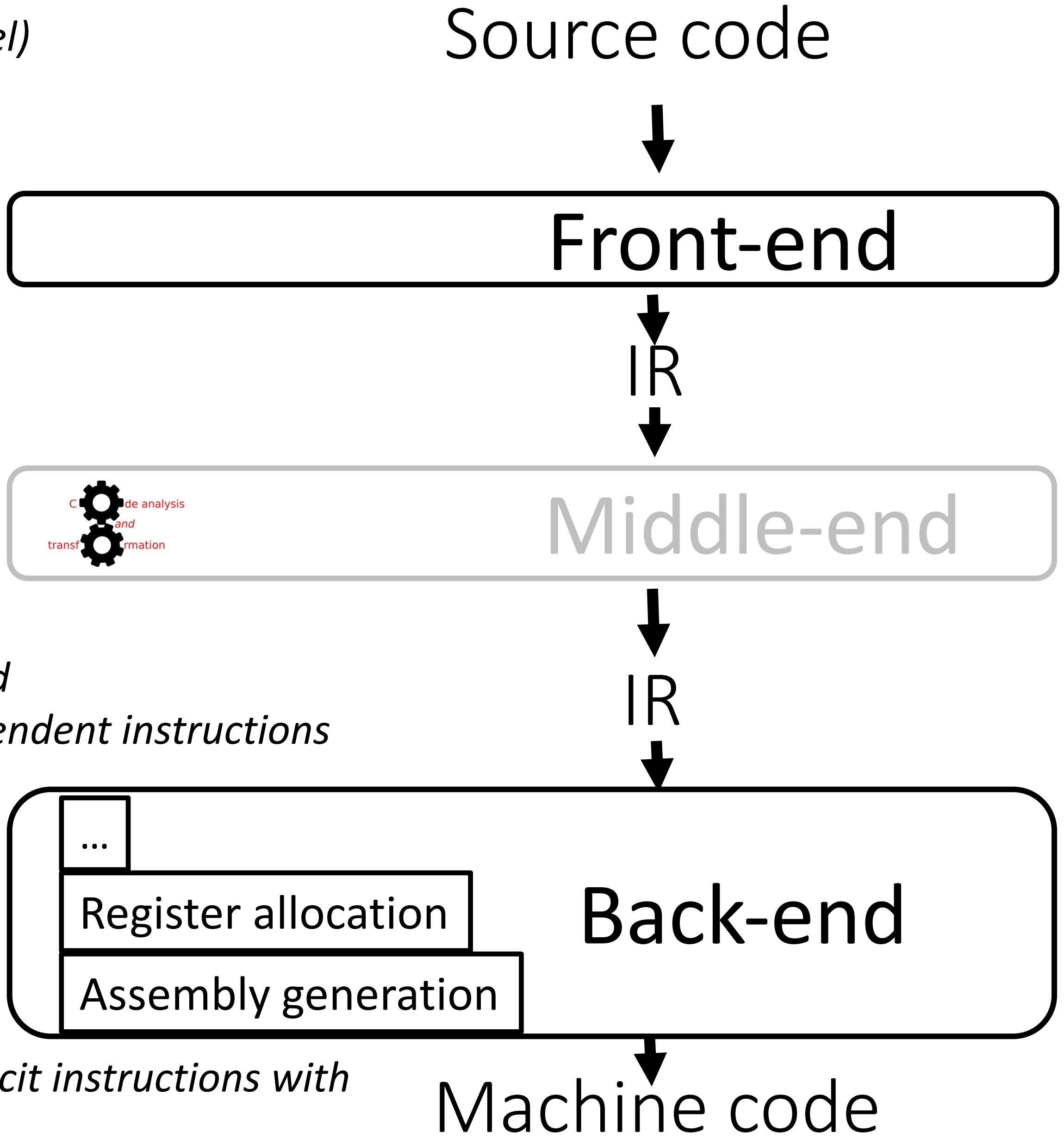
Compiler Construction

L3

Simone Campanoni
simone.campanoni@northwestern.edu

NORTHWESTERN UNIVERSITY
1851

A compiler



Source code

Front-end

IR

Code analysis *and* transformation

Middle-end

IR

…

Register allocation

Assembly generation

Back-end

Machine code

High level (algorithm level) statements

Explicit, simple, and architecture-independent instructions

Only a few registers, explicit instructions with constraints (e.g., lea)

The language needs to help humans to write (efficient and robust) code

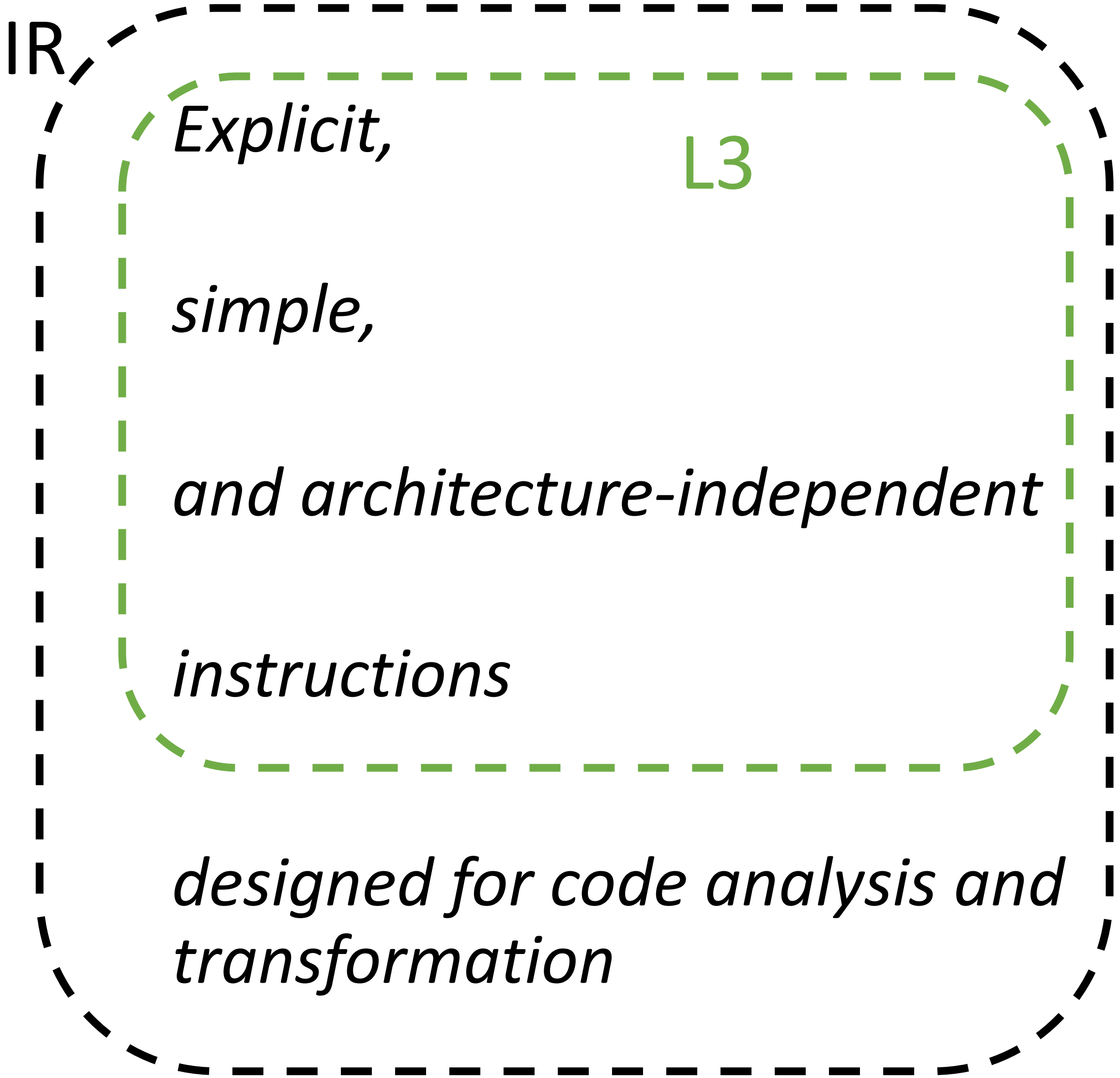
The language needs to be easy to be analyzed and transformed

The language needs to be easy to execute efficiently

Outline

- L3
- Translating L3 to L2: calling convention and labels
- Translating L3 to L2: instruction selection

From L2 to IR going through L3



Explicit,

simple,

and architecture-independent

instructions

designed for code analysis and transformation

Explicit semantic (e.g., add)

add, br, load, store (**no** lea)

no registers, no calling convention

Small piece of computation

L2 language

- Explicit entry point
- Explicit calling convention
- Complex per-instruction semantic
- Registers and variables

```
(@go (@go 0
        rdi <- 5
        mem rsp -8 <- :myF_ret
        call @myF 1
        :myF_ret
        %myRes <- rax
        %myRes @ %myRes %myRes 4
        return )
     (@myF 1
        rax <- rdi
        return
)    )
```

L3 language

- Pre-defined entry point
- Hidden calling convention
- Simple per-instruction semantic
- Variables only

```
define @main (){
   %myRes <- call @myF(5)
   %v1 <- %myRes * 4
   %myRes <- %myRes + %v1
   return
}
define @myF (%p1){
  %p2 <- %p1 + 1
  return %p2
}
```

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 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
```

L3

Explicit signature

```
p      ::= f+
f      ::= define l ( vars ) { i+ }
i      ::=  var <- s | var <- t op t | var <- t cmp t |
             var <- load var | store var <- s |
            return | return t| label | br label | br t label |
            call callee ( args ) | var <- call callee ( args )
callee ::= u | print | allocate | input | tensor-error
vars   ::= | var | var (, var)*
args   ::= | t | t (, t)*
s      ::= t | label | l
t      ::= var | N
u      ::= var | l
op     ::= + | - | * | & | << | >>
cmp    ::= < | <= | = | >= | >
N     ::= (+|-)? [0-9]+
l      ::= @name
label ::= :name
var    ::= %name
name::= sequence of chars matching [a-zA-Z_][a-zA-Z_0-9]*
```

No CISC instructions

No calling convention

The scope of labels is the function!

L3 program examples

```
define @main (){
  %myRes <- call @myF(5)
  %v1 <- %myRes * 4
  %v2 <- %myRes + %v1
  return %v2
}
define @myF (%p1){
  %l1 <- %p1 + 1
  return %l1
}
```

```
define @main (){
    %v1 <- 1
    %v2 <- 2
    %v3 <- %v1 >= %v2
    return %v3
}
```

```
define @myEqual (%p1, %p2){
  %v3 <- %p1 = %p2
  br %v3 :myLabelTrue
  return 0
  :myLabelTrue
  return 1
}
define @main (){
  %ret <- call @myEqual(3,5)
  return %ret
}
```

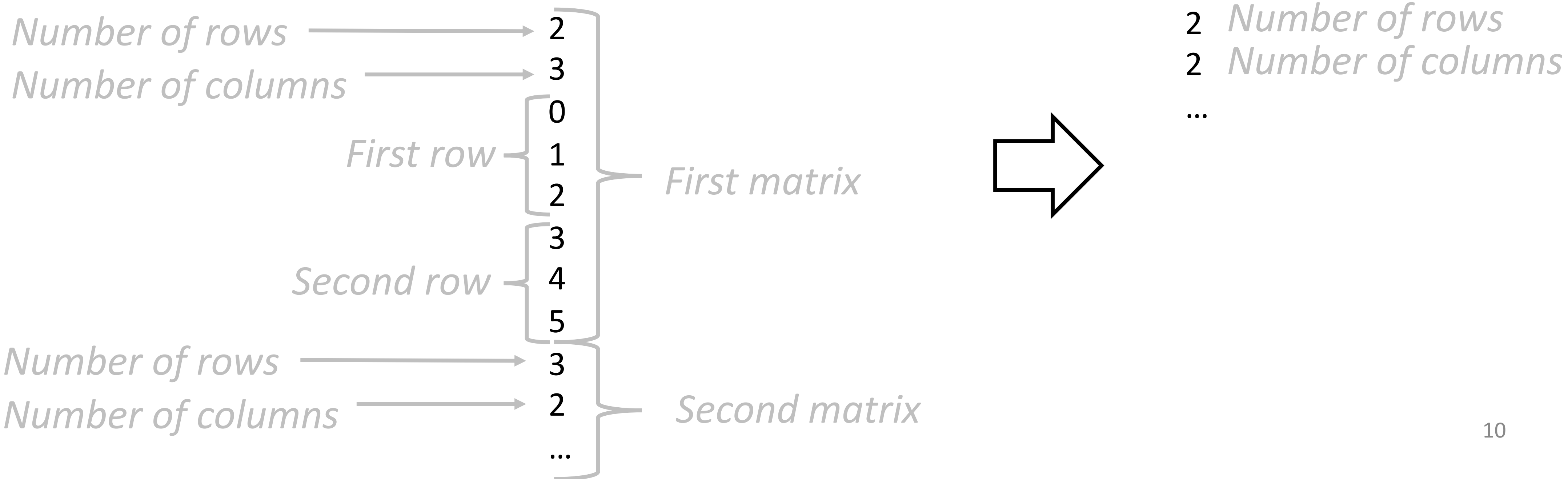
Final notes on L3

As for L2:

- Values are encoded following the same rules of L1
- Same rules for memory heap allocation
- Same undefined behaviors

Now that you know the L3 language

1. Rewrite your sorting L2 program using L3 and
2. Write a new L3 program to perform matrix multiplication (Example of input file = MM.L3.in on canvas)



Number of rows → 2
Number of columns → 3
0
First row 1
2
3
Second row 4
5
First matrix
Number of rows → 3
Number of columns → 2
…
Second matrix

2 Number of rows
2 Number of columns
…

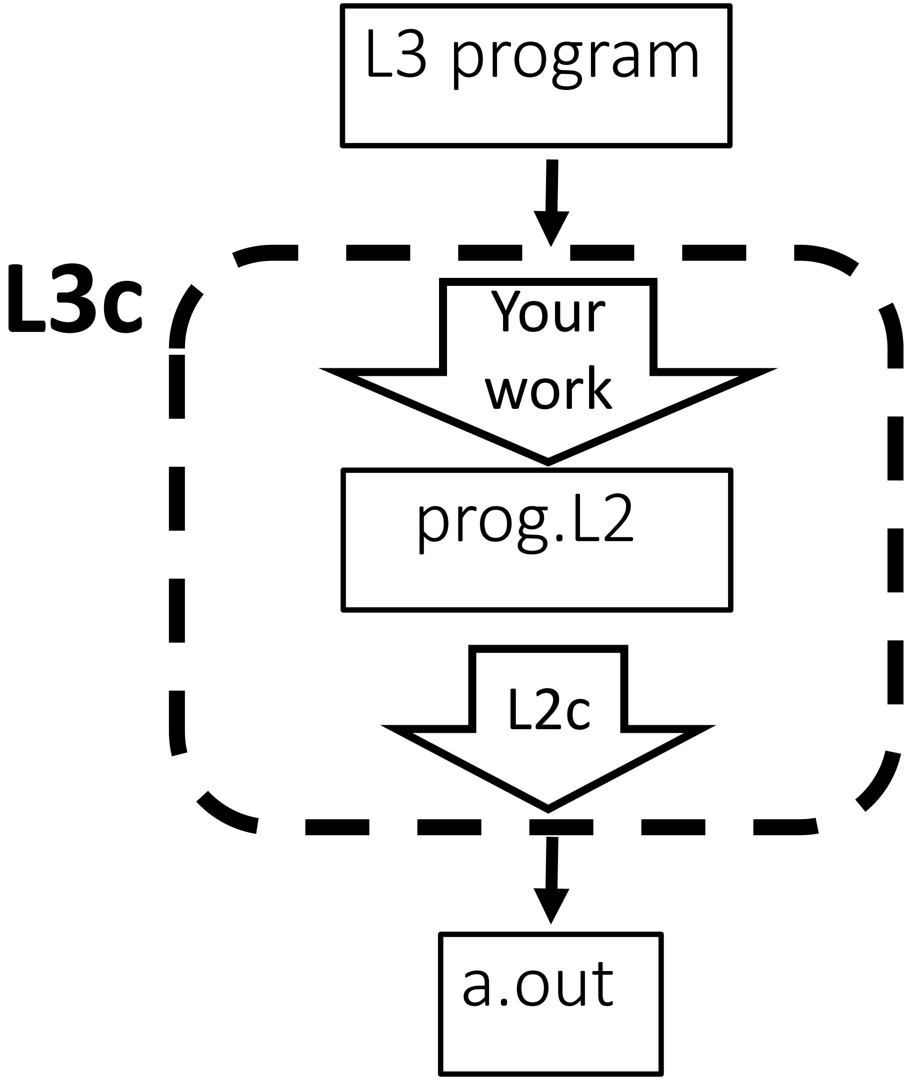
Outline

- L3

- Translating L3 to L2: calling convention and labels

- Translating L3 to L2: instruction selection

The L3 compiler (L3c)



- To build L3c:
  translate an L3 program
  to an equivalent L2

- We need to encode the calling convention
  API -> ABI

- We need to select which L2 instructions to use for the L3 ones
  Instruction selection

L3 parser

- Significantly simpler than the L2 parser
- Pay attention to the L3 grammar

```
i       ::=  ...
             call callee ( args ) | var <- call callee ( args )
callee ::= u | print | allocate | tensor-error
u       ::= var | l
args    ::= t | t (, t)*
t        ::= var | N
```

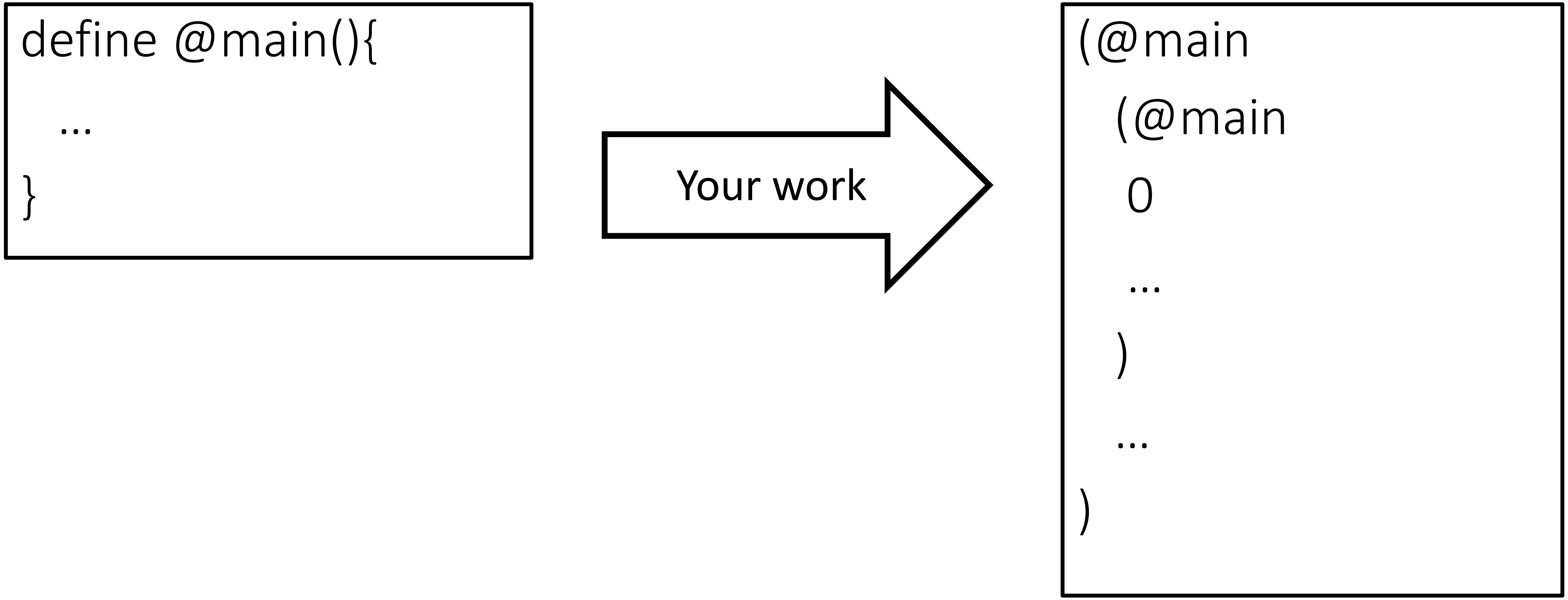
  - Same rule for all call instructions

Parsing an L3 program

```
define @main (){
  %myRes <- call @myF(5)
  call @myF(5)
  return
}
```

```
define @main (){
    %myA <- call allocate(3, 1)
    call allocate(3, 1)
   return
}
```

Entry point



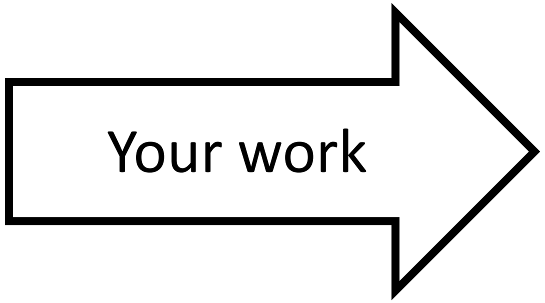
```
define @main(){
   …
}
```

Your work

```
(@main
   (@main
    0
    …
   )
   …
)
```

Making the calling convention explicit: caller

```
define @main(){
  %v1 <- call @myF(3)
  …
}
```



Your work

```
(@main
 (@main
   0
   mem rsp -8 <- :myF_ret
   rdi <- 3
   call @myF 1
   :myF_ret
   %v1 <- rax
   …
 )
)
```

Making the calling convention explicit: callee

```
define @myF (%p1){
  return %p1
}
```

Your work

```
(@myF
  1
  %p1 <- rdi
  rax <- %p1
  return
)
```

Stack arguments, registers, and variables

- L3c is responsible to allocate space on the stack for >6 arguments
- L3c can generate L2 code with registers and variables
- L2c already performs a good register allocation
- Good engineering: don’t replicate functionality
  - L3c should not perform register allocation
  - L3c should use variables always with the only exceptions of implementing the calling convention

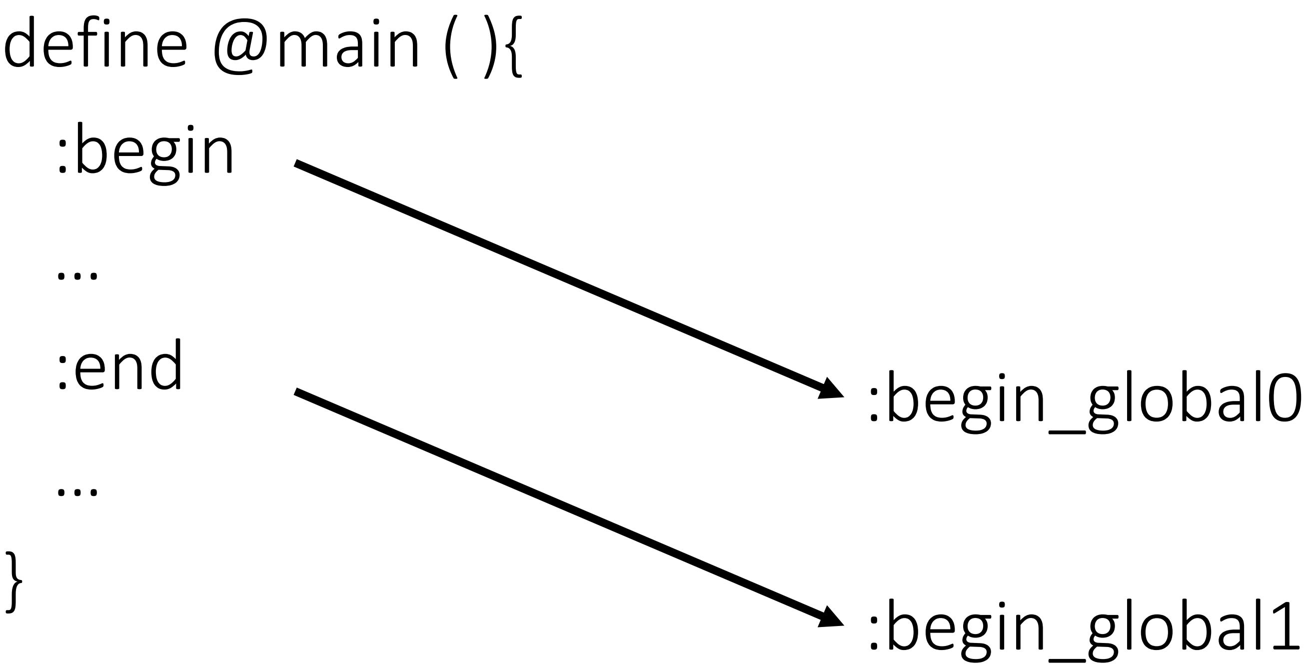
Labels

- The L3 compiler needs to translate
  L3 instruction labels to L2 instruction labels
  - No need to change function names
  - L3 labels: the scope is the function
    - 2 labels with the exact name in 2 different function are possible
  - L2 labels: the scope is the program
    - 2 labels with the exact name are not possible
- A possible mapping from L3 labels to L2 ones:
  1. Find the longest label for the whole L3 program: LL
  2. Append "_global_" to it: LLG
  3. For every L3 label :LABELNAME of a function F,
     generate an L2 label by increasing a global counter and appending
     it to LLG

You can design your own translation scheme (it must be correct)

Label example

- LL is “:begin”
- LLG is “:begin_global_”



```
define @main ( ){
   :begin
   …
   :end
   …
}
```

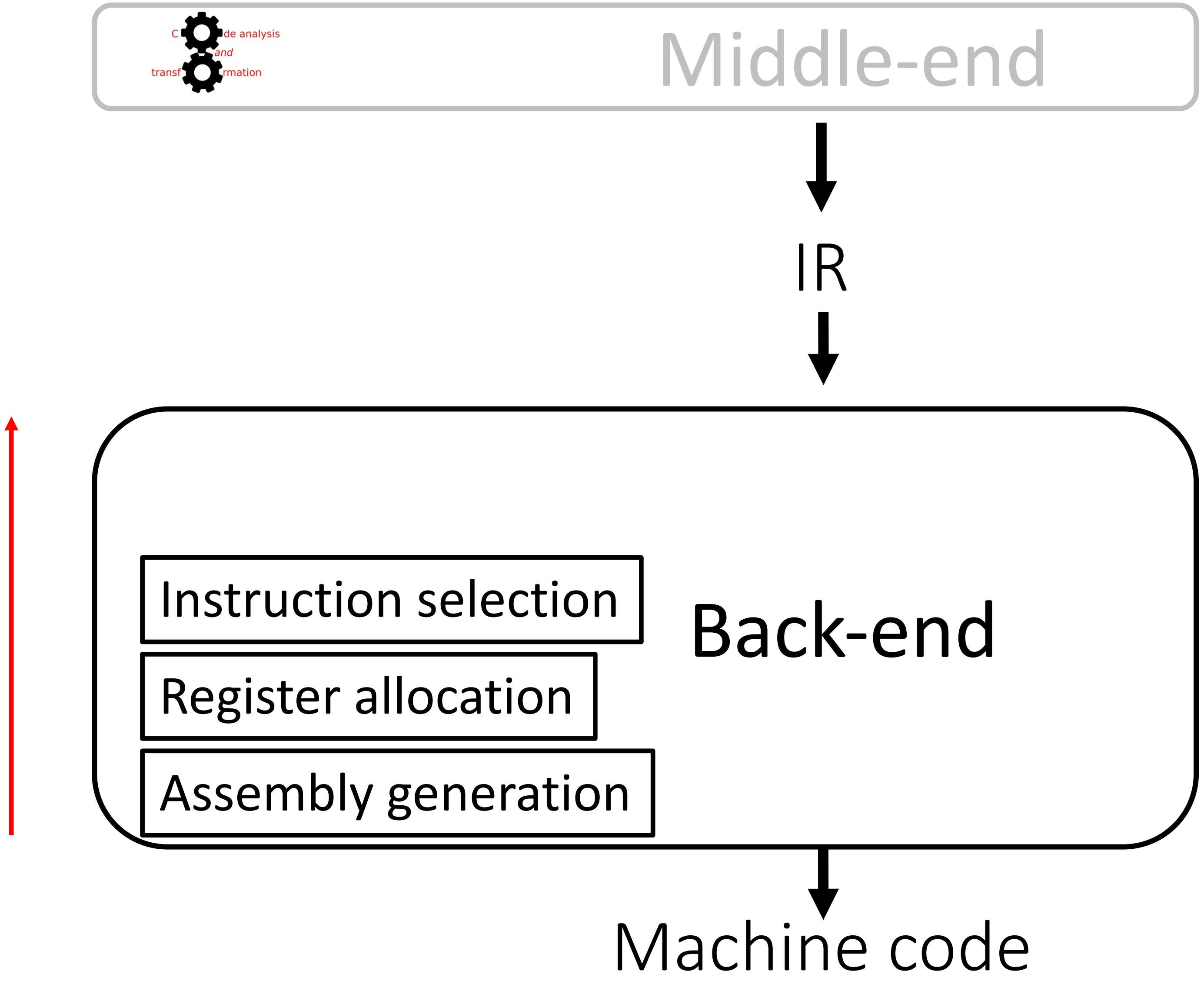
:begin_global0

:begin_global1

Outline

- L3
- Translating L3 to L2: calling convention and labels
- Translating L3 to L2: instruction selection

A compiler



Code analysis *and* transformation

Middle-end

IR

Back-end

Instruction selection

Register allocation

Assembly generation

Machine code

Instruction selection

The process of selecting the lower-level instructions
(assembly instructions)
to use to translate a higher-level representation (e.g., L3)

Instruction selection is intra-procedural

Naive instruction selection for L3

```
define @myF (%p1, %p2){
  %v1 <- %p1 * 4
  …
}
```

Translate L3 instructions one by one and independently with the surrounding ones

```
(@myF
  2
  %p1 <- rdi
  %p2 <- rsi
  %v1 <- %p1
  %v1 *= 4
  …
)
```

Naive translation of an L3 function: problem

```
define @myF (%p1, %p2){
  %v1 <- %p1 * 4
  %v2 <- %v1 + %p2
  …
}
```

Translate L3 instructions one by one and independently with the surrounding ones

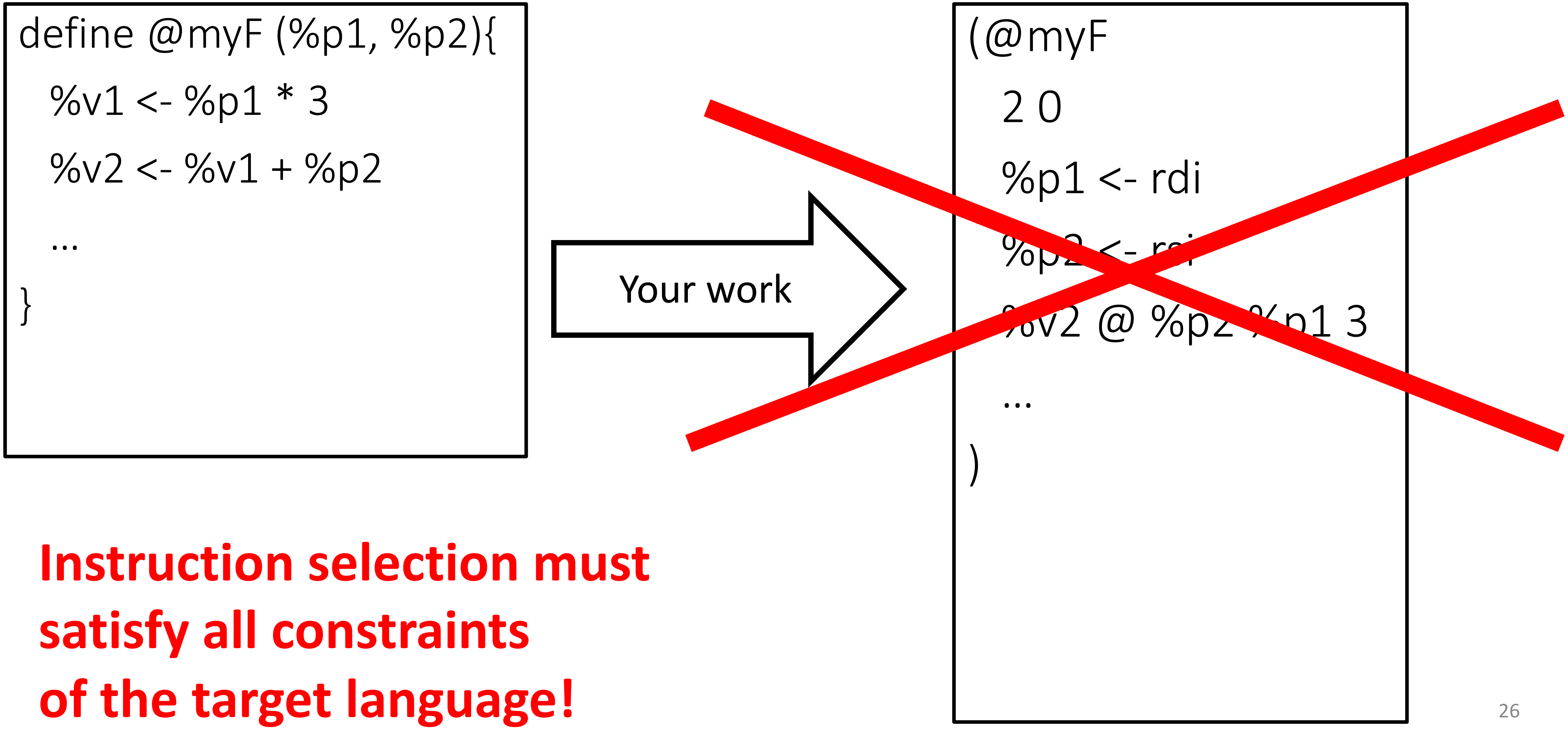
```
(@myF
  2 0
  %p1 <- rdi
  %p2 <- rsi
  %v1 <- %p1
  %v1 *= 4
  %v2 <- %v1
  %v2 += %p2
  …
)
```

Is there a better translation?

```
(@myF
  2 0
  %p1 <- rdi
  %p2 <- rsi
  %v2 @ %p2 %p1 4
  …
)
```

Instruction selection depends on the context!

Instruction selection: it isn't that easy



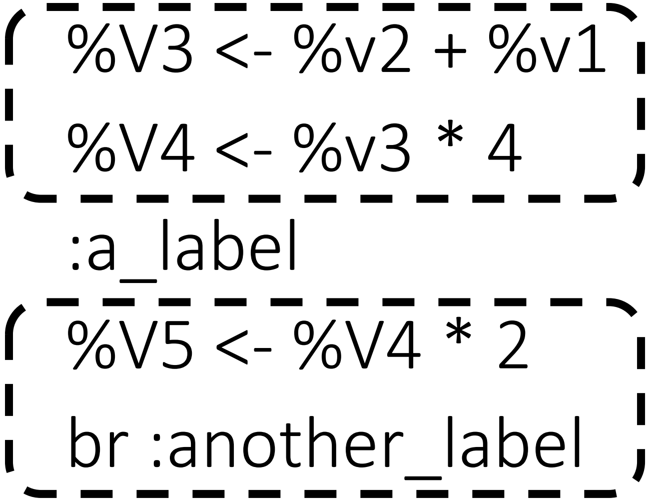
```
define @myF (%p1, %p2){
  %v1 <- %p1 * 3
  %v2 <- %v1 + %p2
  …
}
```

~~(@myF~~
~~2 0~~
~~%p1 <- rdi~~
~~%p2 <- rsi~~
~~%v2 @ %p2 %p1 3~~
~~…~~
~~)~~

Instruction selection must satisfy all constraints of the target language!

Instruction selection: context

- Instruction selection depends on the context
- Context for this class:
  sequence of instructions that does not include
  - a label instruction or
  - a call instruction
- The sequence must end when a branch or a return is encountered
  (the branch or return are part of the context)



```
%V3 <- %v2 + %v1
%V4 <- %v3 * 4
:a_label
%V5 <- %V4 * 2
br :another_label
```

Instruction selection step 1: identify contexts

```
Inst = F.entryPoint()
C = new Context()
While (Inst != nullptr){
  if (Inst is not Label or a call) C.add(Inst)
  if (Inst is Label, Branch, Call, Return) {
       C = new Context()
  }
  Inst = F.nextInst(Inst)
}
Delete empty contexts
```

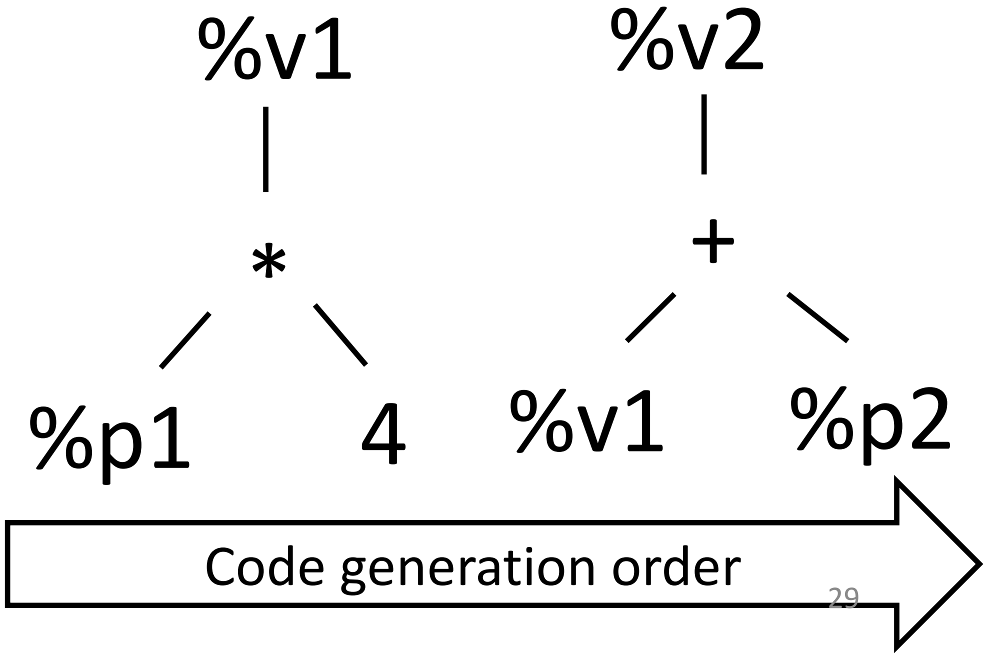
```
:myLabel
%v1 <- %p1 * 4
%v2 <- %v1 + %p2
br :otherLabel
```

Instruction selection step 2: tree generation

We need to generate the tree representation of the instructions of a context, for every context

- Generate a separate tree for every instruction
- The order of the trees define the order of translation/code generation (e.g., the first L2 instructions generated translate the first tree)

```
%v1 <- %p1 * 4
%v2 <- %v1 + %p2
```

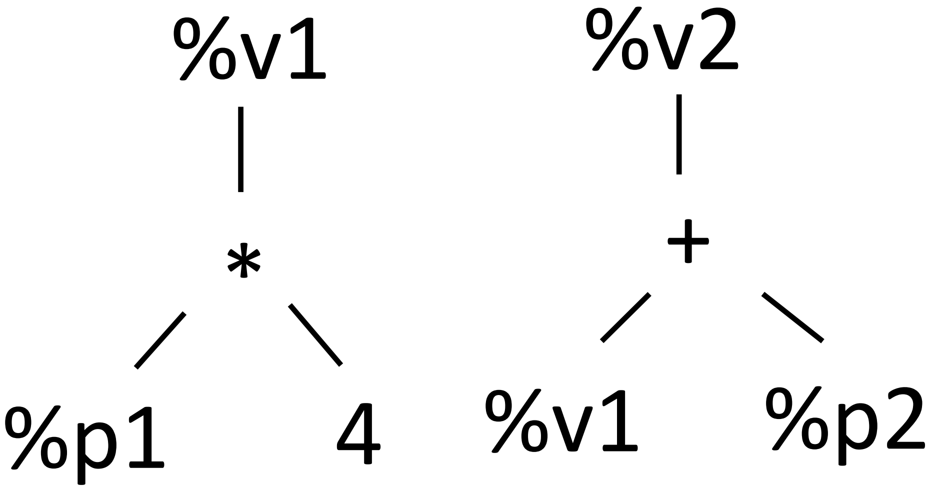


Instruction selection step 3: merging trees

- We perform instruction selection per tree
  - A target instruction (e.g., @ in L2) cannot cover nodes that belong to different trees
  - The bigger is the tree, the more optimal the instruction selection can be
- We aim to make trees as big as possible
  - We have generated the smallest trees (one per instruction)
  - Now we need to merge them as much as possible
  - Quality – complexity tradeoff: this class targets what is reasonable for one week of work

```
%v1 <- %p1 * 4
%v2 <- %v1 + %p2
```

Ideal selection: %v2 @ %p2 %p1 4



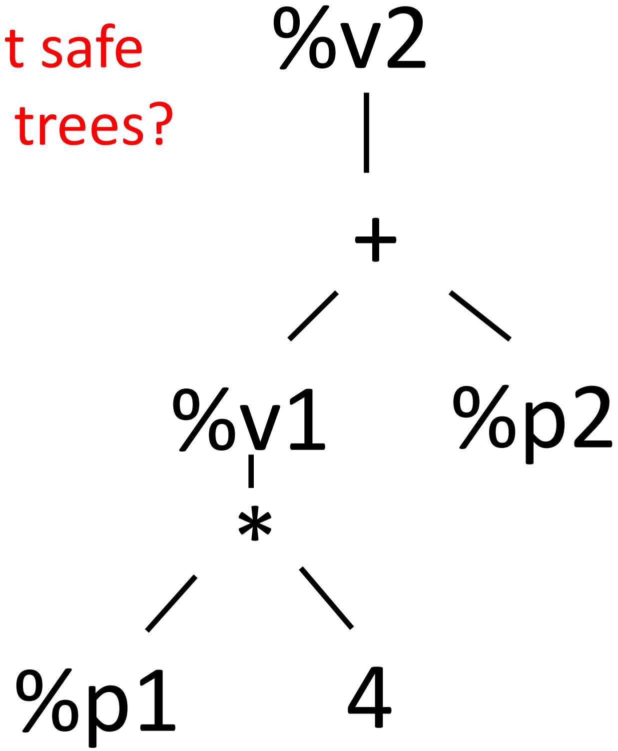
We cannot obtain the ideal selection because the target instruction (@) would cover nodes of different trees

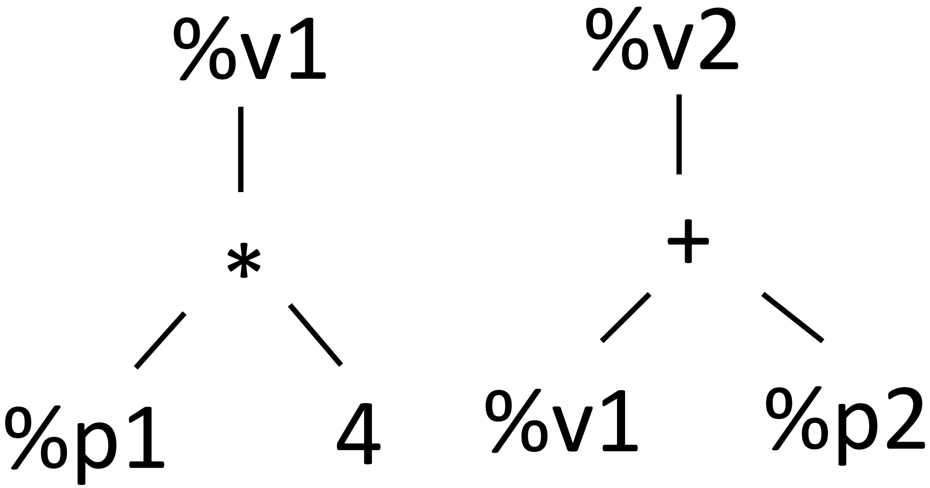
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

```
%v1 <- %p1 * 4
%v2 <- %v1 + %p2
```

When is it safe to merge trees?





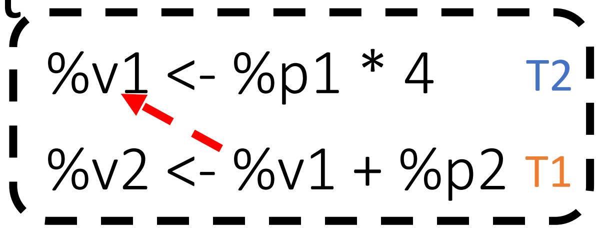
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

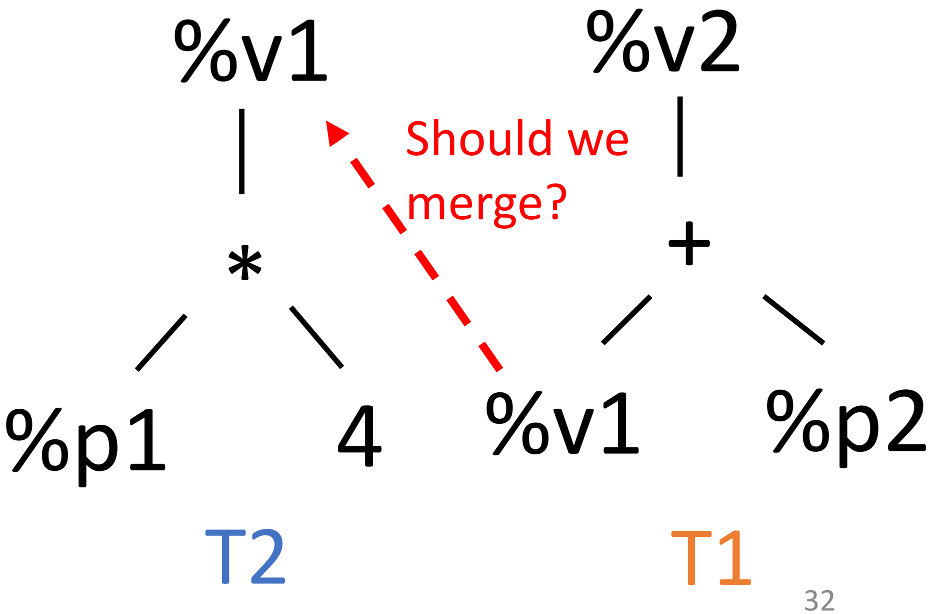
Let T1, T2 be two trees that belong to the same context

I. T1 uses a variable %v defined by T2

II. What else ?

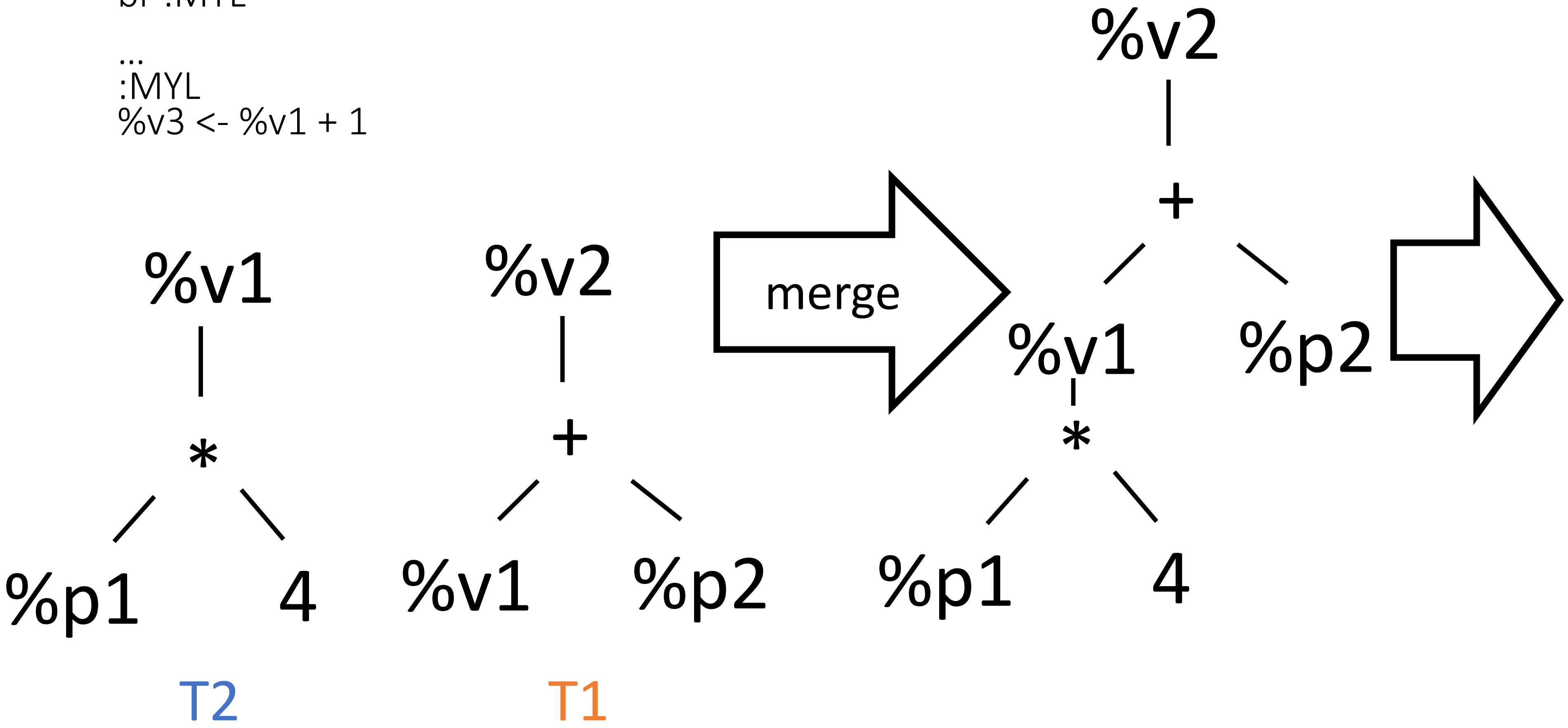


```
%v1 <- %p1 * 4      T2
%v2 <- %v1 + %p2    T1
```



Instruction selection step 3: merging trees

```
%v1 <- %p1 * 4
%v2 <- %v1 + %p2
br :MYL
…
:MYL
%v3 <- %v1 + 1
```



%v2 @ %p2 %p1 4

Is it correct?

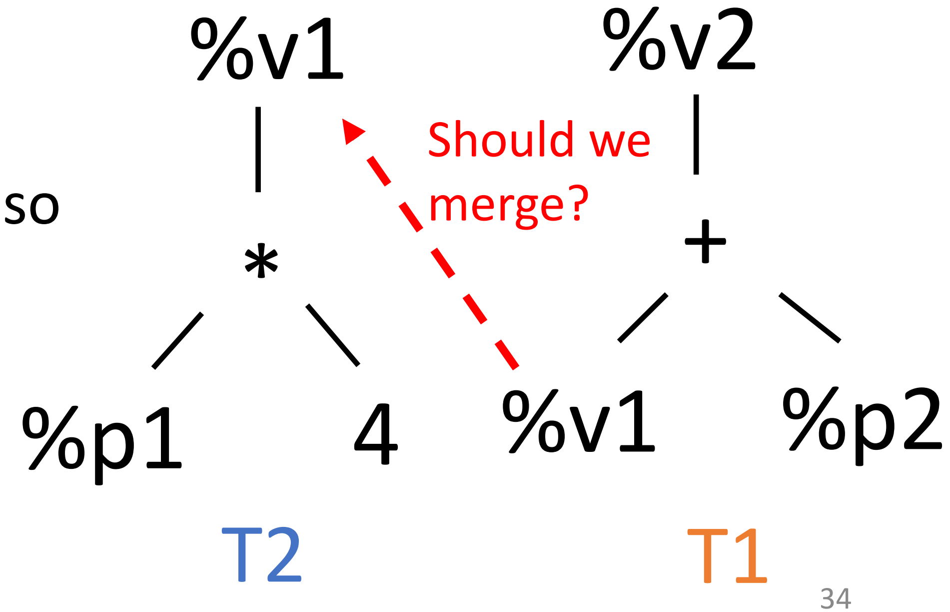
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

```
%v1 <- %p1 * 4      T2
%v2 <- %v1 + %p2 T1
```

Let T1, T2 be two trees that belong to the same context

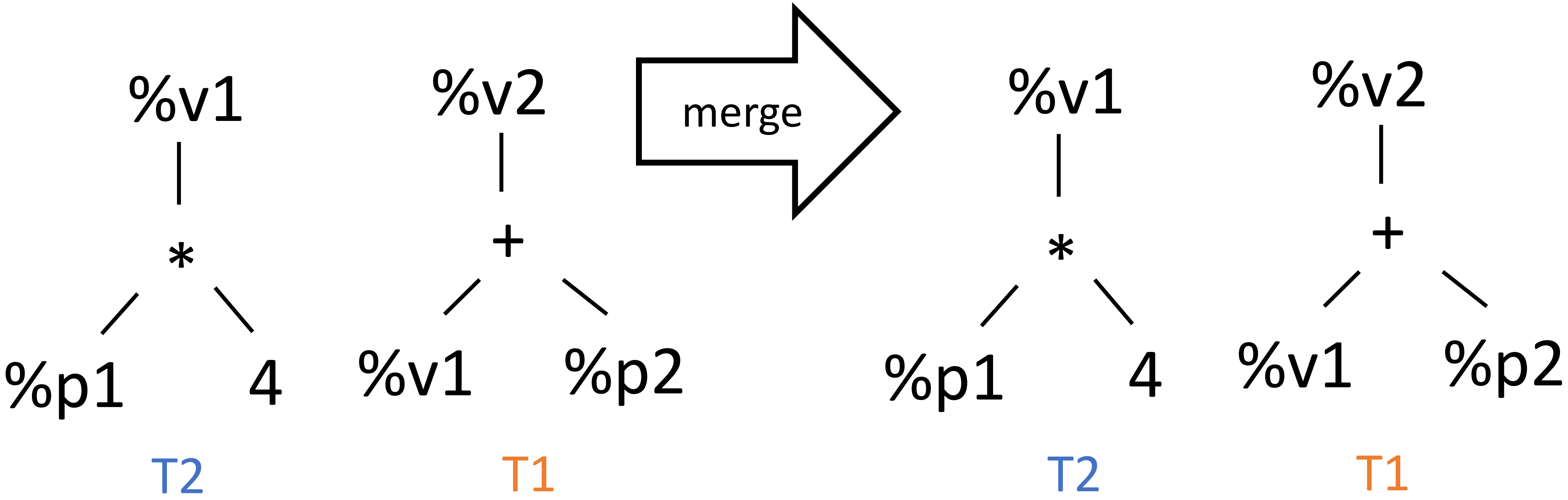
I. T1 uses a variable %v defined by T2
II. Merge T2 into T1 only when it is safe to do so
    A. %v is dead after the instruction related to T1 or %v is only used by T1



Instruction selection step 3: merging trees

```
%v1 <- %p1 * 4
%v2 <- %v1 + %p2
br :MYL

…
:MYL
%v3 <- %v1 + 1
```



Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

```
%v1 <- %p1 * 4      T2
%v2 <- %v1 + %p2 T1
```

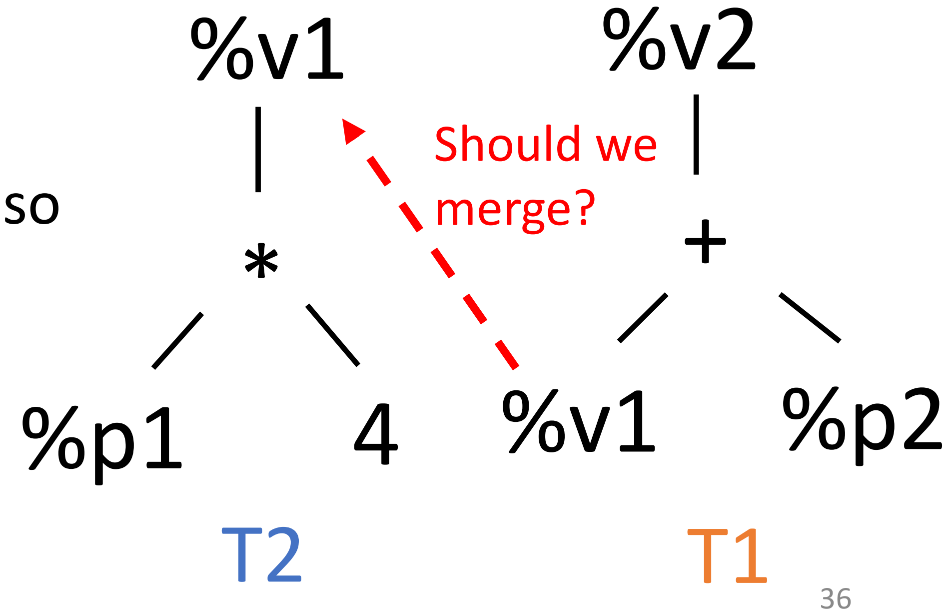
Let T1, T2 be two trees that belong to the same context

I. T1 uses a variable %v defined by T2

II. Merge T2 into T1 only when it is safe to do so

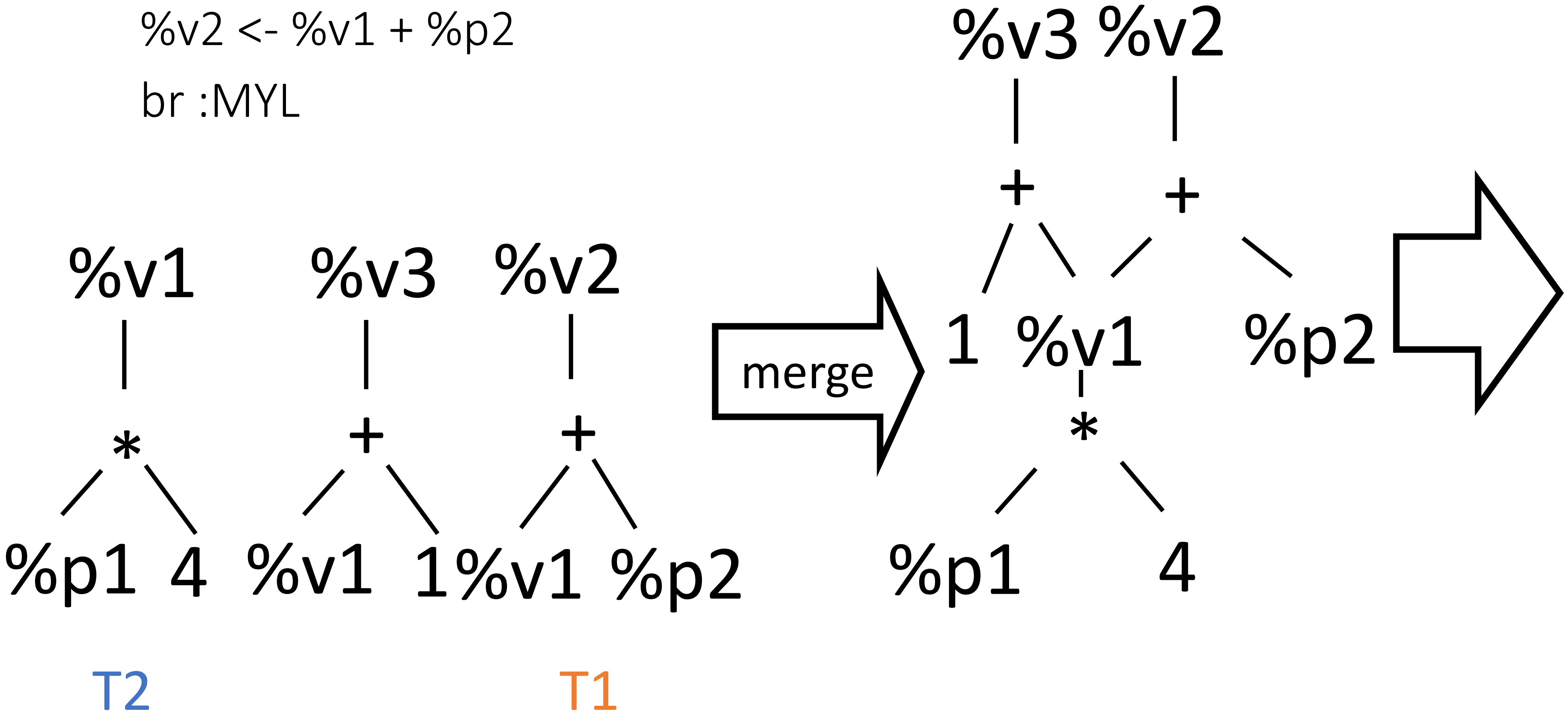
   A. %V is dead after the instruction attached to T1 or %v is only used by T1

   B. What else ?



Instruction selection step 3: merging trees

```
%v1 <- %p1 * 4
%v3 <- %v1 + 1
%v2 <- %v1 + %p2
br :MYL
```



```
%v3 <- %v1 + 1
%v2 @ %p2 %p1 4
…
```

Is it correct?

Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

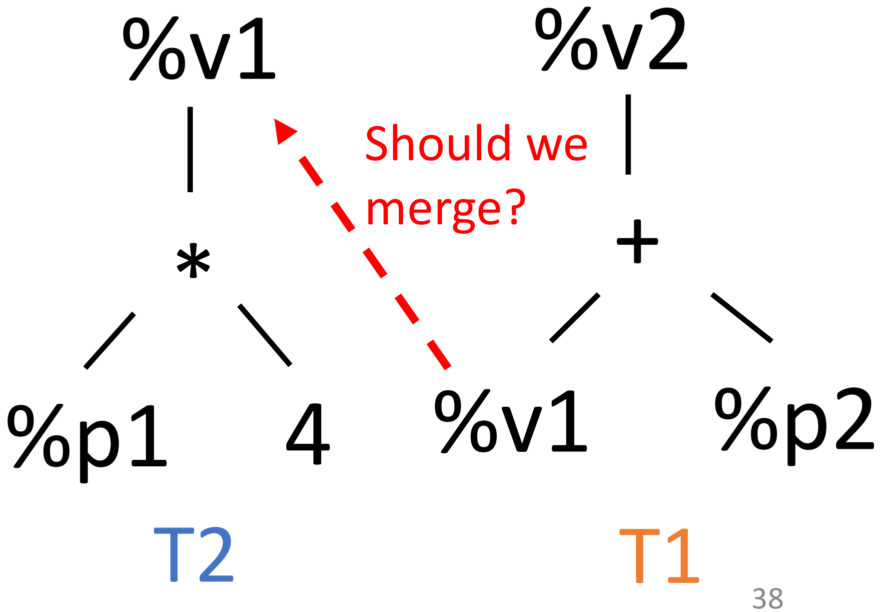
```
%v1 <- %p1 * 4      T2
%v2 <- %v1 + %p2  T1
```

Let T1, T2 be two trees that belong to the same context

I. T1 uses a variable %v defined by T2

II. Merge T2 into T1 only when it is safe to do so
   A. %v is dead after the instruction attached to T1 or %v is only used by T1
   B. No instruction that depends on T2 between T2 and T1

Including T2 in this range



- Dependences exist between instructions
  when they both access a variable or memory location
  and one of them is a write
- For variables the condition B of the previous slide
  becomes the following

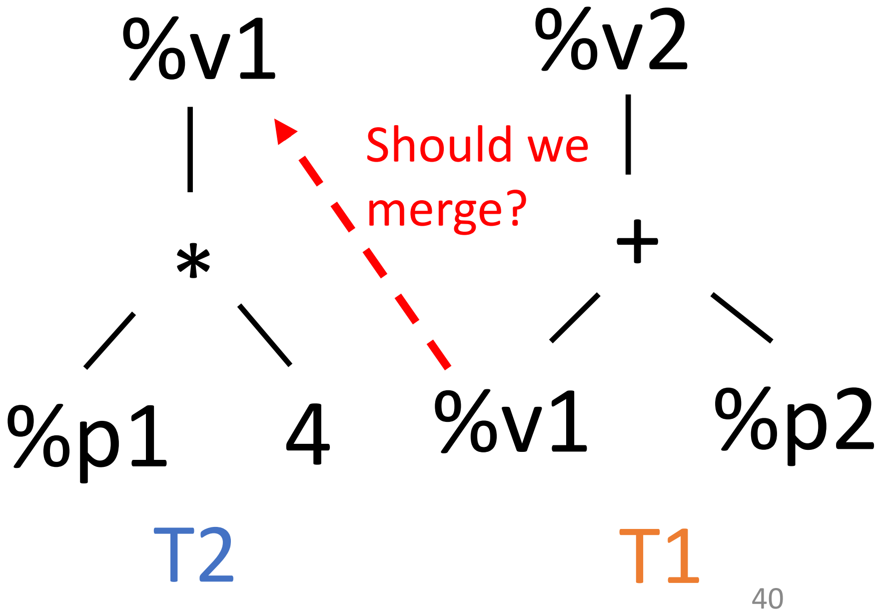
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

```
%v1 <- %p1 * 4       T2
%v2 <- %v1 + %p2     T1
```

Let T1, T2 be two trees that belong to the same context

- I. T1 uses a variable %v defined by T2
- II. Merge T2 into T1 only when it is safe to do so
  - A. %v is dead after the instruction attached to T1 or %v is only used by T1
  - B. No instruction that depends on T2 between $\underbrace{\text{T2 and T1}}_{\textit{Including T2 in this range}}$



Instruction selection step 3: merging trees

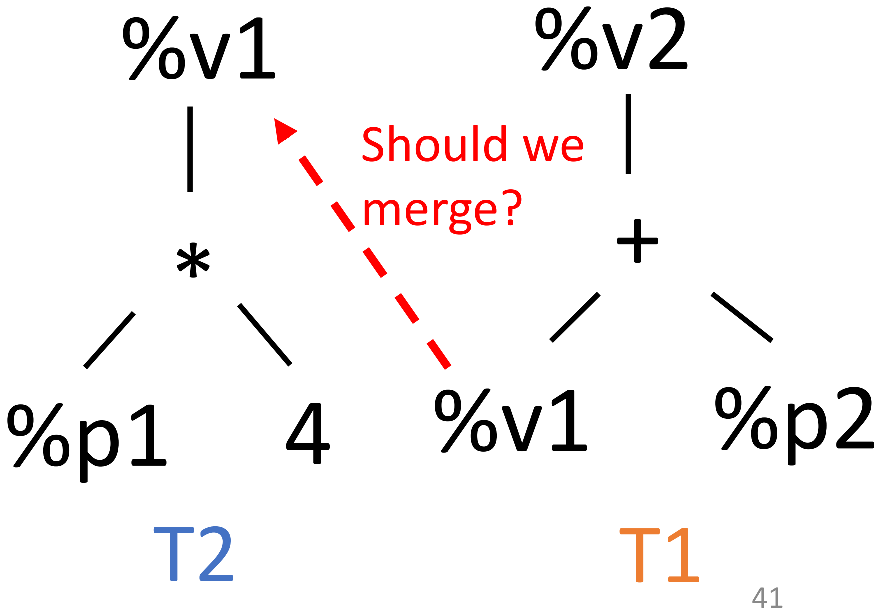
1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

```
%v1 <- %p1 * 4      T2
%v2 <- %v1 + %p2    T1
```

Let T1, T2 be two trees that belong to the same context

- I. T1 uses a variable %v defined by T2
- II. Merge T2 into T1 only when it is safe to do so
  - A. %v is dead after the instruction attached to T1 or %v is only used by T1
  - B. No other uses of %v between T2 and T1 and

Including T2 in this range



Instruction selection step 3: merging trees

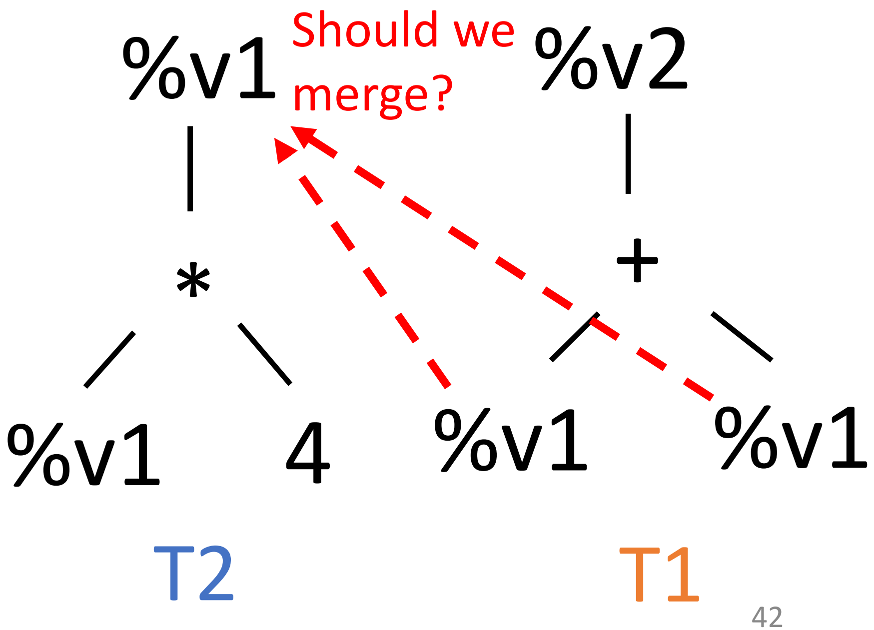
1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

```
%v1 <- %v1 * 4      T2
%v2 <- %v1 + %v1   T1
```

Let T1, T2 be two trees that belong to the same context

- I. T1 uses a variable %v defined by T2
- II. Merge T2 into T1 only when it is safe to do so
  - A. %v is dead after the instruction attached to T1 or %v is only used by T1
  - B. No other uses of %v between T2 and T1 and

Including T2 in this range



Instruction selection step 3: merging trees

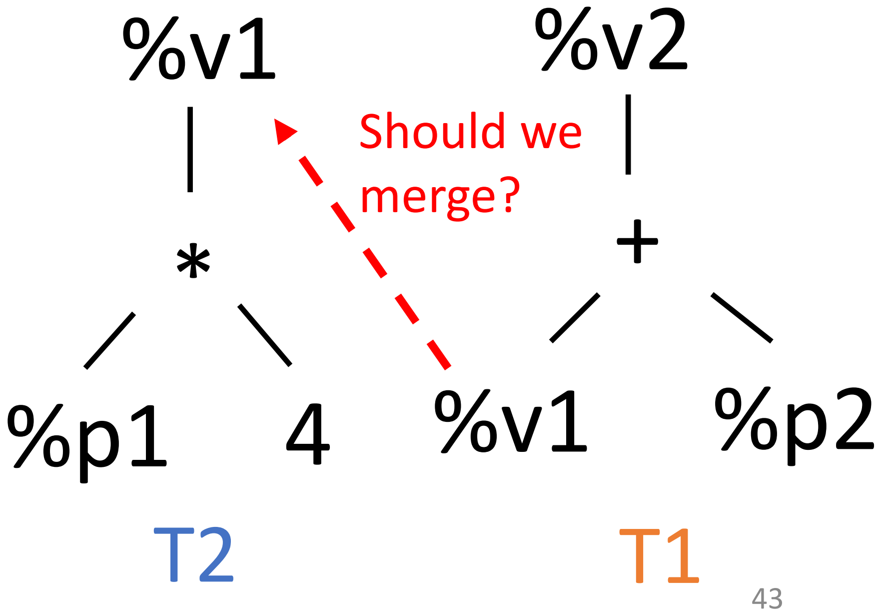
1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

```
%v1 <- %p1 * 4      T2
%v2 <- %v1 + %p2 T1
```

Let T1, T2 be two trees that belong to the same context

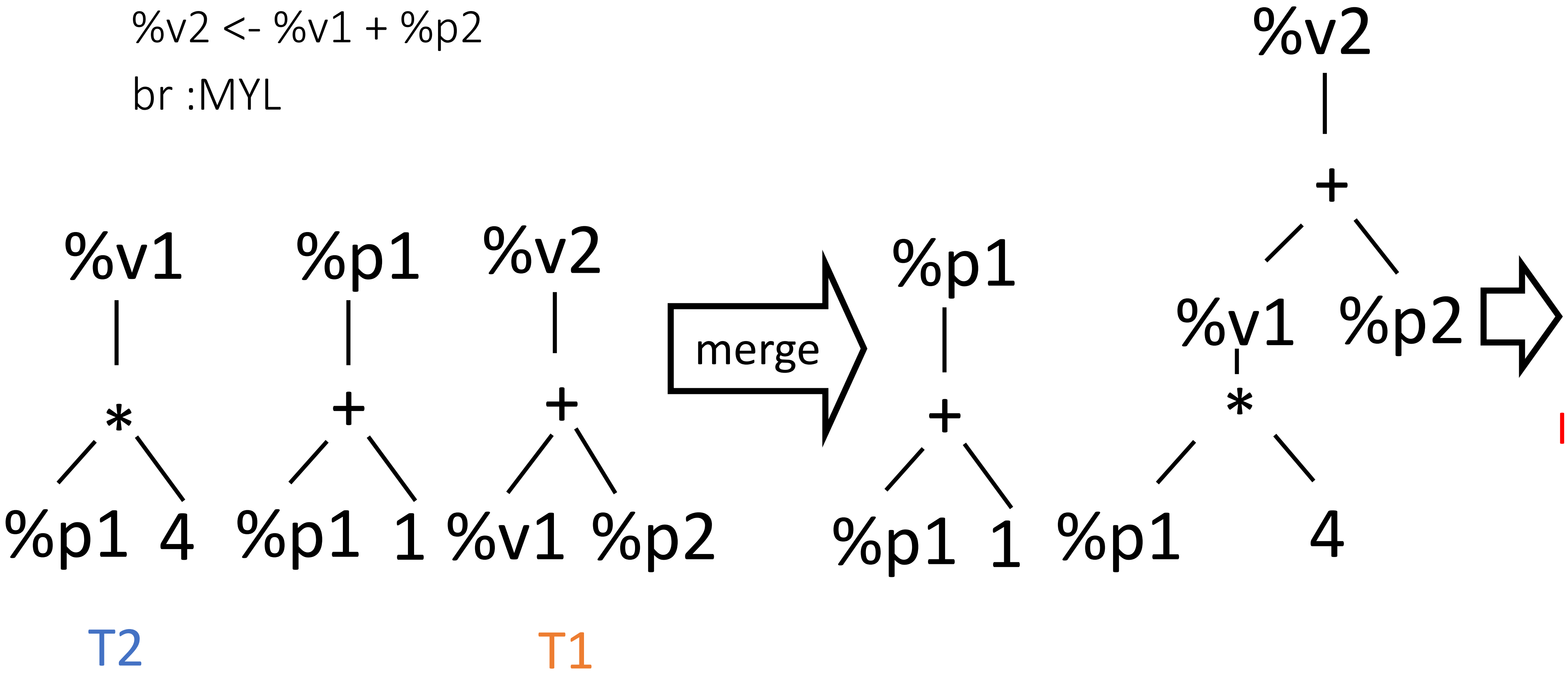
I. T1 uses a variable %v defined by T2
II. Merge T2 into T1 only when it is safe to do so
   A. %v is dead after the instruction attached to T1 or %v is only used by T1
   B. No other uses of %v between T2 and T1 and

What else ?



Instruction selection step 3: merging trees

```
%v1 <- %p1 * 4
%p1 <- %p1 + 1
%v2 <- %v1 + %p2
br :MYL
```



```
%p1 <- %p1 + 1
%v2 @ %p2 %p1 4
```

Is it correct?

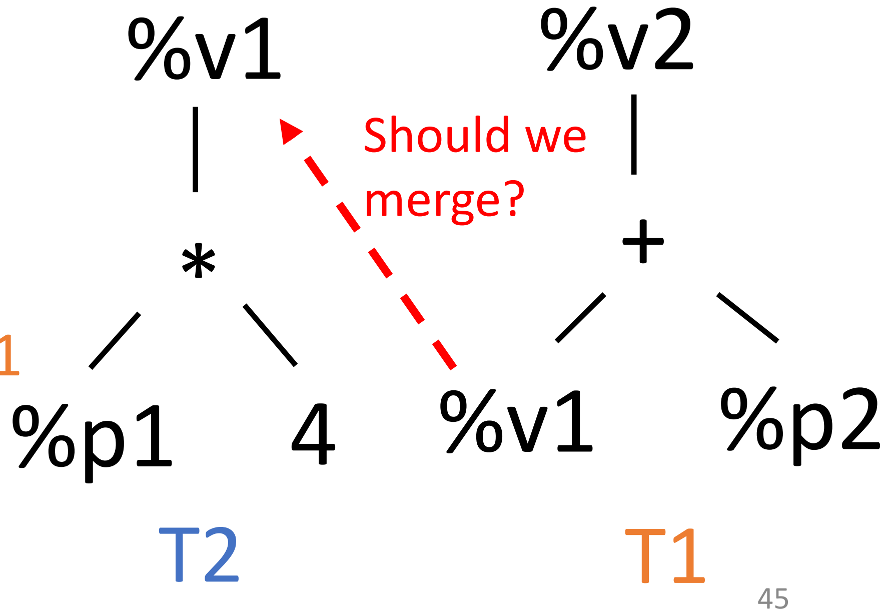
Instruction selection step 3: merging trees

1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

```
%v1 <- %p1 * 4       T2
%v2 <- %v1 + %p2  T1
```

Let T1, T2 be two trees that belong to the same context

- I. T1 uses a variable %V defined by T2
- II. Merge T2 into T1 only when it is safe to do so
  - A. %v is dead after the instruction represented by T1 or %v is only used by T1
  - B. No other uses of %v between T2 and T1 and no definitions of variables used by T2 between T2 and T1



- The previous condition excludes the possibility to have instructions between T2 and T1 that depends on T2

- **Dependence definition**:
  two generic instructions depend on each other
  if they both access a variable or memory location
  and one of them is a write

- If T2 accesses a memory location, then condition B becomes the following

Instruction selection step 3: merging trees

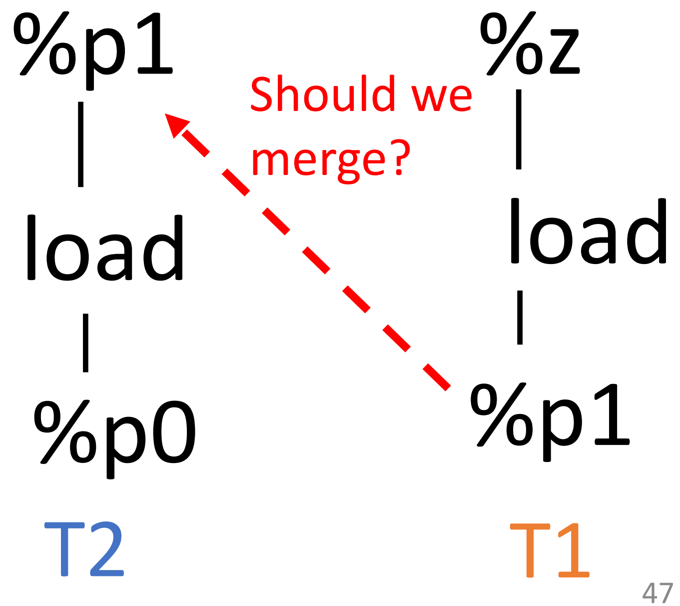
1. Cluster trees that belong to the same context
2. Merge trees (as much as possible) that belong to the same context

Let T1, T2 be two trees that belong to the same context

- I. T1 uses a variable %V defined by T2
- II. Merge T2 into T1 only when it is safe to do so
  - A. %v is dead after the instruction attached to T1 or %v is only used by T1
  - B. No memory instruction between T2 and T1

Including T2 in this range

```
%p1 <- load %p0    T2
store 3, %p2
%z <- load %p1     T1
```

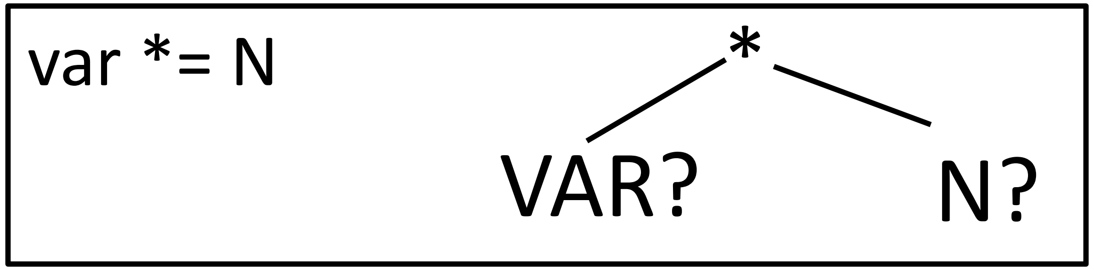


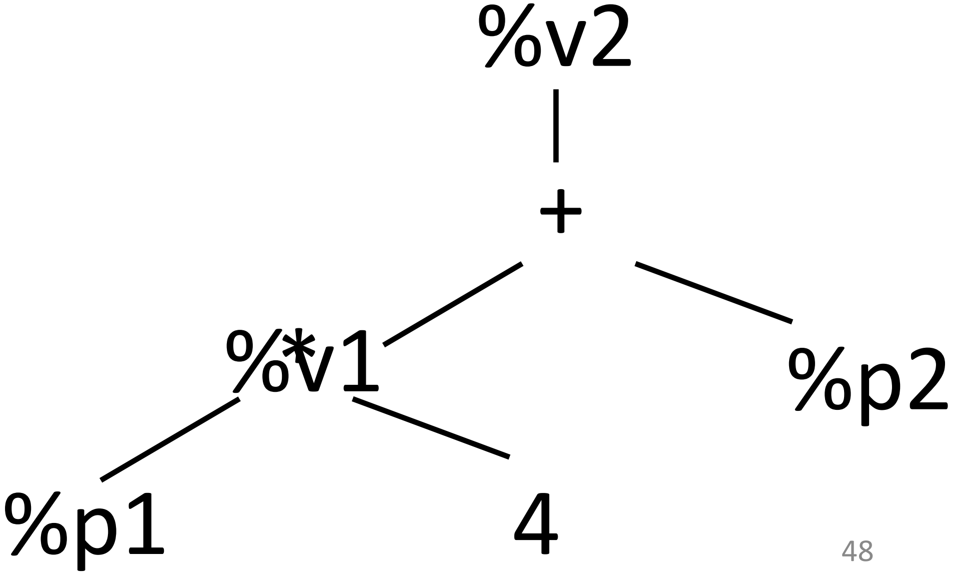
Instruction selection step 4: tiling trees

- Tile = instruction of the target language (e.g., L2) = pattern
- Instruction selectors use pattern-matching on trees with tiles
  - Use a tree-based code representation
  - Each target instruction defines a tile (pattern) that can be used to cover the tree
  - Used tiles (patterns) = selected target instructions to generate

```
%v1 <- %p1 * 4

%v2 <- %v1 + %p2
```


var *= N

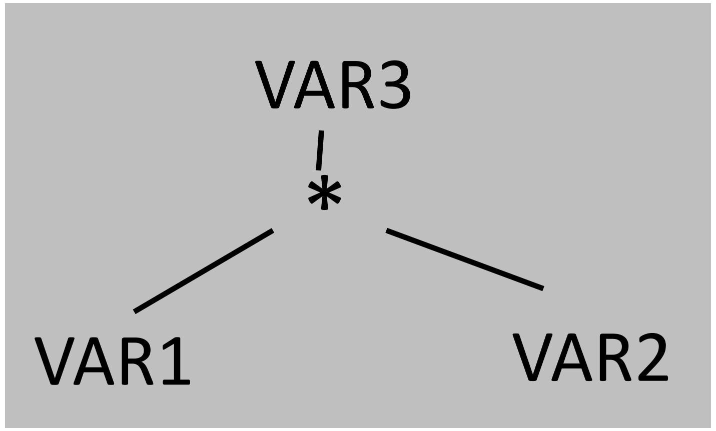


From L3 instructions to L2 instructions

1. Translate L3 instructions of a context into a list of trees
    - Order needs to be preserved
2. Merge as many trees as possible
3. For each tree (in order):
    - **A. Tiling:** cover the tree with L2 tiles
    - **B. Code generation:** from the bottom to the top of the tree:
        - i. Get the next tile
        - ii. Append L2 instructions generated by the current tile

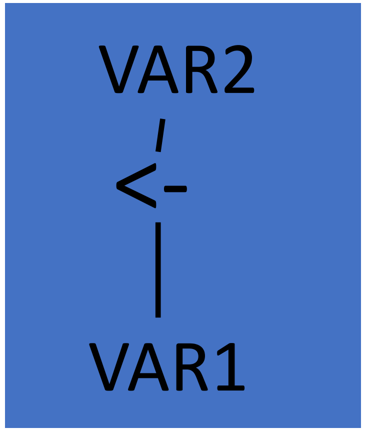
Example: tiles and tiling

? *= ?



VAR3 <- VAR1
VAR3 *= VAR2

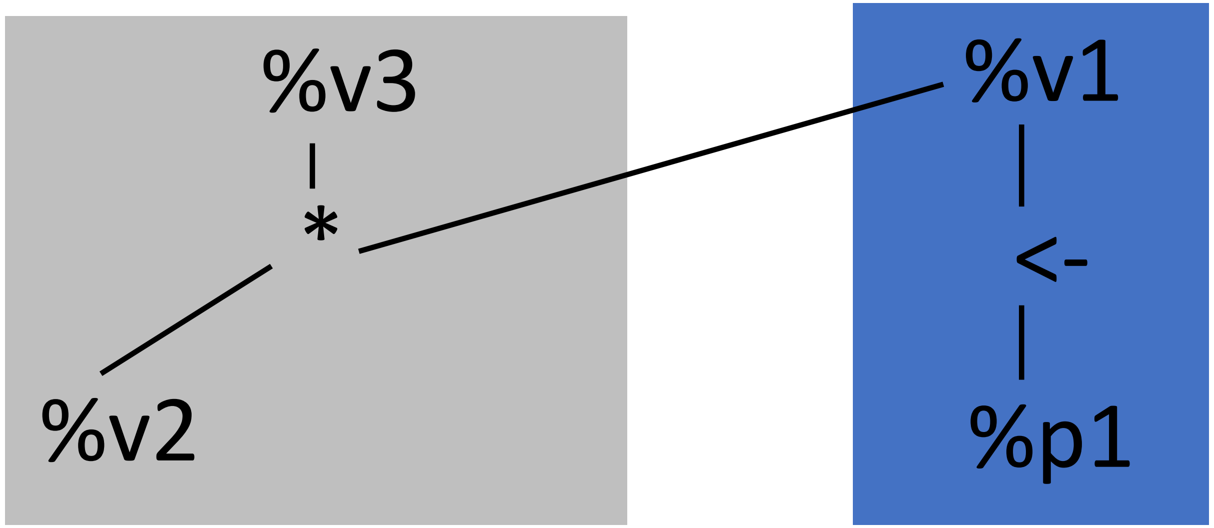
? <- ?



VAR2 <- VAR1

%v1 <- %p1

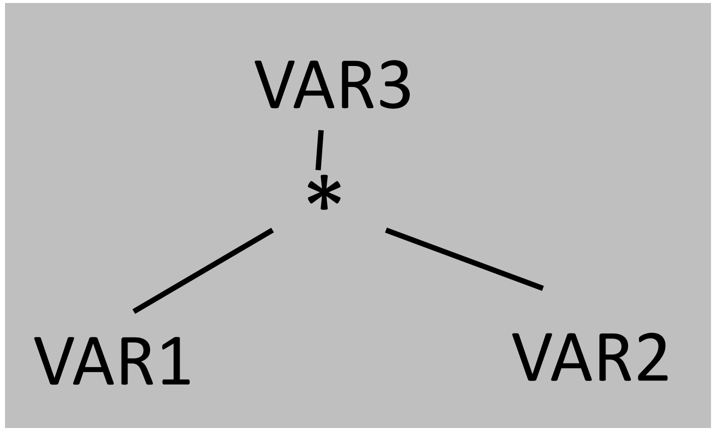
%v3 <- %v2 * %v1



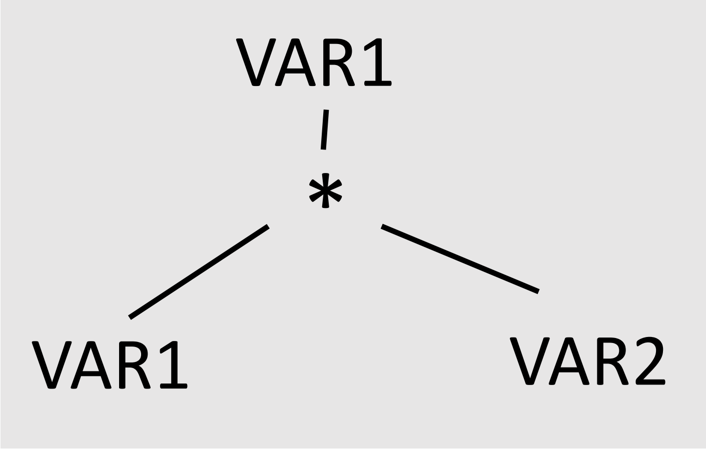
%v1 <- %p1
%v3 <- %v2
%v3 *= %v1

Specialized tiles

? *= ?



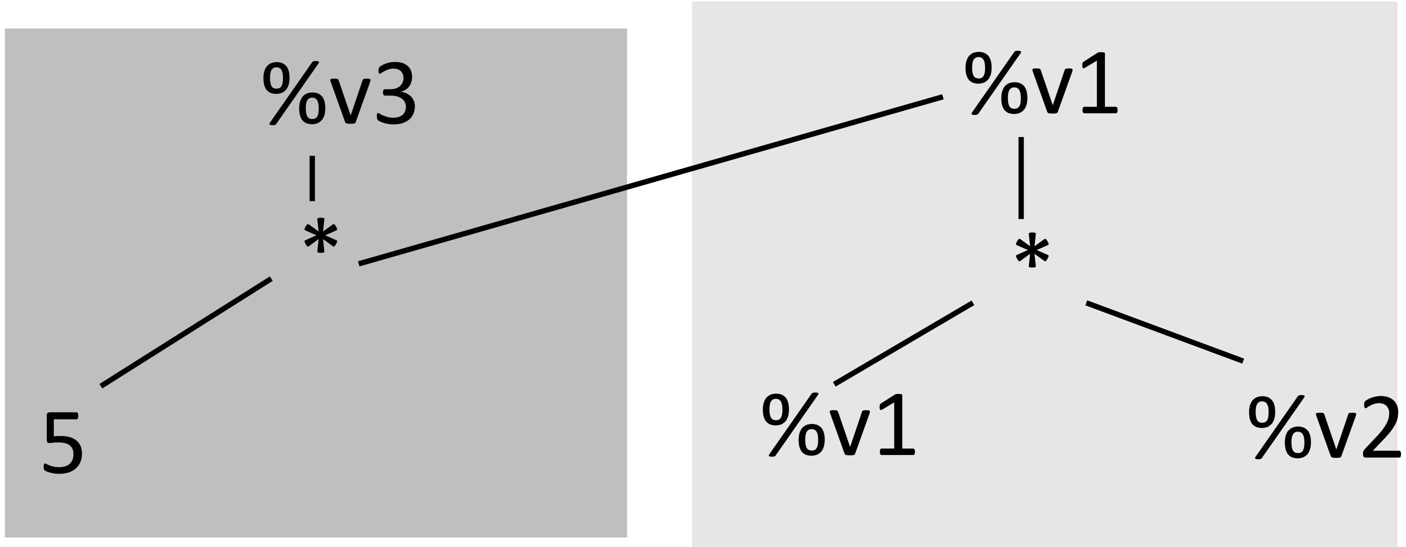
VAR3 <- VAR1
VAR3 *= VAR2



var1 *= var2

%v1 <- %v1*%v2

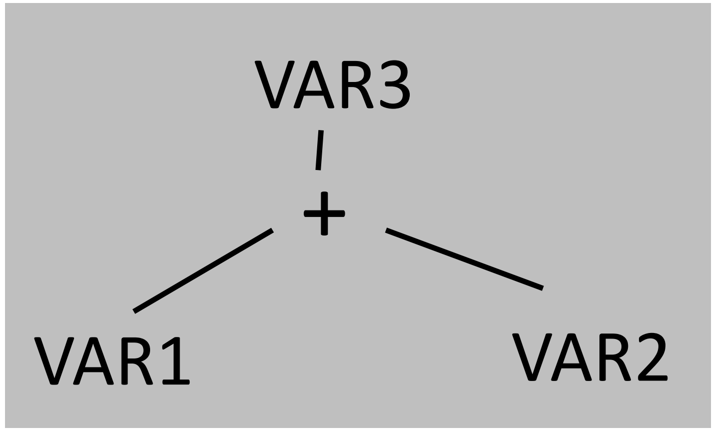
%v3 <- %v1 * 5



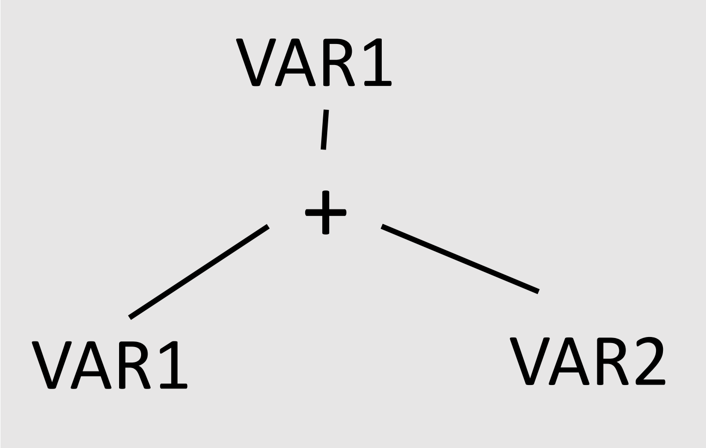
%v1 *= %v2
%v3 <- %v1
%v3 *= 5

Large tiles

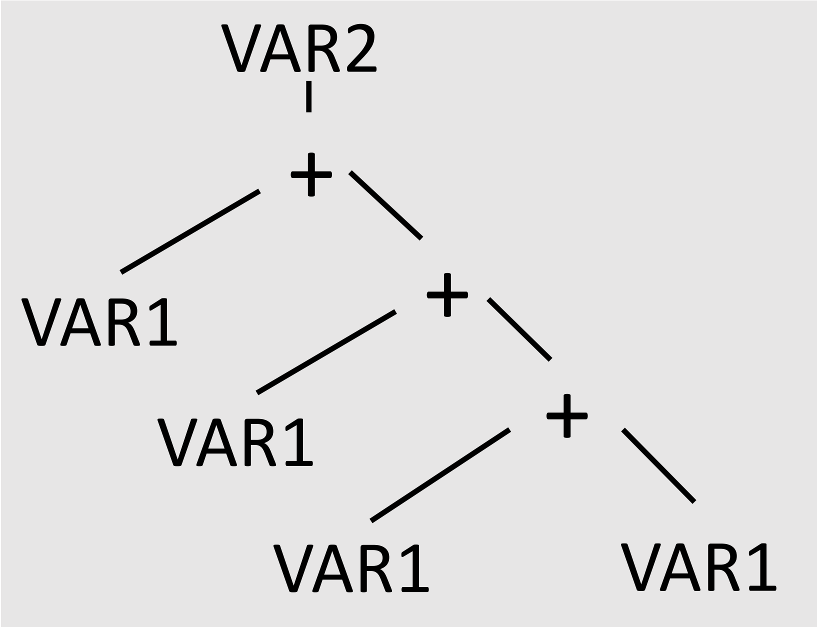
? *= ?



VAR3 <- VAR1
VAR3 += VAR2



var1 += var2



var2 <- var1
var1 <<= 2

Tiles and tiling

- Tiles capture compiler’s understanding of the target instruction set

- In general, for any given tree, many tilings are possible
  - Each resulting in a different instruction sequence

- We ensure pattern coverage by covering, at a minimum, all atomic L3 trees

The instruction selection problem

- Many solutions to cover a tree are possible

- How to pick tiles that cover our tree with minimum execution time?

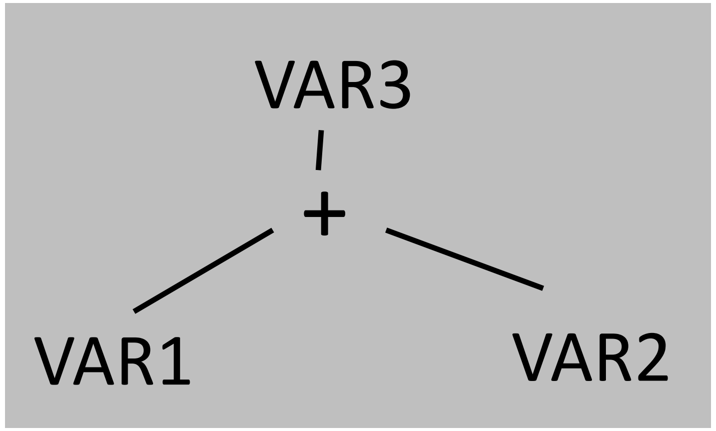
- Need a good selection of tiles
  - Small tiles to make sure we can tile every tree
  - Large tiles for efficiency

Quality of a tile in CC

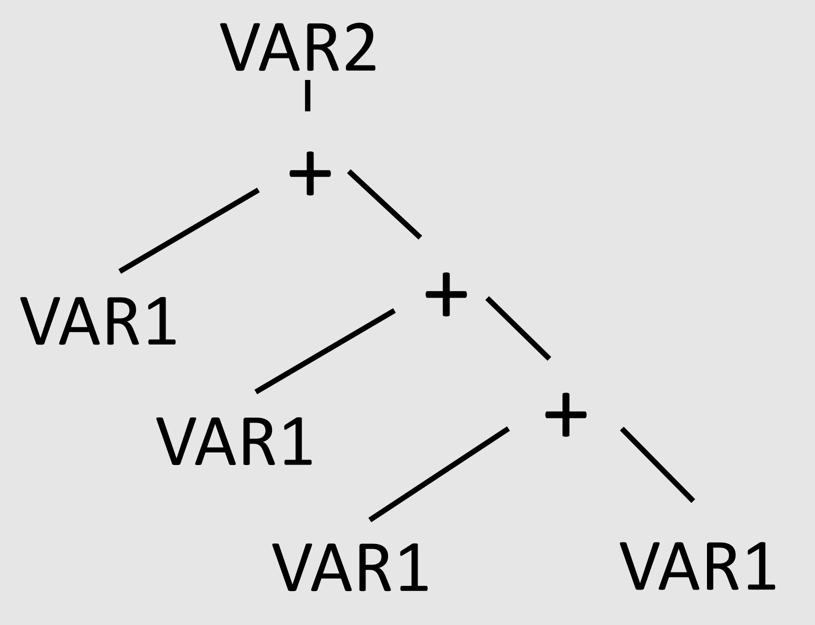
- Instruction selection should prefer high-quality tiles
- The quality of a tile t is related to the latency of the instructions generated by t
- In this class, we use the number of instructions as proxy to the latency
- Hence, if two tiles cover the same sub-tree, then we choose the one that has less instructions
  - Each tile reports the number of instructions generated by it

Tiles in CC

- Tiles need to be designed such that
  a large tile t has <= instructions
  than a possible set of small tiles that cover the same sub-tree
- Hence, we prefer larger tiles: fewer instructions



```
VAR3 <- VAR1
VAR3 += VAR2
```



```
var2 <- var1
var1 <<= 2
```

Quality of a solution of the tiling problem

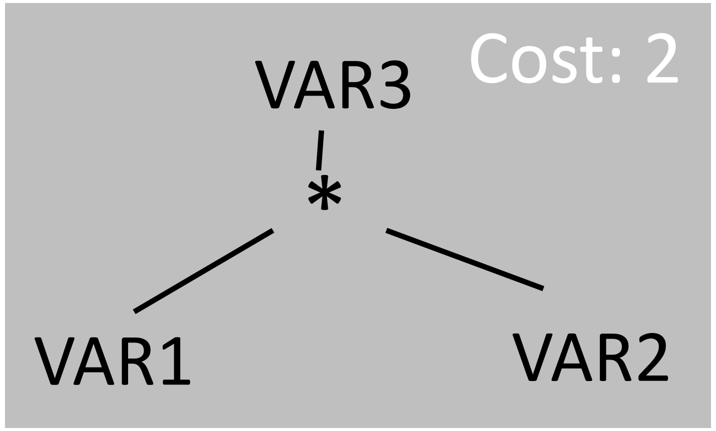
- Tiling problem: choose a set of tiles to cover a tree

- Quality of a tiling solution: the cumulative execution time of all instructions generated to cover a tree

- In instruction selection, we estimate the total execution time as the sum of costs of all tiles
  - *In this class*: the cost of a tile is the number of instructions of it
  - *Hence, in in this class*: the quality of a tiling solution is the total number of instructions generated

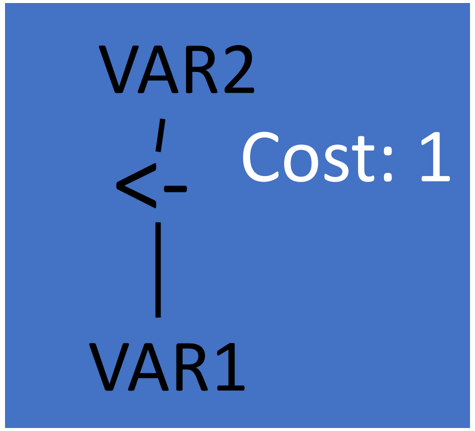
Example of tiling cost for L3

? *= ?



VAR3 <- VAR1
VAR3 *= VAR2

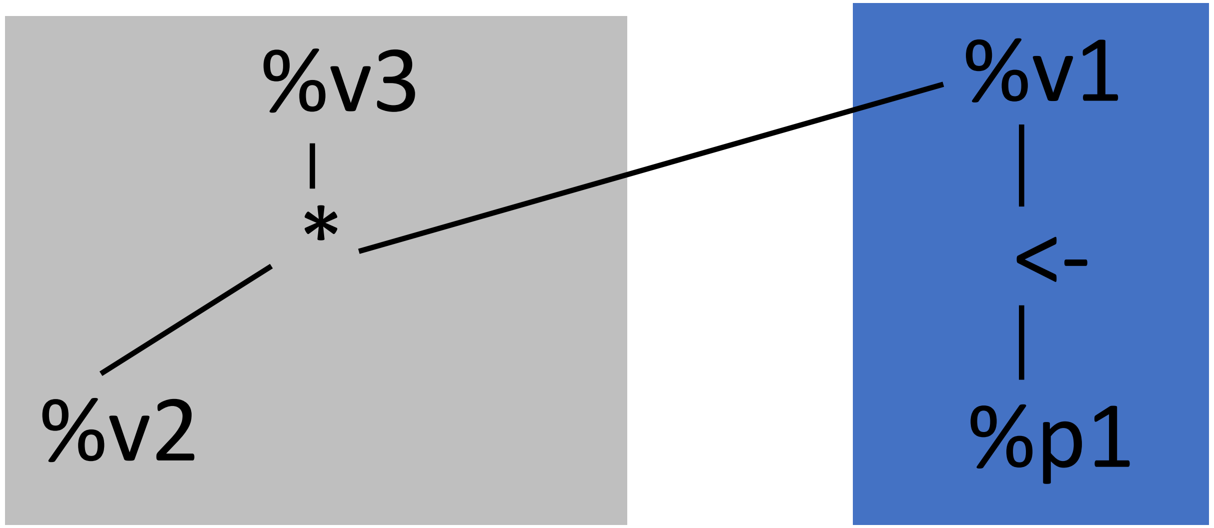
? <- ?



VAR2 <- VAR1

%v1 <- %p1

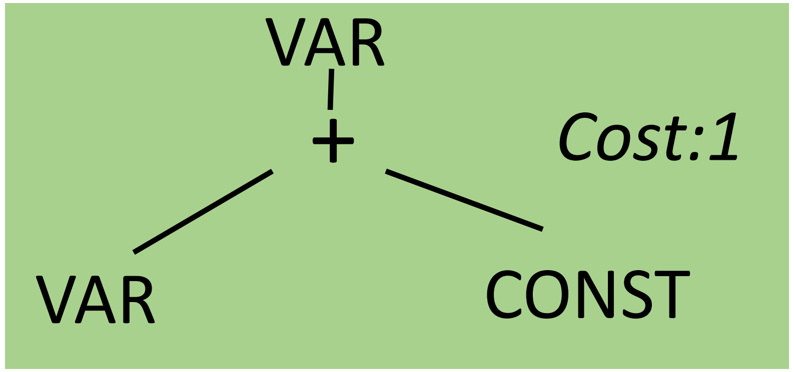
%v3 <- %v2 * %v1

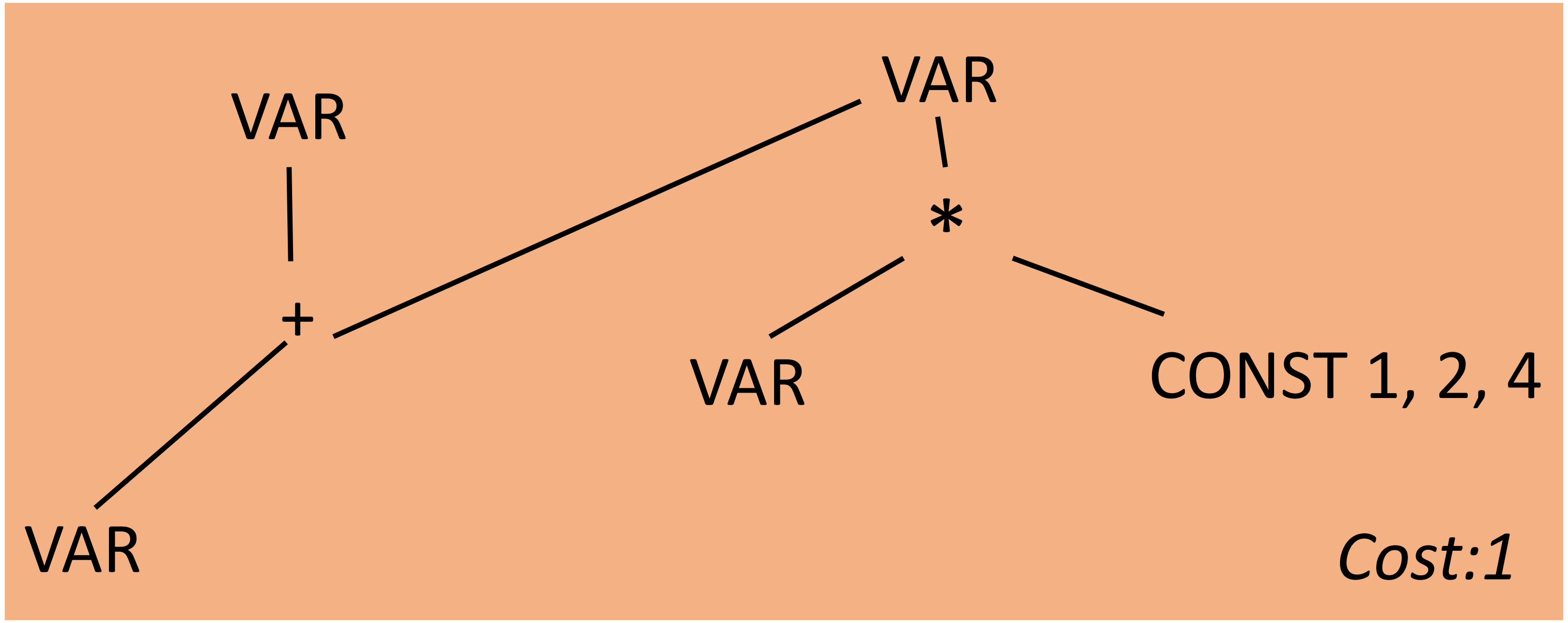


%v1 <- %p1
%v3 <- %v2
%v3 *= %v1

Total cost: 3

Other examples of L2 tiles





Global vs. local optimal solution

- We want the “lowest cost” tiling
  - Take into account cost/delay of each instruction (i.e., timing model)

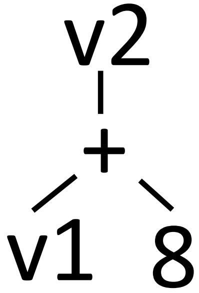
- **Optimum** tiling: lowest-cost tiling
- **Locally Optimal** tiling: no two adjacent tiles can be combined into one tile of lower cost

Locally optimal tilings

- A simple greedy algorithm works extremely well in practice:
  Maximal munch
- Choose the largest pattern with lowest cost,
  i.e., the “maximal munch”
- Algorithm:
  - Start at root
  - Use “biggest” match (in # of nodes)
    - This is the munch
    - Use cost to break ties
  - Recursively apply maximal much
    at each subtree of this munch

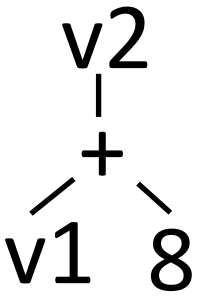
Maximal munch example

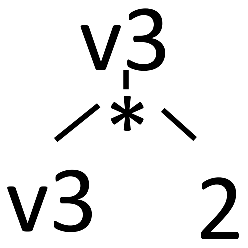
```
→ %v2 <- %v1 + 8
  %v3 <- %v3 * 2
  %v3 <- %v3 * 4
  %v4 <- load %v2
  %v5 <- %v4 + %v3
```



Maximal munch example

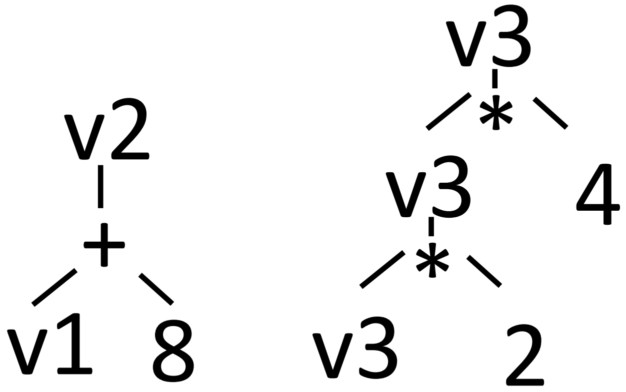
```
 %v2 <- %v1 + 8
→%v3 <- %v3 * 2
 %v3 <- %v3 * 4
 %v4 <- load %v2
 %v5 <- %v4 + %v3
```





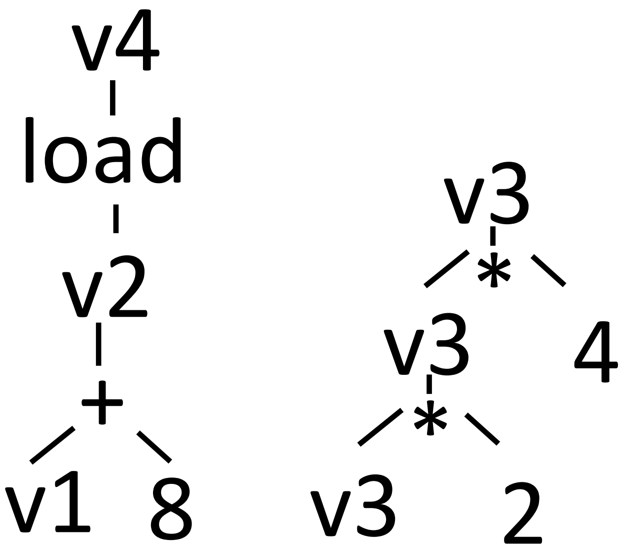
Maximal munch example

```
  %v2 <- %v1 + 8
  %v3 <- %v3 * 2
→ %v3 <- %v3 * 4
  %v4 <- load %v2
  %v5 <- %v4 + %v3
```



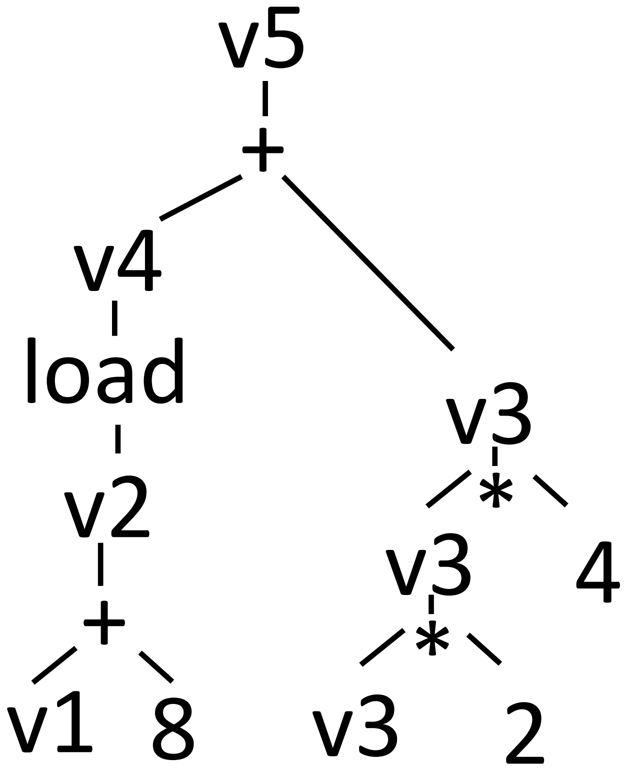
Maximal munch example

```
  %v2 <- %v1 + 8
  %v3 <- %v3 * 2
  %v3 <- %v3 * 4
→ %v4 <- load %v2
  %v5 <- %v4 + %v3
```

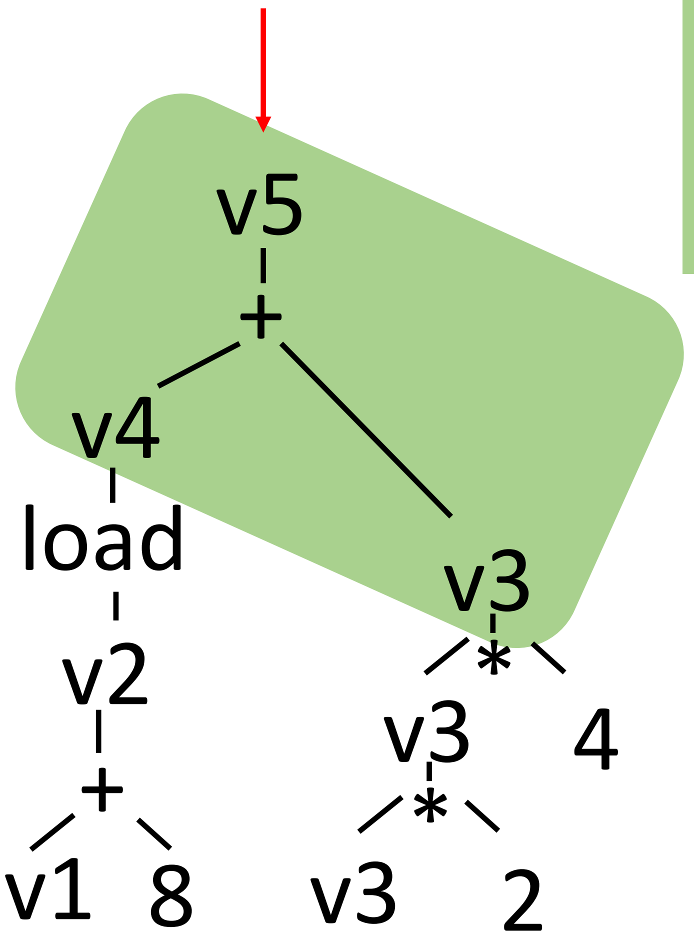


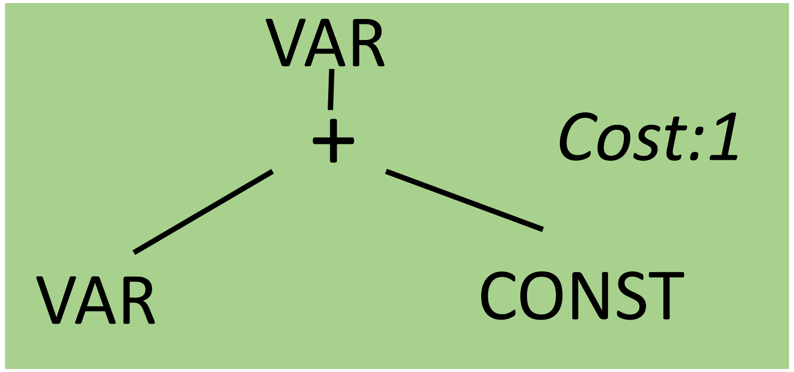
Maximal munch example

```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
→%v5 <- %v4 + %v3
```



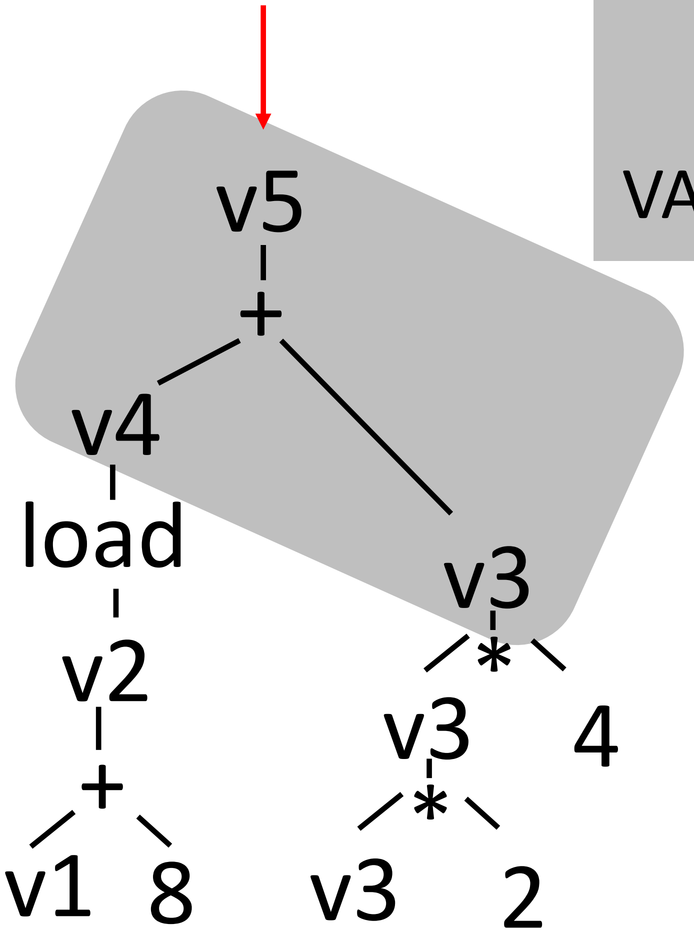
Maximal munch example

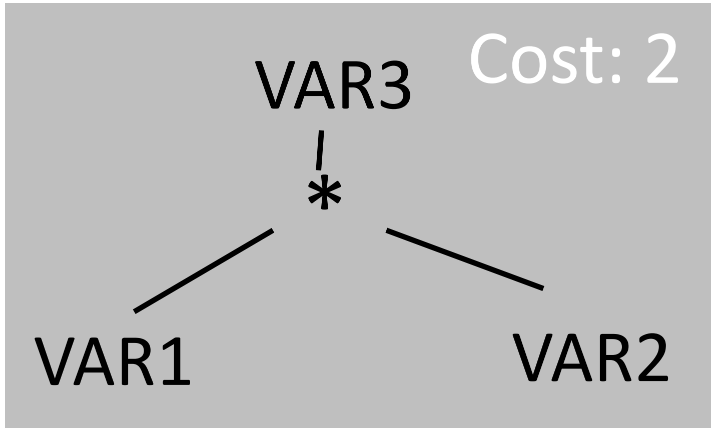




```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
```

Maximal munch example



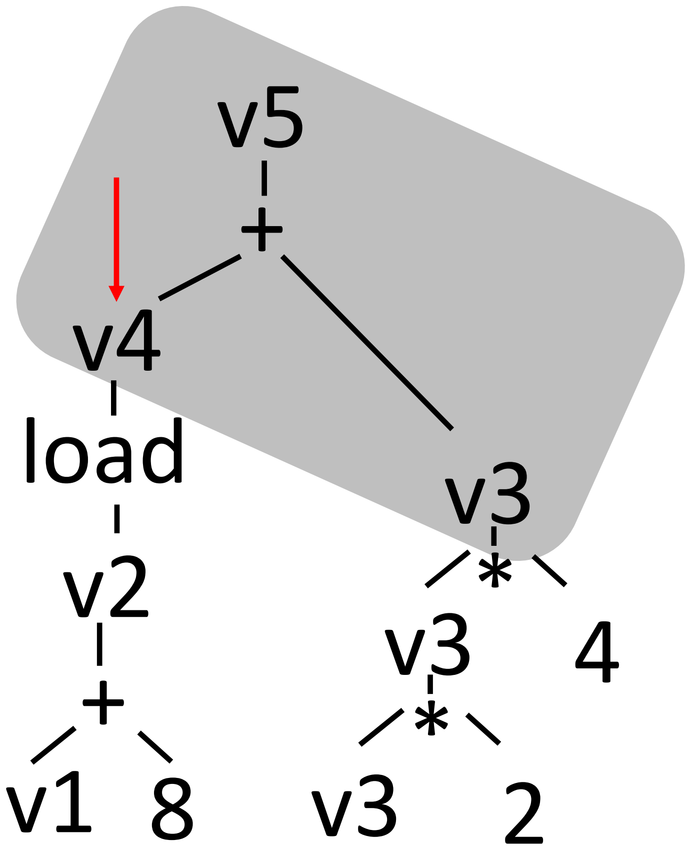


Cost: 2

VAR3 <- VAR1
VAR3 *= VAR2

```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
```

Maximal munch example



%v2 <- %v1 + 8

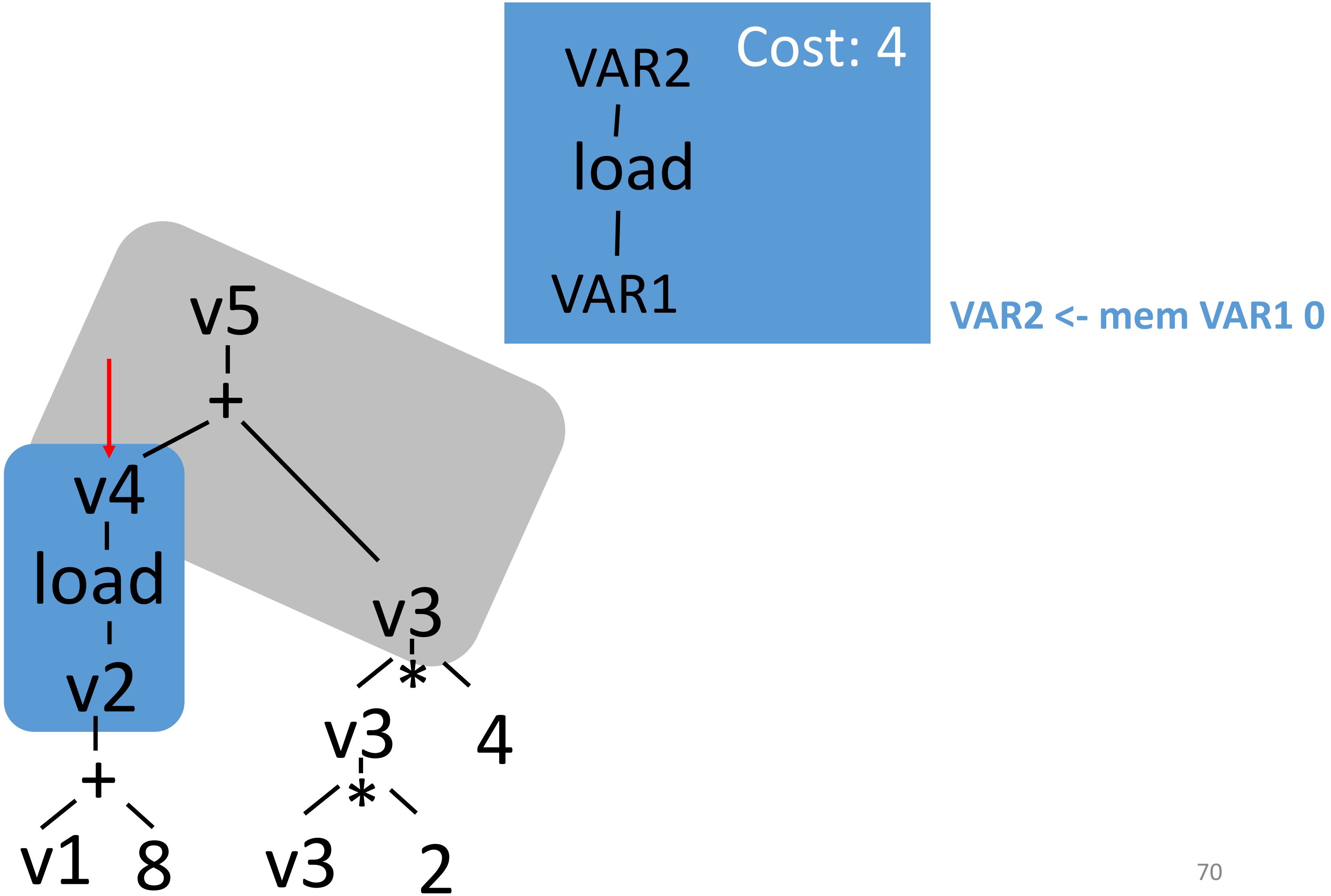
%v3 <- %v3 * 2

%v3 <- %v3 * 4

%v4 <- load %v2

%v5 <- %v4 + %v3

Maximal munch example



%v2 <- %v1 + 8

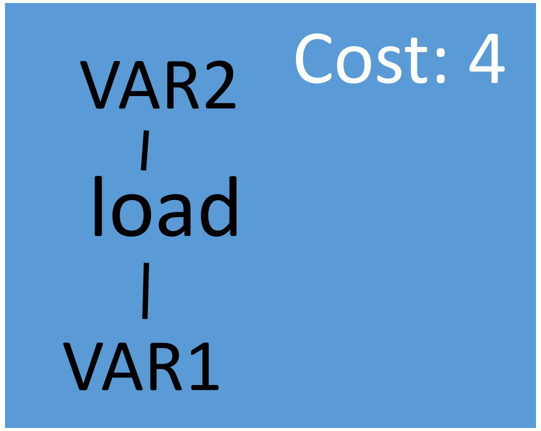
%v3 <- %v3 * 2

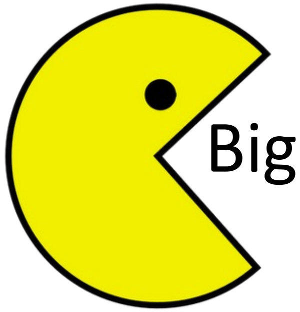
%v3 <- %v3 * 4

%v4 <- load %v2

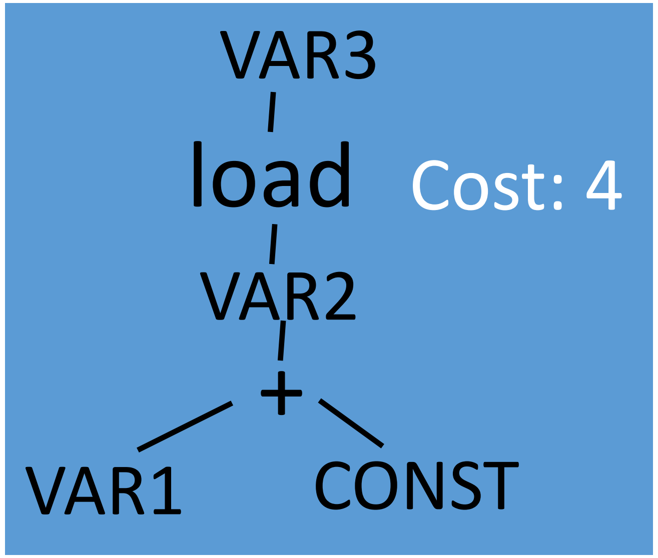
%v5 <- %v4 + %v3

Maximal munch example

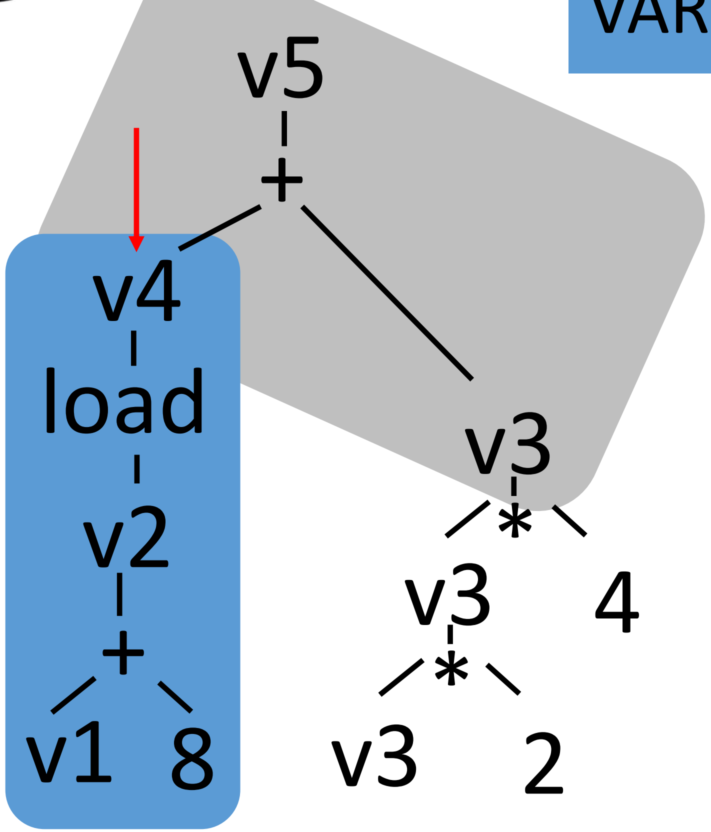




Biggest munch!

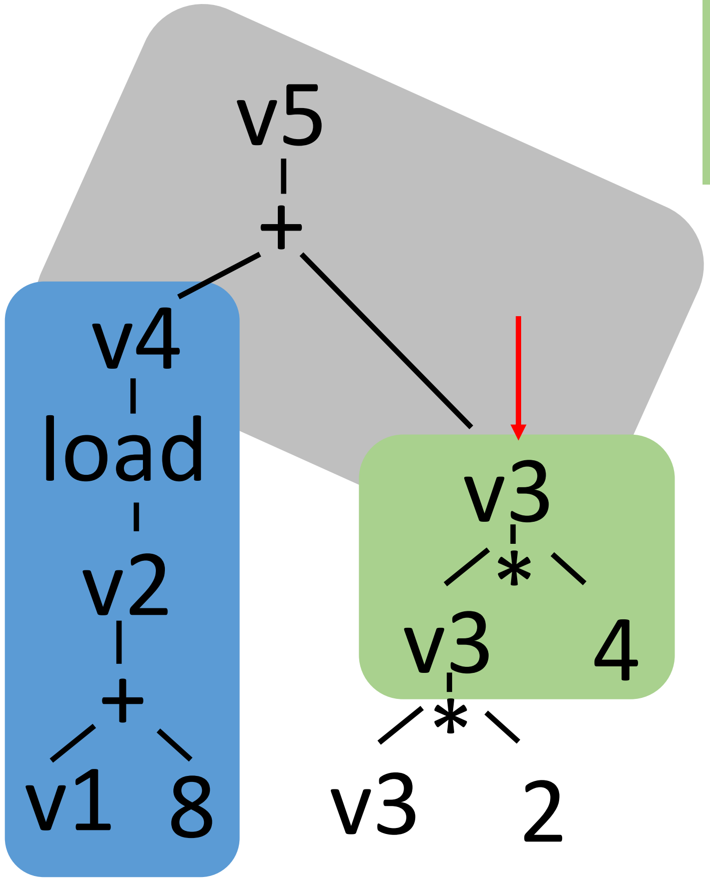


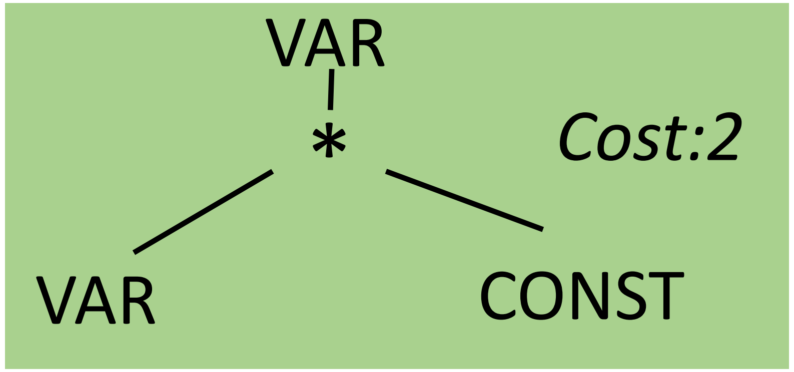
var3 <- mem var1 CONST



```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
```

Maximal munch example





var *= CONST

%v2 <- %v1 + 8

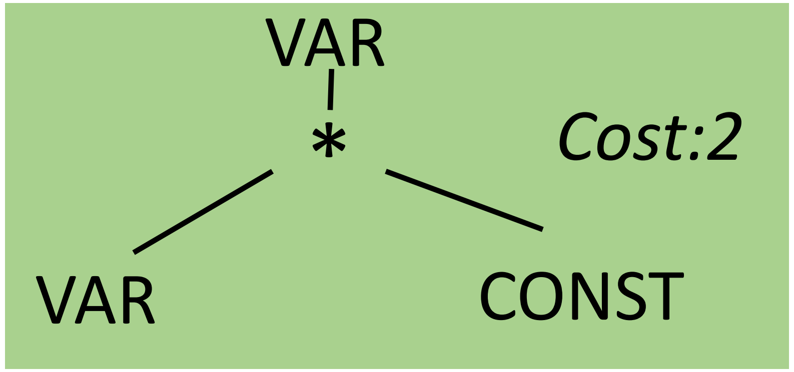
%v3 <- %v3 * 2

%v3 <- %v3 * 4

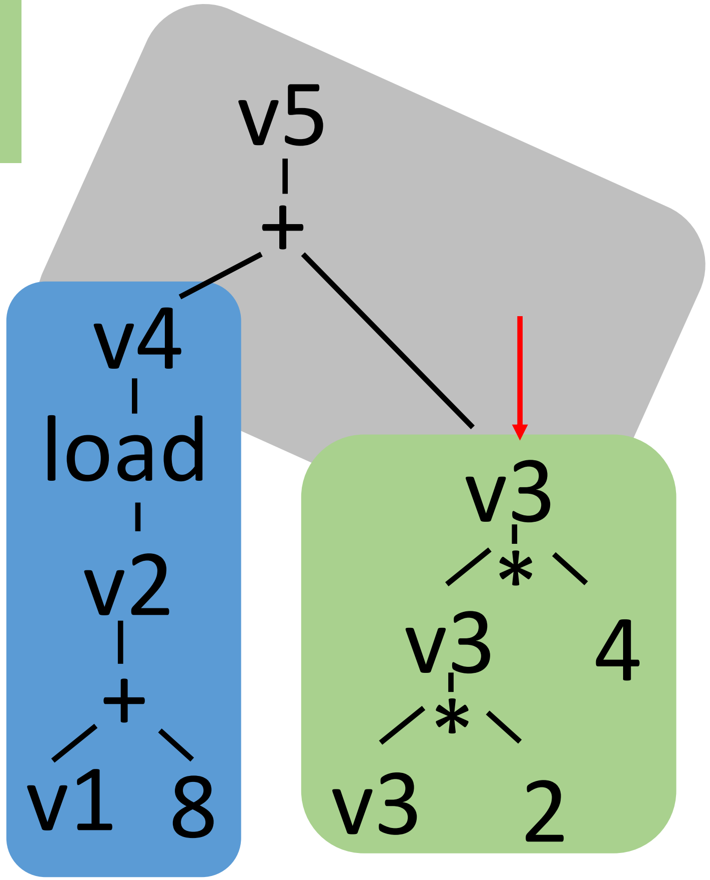
%v4 <- load %v2

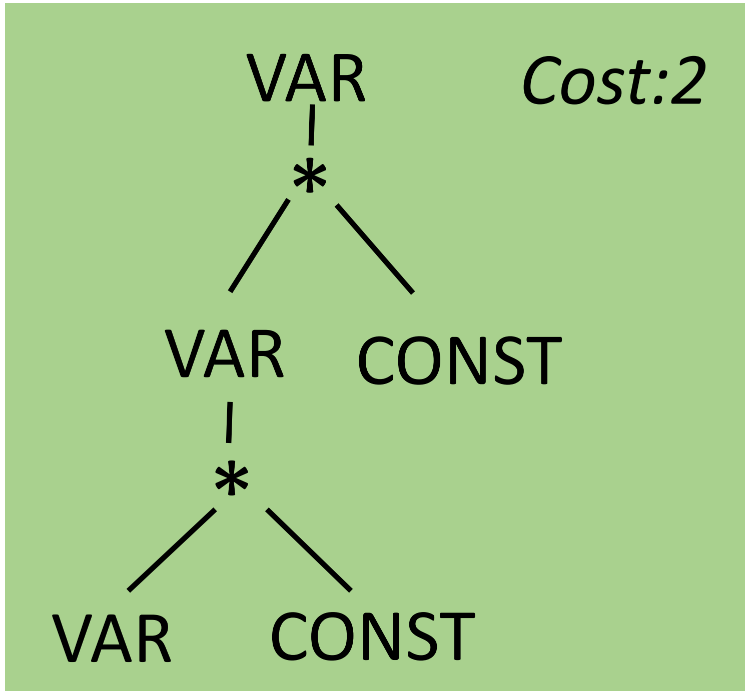
%v5 <- %v4 + %v3

Maximal munch example



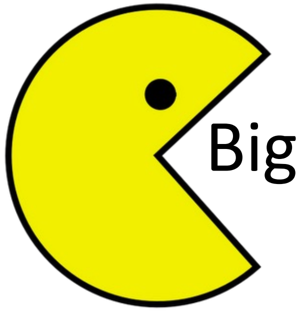
var *= CONST





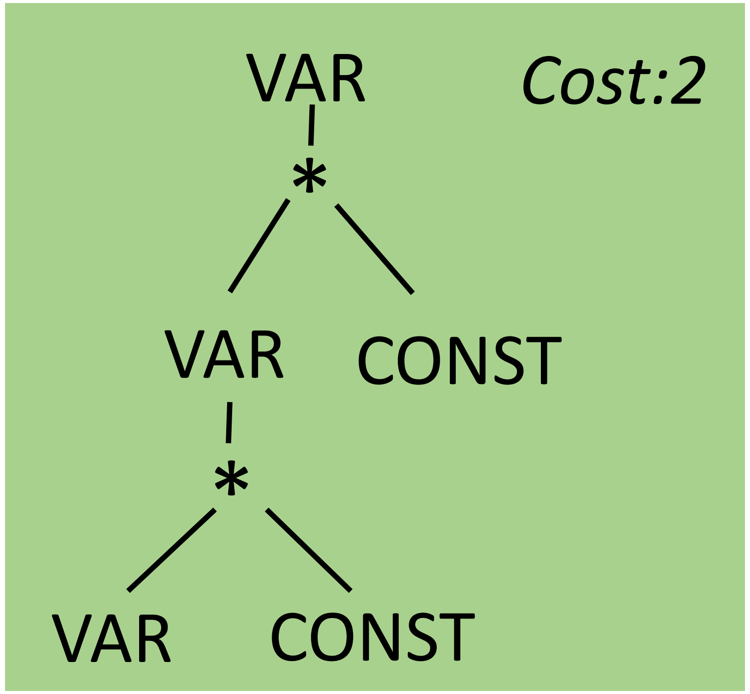
var *= CONST'

```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
```



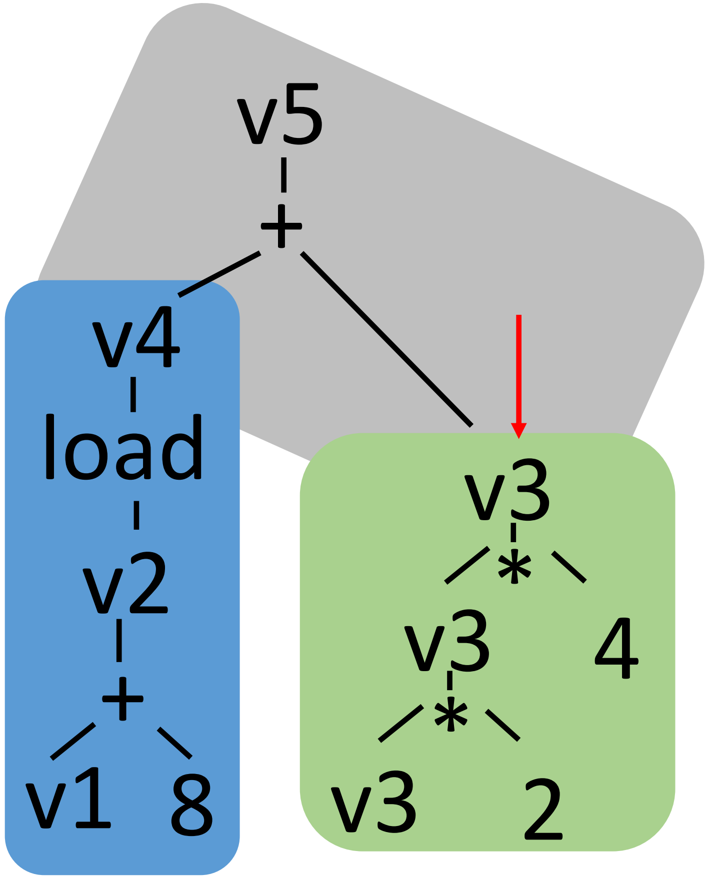
Biggest munch!

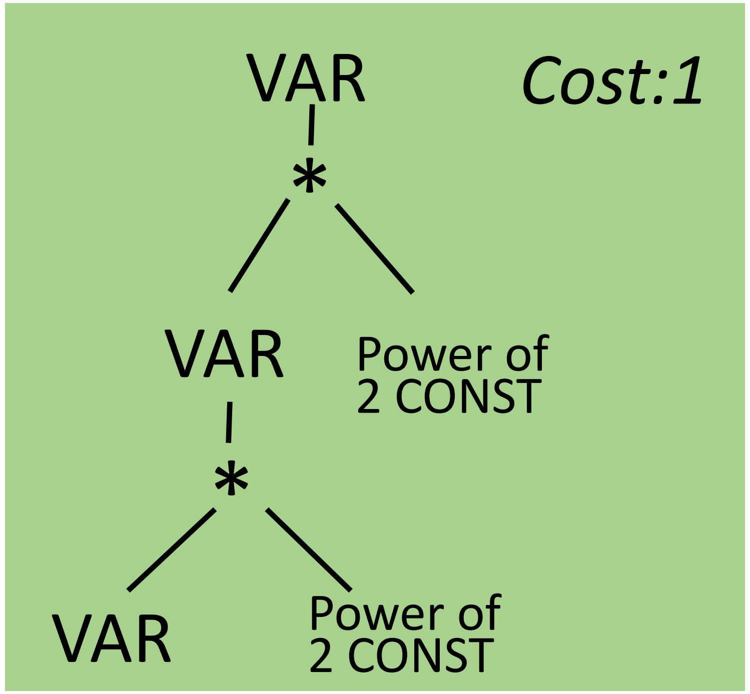
Maximal munch example



var *= CONST’

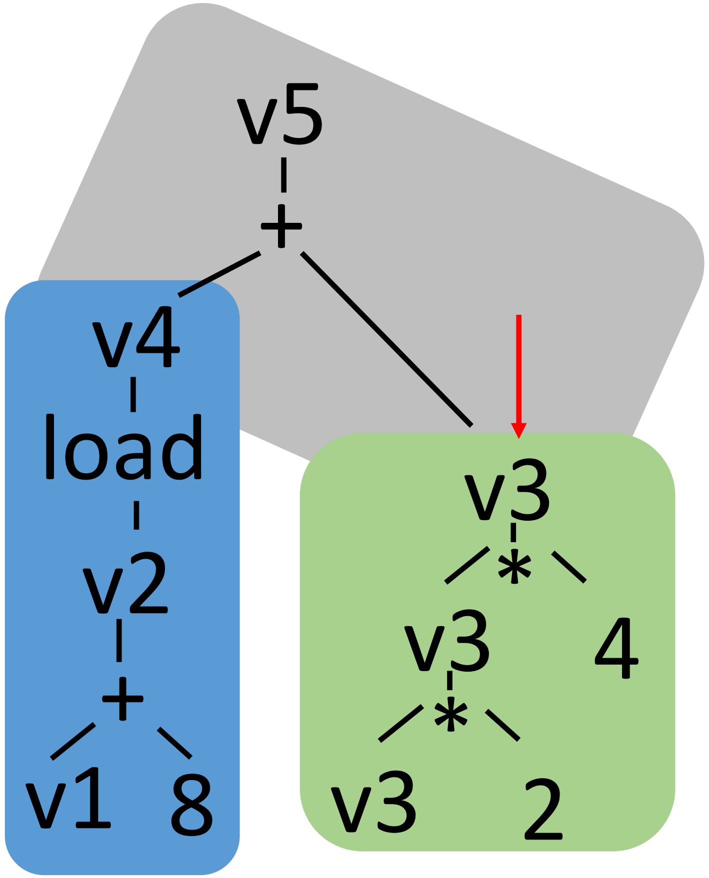
```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
```





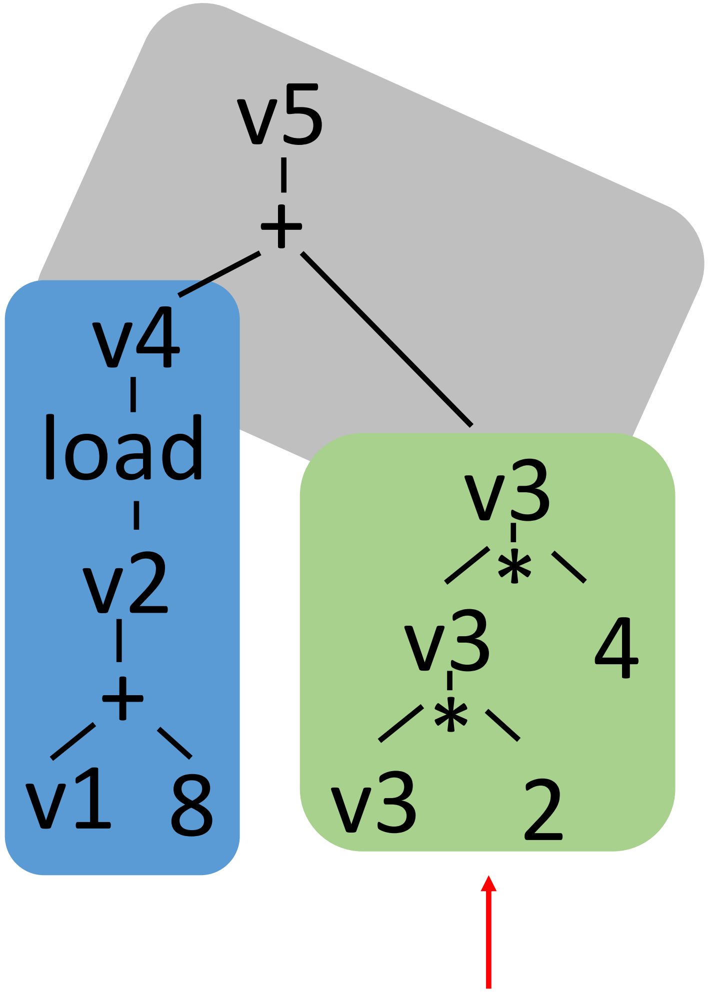
var <<= CONST’

Maximal munch example



```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
```

Maximal munch example



%v3 <<= 8

%v2 <- %v1 + 8

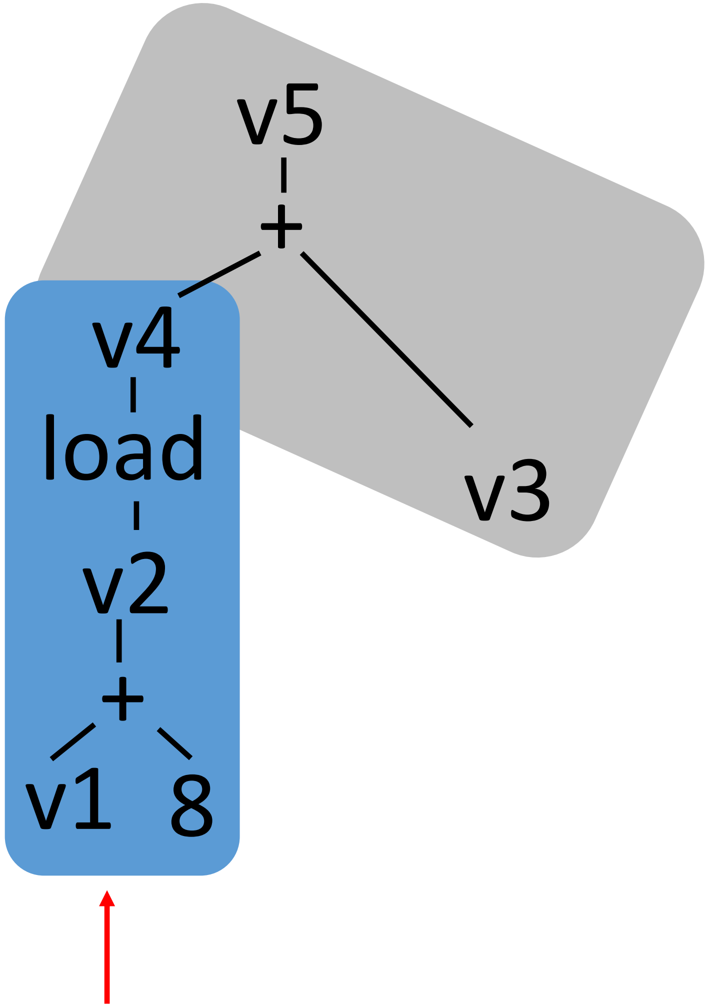
%v3 <- %v3 * 2

%v3 <- %v3 * 4

%v4 <- load %v2

%v5 <- %v4 + %v3

Maximal munch example



%v3 <<= 8

%v2 <- %v1 + 8

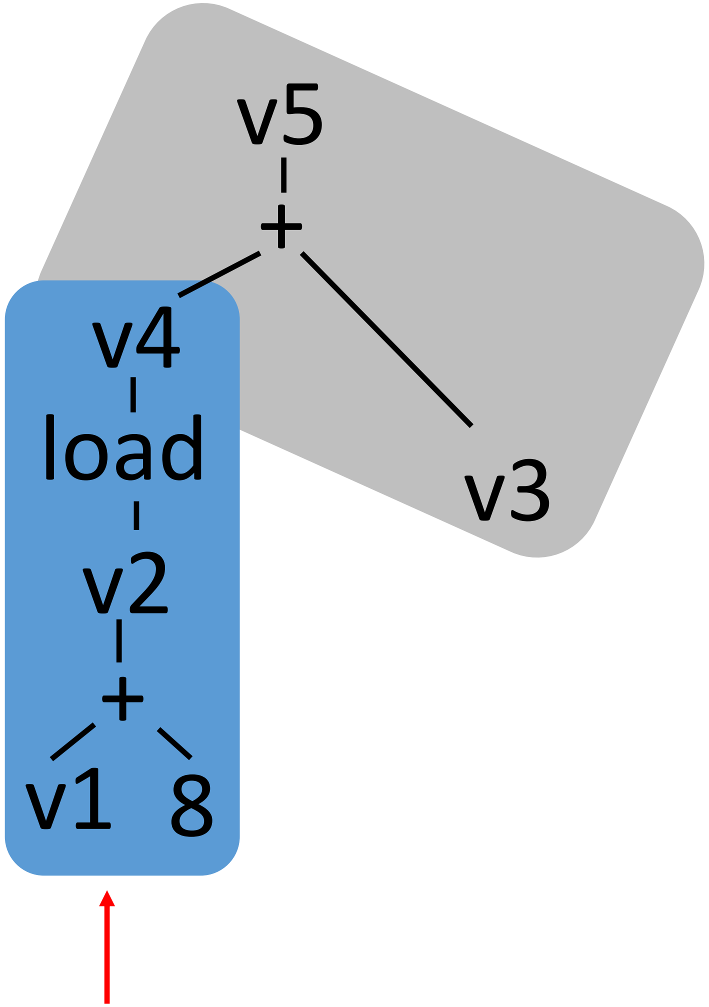
%v3 <- %v3 * 2

%v3 <- %v3 * 4

%v4 <- load %v2

%v5 <- %v4 + %v3

Maximal munch example



%v3 <<= 8

%v4 <- mem %v1 8

%v2 <- %v1 + 8

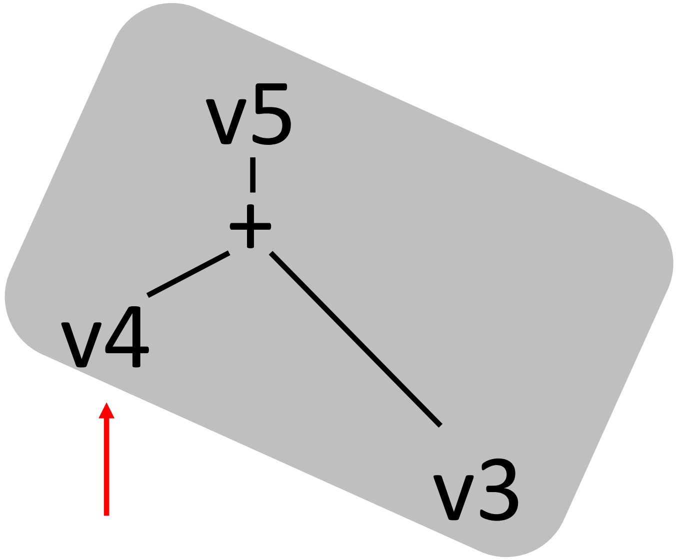
%v3 <- %v3 * 2

%v3 <- %v3 * 4

%v4 <- load %v2

%v5 <- %v4 + %v3

Maximal munch example

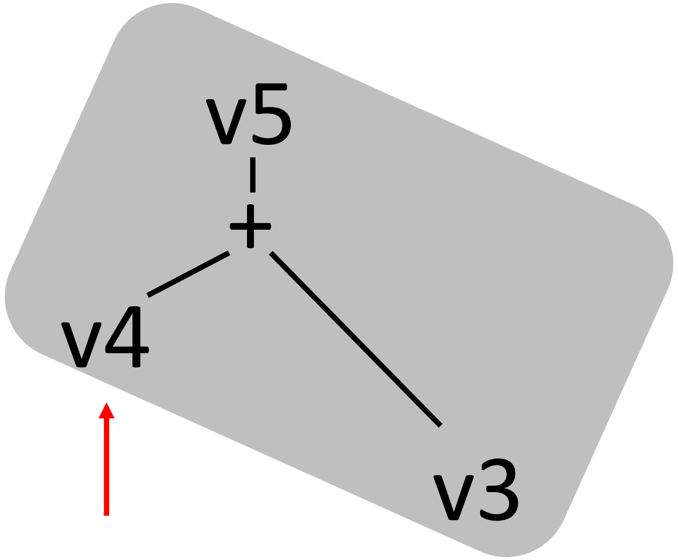


%v3 <<= 8

%v4 <- mem %v1 8

```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
```

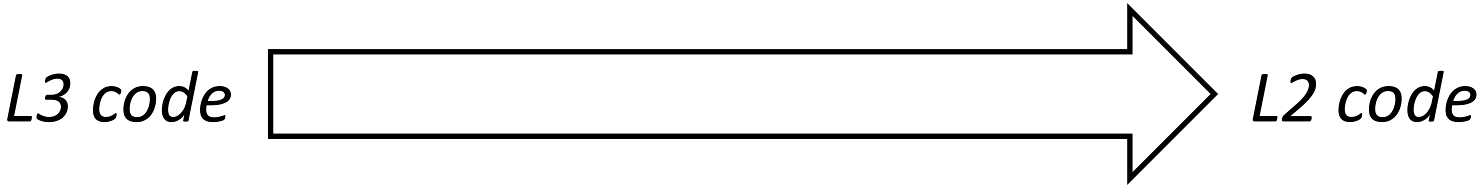
Maximal munch example



```
%v3 <<= 8
%v4 <- mem %v1 8
%v5 <- %v4
%v5 += %v3
```

```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
```

Maximal munch example



L3 code

L2 code

```
%v2 <- %v1 + 8
%v3 <- %v3 * 2
%v3 <- %v3 * 4
%v4 <- load %v2
%v5 <- %v4 + %v3
```

```
%v3 <<= 8
%v4 <- mem %v1 8
%v5 <- %v4
%v5 += %v3
```

Maximal munch

- Maximal munch does not necessarily produce the optimum selection of instructions
- But:
  - it is easy to implement
  - it tends to work “well”
    for current instruction-set architectures

… but if we want the optimum?

Instruction selection complexity

- Finding the optimum for tree: P

- Finding the optimum for DAG: NP
  - Countless number of heuristics proposed (including the one described in this class)
  - Dynamic programming

- Most (all) of programs we run are DAGs

Homework #3: the L3 compiler

For every L3 function f

L3 function f

Label globalization

Instruction selection

Excluding **only** Instruction selection step 3: merging trees

API -> ABI

L2 function

Always have faith in your ability

Success will come your way eventually

Best of luck!