Liveness analysis

Simone Campanoni
simone.campanoni@northwestern.edu
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
A register allocator structure

Register allocator

Map heuristic

f without varX

Current f, varX

Spill

f without variables

f
A graph-coloring register allocator structure

Register allocator
  Code analysis
  Graph coloring

Spill
  \texttt{spill(f, var, prefix)}

\texttt{f with var spilled}

\texttt{f without variables}

\texttt{f}
Task: From Variables to Registers

We built the interference graph. To compute it automatically, we need the liveness analysis.

```plaintext
(@MyVeryImportantFunction 0
	%MyVar1 <- 2
	%MyVar2 <- 40
	%MyVar3 <- %MyVar1
	%MyVar3 += %MyVar2
	print %MyVar3
)
```

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

We need the liveness analysis to map variables to registers.
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
  %myVar1 <- 2
  %myVar2 <- 40
  %myVar3 <- %myVar1
  %myVar3 += %myVar2
  rax <- %myVar3
  return
)
```

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

<table>
<thead>
<tr>
<th>IN (just before)</th>
<th>OUT (just after)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>IN[0]</td>
</tr>
<tr>
<td>{myVar1}</td>
<td>OUT[0], IN[1]</td>
</tr>
<tr>
<td>{myVar1, myVar2}</td>
<td>OUT[1], IN[2]</td>
</tr>
<tr>
<td>{myVar3, myVar2}</td>
<td>OUT[2], IN[3]</td>
</tr>
<tr>
<td>{myVar3}</td>
<td>OUT[3], IN[4]</td>
</tr>
</tbody>
</table>

Live ranges

Interference graph

Graph coloring

@myF w/o variables
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
  • \texttt{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

• \texttt{IN[i]} = \{r10\}
Execution path

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

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

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

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem } x \text{ M} \mid \text{mem } x \text{ M} \leftarrow s \mid w \leftarrow \text{stack-arg } M \mid \]
\[ w \text{ aop } t \mid w \text{ sop } sx \mid w \text{ sop } N \mid \text{mem } x \text{ M} + = t \mid \text{mem } x \text{ M} - = t \mid w + = \text{mem } x \text{ M} \mid w - = \text{mem } x \text{ M} \mid \]
\[ w \leftarrow t \text{ cmp } t \mid \text{cj} \text{jump } t \text{ cmp } t \text{ label} \mid \text{label} \mid \text{goto } \text{label} \mid \]
\[ \text{return} \mid \text{call } u N \mid \text{call } \text{print } 1 \mid \text{call } \text{input } 0 \mid \text{call } \text{allocate } 2 \mid \text{call } \text{tensor-error } F \mid \]
\[ w++ \mid w-- \mid w @ w w E \]

An instruction \( i \) that has only one successor \( s \) and \( s \) is the instruction stored just after \( i \)

\[ \text{rdi }\leftarrow 5 \]
\[ \text{call print 1} \]
\[ \text{r10 }\leftarrow \text{rax } < 5 \]
Successors of an instruction (2)

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem x M} \mid \text{mem x M} \leftarrow s \mid w \leftarrow \text{stack-arg M} \mid \]
\[ w \text{ aop t} \mid w \text{ sop sx} \mid w \text{ sop N} \mid \text{mem x M} \leftarrow t \mid \text{mem x M} \leftarrow t \mid w \leftarrow t \mid w \leftarrow \text{mem x M} \mid w \leftarrow \text{mem x M} \]
\[ w \leftarrow t \text{ cmp t} \mid \text{cjump t cmp t label} \mid \text{label} \mid \text{goto label} \mid \]
\[ \text{return} \mid \text{call u N} \mid \text{call print 1} \mid \text{call input 0} \mid \text{call allocate 2} \mid \text{call tensor-error F} \mid \]
\[ w++ \mid w-- \mid w @ w w E \]

An instruction \( i \) that has only one successor \( s \) but \( s \) is not necessarily the instruction stored just after \( i \)
\[ \text{goto :MY_LABEL_0} \]
\[ :MY_LABEL_0 \]
Successors of an instruction (3)

\[ i \quad ::= \quad w \leftarrow s \mid w \leftarrow \text{mem} \times M \mid \text{mem} \times M \leftarrow s \mid w \leftarrow \text{stack-arg} \ M \mid \]

\[ \quad w \ \text{aop} \ t \mid w \ \text{sop} \ sx \mid w \ \text{sop} \ N \mid \text{mem} \times M \ += \ t \mid \text{mem} \times M \ -= \ t \mid w \ += \ \text{mem} \times M \mid w \ -= \ \text{mem} \times M \mid \]

\[ w \leftarrow t \ \text{cmp} \ t \mid \text{cjump} \ t \ \text{cmp} \ t \ \text{label} \mid \text{label} \mid \text{goto} \ \text{label} \mid \]

\[ \quad \text{return} \mid \text{call} \ u \ N \mid \text{call} \ \text{print} \ 1 \mid \text{call} \ \text{input} \ 0 \mid \text{call} \ \text{allocate} \ 2 \mid \text{call} \ \text{tensor-error} \ F \mid \]

\[ w++ \mid w-- \mid w \ @ \ w \ w \ E \]

An instruction \( i \) that has no successor
Successors of an instruction (4)

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem} \times M \mid \text{mem} \times M \leftarrow s \mid w \leftarrow \text{stack-arg} \times M \mid \]
\[ w \text{ aop} t \mid w \text{ sop} sx \mid w \text{ sop} N \mid \text{mem} \times M \leftarrow t \mid \text{mem} \times M \leftarrow t \mid w \leftarrow \text{mem} \times M \mid w \leftarrow \text{mem} \times M \mid \]
\[ w \leftarrow t \text{ cmp} t \mid \text{cjump} t \text{ cmp} t \text{ label} \mid \text{label} \mid \text{goto} \text{ label} \mid \]
\[ \text{return} \mid \text{call} u \times N \mid \text{call} \text{ print} 1 \mid \text{call} \text{ input} 0 \mid \text{call} \text{ allocate} 2 \mid \text{call} \text{ tensor-error} F \mid \]
\[ w++ \mid w-- \mid w @ w w E \]

An instruction \( i \) that has two successors

\[
\text{cjump} \text{ rax} < 5 :L1 \\
\text{rdi} \leftarrow 1 \\
\text{rsi} \leftarrow 3 \\
:L1
\]
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 defined 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

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

```
rdi++
KILL[i] = \{rdi\}
GEN[i] = \{rdi\}
```
GEN and KILL sets: more examples

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

\[ \text{cjump } \text{rdi } \leq \%v2 : \text{true} \]
\[ \text{KILL}[i] = \{ \} \]
\[ \text{GEN}[i] = \{ \text{rdi, } \%v2 \} \]
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}
  \[ \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \]

• OUT[i] = {all variables live right after instruction i}
  \[ \text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s] \]

- If OUT[i] = {} then
  \[ \text{IN}[i] = \{\%v1\} \]

- If OUT[i] = {\%v2} then
  \[ \text{IN}[i] = \{\%v1\} \]

- If IN[i+1] = {\%v1}, then
  \[ \text{OUT}[i] = \{\%v1\} \]

- If IN[i+1] = {\%v1}, IN[i+j] = {\%v2}, then
  \[ \text{OUT}[i] = \{\%v1, \%v2\} \]

- i+1: \%v2 \leftarrow \%v1 \\
  \text{cjump } \%v = 1 :s2
Algorithm

for (each instruction $i$) {
    \text{GEN}[$i$] = ...
    \text{KILL}[$i$] = ...
}
for (each instruction $i$) \text{IN}[$i$] = \text{OUT}[$i$] = { };
do{
    for (each instruction $i$){
        \text{IN}[$i$] = \text{GEN}[$i$] \cup (\text{OUT}[$i$] – \text{KILL}[$i$])
        \text{OUT}[$i$] = \text{U}_s \text{ a successor of } i \ \text{IN}[$s$]
    }
} while (changes to any IN or OUT occur);
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
## Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u,</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td></td>
<td>args used}</td>
<td></td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td>{ args used}</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td>return</td>
<td>{ rax,</td>
<td>{ }</td>
</tr>
<tr>
<td></td>
<td>callee save registers}</td>
<td></td>
</tr>
</tbody>
</table>

*The reason why call and return instructions must be treated with the above special rules will be explained at the next lecture*
Let’s run an example to show the computation of the liveness analysis
Code example

@@myF
0
%a <- 2  // 1
rax <- %a // 2
return  // 3
)
## Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u, args used}</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td>{ args used}</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{}</td>
</tr>
</tbody>
</table>
# Registers

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Result</th>
<th>Caller save</th>
<th>Callee save</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdi</td>
<td>rax</td>
<td>r10</td>
<td>r12</td>
</tr>
<tr>
<td>rsi</td>
<td></td>
<td>r11</td>
<td>r13</td>
</tr>
<tr>
<td>rdx</td>
<td></td>
<td>r8</td>
<td>r14</td>
</tr>
<tr>
<td>rcx</td>
<td></td>
<td>r9</td>
<td>r15</td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td>rax</td>
<td>rbp</td>
</tr>
<tr>
<td>r9</td>
<td></td>
<td>rcx</td>
<td>rbx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdi</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rdx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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[$i$] U (OUT[$i$] – KILL[$i$])
        OUT[$i$] = U $s$ a successor of $i$ IN[$s$]
    }
} while (changes to any IN or OUT occur);
```
### Code example

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

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>%a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rax</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

for (each instruction $i$) {
    \begin{align*}
    \text{GEN}[i] &= \ldots \\
    \text{KILL}[i] &= \ldots
    \end{align*}
}

for (each instruction $i$) \text{IN}[i] = \text{OUT}[i] = \{ \} ;

do{
    \begin{align*}
    \text{for (each instruction $i$)}\{
        \text{IN}[i] &= \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
        \text{OUT}[i] &= \bigcup_{s \text{ a successor of } i} \text{IN}[s]
    \}
    \} \text{ while (changes to any IN or OUT occur)};

### Code example

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

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ %a }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{ %a }</td>
<td>{ rax }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{ rax, r12}</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
</tbody>
</table>
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[$i$] $\cup$ (OUT[$i$] $-$ KILL[$i$])
        OUT[$i$] = $\cup_{s \text{ a successor of }i} \text{ IN}[s]$
    }
} while (changes to any IN or OUT occur);
## Code example

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

### GEN, KILL, IN, OUT

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{ }</td>
<td>{%a}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>2</td>
<td>{%a}</td>
<td>{rax}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>3</td>
<td>{rax, r12}</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
</tbody>
</table>

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]

\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Code example

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

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{%a}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{rax, r12}</td>
<td>{ }</td>
<td>{rax, r12}</td>
<td>{ }</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{IN}[i] & = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] & = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\end{align*}
\]
Code example

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

```
<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{ }</td>
<td>{%a}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td></td>
<td>{%a}</td>
<td>{rax}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td></td>
<td>{rax, r12}</td>
<td>{ }</td>
<td>{rax, r12}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
```

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]

\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Code example

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

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ %a }</td>
<td>{ %a }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{ %a }</td>
<td>{ rax }</td>
<td>{ %a, r12 }</td>
<td>{ rax, r12 }</td>
</tr>
<tr>
<td>{ rax, r12 }</td>
<td>{ }</td>
<td>{ rax, r12 }</td>
<td>{ }</td>
</tr>
</tbody>
</table>

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]

\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
### Code example

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

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{}</td>
<td>{%a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{%a, r12}</td>
<td>{rax, r12}</td>
<td></td>
</tr>
<tr>
<td>{rax, r12}</td>
<td>{}</td>
<td>{rax, r12}</td>
<td>{}</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{}</td>
<td>{%a}</td>
<td>{r12}</td>
<td>{%a, r12}</td>
</tr>
<tr>
<td></td>
<td>{%a}</td>
<td>{rax}</td>
<td>{%a, r12}</td>
<td>{rax, r12}</td>
</tr>
<tr>
<td></td>
<td>{rax, r12}</td>
<td>{}</td>
<td>{rax, r12}</td>
<td>{}</td>
</tr>
</tbody>
</table>

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}
for (each instruction $i$)  IN[$i$] = OUT[$i$] = { };
do{
    for (each instruction $i$){
        IN[$i$]     = GEN[$i$] U (OUT[$i$] – KILL[$i$])
        OUT[$i$] = U $s$ 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
)

\[
\begin{array}{cccc}
\text{GEN} & \text{KILL} & \text{IN} & \text{OUT} \\
\{\} & \{%a\} & \{r12\} & \{%a, r12\} \\
\{%a\} & \{rax\} & \{%a, r12\} & \{rax, r12\} \\
\{rax, r12\} & \{\} & \{rax, r12\} & \{\} \\
\end{array}
\]
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[$i$] U (OUT[$i$] – KILL[$i$])
        OUT[$i$] = $\cup_s$ a successor of $i$ IN[$s$]
    }
} while (changes to any IN or OUT occur);
### Code example

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

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{%a}</td>
<td>{r12}</td>
<td>{%a, r12}</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{%a, r12}</td>
<td>{rax, r12}</td>
</tr>
<tr>
<td>{rax, r12}</td>
<td>{ }</td>
<td>{rax, r12}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
```

- 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[0] -> (r13 r15 rax r14 rbp r12 rbx)
IN[1] -> (r13 r15 rax r14 rbp %myVar1 r12 rbx)
IN[2] -> (r13 r15 rax r14 rbp %myVar2 r12 %myVar1 rbp r12)
IN[3] -> (r13 r15 rax r14 rbp r12 rbx)

OUT[0] -> (r13 r15 rax r14 rbp r12 %myVar1 r12)
OUT[1] -> (r13 r15 rax r14 rbp %myVar2 r12 %myVar1 r12)
OUT[2] -> (r13 r15 rax r14 rbp r12 rbx)
OUT[3] -> ( )

IN
OUT
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!