Compilation (#6a) : SSA ENSLOnly

Yannick Zakowski Gabriel Radanne

Master 1, ENS de Lyon et Dpt Info, Lyon1

2022-2023

Credits

Source http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/StaticSingleAssignment.pdf

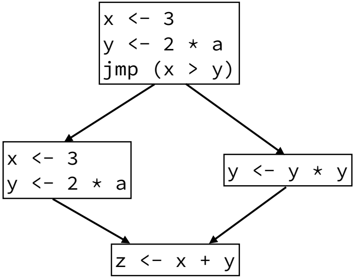
- The SSA book (collective)
- Modern Compiler Implementation in C/Java/ML (Andrew Appel)
- Fernando Magno Quintao Pereira's course https://www.youtube.com/user/pronesto/videos
- Adrian Sampson's course https://www.cs.cornell.edu/courses/cs6120/2020fa/

Motivation: It's all about information

Compilers alternate between two tasks:

1. *computing* some information (invariants) of the program
2. *using* this information to justify some program transformations

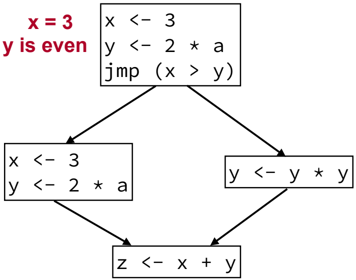
Dataflow analyses associate facts to every program point:
* a fact is *associated to a definition-site* of a variable
* a fact is *exploited at a use-site* of a variable



Motivation: It's all about information

Compilers alternate between two tasks:
1. *computing* some information (invariants) of the program
2. *using* this information to justify some program transformations

Dataflow analyses associate facts to every program point:
* a fact is *associated to a definition-site* of a variable
* a fact is *exploited at a use-site* of a variable

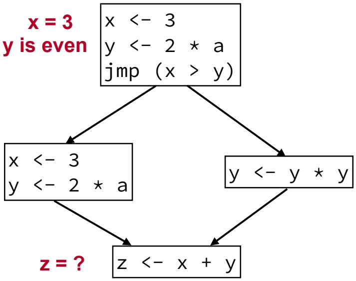


Motivation: It’s all about information

Compilers alternate between two tasks:

1. *computing* some information (invariants) of the program
2. *using* this information to justify some program transformations

Dataflow analyses associate facts to every program point:
* a fact is *associated to a definition-site* of a variable
* a fact is *exploited at a use-site* of a variable



What do we know about x and y?

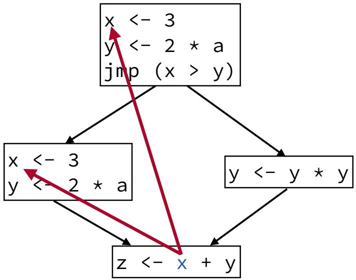
Motivation: It's all about information

Compilers alternate between two tasks:
1. *computing* some information (invariants) of the program
2. *using* this information to justify some program transformations

Dataflow analyses associate facts to every program point:
* a fact is *associated to a definition-site* of a variable
* a fact is *exploited at a use-site* of a variable

One solution: use a data-structure, the *def-use and use-def chains*

M def and N use of a variable: O(N * M) space and time

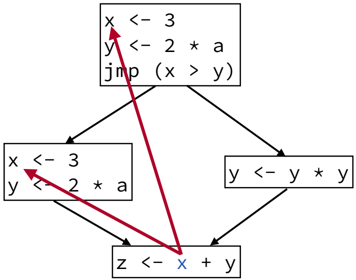


Motivation: It's all about information

Compilers alternate between two tasks:

1. *computing* some information (invariants) of the program
2. *using* this information to justify some program transformations

Dataflow analyses associate facts to every program point:
* a fact is *associated to a definition-site* of a variable
* a fact is *exploited at a use-site* of a variable



One solution: use a data-structure, the *def-use and use-def chains*

M def and N use of a variable: O(N * M) space and time

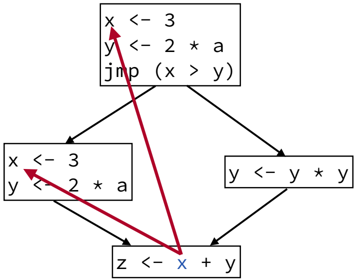
Could we enforce this structure to be trivial by definition? Sure, let's have M = 1!

Motivation: It’s all about information

Compilers alternate between two tasks:
1. *computing* some information (invariants) of the program
2. *using* this information to justify some program transformations

Dataflow analyses associate facts to every program point:
* a fact is *associated to a definition-site* of a variable
* a fact is *exploited at a use-site* of a variable

One solution: use a data-structure, the *def-use and use-def chains*



M def and N use of a variable: O(N * M) space and time

Could we enforce this structure to be trivial by definition? Sure, let’s have M = 1!

We want to enforce an invariant by construction: we want an *intermediate representation*

Single Static Assignment (SSA)

Each variable has exactly one definition in the syntax[1]

Use-def chains are explicit in the syntax of the program -> Many optimizations are simplified

[1] Dynamically, it can be defined many times: it is not “Single Assignment”!

Single Static Assignment (SSA)

Each variable has exactly one definition in the syntax[1]

Use-def chains are explicit in the syntax of the program -> Many optimizations are simplified

Introduced in 1988:
"Global value numbers and redundant computations" by Rosen, Wegman and Zadeck

Used in most modern compilers: GCC, llvm, HotSpot…

We will consider here more specifically Control Flow Graphs in SSA form

[1] Dynamically, it can be defined many times: it is not "Single Assignment"!

Converting to SSA form: informally

Converting to SSA form: informally

Each variable has exactly one definition in the syntax

Converting straight code

bl:

```
a  <- x  + 1
b  <- a  * 2
a  <- a  + b
c  <- x  * a
b  <- c  - 1
```

Converting straight code

```
bl:
a <- x + 1
b <- a * 2
a <- a + b
c <- x * a
b <- c - 1
```

Converting straight code

```
bl:
a <- x + 1
b <- a * 2
a <- a + b
c <- x * a
b <- c - 1
```

→

```
bl:
a1 <- x  + 1
b1 <- a1 * 2
a2 <- a1 + b1
c1 <- x  * a2
b2 <- c1 - 1
```

Converting straight code

```
bl:
a <- x + 1
b <- a * 2
a <- a + b
c <- x * a
b <- c - 1
```

```
bl:
a1 <- x  + 1
b1 <- a1 * 2
a2 <- a1 + b1
c1 <- x  * a2
b2 <- c1 - 1
```

Rule 1: use a fresh index at each def-site

Converting straight code

```
bl:
a <- x + 1
b <- a * 2
a <- a + b
c <- x * a
b <- c - 1
```

→

```
bl:
a1 <- x  + 1
b1 <- a1 * 2
a2 <- a1 + b1
c1 <- x  * a2
b2 <- c1 - 1
```

Rule 1: use a fresh index at each def-site

Converting straight code

```
bl:
a <- x + 1
b <- a * 2
a <- a + b
c <- x * a
b <- c - 1
```

```
bl:
a1 <- x  + 1
b1 <- a1 * 2
a2 <- a1 + b1
c1 <- x  * a2
b2 <- c1 - 1
```

Rule 1: use a fresh index at each def-site

Rule 2: use the most recent definition at each use-site

Converting disjunction points

```
bi:
x <- 3 * y
jmp (x = 0)
```

```
bl:
a  <- x  + 1
b  <- a  * 2
a  <- a  + b
c  <- x  * a
b  <- c  - 1
```

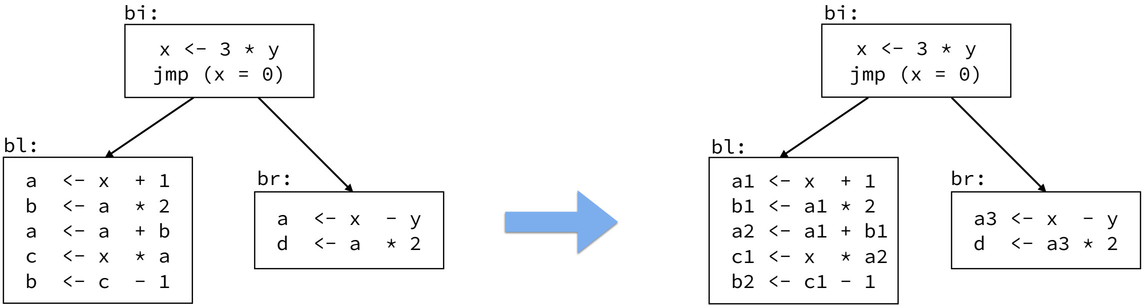
```
br:
a  <- x  - y
d  <- a  * 2
```

```
a1 <- x  + 1
b1 <- a1 * 2
a2 <- a1 + b1
c1 <- x  * a2
b2 <- c1 - 1
```

Rule 1: use a fresh index at each def-site

Rule 2: use the most recent definition at each use-site

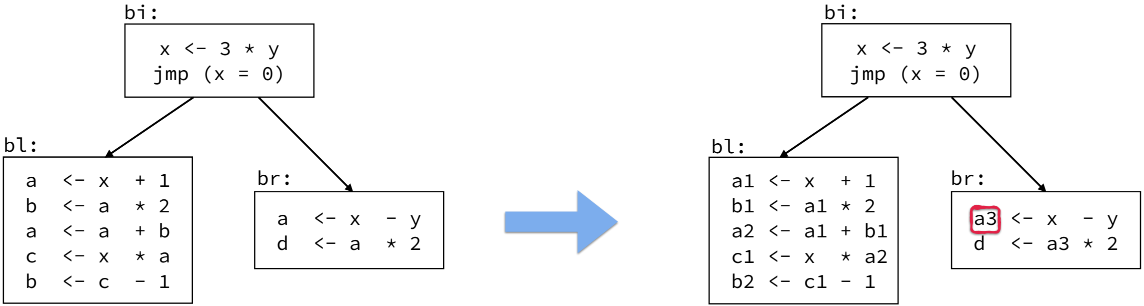
Converting disjunction points



Rule 1: use a fresh index at each def-site

Rule 2: use the most recent definition at each use-site

Converting disjunction points

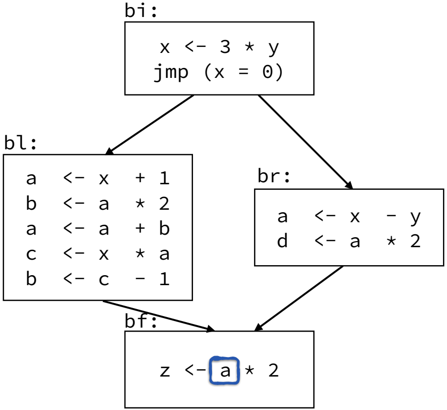


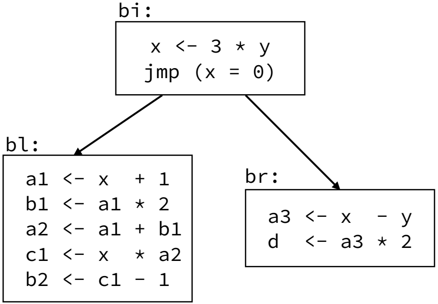
Freshness is global

Rule 1: use a fresh index at each def-site

Rule 2: use the most recent definition at each use-site

Converting merging points: Φ-nodes

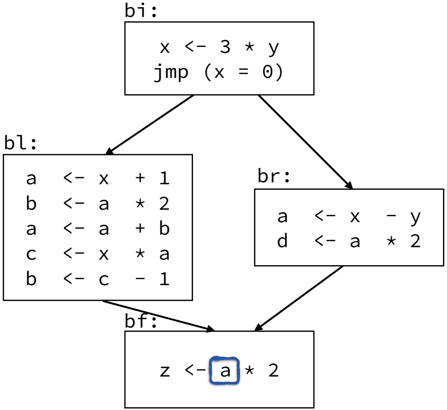


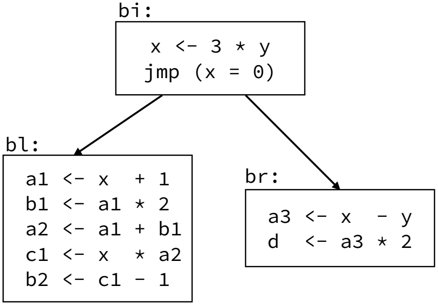


Rule 1: use a fresh index at each def-site

Rule 2: use the most recent definition at each use-site

Converting merging points: Φ-nodes



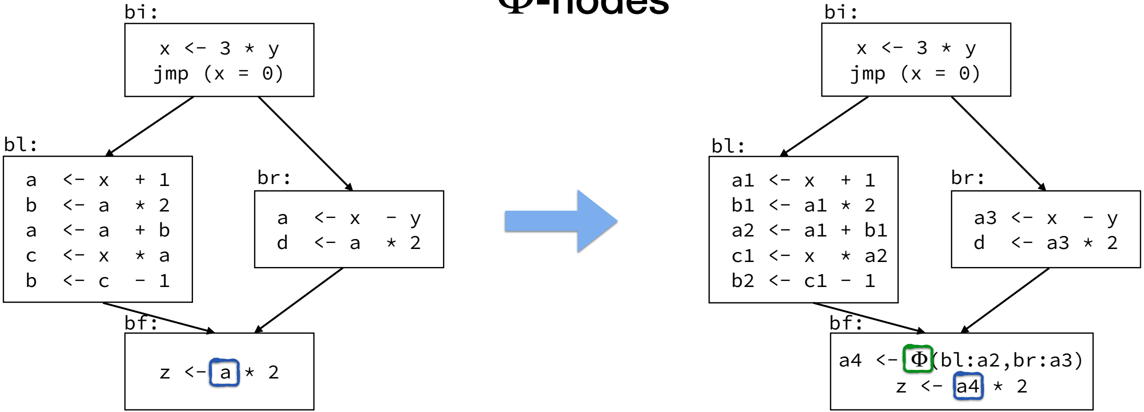


Rule 1: use a fresh index at each def-site

Rule 2: use the most recent definition at each use-site

?

Converting merging points: Φ-nodes

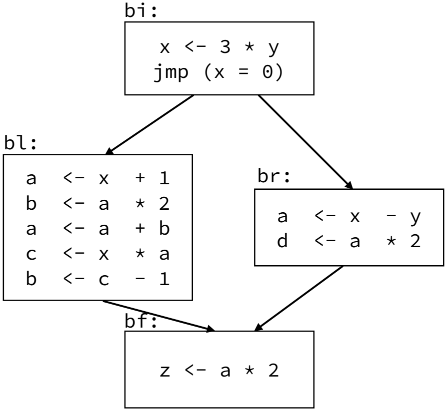


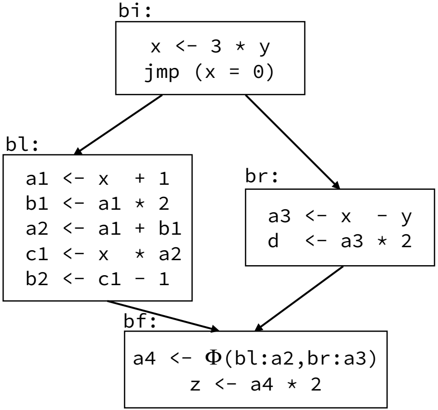
Rule 1: use a fresh index at each def-site

Rule 2: use the most recent definition at each use-site

Rule 3: at merge points, introduce Φ-nodes

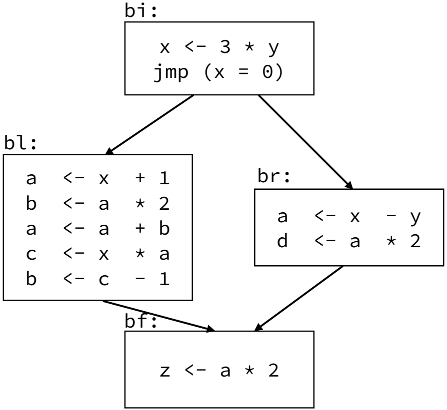
Converting merging points: Φ-nodes

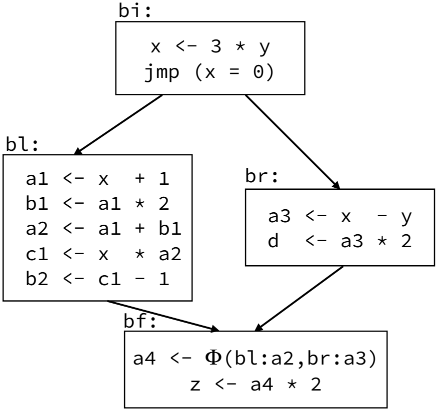




Goal: to decide when to introduce Φ-nodes

Converting merging points: Φ-nodes

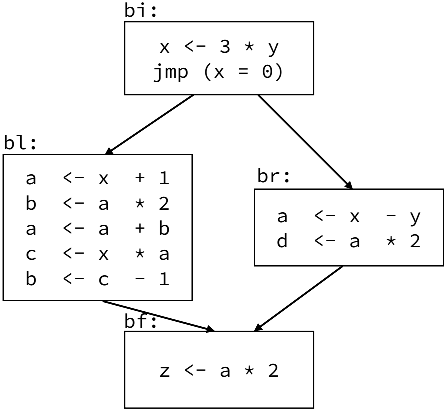


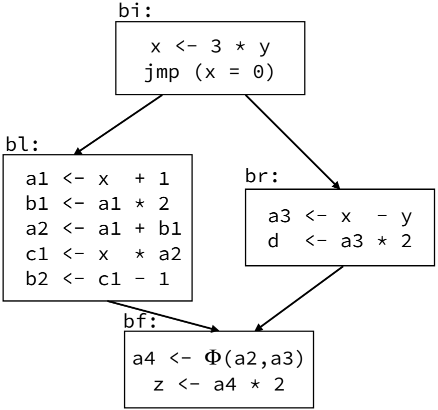


Goal: to decide when to introduce Φ-nodes

One per variable at every join point?

Converting merging points: Φ-nodes





Goal: to decide when to introduce *few* Φ-nodes

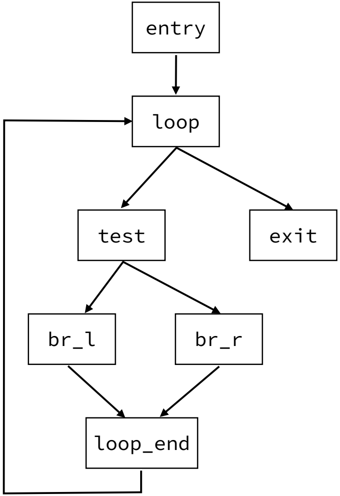
~~One per variable at every join point?~~

Converting to SSA form: an algorithm

Converting to SSA form: an algorithm

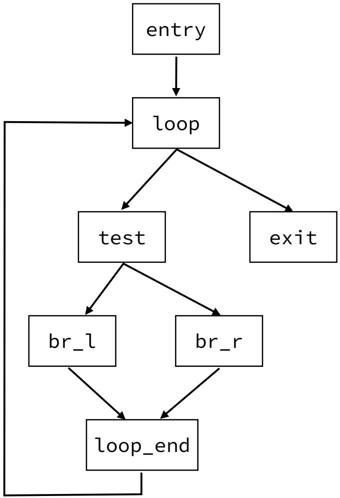
Goal: to decide when to introduce *few* Φ-nodes

The domination relation



A dominates B if any path from `entry` to B contains A

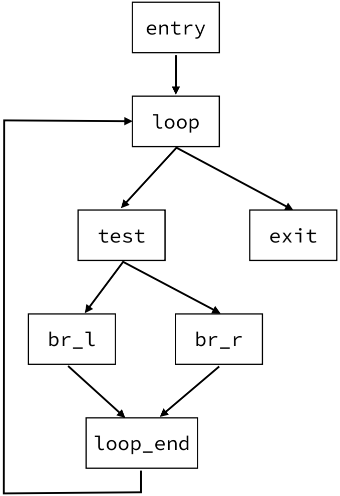
The domination relation



A dominates B if any path from `entry` to B contains A

Can you annotate the nodes with their dominators?

The domination relation

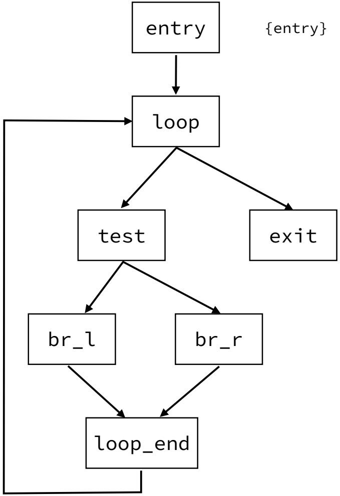


A dominates B if any path from `entry` to B contains A

The domination relation

It's reflexive

A dominates B if any path from `entry` to B contains A

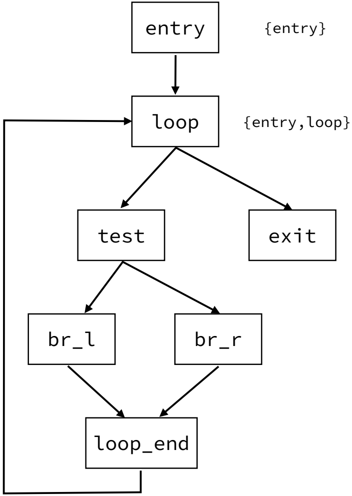


{entry}

The domination relation

It's reflexive

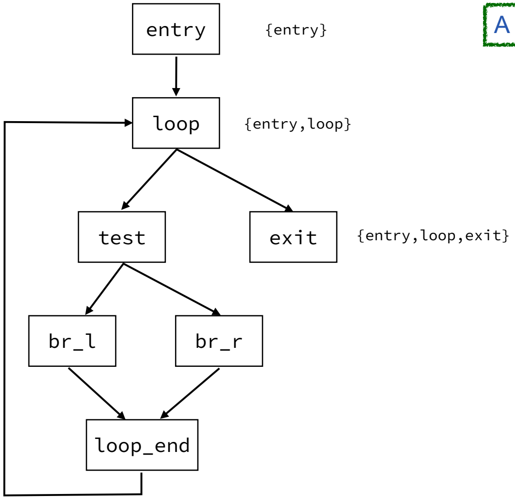
A dominates B if any path from `entry` to B contains A



The domination relation

It's reflexive

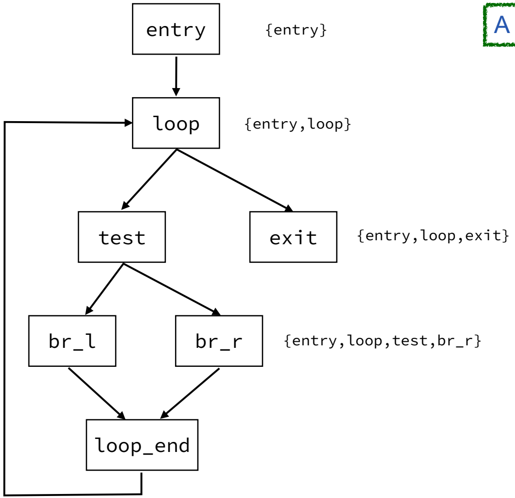
A dominates B if any path from `entry` to B contains A



The domination relation

It's reflexive

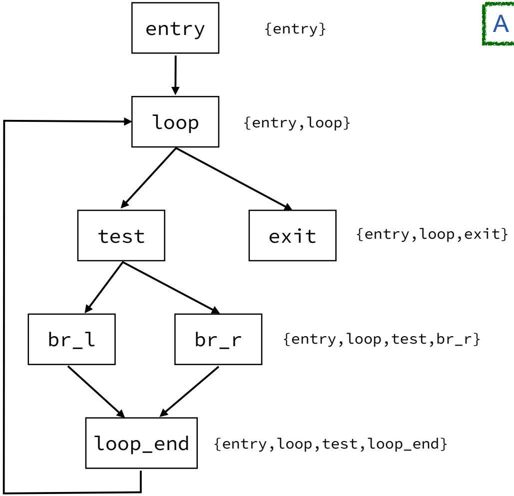
A dominates B if any path from `entry` to B contains A



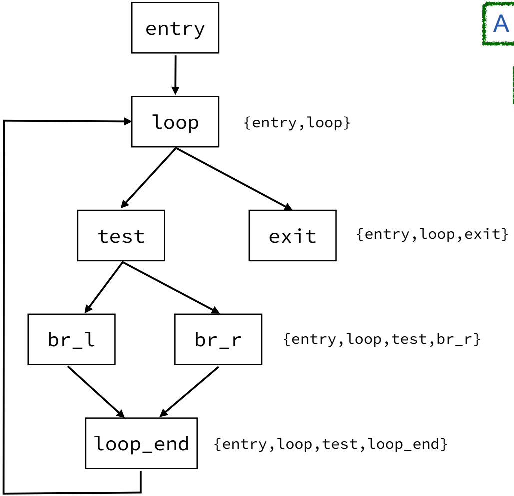
The domination relation

It's reflexive

A dominates B if any path from `entry` to B contains A



The domination relation



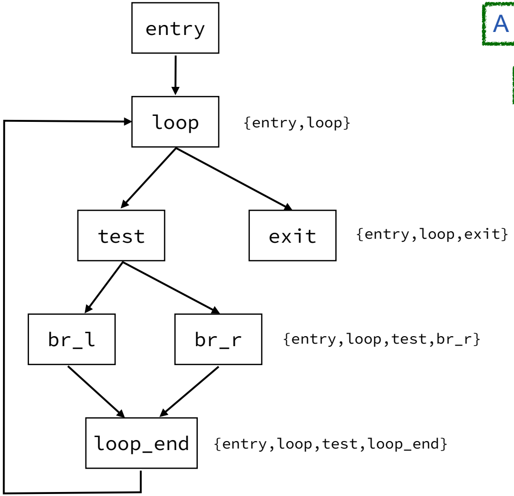
A dominates B if any path from `entry` to B goes through A

A *strictly dominates* B if A dominates B and A is not B

The *domination tree* stores the domination relation
A is parent to B if:

- A strictly dominates B
- A does not strictly dominates any C that strictly dominates B

The domination relation



A dominates B if any path from `entry` to B goes through A

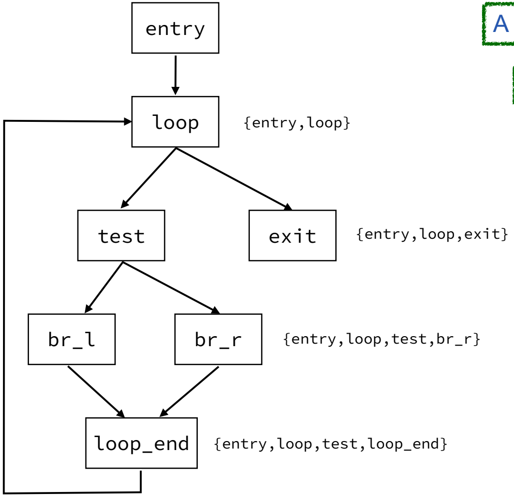
A *strictly dominates* B if A dominates B and A is not B

The *domination tree* stores the domination relation
A is parent to B if:

- A strictly dominates B
- A does not strictly dominates any C that strictly dominates B

Can you build the domination tree of the CFG to the left?

The domination relation



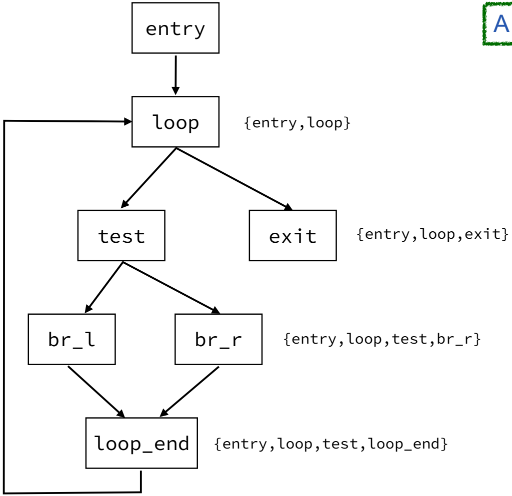
A dominates B if any path from `entry` to B goes through A

A *strictly dominates* B if A dominates B and A is not B

The *domination tree* stores the domination relation
A is parent to B if:

- A strictly dominates B
- A does not strictly dominates any C that strictly dominates B

The domination relation

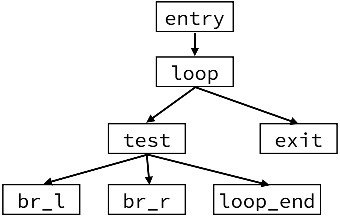


A dominates B if any path from `entry` to B goes through A

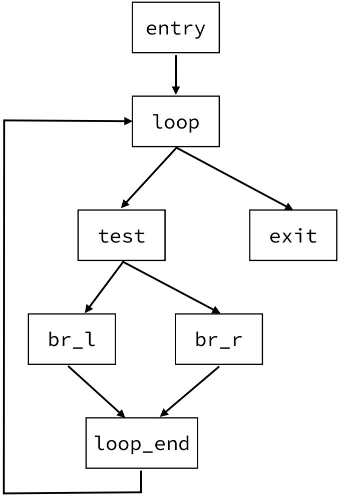
A *strictly dominates* B if A dominates B and A is not B

The *domination tree* stores the domination relation
A is parent to B if:

- A strictly dominates B
- A does not strictly dominates any C that strictly dominates B



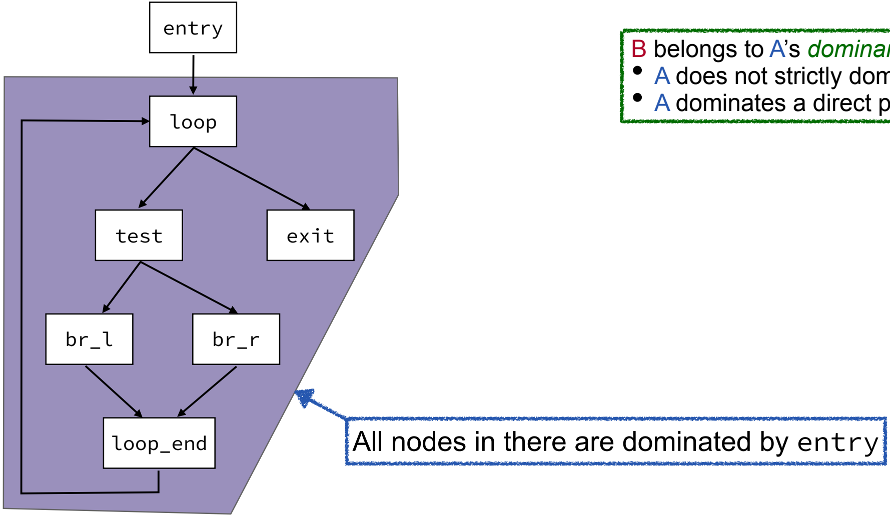
Dominance frontier



B belongs to A's *dominance frontier* if:

- A does not strictly dominate B
- A dominates a direct predecessor of B

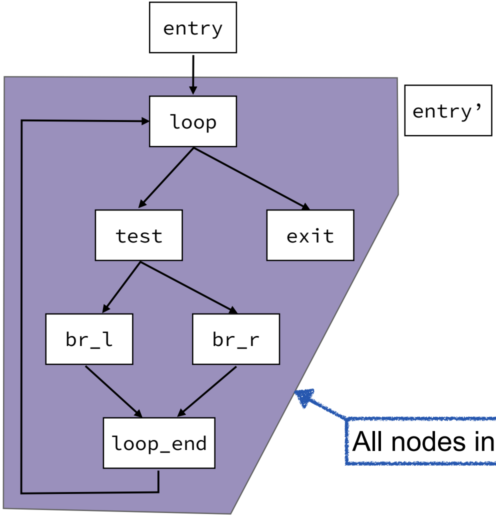
Dominance frontier



B belongs to A's *dominance frontier* if:

- A does not strictly dominate B
- A dominates a direct predecessor of B

Dominance frontier

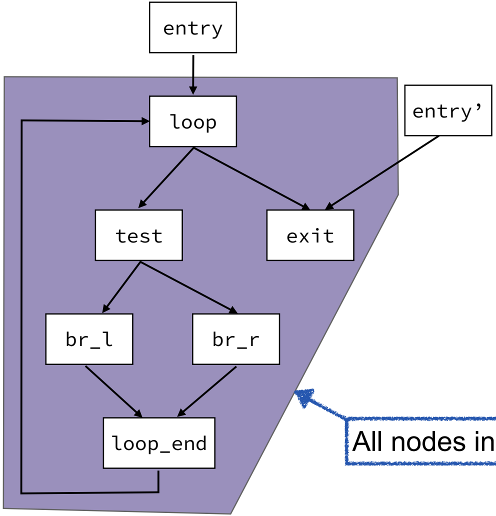


B belongs to A’s *dominance frontier* if:

- A does not strictly dominate B
- A dominates a direct predecessor of B

All nodes in there are dominated by `entry`

Dominance frontier

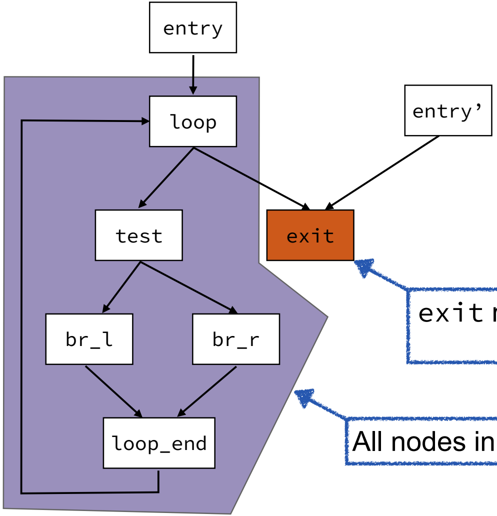


B belongs to A's *dominance frontier* if:

- A does not strictly dominate B
- A dominates a direct predecessor of B

All nodes in there are dominated by `entry`

Dominance frontier



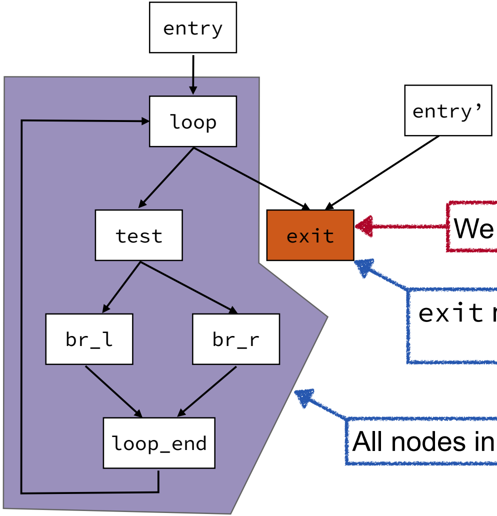
B belongs to A's *dominance frontier* if:

- A does not strictly dominate B
- A dominates a direct predecessor of B

`exit` now barely escapes the influence of `entry`: it is in its dominance frontier

All nodes in there are dominated by `entry`

Dominance frontier



B belongs to A's *dominance frontier* if:

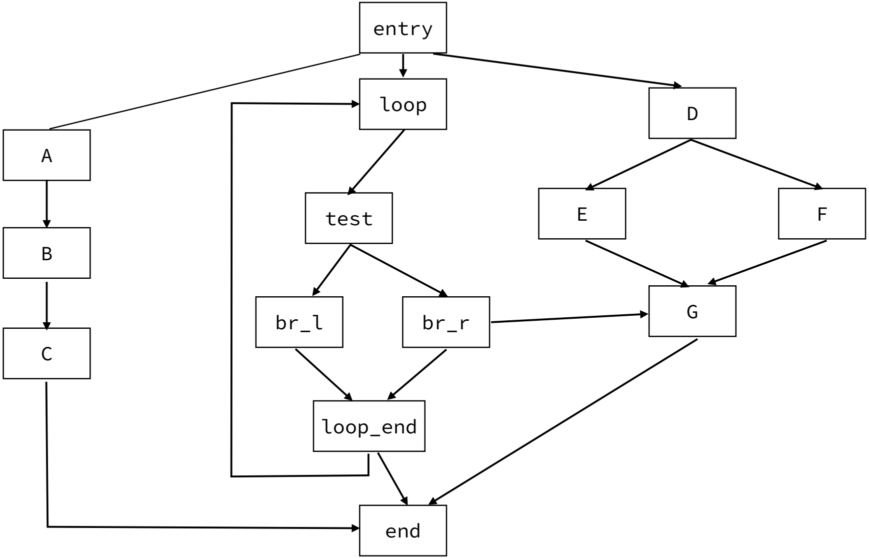
- A does not strictly dominate B
- A dominates a direct predecessor of B

We need a Φ-node right there!

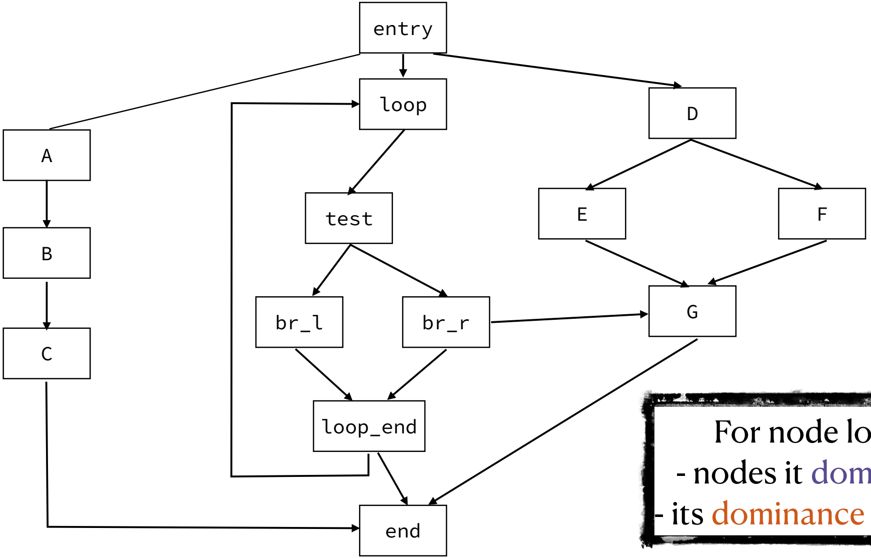
`exit` now barely escapes the influence of `entry`: it is in its dominance frontier

All nodes in there are dominated by `entry`

Dominance frontier

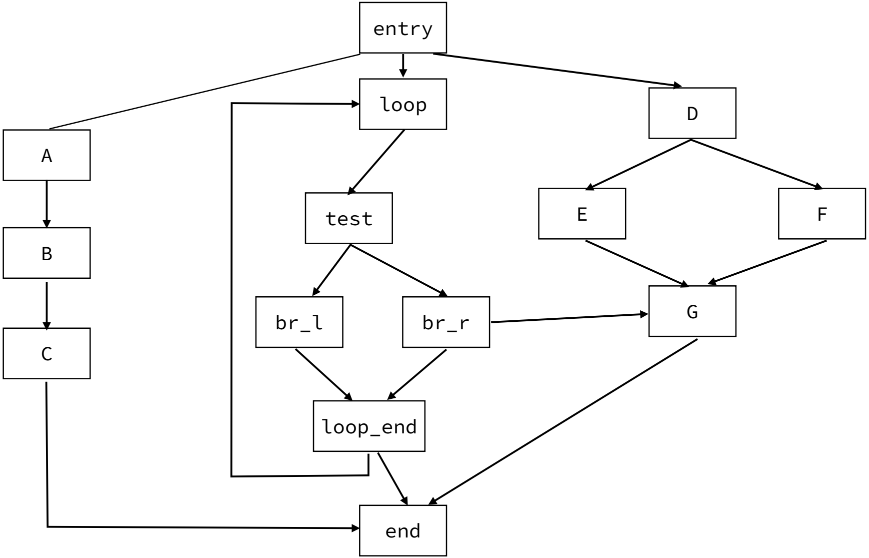


Dominance frontier

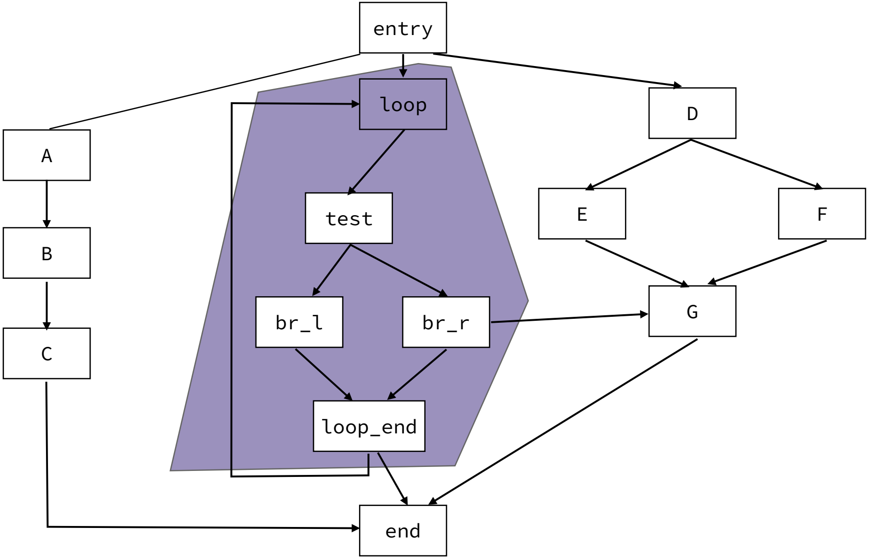


For node loop:
- nodes it dominates
- its dominance frontier

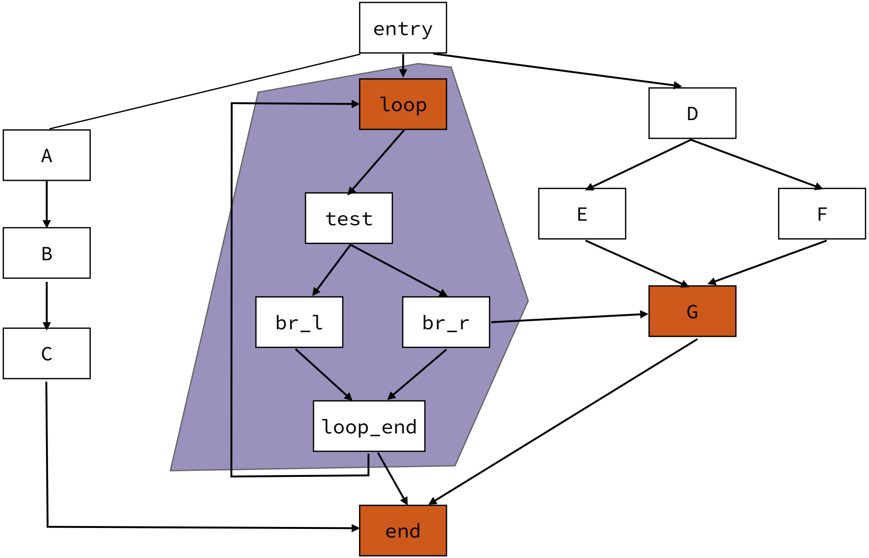
Dominance frontier



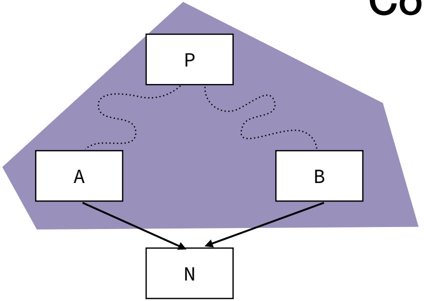
Dominance frontier



Dominance frontier

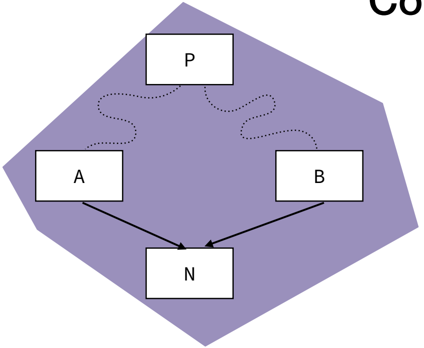


Computing dominators



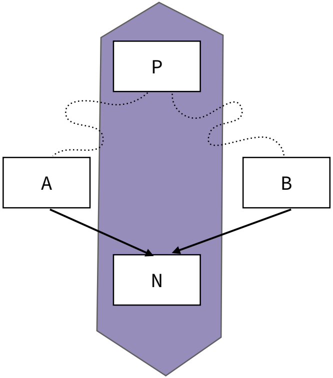
If P dominates A and B, then P dominates N

Computing dominators



If P dominates A and B, then P dominates N

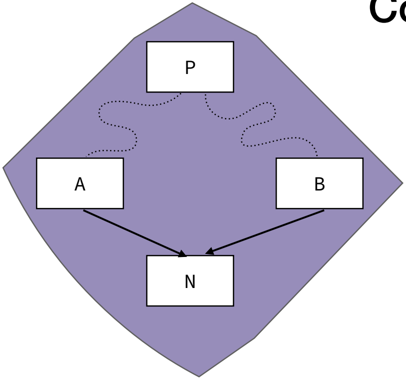
Computing dominators



If P dominates A and B, then P dominates N

If P dominates N, then P dominates A and B

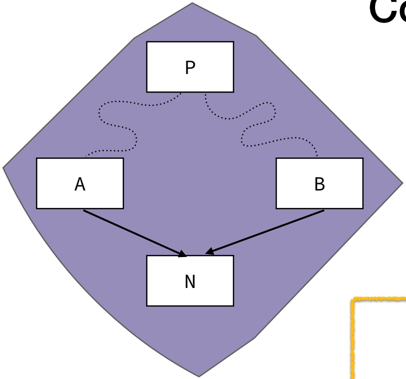
Computing dominators



If P dominates A and B, then P dominates N

If P dominates N, then P dominates A and B

Computing dominators



If P dominates A and B, then P dominates N

If P dominates N, then P dominates A and B

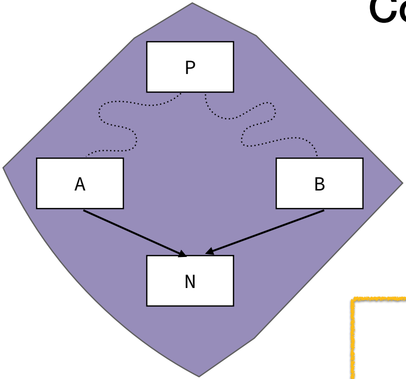
Let D[n] be the set of nodes dominating n

$$D[\text{entry}] \triangleq \{\text{entry}\}$$

$$D[n] \triangleq \{n\} \cup \Big(\bigcap_{p \in \mathit{pred}(n)} D[p] \Big)$$

As is traditional, this system of equations can be solve by iteration[1]

Computing dominators



If P dominates A and B, then P dominates N

If P dominates N, then P dominates A and B

Let D[n] be the set of nodes dominating n

$$D[\text{entry}] \triangleq \{\text{entry}\}$$

$$D[n] \triangleq \{n\} \cup \left(\bigcap_{p \in pred(n)} D[p] \right)$$

As is traditional, this system of equations can be solve by iteration[1]

Complexity?

[1]: For a more efficient algorithm, see Lengauer and Tarjan's 1979

"A fast algorithm for finding dominators in a flowgraph"

Computing the dominance frontier

`G` : ambient cfg
`DT`: Dominance Tree of `G`
`DF`: map from nodes to sets of nodes

```
computeDF(n) ::=
```

Computing the dominance frontier

`G` : ambient cfg
`DT`: Dominance Tree of `G`
`DF`: map from nodes to sets of nodes

"Obvious", immediate frontier

```
computeDF(n) ::=
  S <- {y | y successor of n in G but not in DT}
```

Computing the dominance frontier

`G` : ambient cfg
`DT`: Dominance Tree of `G`
`DF`: map from nodes to sets of nodes

“Obvious”, immediate frontier

```
computeDF(n) ::=
  S <- {y | y successor of n in G but not in DT}
  for c in children(n) in DT:
    computeDF(c)
    for each w in DF[c]:
      if n does not dominate w:
        S <- S ∪ {w}
```

The rest of the frontier is inherited from the other children

Computing the dominance frontier

`G` : ambient cfg
`DT`: Dominance Tree of `G`
`DF`: map from nodes to sets of nodes

"Obvious", immediate frontier

```
computeDF(n) ::=
  S <- {y | y successor of n in G but not in DT}
  for c in children(n) in DT:
    computeDF(c)
    for each w in DF[c]:
      if n does not dominate w:
        S <- S ∪ {w}
  DF[n] <- S
```

The rest of the frontier is inherited from the other children

Computing the dominance frontier

`G` : ambient cfg
`DT`: Dominance Tree of `G`
`DF`: map from nodes to sets of nodes

“Obvious”, immediate frontier

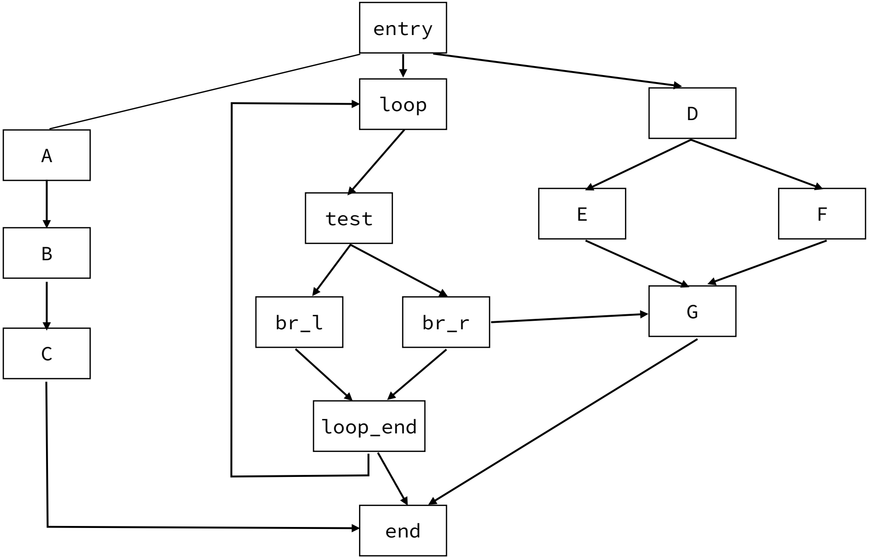
```
computeDF(n) ::=
  S <- {y | y successor of n in G but not in DT}
  for c in children(n) in DT:
    computeDF(c)
    for each w in DF[c]:
      if n does not dominate w:
        S <- S ∪ {w}
  DF[n] <- S

DF ::= computeDF(entry)
```

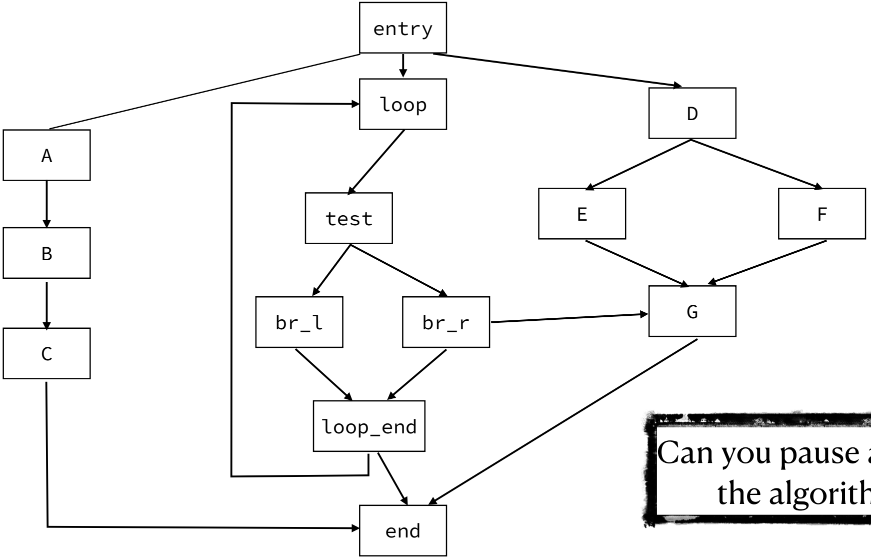
The rest of the frontier is inherited from the other children

We kickstart the pass from the entry

Computing the dominance frontier

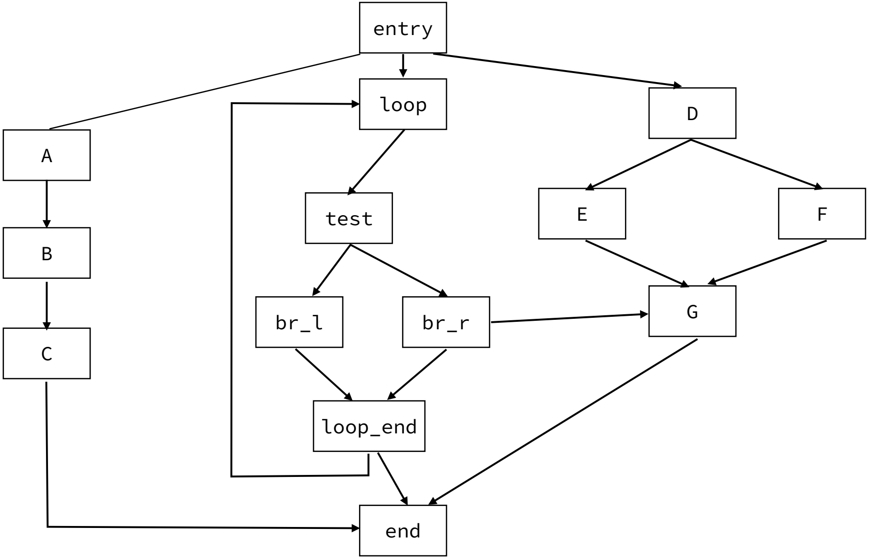


Computing the dominance frontier



Can you pause and run the algorithm?

Computing the dominance frontier

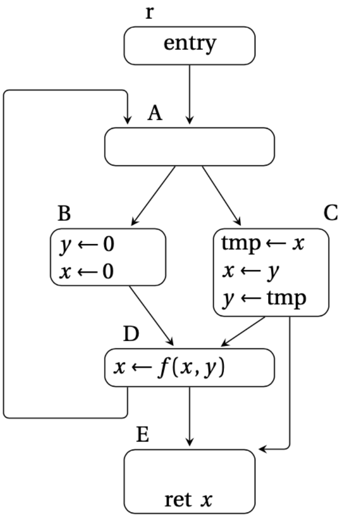


Taking stock

We want to convert a cfg to SSA-form

- The key difficulty is to figure out *where* exactly Φ-nodes are needed
- We observed *the dominance frontier* of a node seems to be the right notion
- We saw how to construct the dominance frontier, based on the construction of *the dominance tree*

We can now turn to the construction!



Exemple taken from the SSA book

Inserting Φ-nodes

```
Insert-phi ::=

   for x in Vars:
      for d in Defs(x):
          for b in DF(d):
              if there are no Φ-node associated to x in b:
                 add one such Φ-node
                 add b to Defs(x)
```

Inserting Φ-nodes

We have not yet renamed: x can have several def-sites

```
Insert-phi ::=

  for x in Vars:
     for d in Defs(x):
         for b in DF(d):
             if there are no Φ-node associated to x in b:
                add one such Φ-node
                add b to Defs(x)
```

Inserting Φ-nodes

We have not yet renamed: x can have several def-sites

Blocks containing at least one def-site of x

```
Insert-phi ::=

  for x in Vars:
      for d in Defs(x):
          for b in DF(d):
              if there are no Φ-node associated to x in b:
                  add one such Φ-node
                  add b to Defs(x)
```

Inserting Φ-nodes

We have not yet renamed: x can have several def-sites

Blocks containing at least one def-site of x

```
Insert-phi ::=

   for x in Vars:
      for d in Defs(x):
          for b in DF(d):
              if there are no Φ-node associated to x in b:
                 add one such Φ-node
                 add b to Defs(x)
```

A Φ-node is a new definition site!

Inserting Φ-nodes

We have not yet renamed: x can have several def-sites

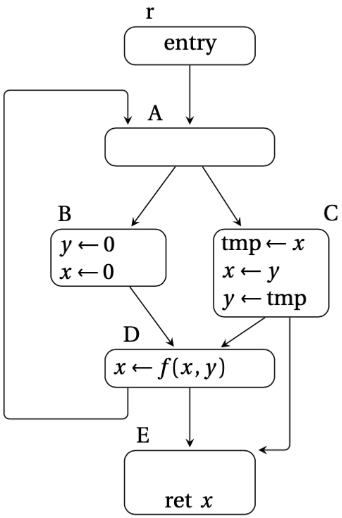
Blocks containing at least one def-site of x

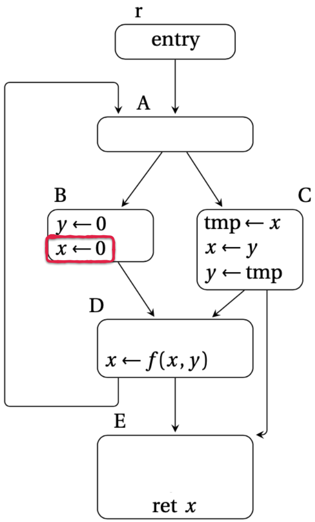
```
Insert-phi ::=

  for x in Vars:
     for d in Defs(x):
         for b in DF(d):
             if there are no Φ-node associated to x in b:
                add one such Φ-node
                add b to Defs(x)
```

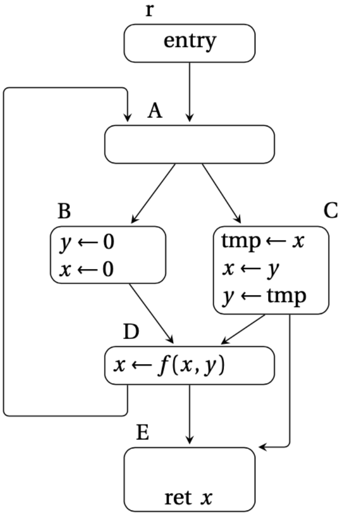
A Φ-node is a new definition site!

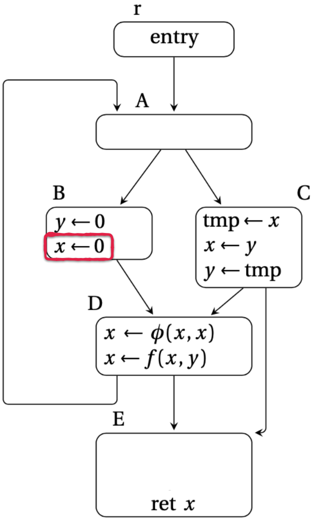
Convince yourself it converges!



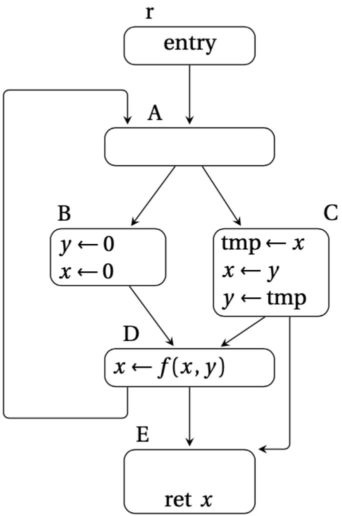


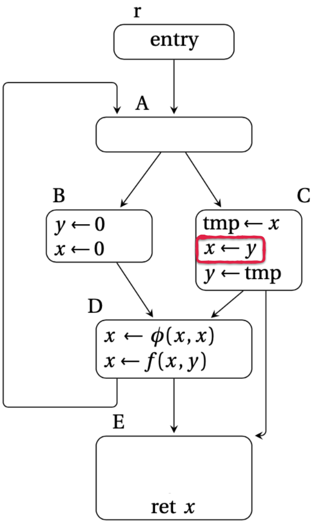
Exemple taken from the SSA book



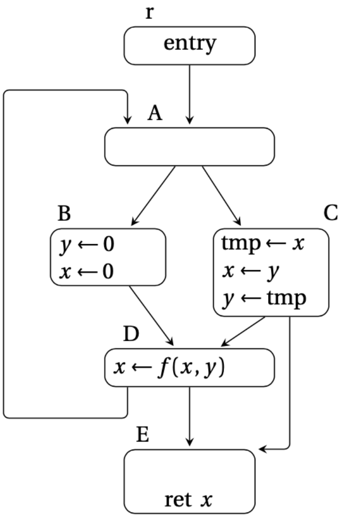


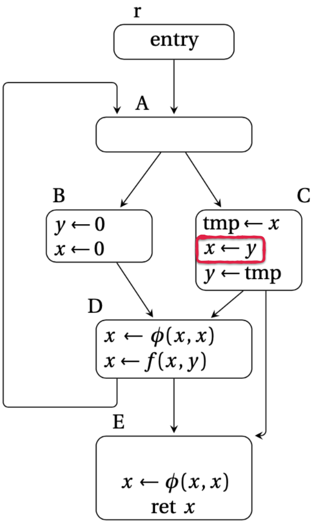
Exemple taken from the SSA book



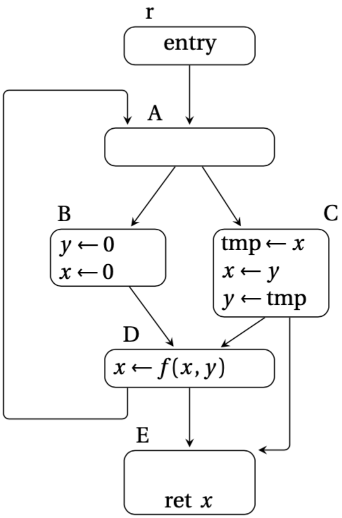


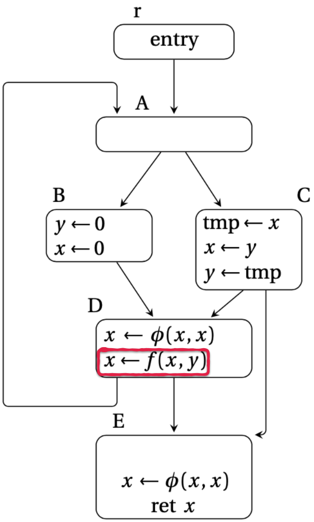
Exemple taken from the SSA book



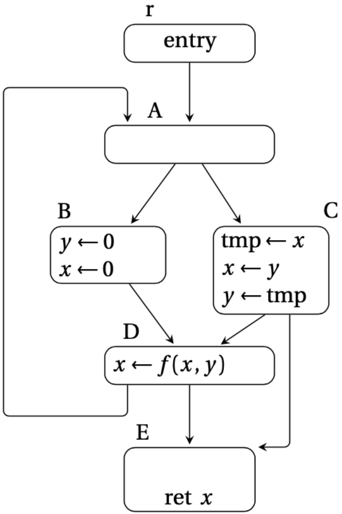


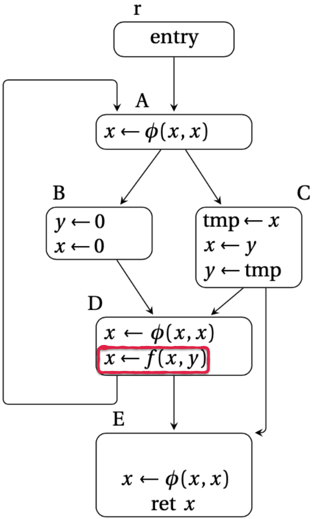
Exemple taken from the SSA book



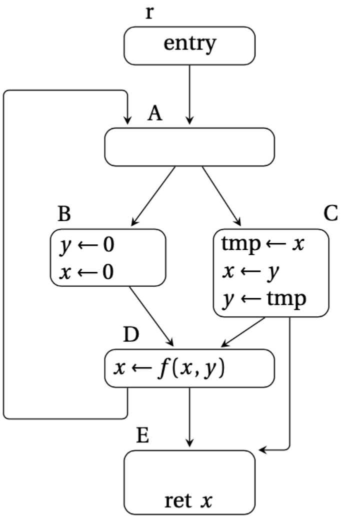


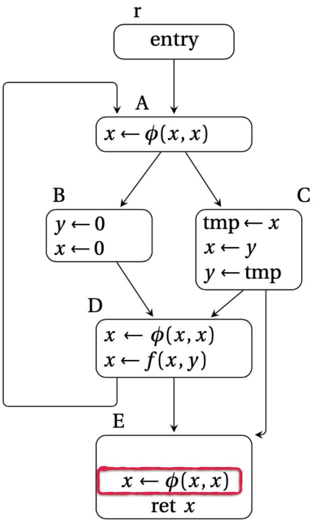
Exemple taken from the SSA book



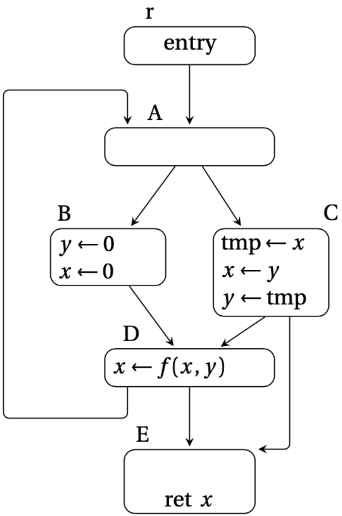


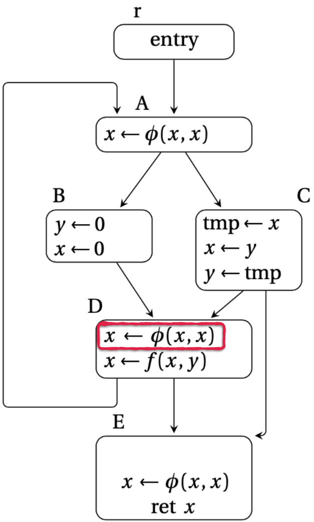
Exemple taken from the SSA book



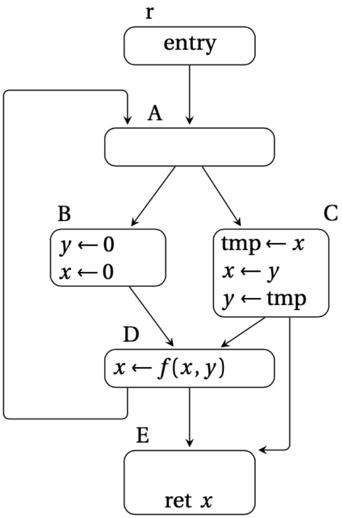


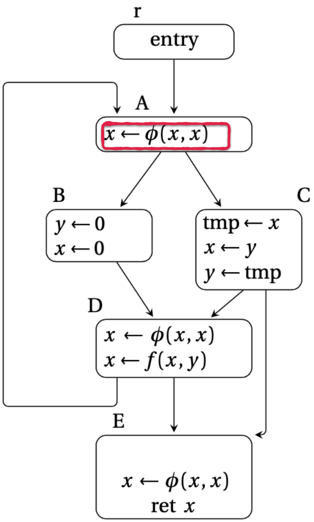
Exemple taken from the SSA book



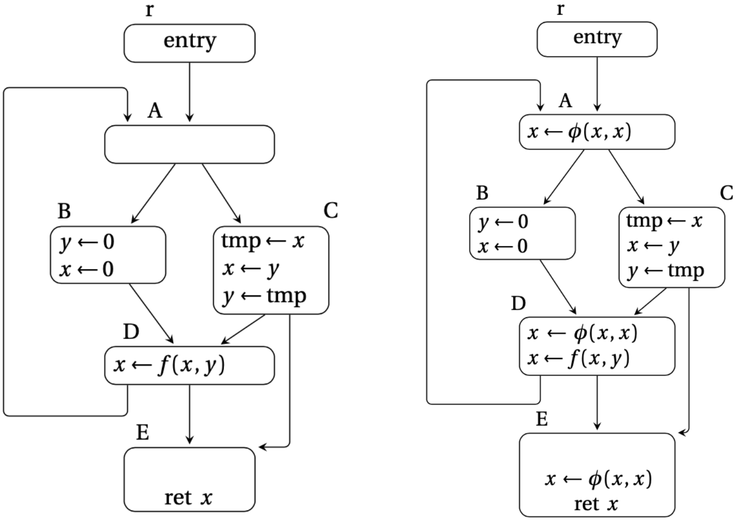


Exemple taken from the SSA book

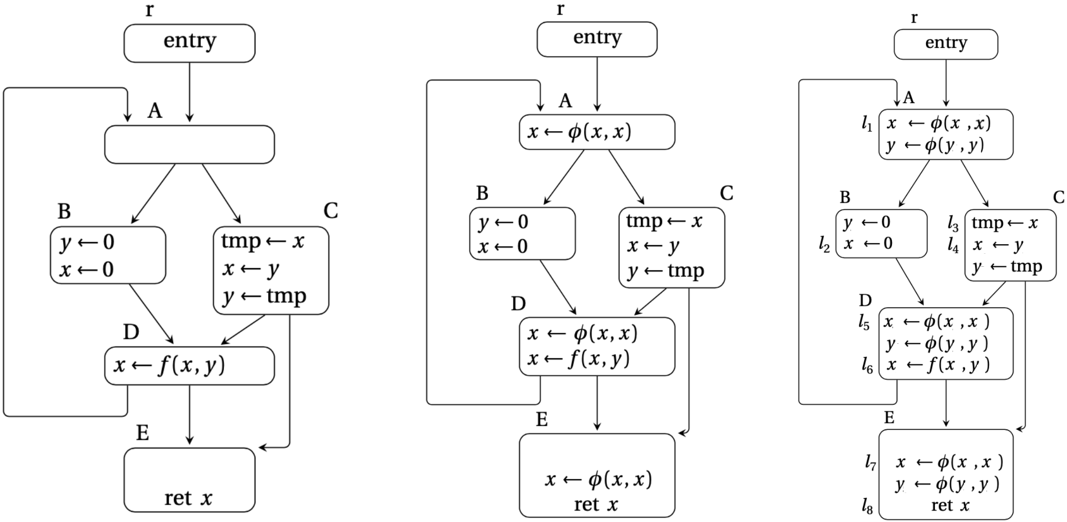




Exemple taken from the SSA book



Exemple taken from the SSA book



Exemple taken from the SSA book

Renaming variables

`stack[x]` : for each variable, we maintain a stack of names (“x_i”)

```
rename_aux(block) ::=
```

```
rename() ::= rename_aux(entry)
```

Renaming variables

`stack[x]` : for each variable, we maintain a stack of names ("x_i")

```
rename_aux(block) ::=
  for ins := y <- e in instr(block):
    for each var x in e, replace x by stack[x]
    generate a fresh name y' for y
    push y' on top of stack[y]
```

```
rename() ::= rename_aux(entry)
```

Renaming variables

`stack[x]` : for each variable, we maintain a stack of names ("x_i")

```
rename_aux(block) ::=
  for ins := y <- e in instr(block):
    for each var x in e, replace x by stack[x]
    generate a fresh name y' for y
    push y' on top of stack[y]
  for each s successor of block:
    for each Φ-node p of s:
      if x is read coming from block, replace x with stack[x]
```

```
rename() ::= rename_aux(entry)
```

Renaming variables

`stack[x]` : for each variable, we maintain a stack of names (“x_i”)

```
rename_aux(block) ::=
  for ins := y <- e in instr(block):
    for each var x in e, replace x by stack[x]
    generate a fresh name y’ for y
    push y’ on top of stack[y]
  for each s successor of block:
    for each Φ-node p of s:
      if x is read coming from block, replace x with stack[x]
  for each successor b of block in the DT:
    rename_aux(b)
```

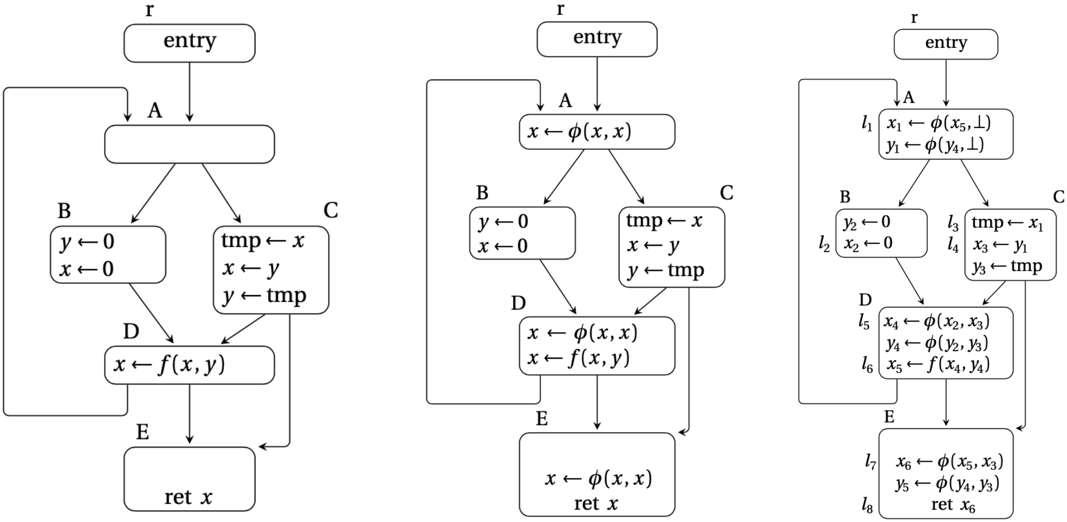
```
rename() ::= rename_aux(entry)
```

Renaming variables

`stack[x]` : for each variable, we maintain a stack of names ("x_i")

```
rename_aux(block) ::=
  for ins := y <- e in instr(block):
    for each var x in e, replace x by stack[x]
    generate a fresh name y' for y
    push y' on top of stack[y]
  for each s successor of block:
    for each Φ-node p of s:
      if x is read coming from block, replace x with stack[x]
  for each successor b of block in the DT:
    rename_aux(b)
  pop from stack all variables introduced in this function call

rename() ::= rename_aux(entry)
```



Exemple taken from the SSA book

Converting out of SSA form

From SSA to machine code

Processors do not support Φ-nodes, we need to compile them away!

From SSA to machine code

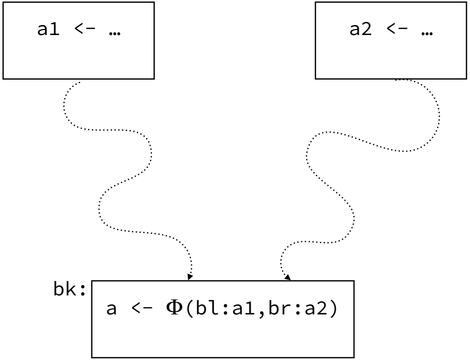
Processors do not support Φ-nodes, we need to compile them away!

bk:

```
a <- Φ(bl:a1,br:a2)
```

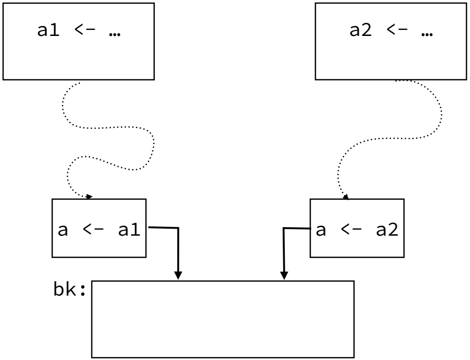
From SSA to machine code

Processors do not support Φ-nodes, we need to compile them away!



From SSA to machine code

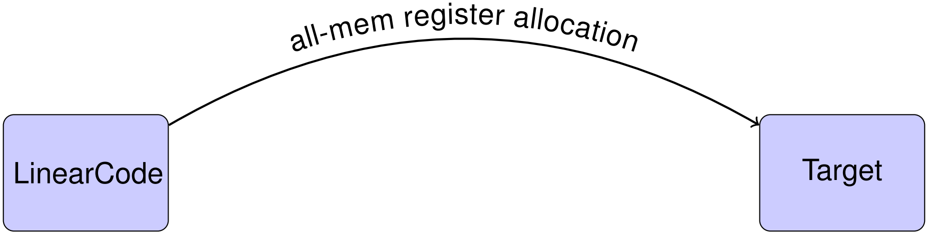
Processors do not support Φ-nodes, we need to compile them away!



A good register allocator should then take care of eliminating needlessly introduced mov

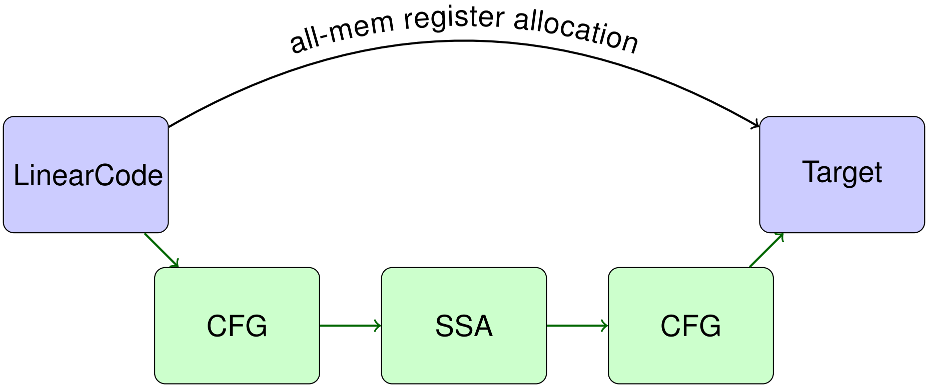
Code Generation

In previous labs



Code Generation

In previous labs



In next labs

Steps

1. Implement Leader algorithm (from Linear code to CFG)
2. Implement SSA entry (dominance frontier and ϕ-insertion)
3. Implement SSA exit

To SSA and back again

```
i=1; j=1; k=0;
while (k < 100) {
  if (j < 20) {
    j=i;
    k=k+1;
  } {
    j=k;
    k=k+2;
}}
return j;
```

(Exercise taken from Fernando Pereira)

1. Draw the CFG
2. Compute the Dominance Tree and the Frontier
3. Convert to SSA
4. Convert out of SSA

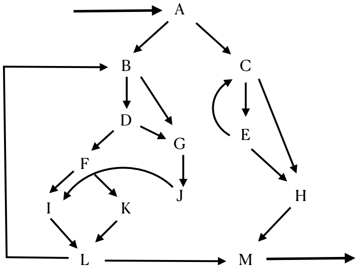
To SSA and back again

```
i=1; j=1; k=0;
while (k < 100) {
  if (j < 20) {
    j=i;
    k=k+1;
  } {
    j=k;
    k=k+2;
}}
return j;
```

To SSA and back again

```
i=1; j=1; k=0;
while (k < 100) {
  if (j < 20) {
    j=i;
    k=k+1;
  } {
    j=k;
    k=k+2;
}}
return j;
```

Testing dominance in constant time



We wish to test in constant time whether a given node dominates another. We assume that we have already computed the dominance tree, and allow ourself to this end a little pre-processing.

Q1. Draw the dominance tree of the graph on the right

Q2. Write an instrumented depth-first traversal labeling each node of the dominance tree with two numbers:

- N: the order in which that node was visited
- A: the maximum N among the node's descendants

Q3. Prove that these annotations can be used to test dominance in constant time.

Text adapted from an exercise designed by Fernando Pereira

Summary

1. SSA Control Flow Graph
2. LAB: CFG + SSA
3. Exercises