Compilation and Program Analysis (#6) :

Intermediate Representations: CFG, DAGs (Instruction Selection and Scheduling), SSA

Laure Gonnord

http://laure.gonnord.org/pro/teaching/capM1.html

Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

oct 2017

Big picture

source code

↓

$\boxed{\text{lexical+syntactic analysis + typing}}$

↓

decorated AST

↓

$\boxed{\text{code production (numerous phases)}}$

↓

assembly language

In context 1/2

In the last course we saw the need for a better data structure to propagate and infer information. We need :

- A data structure that helps us to reason about the flow of the program.
- Which embeds our three address code.

▶ Control-Flow Graph.

In context 2/2

decorated AST

↓

$\boxed{\text{IR Construction}}$

↓

Control-Flow Graph

↓

$\boxed{\text{Clever analyses/code generation}}$

↓

assembly language

Definitions

Definition (Basic Block)

Basic block : largest (3-address LC-3*) instruction sequence without label. (except at the first instruction) and without jumps and calls.*

Definition (CFG)

It is a directed graph whose vertices are basic blocks, and edge $B_1 \rightarrow B_2$ *exists if* B_2 *can follow immediately* B_1 *in an execution.*

- two optimisation levels : local (BB) and global (CFG)

Identifying Basic Blocks (from 3@code)

- The first instruction of a basic block is called a **leader**.
- We can identify leaders via these three properties :
  1. The first instruction in the intermediate code is a leader.
  2. Any instruction that is the target of a conditional or unconditional jump is a leader.
  3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Once we have found the leaders, it is straighforward to find the basic blocks : for each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.

Exercise

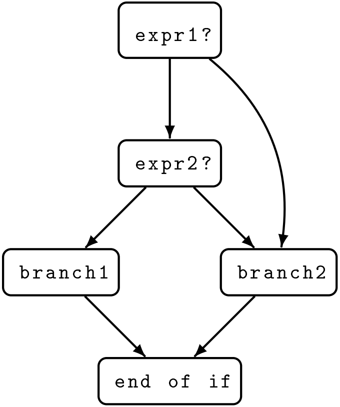
Generate the “high level” CFG for the given program :

```
p:=0;i:=1;
while (i <= 20) do
  if p>60 then
     p:=0;i:=5;
  endif
  i:=2*i+1;
done
k:=p*3;
```

(inside your compiler, blocks will be a list of 3@-LC-3 code)

CFG for tests

```
if (expr1 and expr2)
  ...branch1...
else
  ...branch2...
```

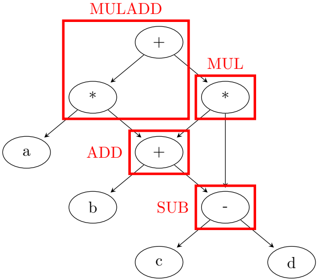


(blocks are subgraphs)

Big picture

- Front-end → a CFG where nodes are basic blocks.
- Basic blocks → DAGs that explicit common computations

```
u1 := c - d
u2 := b + u1
u3 := a * u2
u4 := u2 * u1
u5 := u3 + u4
```



▶ choose instructions(**selection**) and order them (**scheduling**).

1. Control flow Graph

2. Basic Bloc DAGs, instruction selection/scheduling
   - Instruction Selection
   - Instruction Scheduling

3. SSA Control Flow Graph
   - SSA Construction
   - Example
   - Out of SSA !

Instruction Selection

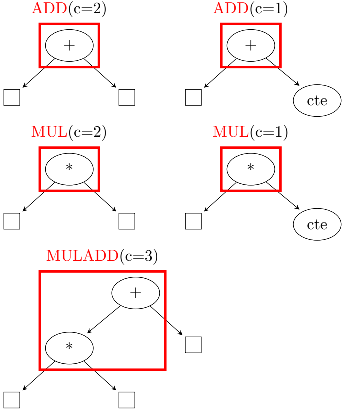
The problem of selecting instructions is a DAG-partitioning problem. But what is the objective ?

The best instructions :

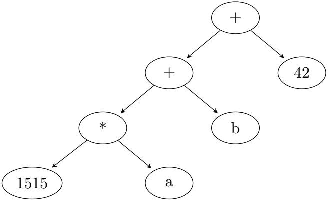
- cover bigger parts of computation.
- cause few memory accesses.

▶ Assign a cost to each instruction, depending on their addressing mode.

Instruction Selection : an example



What is the optimal instruction selection for :



- Finding a tiling of minimal cost : it is **NP-complete** (SAT reduction).

Tiling trees / DAGs, in practise

For tiling :

- There is an optimal algorithm for **trees** based on dynamic programing.
- For DAGs we use heuristics (decomposition into a forest of trees, . . .)

▶ The litterature is pletoric on the subject.

Instruction Scheduling, what for ?

We want an evaluation order for the instructions that we choose with **Instruction Scheduling**.

A scheduling is a function θ that associates a **logical date** to each instruction. To be correct, it must respect data dependancies :

```
(S1) u1 := c - d
(S2) u2 := b + u1
```

implies $\theta(S1) < \theta(S_2)$.

- How to choose among many correct schedulings ? depends on the target architecture.

Architecture-dependant choices

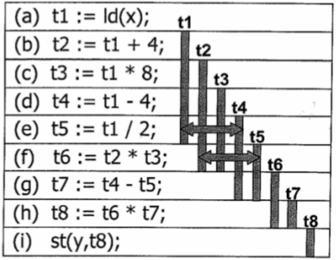
The idea is to exploit the different ressources of the machine at their best :

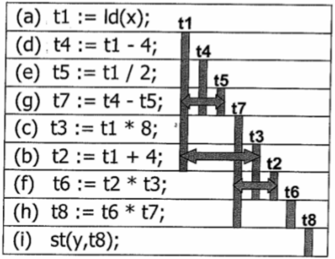
- instruction parallelism : some machine have parallel units (subinstructions of a given instruction).
- prefetch : some machines have non-blocking load/stores, we can run some instructions between a load and its use (hide latency !)
- pipeline.
- registers : see next slide.

(sometimes these criteria are incompatible)

Register use

Some schedules induce less **register pressure** :





- How to find a schedule with less register pressure ?

Scheduling wrt register pressure

Result : this is a linear problem on trees, but NP-complete on DAGs (Sethi, 1975).

- Sethi-Ullman algorithm on trees, heuristics on DAGs

Sethi-Ullman algorithm on trees

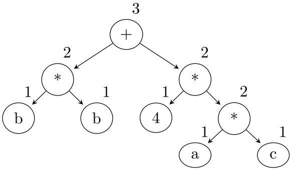
$\rho(node)$ denoting the number of (pseudo)-registers necessary to compute a node :

- $\rho(leaf) = 1$
- $\rho(nodeop(e_1, e_2)) = \begin{cases} max\{\rho(e_1), \rho e_2\} & \text{if } \rho(e_1) \neq \rho(e_2) \\ \rho(e_1) + 1 & \text{else} \end{cases}$

(the idea for non "balanced" subtrees is to execute the one with the biggest ρ first, then the other branch, then the op. If the tree is balanced, then we need an extra register)

► then the code is produced with postfix tree traversal, the biggest register consumers first.

Sethi-Ullman algorithm on trees - an example



	tmp_1	tmp_2	tmp_3	tmp_4
`mul tmp1, b b`				
`mul tmp2, a c`				
`ldi tmp3, 4`				
`mul tmp4, tmp2, tmp3`				
`mul tmp5, tmp1 ,temp4`				

Conclusion (instruction selection/scheduling)

Plenty of other algorithms in the literature :

- Scheduling DAGs with heuristics, ...
- Scheduling loops (M2 course on advanced compilation)

Practical session :

- we have (nearly) no choice for the instructions in the LEIA ISA.
- evaluating the impact of scheduling is a bit hard.

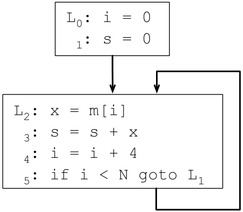
We won't implement any of the previous algorithms.

Credits

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

The Static Single Assignment Form

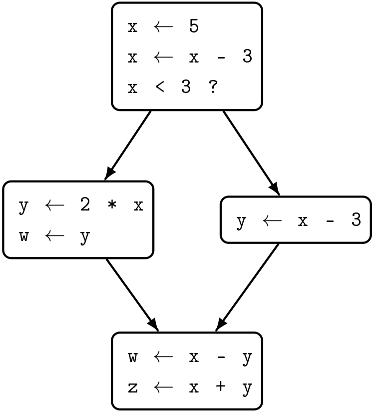
- This name comes out of the fact that each variable has only one definition site in the program.
- In other words, the entire program contains only one point where the variable is assigned a value.
- Were we talking about Single Dynamic Assignment, then we would be saying that during the <u>execution</u> of the program, the variable is assigned only once.

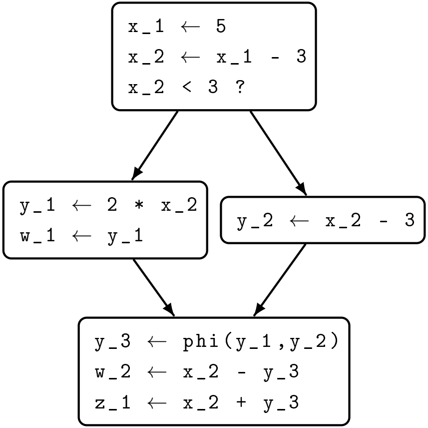


Variable i has two static assignment sites: at L_0 and at L_4; thus, this program is not in Static Single Assignment form. Variable s, also has two static definition sites. Variable x, on the other hand, has only one static definition site, at L_2. Nevertheless, x may be assigned many times dynamically, i.e., during the execution of the program.

A first Example (Cytron 1991)

Each variable is assigned only once (Static Single Assigment form) :





Pro/cons

- \- Another IR, and cost of contruction/deconstruction
- \+ (some) Analyses/optimisations are easier to perform (like register allocation) : `http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/SSABasedRA.pdf`

1. Control flow Graph

2. Basic Bloc DAGs, instruction selection/scheduling
   - Instruction Selection
   - Instruction Scheduling

3. SSA Control Flow Graph
   - SSA Construction
   - Example
   - Out of SSA !

Converting Straight-Line Code into SSA Form

- We call a program without branches a piece of "straight-line code".
- Converting a straight-line program, e.g., a basic block, into SSA is fairly straightforward.

```
L0: a = x + y
 1: b = a - 1
 2: a = y + b
 3: b = 4 * x
 4: a = a + b
```

Can you convert this program into SSA form?

for each variable a:
  Count[a] = 0
  Stack[a] = [0]

rename_basic_block(B) =
  for each instruction S in block B:
    for each use of a variable x in S:
      i = top(Stack[x])
      replace the use of x with x_i
    for each variable a that S defines
      count[a] = Count[a] + 1
      i = Count[a]
      push i onto Stack[a]
      replace definition of a with a_i

Converting Straight-Line Code into SSA Form

- We call a program without branches a piece of "straight-line code".
- Converting a straight-line program, e.g., a basic block, into SSA is fairly straightforward.

$$
\begin{array}{rl}
L_0: & a_1 = x_0 + y_0 \\
{}_1: & b_1 = a_1 - 1 \\
{}_2: & a_2 = y_0 + b_1 \\
{}_3: & b_2 = 4 * x_0 \\
{}_4: & a_3 = a_2 + b_2
\end{array}
$$

for each variable a:

  Count[a] = 0

  Stack[a] = [0]

rename_basic_block(B) =

  for each instruction S in block B:

    for each use of a variable x in S:

      i = top(Stack[x])

      replace the use of x with x_i

    for each variable a that S defines

      count[a] = Count[a] + 1

      i = Count[a]

      push i onto Stack[a]

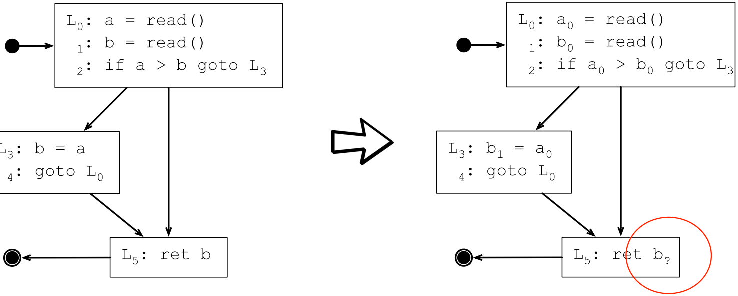
      replace definition of a with a_i

Notice that we could do without the **stack**. How? But we will need it to generalize this method.

Phi-Functions

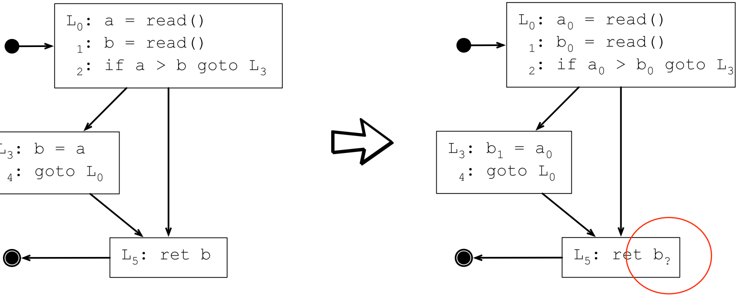
Having just one static assignment site for each variable brings some challenges, once we stop talking about straight-line programs, and start dealing with more complex flow graphs.

One important question is: once we convert this program to SSA form, which definition of b should we use at L_5?



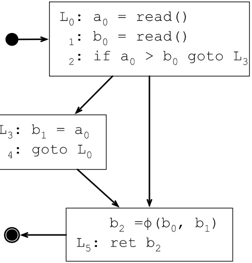
Phi-Functions

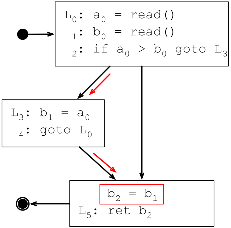
The answer to this question is: *it depends*! Indeed, the definition of b that we will use at L_5 will depend on which path execution flows. If the execution flow reaches L_5 coming from L_4, then we must use b_1. Otherwise, execution must reach L_5 coming from L_2, in which case we must use b_0

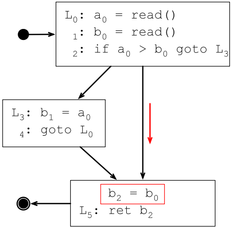


Phi-Functions

In order to represent this kind of behavior, we use a special notation: the phi-function. Phi-functions have the semantics of a multiplexer, copying the correct definition, depending on which path they are reached by the execution flow.



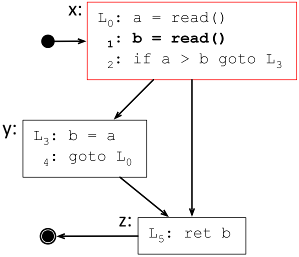




What happens once we have multiple phi-functions at the beginning of a block?

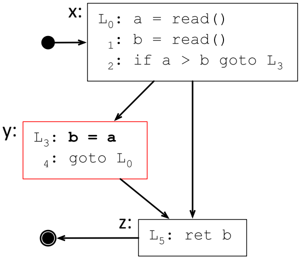
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable <u>b</u> at node <u>z</u> of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of b
  - There is a block y (with y ≠ x) containing a definition of b
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



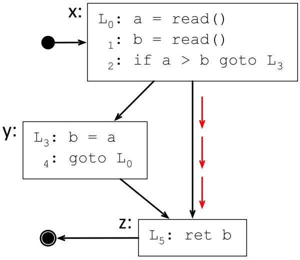
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable b at node z of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of b
  - There is a block y (with y ≠ x) containing a definition of b
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



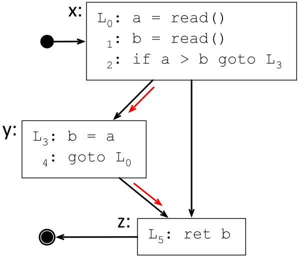
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable b at node z of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of b
  - There is a block y (with y ≠ x) containing a definition of b
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



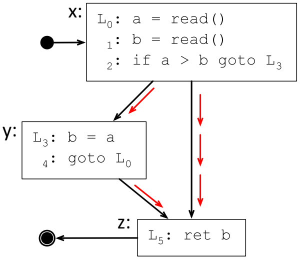
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable b at node z of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of b
  - There is a block y (with y ≠ x) containing a definition of b
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



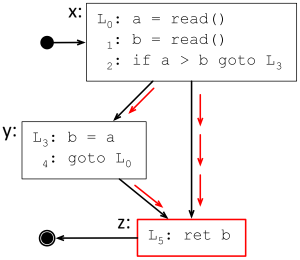
Criteria for Inserting Phi-Functions

- There should be a phi-function for variable b at node z of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of b
  - There is a block y (with y ≠ x) containing a definition of b
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



Criteria for Inserting Phi-Functions

- There should be a phi-function for variable b at node z of the flow graph exactly when all of the following are true:
  - There is a block x containing a definition of b
  - There is a block y (with y ≠ x) containing a definition of b
  - There is a nonempty path P_{xz} of edges from x to z
  - There is a nonempty path P_{yz} of edges from y to z
  - Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
  - The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



Iterative Creation of Phi-Functions

- When we insert a new phi-function in the program, we are creating a new definition of a variable.
- This new definition may raise the necessity of new phi-functions in the code.
- Thus, the path convergence criteria must be used iteratively, until we reach a fixed point:

What is the complexity of this algorithm?

while there are nodes *x*, *y*, and *z* satisfying the path-convergence criteria and *z* does not contain a phi-function for variable *a* **do**:

  insert $a = \phi(a, a, \ldots, a)$ at node *z*, with as many parameters as *z* has predecessors.

Dominance Property of SSA Form

The previous algorithm is a bit too expensive. Let's see a faster one. But, to do it, we will need the notion of dominance frontier.

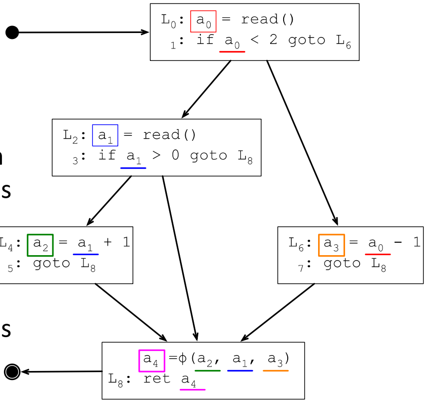
- A node *d* of a rooted, directed graph dominates another node *n* if every path from the root node to *n* goes through *d*.
- In Strict♤ SSA form programs, definitions of variables dominate their uses:
  - If x is the i-th argument of a phi-function in block n, then the definition of x dominates the i-th predecessor of n.
  - If x is used in a non-phi statement in block n, then the definition of x dominates node n.

♤: A program is strict if every variable is initialized before it is used.

Where have we heard of dominance before?

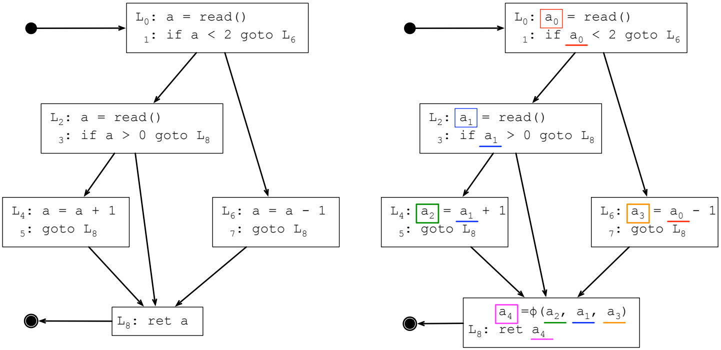
Dominance Property of SSA Form

- In <u>Strict</u> SSA form programs, definitions of variables dominate their uses:
  - If x is the i-th argument of a phi-function in block n, then the definition of x dominates the i-th predecessor of n.
  - If x is used in a non-phi statement in block n, then the definition of x dominates node n.



How does this observation helps us to build SSA form?

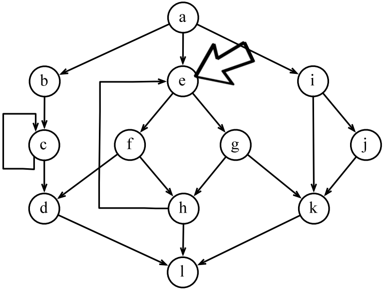
Dominance Property of SSA Form



For one, we can distribute phi-functions here and there, and then we only have to worry about one thing: we must ensure that every use of a variable v has the same name as the instance of v that dominates that use.

The Dominance Frontier

- There is an algorithm more efficient than the iterative application of the path-convergence criteria, which is almost linear time on the size of the program.
  - This algorithm relies on the notion of dominance frontier
- A node x strictly dominates w if x dominates w and x ≠ w.
- The dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.

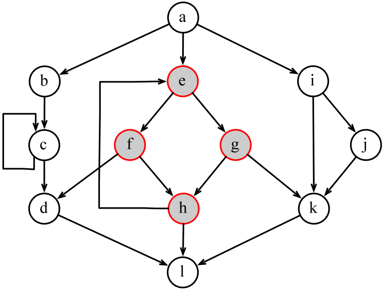


What are the nodes that "e" dominates?

The Dominance Frontier

- There is an algorithm more efficient than the iterative application of the path-convergence criteria, which is almost linear time on the size of the program.
  - This algorithm relies on the notion of dominance frontier
- A node x strictly dominates w if x dominates w and x ≠ w.
- The dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.

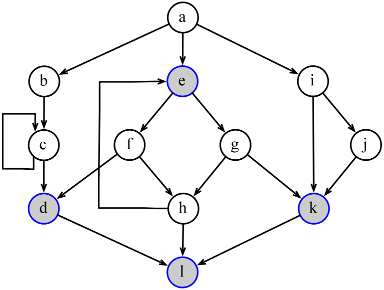
What are the nodes in the dominance frontier of e?



The Dominance Frontier

- There is an algorithm more efficient than the iterative application of the path-convergence criteria, which is almost linear time on the size of the program.
  - This algorithm relies on the notion of dominance frontier
- A node x strictly dominates w if x dominates w and x ≠ w.
- The dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.

Why is e included in its dominance frontier?



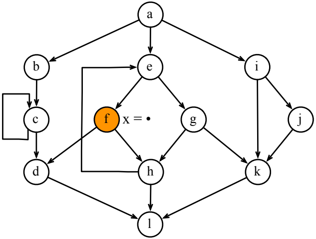
The Dominance Frontier Criterion

- **Dominance-Frontier Criterion**: Whenever node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a phi-function for a.

- **Iterated dominance frontier**: since a phi-function itself is a kind of definition, we must iterate the dominance-frontier criterion until there are no nodes that need phi-functions.

Theorem: the iterated dominance frontier criterion and the iterated path-convergence criteria specify exactly the same set of nodes at which to put phi-functions.

The Dominance Frontier Criterion

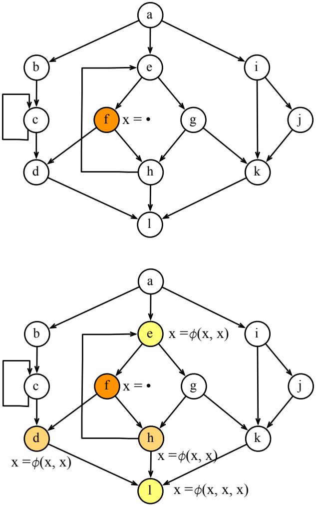


Where should we place phi-functions due to the definition of x at block f?

- **Dominance-Frontier Criterion**: Whenever node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a phi-function for a.

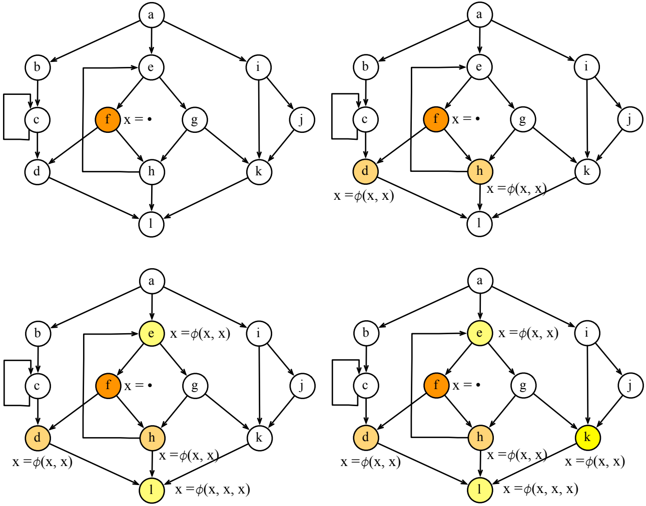
- **Iterated dominance frontier**: since a phi-function itself is a kind of definition, we must iterate the dominance-frontier criterion until there are no nodes that need phi-functions.

The Dominance Frontier Criterion



Is there any other place that should receive phi-functions?

The Dominance Frontier Criterion



Computing the Dominance Frontier

We compute the dominance frontier of the nodes of a graph by iterating the following equations:

$$DF[n] = DF_{local}[n] \cup \{ DF_{up}[c] \mid c \in children[n] \}$$

Where:

- $DF_{local}[n]$: the successors of n that are not strictly dominated by n
- $DF_{up}[c]$: nodes in the dominance frontier of c that are not strictly dominated by n.
- children[n]: the set of children of node n in the dominator tree

1) It should be clear why we need $DF_{local}[n]$, right?

2) But, why do we have **this** second part of the equation?

Computing the Dominance Frontier

We compute the dominance frontier of the nodes of a graph by iterating the following equations:

$DF[n] = DF_{local}[n] \cup \{ DF_{up}[c] \mid c \in children[n] \}$

Where:

- $DF_{local}[n]$: the successors of n that are not strictly dominated by n
- $DF_{up}[c]$: nodes in the dominance frontier of c that are not strictly dominated by n.
- children[c]: the set of children of node c in the dominator tree

The algorithm below computes the dominance frontier of every node in the CFG. It must be called from the root node:

computeDF[n]:
S = {}
for each node y in succ[n]
  if idom(y) ≠ n
    S = S ∪ {y}
for each child c of n in the dom-tree
  computeDF[c]
  for each w ∈ DF[c]
    if n does not dom w, or n = w
      S = S ∪ {w}
DF[n] = S

Inserting Phi-Functions

place-phi-functions:

for each node n:

for each variable a $\in A_{orig}[n]$:

defsites[a] = defsites[a] $\cup$ [n]

for each variable a:

W = defsites[a]

while W $\neq$ empty list

remove some node n from W

for each y in DF[n]:

if a $\notin A_{phi}[y]$

insert-phi(y, a)

$A_{phi}[y] = A_{phi}[y] \cup \{a\}$

if a $\notin A_{orig}[y]$

W = W $\cup$ {y}

insert-phi(y, a):

insert the statement a = ϕ(a, a, …, a) at the top of block y, where the phi-function has as many arguments as y has predecessors

Where:

- $A_{orig}[n]$: the set of variables defined at node "n"
- $A_{phi}[y]$: the set of variables that have phi-functions at node "y"

Notice that W can grow, due to **this** union. How do we know that this algorithm terminates?

Renaming Variables

- We already have a procedure that renames variables in straight-line segments of code
- We must now extend this procedure to handle general control flow graphs.

How should we extend this algorithm to handle general CFGs?

for each variable a:
  Count[a] = 0
  Stack[a] = [0]
rename-basic-block(B):
  for each instruction S in block B:
    for each use of a variable x in S:
      i = top(Stack[x])
      replace the use of x with x_i
    for each variable a that S defines
      count[a] = Count[a] + 1
      i = Count[a]
      push i onto Stack[a]
      replace definition of a with a_i

Renaming Variables

Does this algorithm ensure that the definition of a variable dominates all its uses?

Child is the successor of n in the dominator tree. Why we cannot use the successors of n in the CFG?

rename(n):

rename-basic-block(n)

for each successor Y of n, **where** n is the j-th predecessor of Y:

for each phi-function f in Y, **where** the operand of f is 'a'

i = top(Stack[a])

replace j-th operand with a_i

for each child X of n:

rename(X)

for each instruction S $\in$ n:

for each variable v that S defines:

pop Stack[v]

1. Control flow Graph

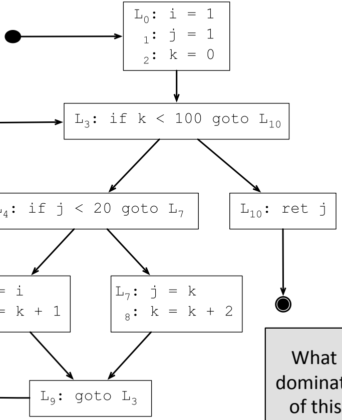
2. Basic Bloc DAGs, instruction selection/scheduling
   - Instruction Selection
   - Instruction Scheduling

3. SSA Control Flow Graph
   - SSA Construction
   - Example
   - Out of SSA !

Putting it All Together

- Lets convert the following program to SSA form:

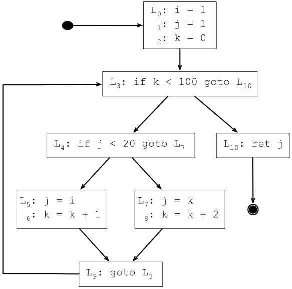
```
i = 1
j = 1
k = 0
while k < 100
   if j < 20
      j = i
      k = k + 1
   else
      j = k
      k = k + 2
return j
```



What is the dominator tree of this CFG?

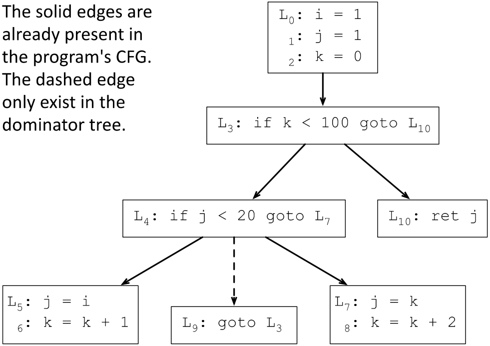
Putting it All Together

Can you compute the dominance frontier of each node?

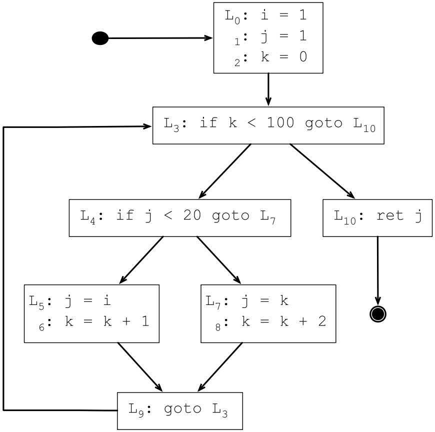


The Dominator Tree:

The solid edges are already present in the program's CFG. The dashed edge only exist in the dominator tree.



Computing the Dominance Frontier



The dominance frontier of each node is listed below:

L_0: {}

L_3: {L_3}

L_4: {L_3}

L_5: {L_9}

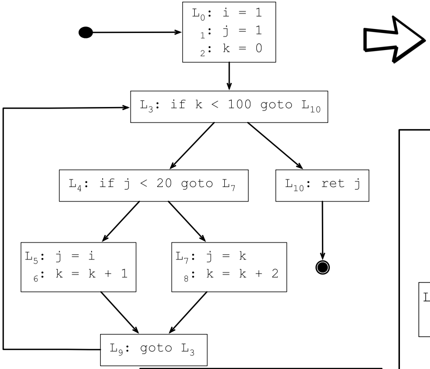
L_7: {L_9}

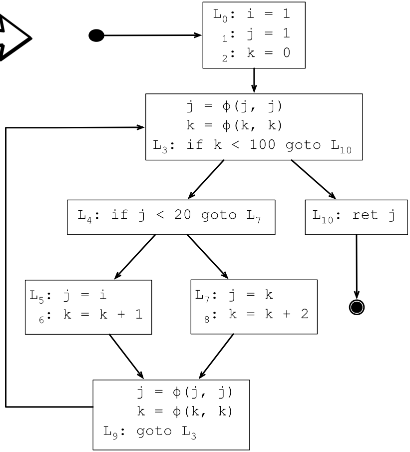
L_9: {L_3}

L_{10}: {}

Can you insert phi-functions in the CFG on the left, given these dominance frontiers?

Inserting Phi-Functions

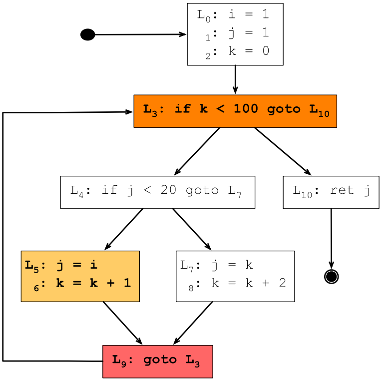




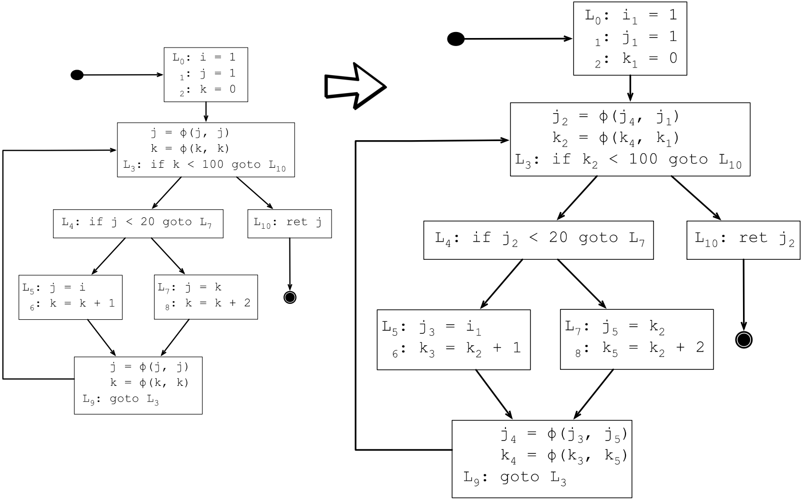
Which succession of events has forced us to add phi functions onto L_3?

Iterated Dominance Frontier

- Node L_5 does not dominate L_9, although L_9 is a successor of L_5. Therefore, L_9 is in the dominance frontier of L_5. L_9 should have a phi-function for every variable defined inside L_5.
- We repeat the process for L_9, after all, we are considering the iterated dominance frontier.
- L_3 is in the dominance frontier of L_9, and should also have a phi-function for every variable defined in L_5. Notice that these variables are now redefined at L_9, due to the phi-functions.



After Variable Renaming



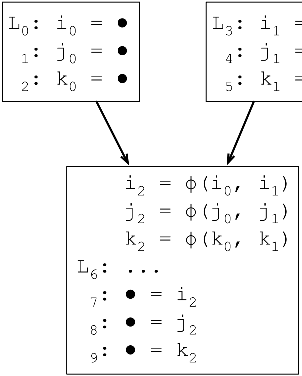
Demo !

cf `demossa.c` and Exercise sheet.

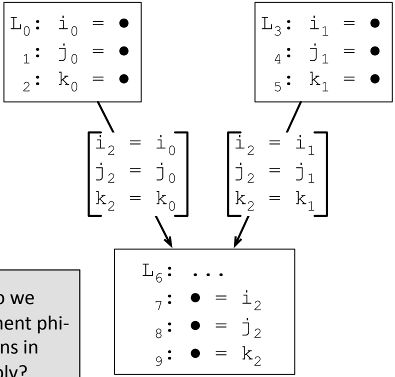
1. Control flow Graph
2. Basic Bloc DAGs, instruction selection/scheduling
   - Instruction Selection
   - Instruction Scheduling
3. SSA Control Flow Graph
   - SSA Construction
   - Example
   - Out of SSA !

Phi-Functions

A set of N phi-functions with M arguments each at the beginning of a basic block represents M parallel copies. Each copy reads N inputs, and writes on N outputs.



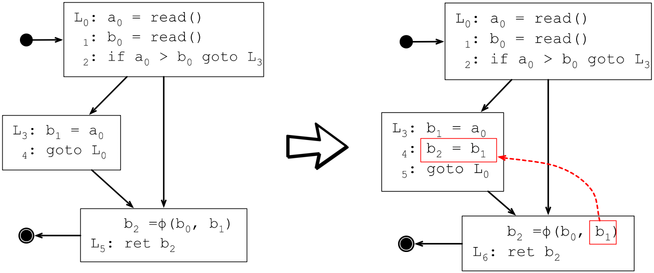
How do we implement phi-functions in assembly?



SSA Elimination

Compilers that use the SSA form usually contain a step, before the generation of actual assembly code, in which phi-functions are replaced by ordinary instructions. Normally these instructions are simple copies.

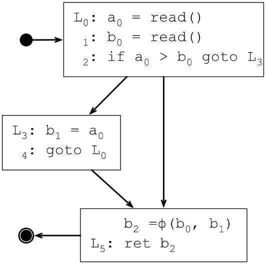
> And where would we place the copy $b_2 = b_0$? Why is this an important question at all?

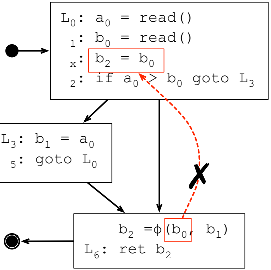


Critical Edges

The placement of the copy $b_2 = b_0$ is not simple, because the edge that links L_2 to L_5 is *critical*. A critical edge connects a block with multiple successors to a block with multiple predecessors.

If we were to put the copy between labels L1 and L2, then we would be creating a partial redundancy.





1) have you heard of critical edges before? How so?

2) How can we solve this conundrum?

Edge Splitting

We can solve this problem by doing *critical edge splitting*. This CFG transformation consists in adding an empty basic block (empty, except by – perhaps – a goto statement) between each pair of blocks connected by a critical edge.

Ok, but let's go back into SSA construction: where to insert phi-functions?