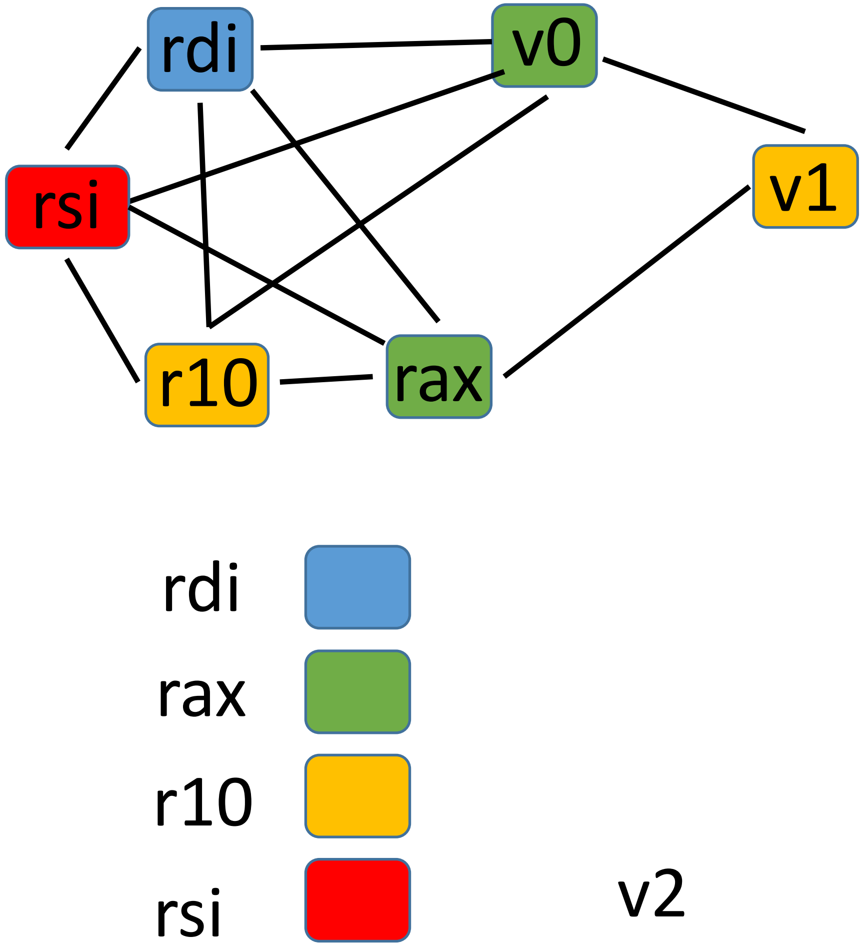
```
(@myF 3
  %v0 <- rdi
  %v0 += rdi
  %v0 += rsi
  %v0 += r10
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)
```




```
@myf(%p0, %p1, %p2){
   return (%p0 *2 + %p1 + %p2) * 3
}
```

We just need 1 register

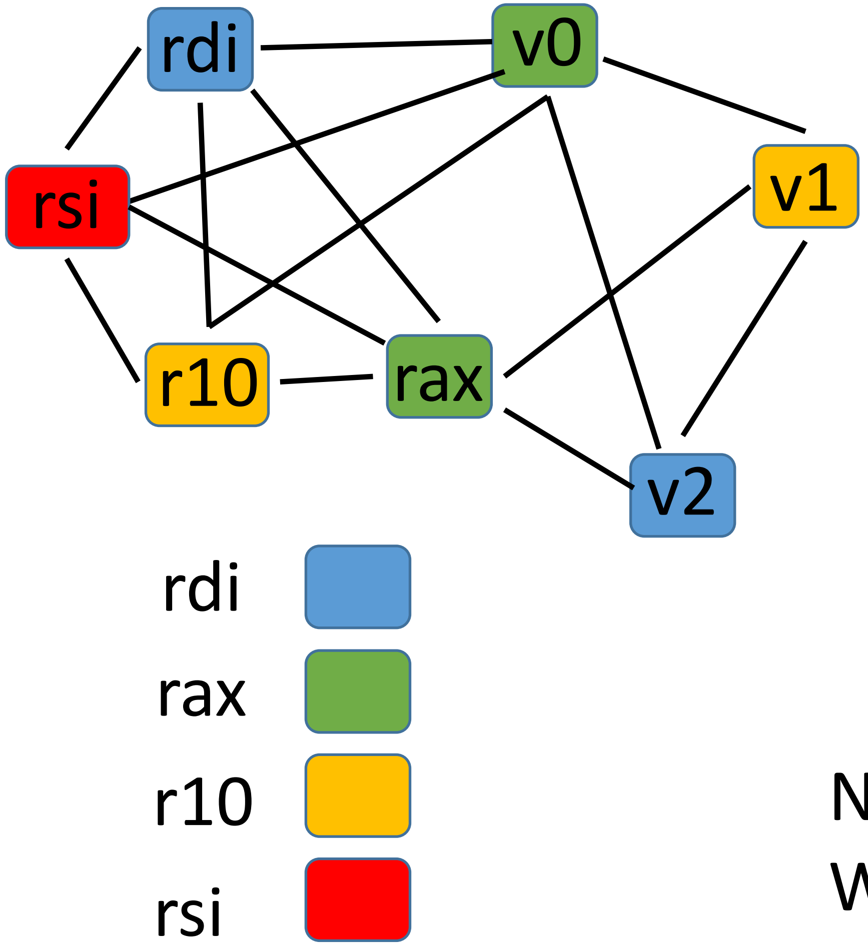
```
(@myF 3
  %v0 <- rdi
  %v0 += rdi
  %v0 += rsi
  %v0 += r10
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)
```




```
@myf(%p0, %p1, %p2){
    return (%p0 *2 + %p1 + %p2) * 3
}
```

We just need 1 register

```
(@myF 3
  %v0 <- rdi
  %v0 += rdi
  %v0 += rsi
  %v0 += r10
  %v1 <- %v0
  %v2 <- %v0
  rax <- %v0
  rax += %v1
  rax += %v2
  return
)
```




```
@myf(%p0, %p1, %p2){
    return (%p0 *2 + %p1 + %p2) * 3
}
```

We just need 1 register

No spilling necessary ☺

We need 3 registers ☹

Outline

- Graph coloring
- Heuristics
- L2c

Heuristics

- You need to decide the heuristics to use

- Next slides describe simple heuristics you can implement

  (but you don’t have to. You can implement your own heuristics as long as you implement the coloring algorithm)

- We will see more advanced heuristics later
  - You don’t have to implement them to complete your homework
  - But if you do:
    your L2 compiler will generate more performant code
  - At the end of this class: all final compilers will compete

A coloring algorithm

Algorithm:

1. Repeatedly select a node and remove it from the graph, putting it on top of a stack

2. When the graph is empty, rebuild it
   - Select a color on each node as it comes back into the graph, making sure no adjacent nodes have the same color
   - If there are not enough colors, the algorithm fails
     - Spilling comes in here
     - Select the nodes you want to spill

Heuristic: select the nodes to remove

Observation:

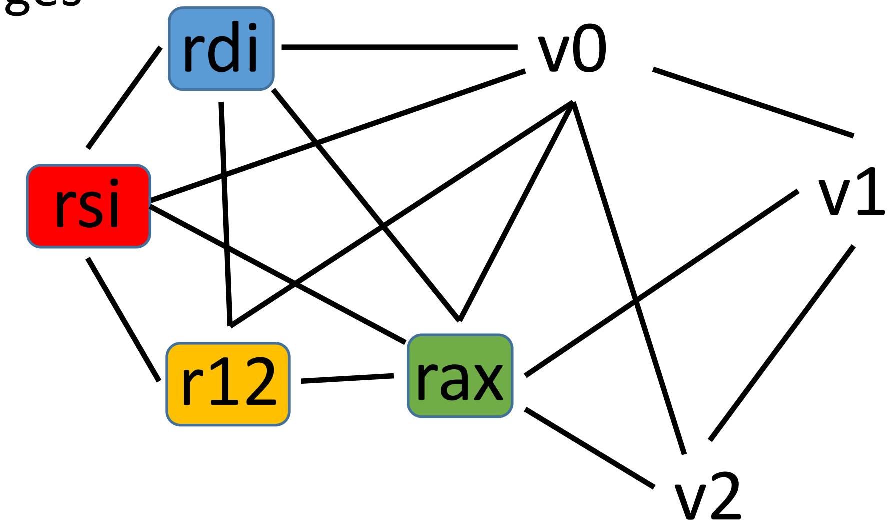
- Suppose G contains a node m with < K adjacent nodes
- Let G' be the graph G without m
- If G' can be colored with K colors, then so can G

Heuristic: *You can create your own heuristic*

- Remove all nodes with #edges < #colors (15 in L1), starting with the one with most edges and recalculating #edges after each removal
- After all nodes with < 15 edges are removed, remove the remaining ones starting from the one with the highest number of edges

Let us assume we have only 4 registers. Hence, the heuristics is

- Remove all nodes with #edges < 4,
  starting with the one with most edges and
  recalculating #edges after each removal
- After all nodes with < 4 edges are removed,
  remove the remaining ones starting from the one
  with the highest number of edges



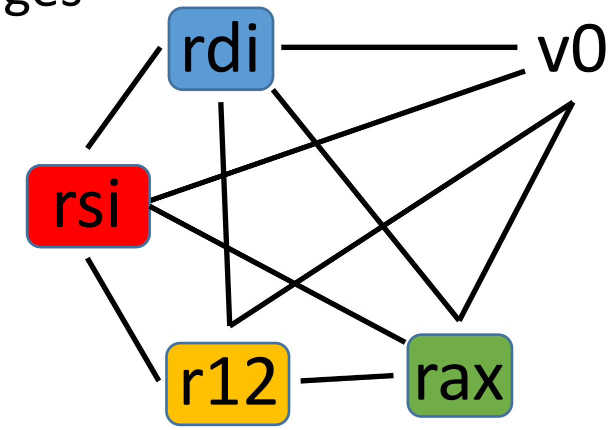
Node	Degree
v0	6
v1	3
v2	3

v1

v2

Let us assume we have only 4 registers. Hence, the heuristics is

- Remove all nodes with #edges < 4,
  starting with the one with most edges and
  recalculating #edges after each removal
- After all nodes with < 4 edges are removed,
  remove the remaining ones starting from the one
  with the highest number of edges



Node	Degree
v0	4

v0

v1

v2

A coloring algorithm

Algorithm:

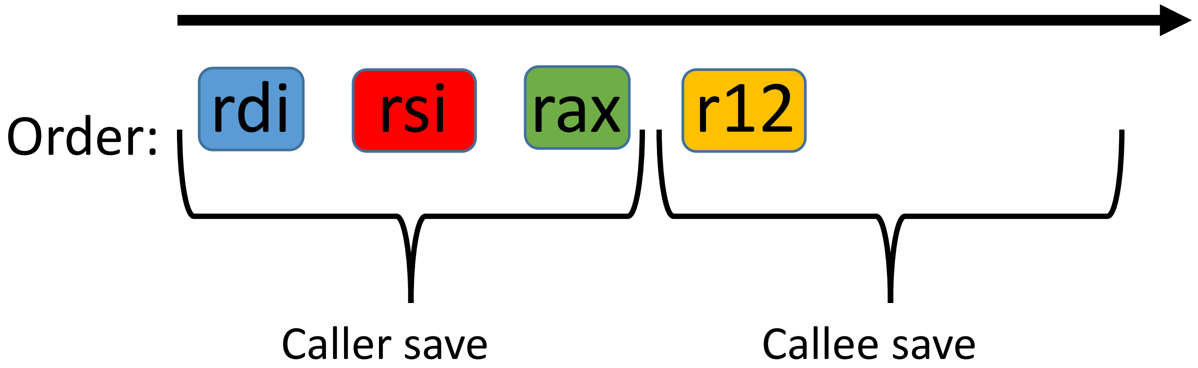
1. Repeatedly select a node and remove it from the graph, putting it on top of a stack

2. When the graph is empty, rebuild it
   - Select a color on each node as it comes back into the graph, making sure no adjacent nodes have the same color
   - If there are not enough colors, the algorithm fails
     - Spilling comes in here
     - Select the nodes you want to spill

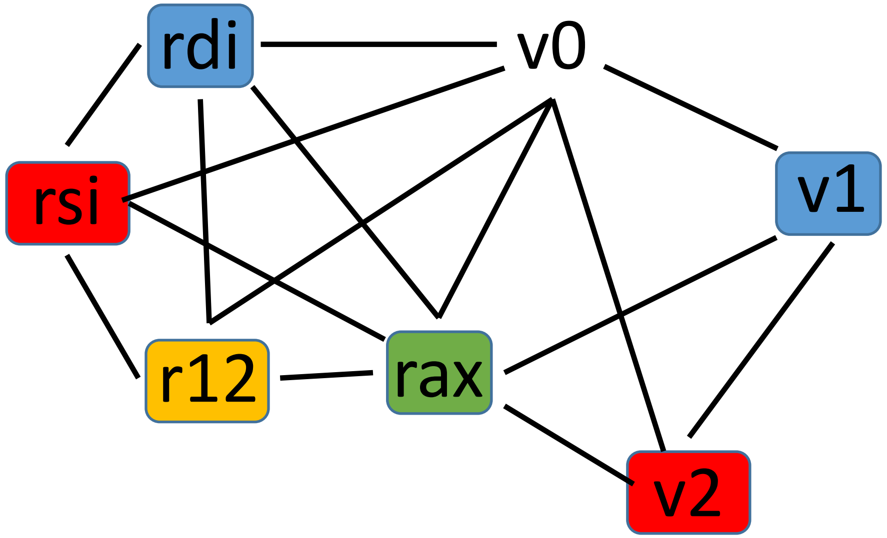
Heuristic: select the color to use

Heuristic:

- Sort the colors at design time starting from caller save registers
- Use the lowest free color



No color is available!



Caller save	Callee save
rdi	r12
rsi	
rax	

v0

v1

v2

A coloring algorithm

Algorithm:

1. Repeatedly select a node and remove it from the graph, putting it on top of a stack

2. When the graph is empty, rebuild it
   - Select a color on each node as it comes back into the graph, making sure no adjacent nodes have the same color
   - If there are not enough colors, the algorithm fails
     - Spilling comes in here
     - Select the nodes you want to spill

Heuristic: select the variables to spill

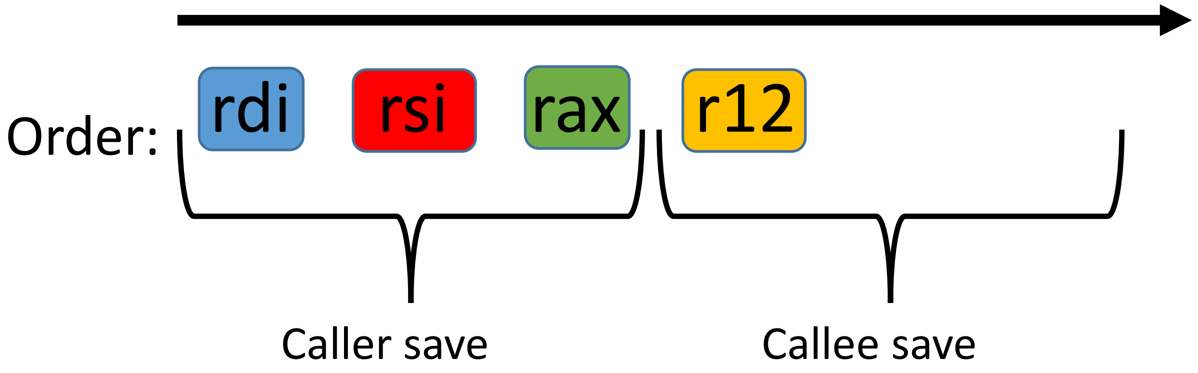
Constraint:
Never spill a variable created by a previous spill (to avoid infinite spilling)

Observation:
Every time you spill:

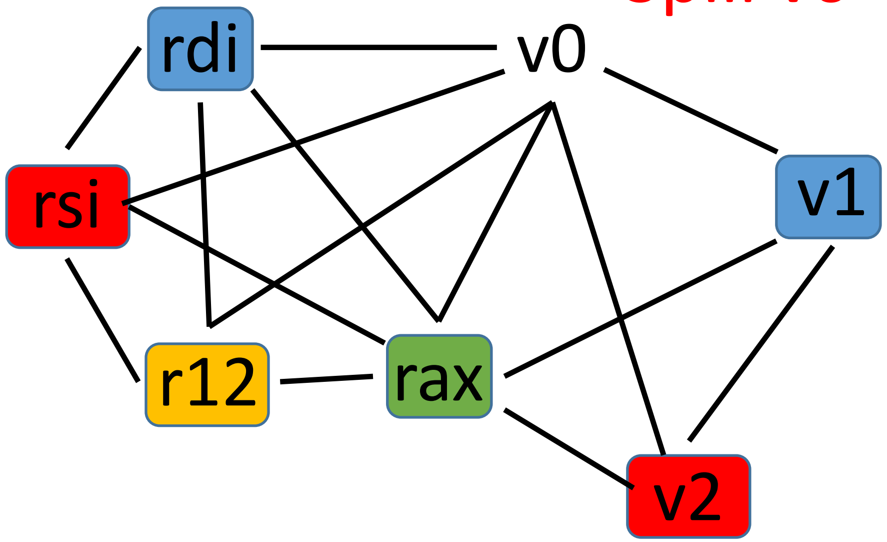
- Liveness analysis
- Interference graph
- Graph coloring

Heuristic: *You can create your own heuristic (e.g., spill only one variable at a time)*

- Add all nodes to the graph at step 2 of the algorithm
- Mark all nodes that represent variables that have no color
- Spill all variables represented by these marked nodes



Spill v0

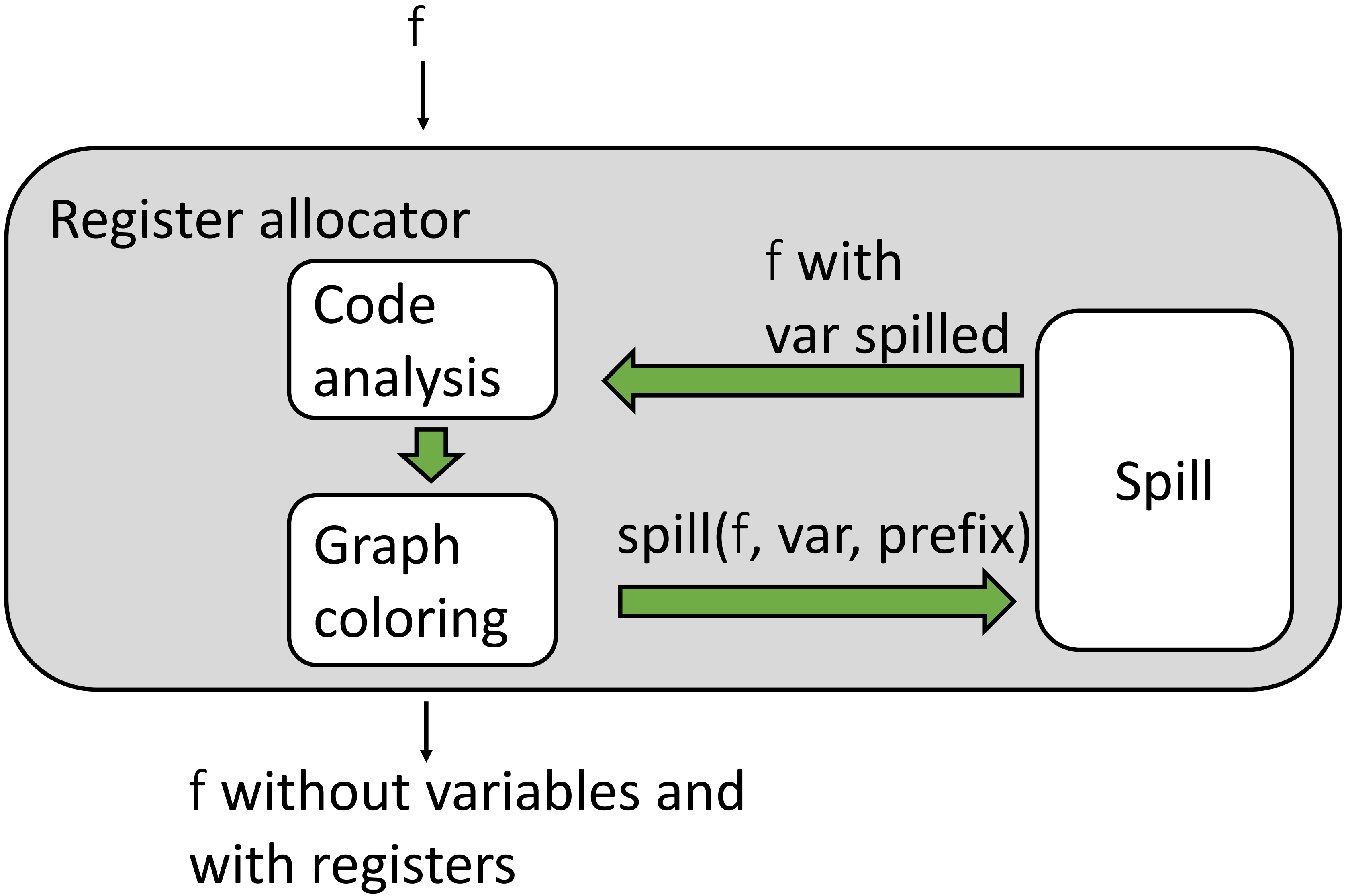


Caller save	Callee save
rdi	r12
rsi	
rax	

v0

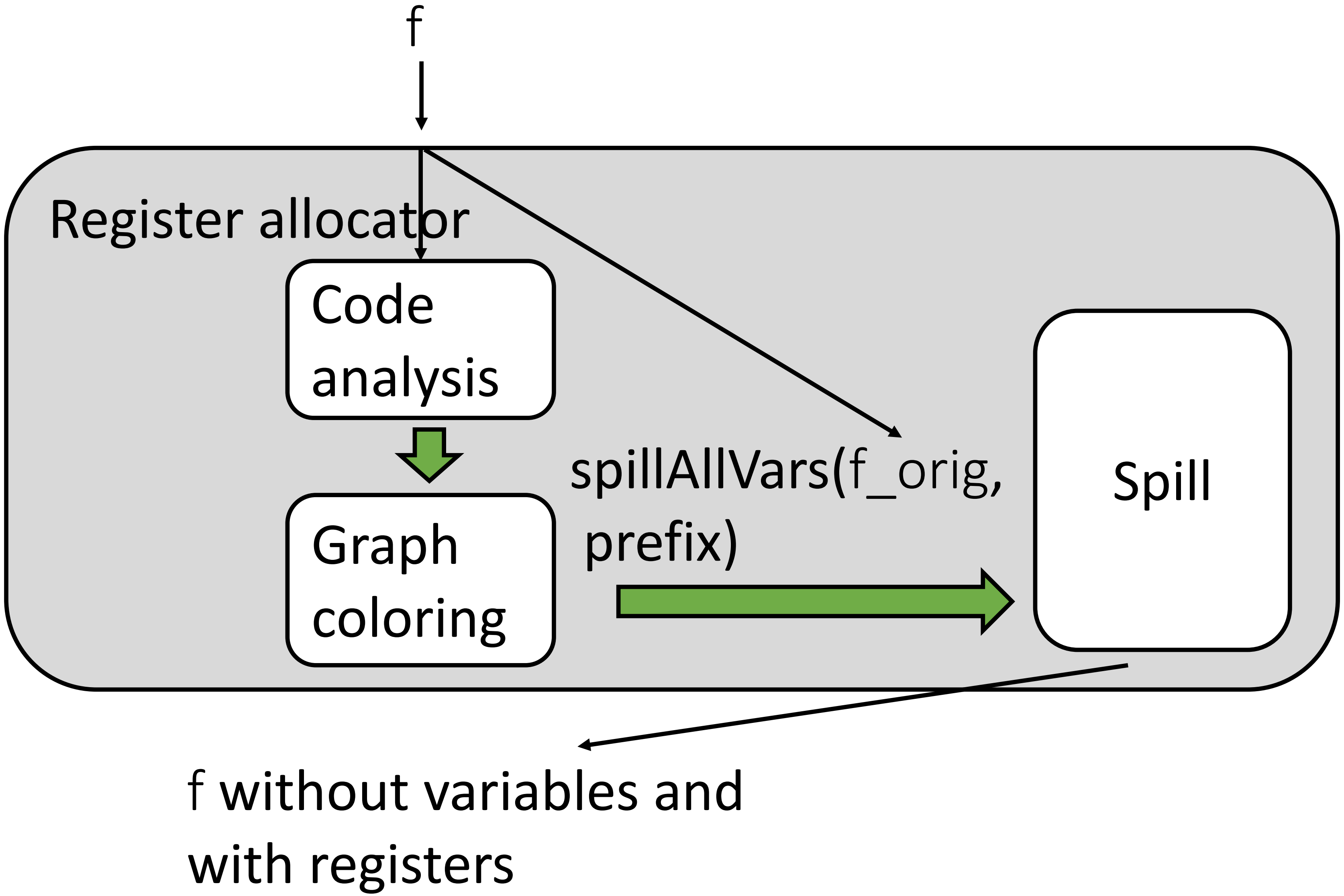
v1

v2


f
Register allocator
Code analysis
Graph coloring
f with var spilled
spill(f, var, prefix)
Spill
f without variables and with registers

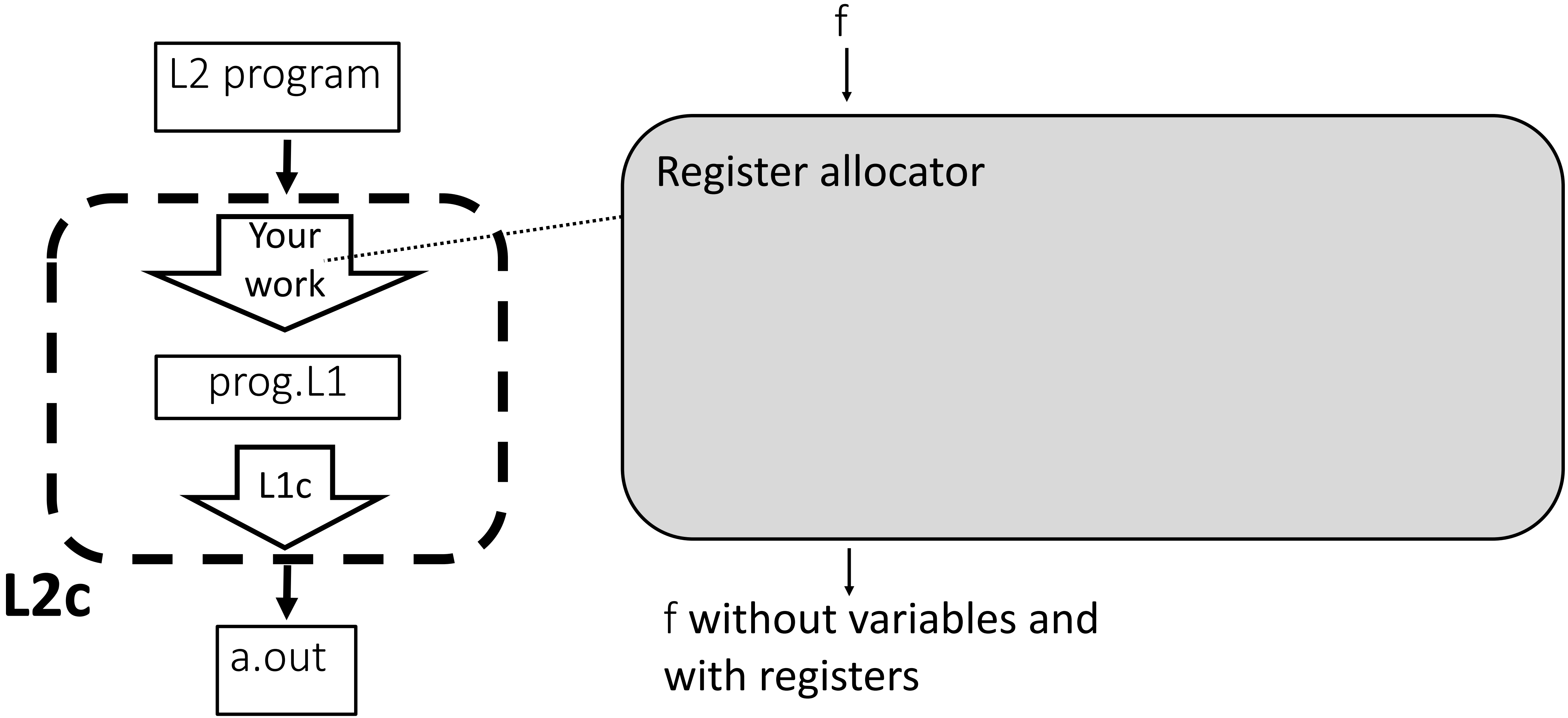
It can happen (it's rare) that the graph coloring:

- Cannot color all variables
- Cannot spill any variable



Outline

- Graph coloring
- Heuristics
- L2c


L2 program
Your work
prog.L1
L1c
L2c
a.out
f
Register allocator
f without variables and with registers

L2c

- Generating assembly from an L2 program

  cd L2 ; ./L2c tests/test25.L2

- L2c steps (this is useful to know to debug your work):
  1) Generate an L1 program from an L2 one

     L2/bin/L2 is invoked to generate L2/prog.L1
     (the name of the output file of your L2 compiler has to always be prog.L1)
  2) Generate assembly code from the generated L1 program

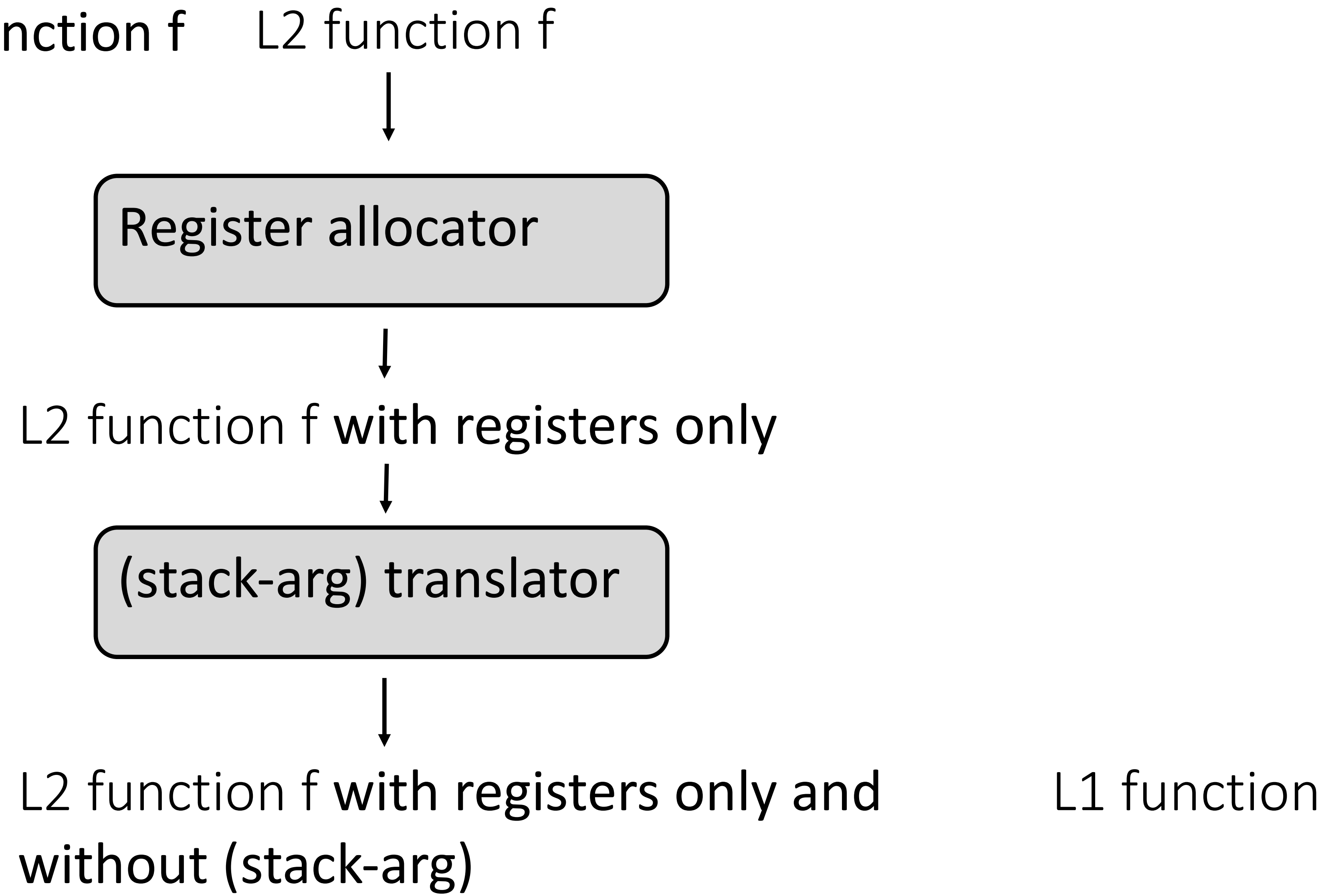
     L1/bin/L1 compiler is invoked to translate L2/prog.L1

     The output is L1/prog.S
  3) The GNU assembler and linker are invoked to generate the binary

     The standalone binary generated is L2/a.out

Homework #2: the L2 compiler

For every L2 function f



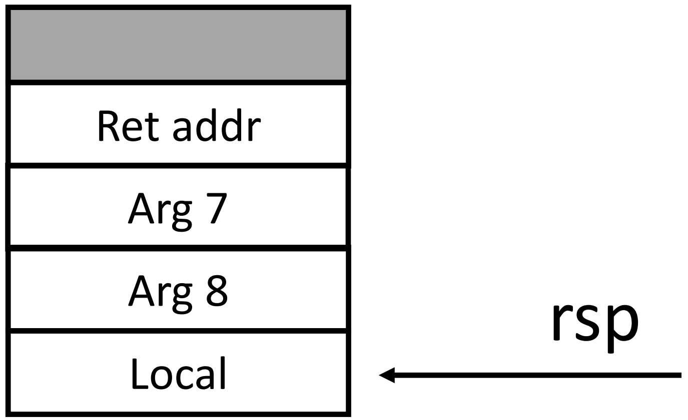
The new L2 instruction

- It accesses stack-based arguments

  w <- stack-arg M

- It is equivalent to
  w <- mem rsp ?
  where ? is M plus the number of bytes of the stack space used for local variables
- stack-arg 0 is always
  the last stack argument
- stack-arg 8 is always
  the second to last argument

```
(@myF
  8 1
  r10 <- stack-arg 0
  r10 += 2
  rdi <- r10
  call print 1
  return
)
```



Compiling your L2 compiler

- Build your L1 compiler:
  - Keep your L1 compiler sources in L1/src
  - Compile your L1 compiler:
    cd L1 ; make -j

- Build your L2 compiler:
  - Build your homework #2 under L2/src
  - Write new code to complete the translation from L2 to L1 in L2/src
  - Compile your L2 compiler:
    cd L2 ; make -j

```
bin
C
IR
L1
L2
L3
LA
LB
LC
LD
lib
Makefile
scripts
```

Testing your full L2 compiler for homework #2

- Under L2/tests there are the L2 programs to translate

- To test:
  - To check all tests: cd L2; make test
  - To check one test: ./L2c tests/test25.L2

- The output of a binary your compiler generates are in L2/tests
  - For example,
    the output of L2/tests/test25.L2f
    is L2/tests/test25.L2.out

Tips about debugging your L2 compiler

- Keep two frameworks (downloaded from Canvas) around at all time
  - Framework 1: this is where you keep **your** source code and **your** compilers
  - Framework 2: this is the framework left completely untouched.
    - Hence, **our** compilers are here
    - Never run “make clean” on this framework (it will delete **our** compilers)
- Debugging your work
  - First check if the problem is your L2 compiler
    - Manually inspect L2/prog.L1
      to check if the semantics of the translated L2 program matches L2/prog.L1
    - If the problem is your L2 compiler (the semantics don’t match),
      then debug just your L2 source code (L2/src/*)
  - If you think your L2 compiler is correct, then
    debug your L1 compiler (next slide)

Tips about debugging your L1 compiler

- Double check whether the problem is actually your L1 compiler:
  - Go to Framework2 where L1/bin/L1 is **our** L1 compiler
  - Invoke **our** L1 compiler (disabling our optimizations)
    to translate the L1 program generated by **your** L2 compiler
    cd L1 ; ./L1c –O0 PATH_Framework1/L2/prog.L1

    (where PATH_Framework1 is where you have Framework1)
  - Run the binary generated by **our** L1 compiler and check its output
    - ./a.out &> tempOutput.txt ; vimdiff tempOutput.txt ../L2/tests/test25.L2.out ;
    - Notice that you are still inside Framework2
- If the output matches the oracle one, then you know the problem is your L1 compiler
  - Check the output of your L1 compiler (PATH_Framework1/L1/prog.S) and compare it with the output of our L1 compiler
  - vimdiff PATH_Framework1/L1/prog.S PATH_Framework2/L1/prog.S

Final notes about debugging your L2 compiler

- Comparing the output of our L2 compiler with yours could be misleading
- Our L2 compiler implements a slightly more elaborate heuristics (see Advanced_graph_coloring.pdf) than the ones described in these slides
- But if you are curious, run our compiler with -v option
  ./L2c –v tests/test0.L2

Always have faith in your ability

Success will come your way eventually

Best of luck!