The Static Single Assignment Form:

Construction and Application to Program Optimizations - Part 1

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

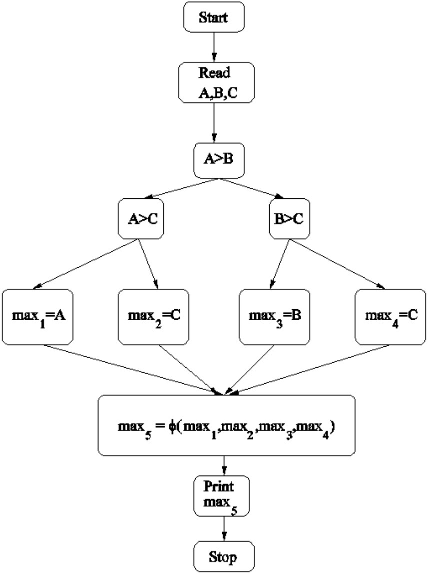
NPTEL Course on Compiler Design

The SSA Form: Introduction

- A new intermediate representation
- Incorporates *def-use* information
- Every variable has exactly one definition in the program text
  - This does not mean that there are no loops
  - This is a *static* single assignment form, and not a *dynamic* single assignment form
- Some compiler optimizations perform better on SSA forms
  - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
- A *sparse* intermediate representation
  - If a variable has N uses and M definitions, then *def-use chains* need space and time proportional to $N.M$
  - But, the corresponding instructions of uses and definitions are only $N + M$ in number
  - SSA form, for most realistic programs, is linear in the size of the original program

A Program in non-SSA Form and its SSA Form

```
read A,B,C
if (A>B)
   if (A>C) max = A
   else max = C
else if (B>C) max = B
       else max = C
printf (max)
```



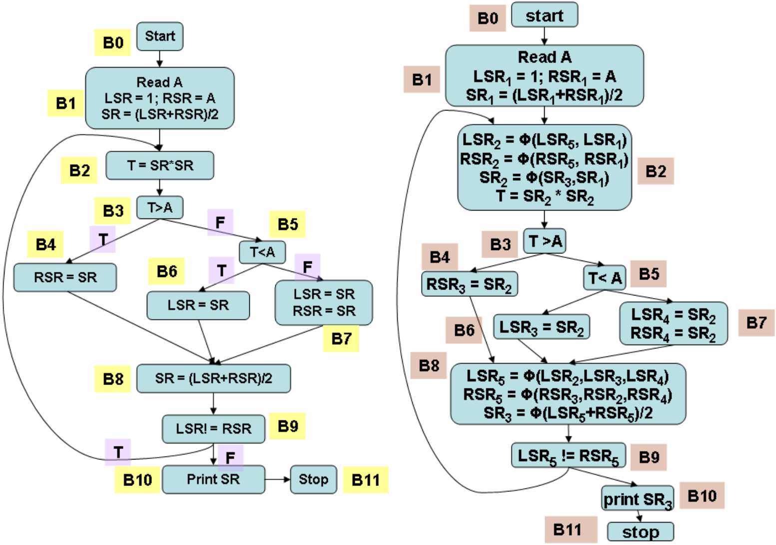
SSA Form: A Definition

- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow control remains the same as in the non-SSA form
- A special merge operator, ϕ, is used for selection of values in join nodes
- Not every join node needs a ϕ operator for every variable
- No need for a ϕ operator, if the same definition of the variable reaches the join node along all incoming edges
- Often, the SSA form is augmented with *u-d* and *d-u* chains to facilitate design of faster algorithms
- Translation from SSA to machine code introduces copy operations, which may introduce some inefficiency

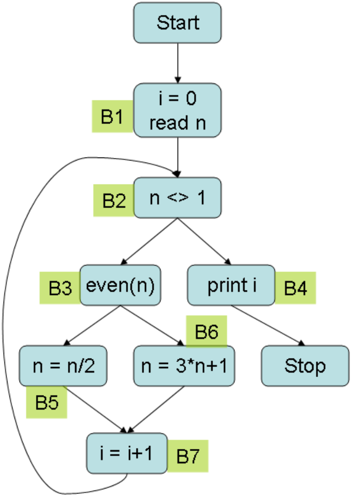
Program 2 in non-SSA Text Form

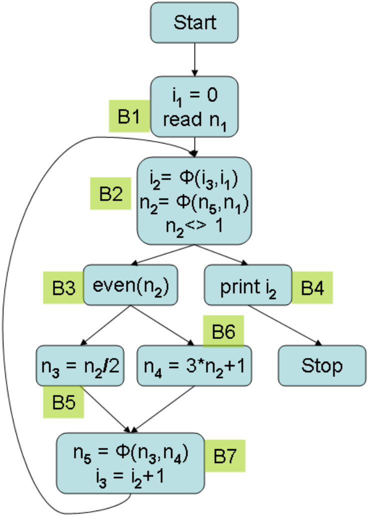
```
{ Read A; LSR = 1; RSR = A;
  SR = (LSR+RSR)/2;
  Repeat {
       T = SR*SR;
       if (T>A) RSR = SR;
       else if (T<A) LSR = SR;
              else { LSR = SR; RSR = SR}
       SR = (LSR+RSR)/2;
  Until (LSR ≠ RSR);
  Print SR;
}
```

Program 2 in non-SSA Flow Graph Form



Program 3 in non-SSA and SSA Form





Conditions on the SSA form

After translation, the SSA form should satisfy the following conditions for every variable v in the original program.

1. If two non-null paths from nodes X and Y each having a definition of v converge at a node p, then p contains a trivial ϕ-function of the form $v = \phi(v, v, ..., v)$, with the number of arguments equal to the in-degree of p.
2. Each appearance of v in the original program or a ϕ-function in the new program has been replaced by a new variable v_i, leaving the new program in SSA form.
3. Any use of a variable v along any control path in the original program and the corresponding use of v_i in the new program yield the same value for both v and v_i.

Conditions on SSA Forms

- Condition 1 in the previous slide is recursive.
  - It implies that ϕ-assignments introduced by the translation procedure will also qualify as assignments to *v*
  - This in turn may lead to introduction of more ϕ-assignments at other nodes
- It would be wasteful to place ϕ-functions in all join nodes
- It is possible to locate the nodes where ϕ-functions are *essential*
- This is captured by the *dominance frontier*

The Join Sets and ϕ Nodes

Given $\mathcal{S}$: set of flow graph nodes, the set $JOIN(\mathcal{S})$ is

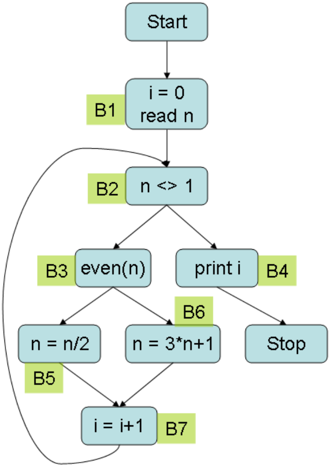
- the set of all nodes *n*, such that there are at least two non-null paths in the flow graph that start at two distinct nodes in $\mathcal{S}$ and converge at *n*
  - The paths considered should not have any other common nodes apart from *n*
- The *iterated join set*, $JOIN^+(\mathcal{S})$ is

$$\begin{aligned} JOIN^{(1)}(\mathcal{S}) &= JOIN(\mathcal{S}) \\ JOIN^{(i+1)}(\mathcal{S}) &= JOIN(\mathcal{S} \cup JOIN^{(i)}(\mathcal{S})) \end{aligned}$$

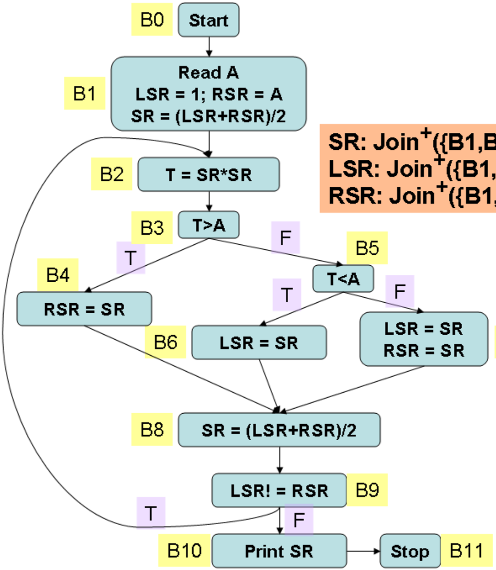
- If $\mathcal{S}$ is the set of assignment nodes for a variable *v*, then $JOIN^+(\mathcal{S})$ is precisely the set of flow graph nodes, where ϕ-functions are needed (for *v*)
- $JOIN^+(\mathcal{S})$ is termed the *dominance frontier*, $DF(\mathcal{S})$, and can be computed efficiently

JOIN Example -1

- variable i: $JOIN^+(\{B1, B7\}) = \{B2\}$
- variable n: $JOIN^+(\{B1, B5, B6\}) = \{B2, B7\}$



JOIN Example - 2



SR: $Join^{+}(\{B1,B8\}) = \{B2\}$
LSR: $Join^{+}(\{B1,B6,B7\}) = \{B2,B8\}$
RSR: $Join^{+}(\{B1,B4,B7\}) = \{B2,B8\}$

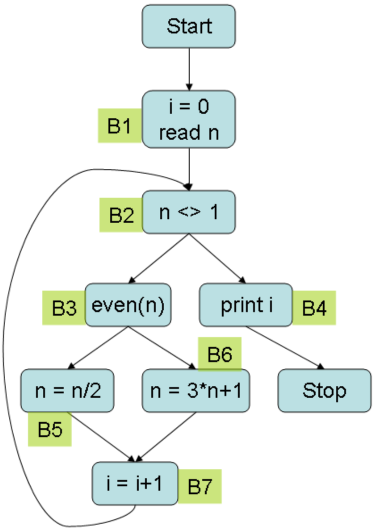
Dominators and Dominance Frontier

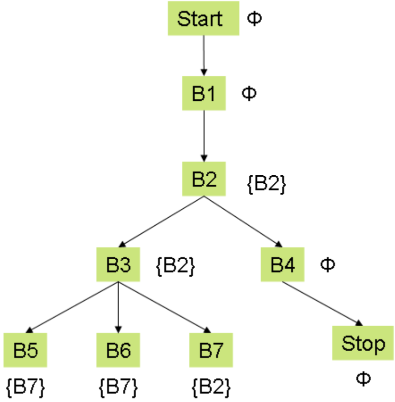
- Given two nodes x and y in a flow graph, x *dominates* y ($x \in dom(y)$) , if x appears in all paths from the *Start* node to y
- The node x *strictly dominates* y, if x dominates y and $x \neq y$
- x is the *immediate dominator* of y (denoted $idom(y)$), if x is the closest strict dominator of y
- A *dominator tree* shows all the immediate dominator relationships
- The *dominance frontier* of a node x, $DF(x)$, is the set of all nodes y such that
  - x dominates a predecessor of y ($p \in preds(y)$ *and* $x \in dom(p)$)
  - but x does not strictly dominate y ($x \notin dom(y) - \{y\}$)

Dominance frontiers - An Intuitive Explanation

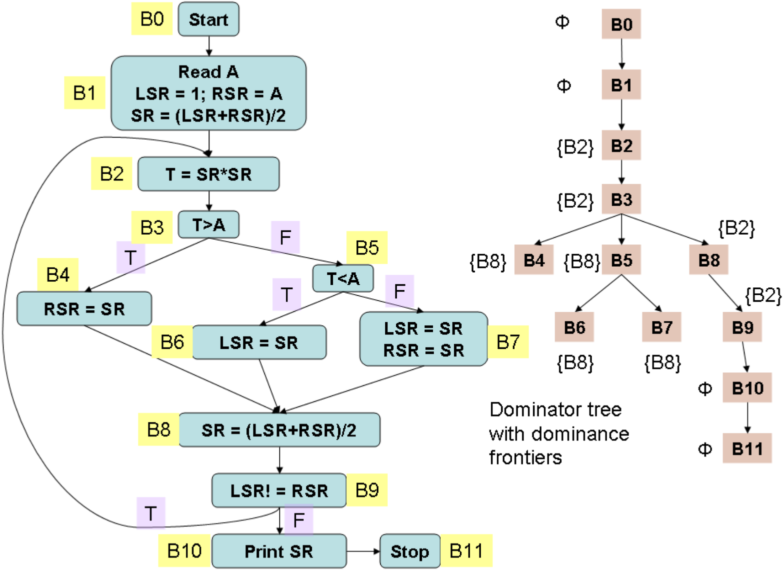
- A definition in node *n* forces a ϕ-function in join nodes that lie just outside the region of the flow graph that *n* dominates; hence the name *dominance frontier*
- Informally, $DF(x)$ contains the *first* nodes reachable from *x* that *x* does not dominate, on *each* path leaving *x*
  - In example 1 (next slide), $DF(B1) = \emptyset$, since B1 dominates all nodes in the flow graph except *Start* and B1, and there is no path from B1 to *Start* or B1
  - In the same example, $DF(B2) = \{B2\}$, since B2 dominates all nodes except *Start*, B1, and B2, and there is a path from B2 to B2 (via the back edge)
  - Continuing in the same example, B5, B6, and B7 do not dominate any node and the first reachable nodes are B7, B7, and B2 (respectively). Therefore, $DF(B5) = DF(B6) = \{B7\}$ and $DF(B7) = \{B2\}$
  - In example 2 (second next slide), B5 dominates B6 and B7, but not B8; B8 is the first reachable node from B5 that B5 does not dominate; therefore, $DF(B5) = \{B8\}$

DF Example - 1


Start
i = 0
read n
B1
B2
n <> 1
B3
even(n)
print i
B4
B6
n = n/2
n = 3*n+1
Stop
B5
i = i+1
B7


Start Φ
B1 Φ
B2 {B2}
B3 {B2}
B4 Φ
B5 {B7}
B6 {B7}
B7 {B2}
Stop Φ

DF Example - 2


B0
Start
B1
Read A
LSR = 1; RSR = A
SR = (LSR+RSR)/2
B2
T = SR*SR
B3
T>A
T
F
B4
RSR = SR
B5
T<A
T
F
B6
LSR = SR
B7
LSR = SR
RSR = SR
B8
SR = (LSR+RSR)/2
B9
LSR! = RSR
T
F
B10
Print SR
Stop
B11
Φ B0
Φ B1
{B2} B2
{B2} B3
{B8} B4
{B8} B5
{B2} B8
B6 {B8}
B7 {B8}
{B2} B9
Φ B10
Φ B11
Dominator tree
with dominance
frontiers

Computation of Dominance Frontiers - 2

1. Identify each join node x in the flow graph
2. For each predecessor, p of x in the flow graph, traverse the dominator tree upwards from p, till $idom(x)$
3. During this traversal, add x to the DF-set of each node met

- In example 1 (second previous slide), consider the join node B2; its predecessors are B1 and B7
  - B1 is also $idom(B2)$ and hence is not considered
  - Starting from B7 in the dominator tree, in the upward traversal till B1 (i.e., $idom(B2)$) B2 is added to the DF sets of B7, B3, and B2
- In example 2 (previous slide), consider the join node B8; its predecessors are B4, B6, and B7
  - Consider B4: B8 is added to $DF(B4)$
  - Consider B6: B8 is added to $DF(B6)$ and $DF(B5)$
  - Consider B7: B8 is added to $DF(B7)$; B8 has already been added to $DF(B5)$
  - All the above traversals stop at B3, which is $idom(B8)$

DF Algorithm

```
{
  for all nodes n in the flow graph do
  DF(n) = ∅;
  for all nodes n in the flow graph do {
  /* It is enough to consider only join nodes */
  /* Other nodes automatically get their DF sets /*
  /* computed during this process /*
    for each predecessor p of n in the flow graph do {
       t = p;
       while (t ≠ idom(n)) do {
          DF(t) = DF(t) ∪ {n};
          t = idom(t);
       }
    }
  }
}
```

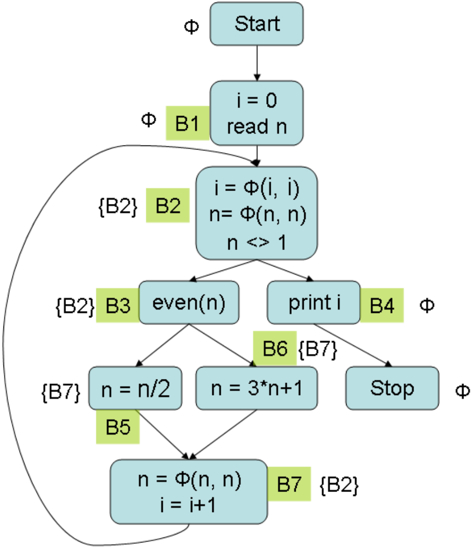
Minimal SSA Form Construction 1

1. Compute *DF* sets for each node of the flow graph
2. For each variable v, place trivial ϕ-functions in the nodes of the flow graph using the algorithm *place-phi-function(v)*
3. Rename variables using the algorithm *Rename-variables(x,B)*

ϕ-Placement Algorithm

- The ϕ-placement algorithm picks the nodes n_i with assignments to a variable
- It places trivial ϕ-functions in all the nodes which are in $DF(n_i)$, for each i
- It uses a work list (i.e., queue) for this purpose

ϕ-function placement Example



Dominance frontier is written beside BB no.

The function *place-phi-function(v)* - 1

```
function Place-phi-function(v) // v is a variable
// This function is executed once for each variable in the flow graph
begin
  // has-phi(B) is true if a φ-function has already
  // been placed in B
  // processed(B) is true if B has already been processed once
  // for variable v
  for all nodes B in the flow graph do
     has-phi(B) = false; processed(B) = false;
  end for
  W = ∅; // W is the work list
  // Assignment-nodes(v) is the set of nodes containing
  // statements assigning to v
  for all nodes B ∈ Assignment-nodes(v) do
     processed(B) = true; Add(W, B);
  end for
```

The function *place-phi-function(v)* - 2

```
while W ≠ ∅ do
begin
  B = Remove(W);
  for all nodes y ∈ DF(B) do
    if (not has-phi(y)) then
    begin
      place < v = φ(v, v, ..., v) > in y;
      has-phi(y) = true;
      if (not processed(y)) then
      begin processed(y) = true;
          Add(W, y);
      end
    end
  end for
end
end
```