Basics and Constant Propagation

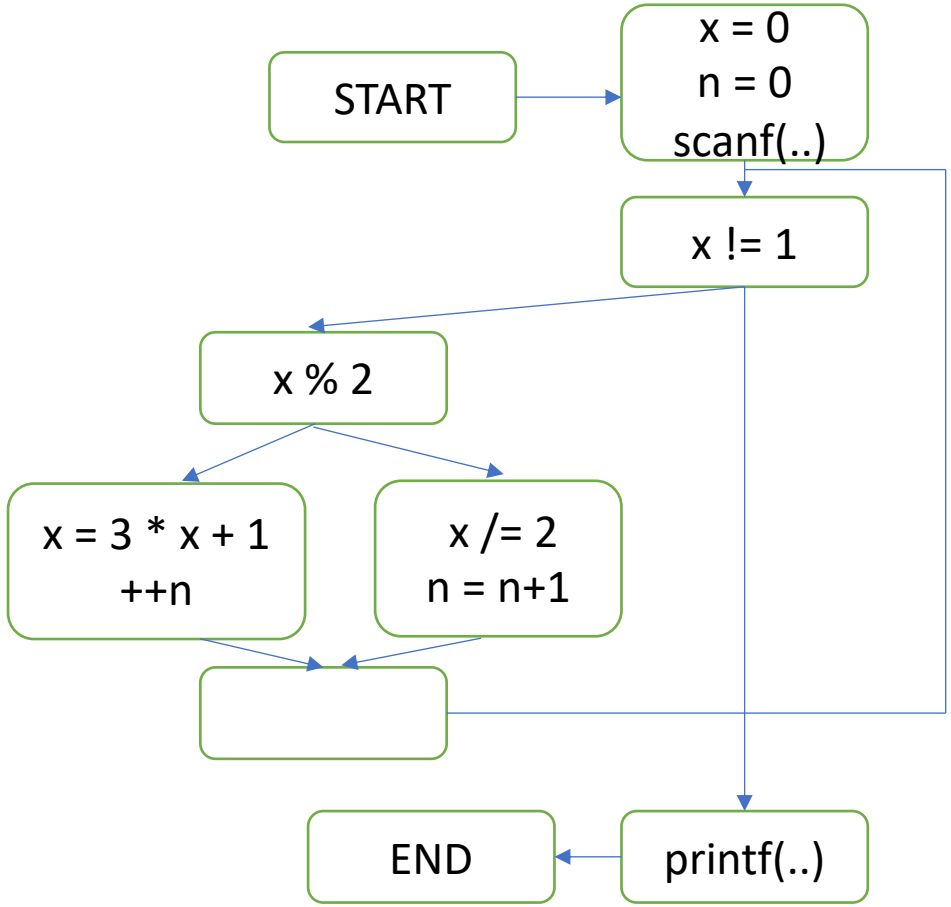
Control-Flow Graph

- A graph representation of intermediate code
- With nodes as basic- blocks and edges as the control-flow

- **Basic blocks** are maximal sequences of consecutive three-address instructions with the properties that
  - Flow of control can only enter the basic block through the first instruction in the block (no jumps into the middle of the block)
  - Control will leave the block without halting or branching except possibly at the last instruction in the block

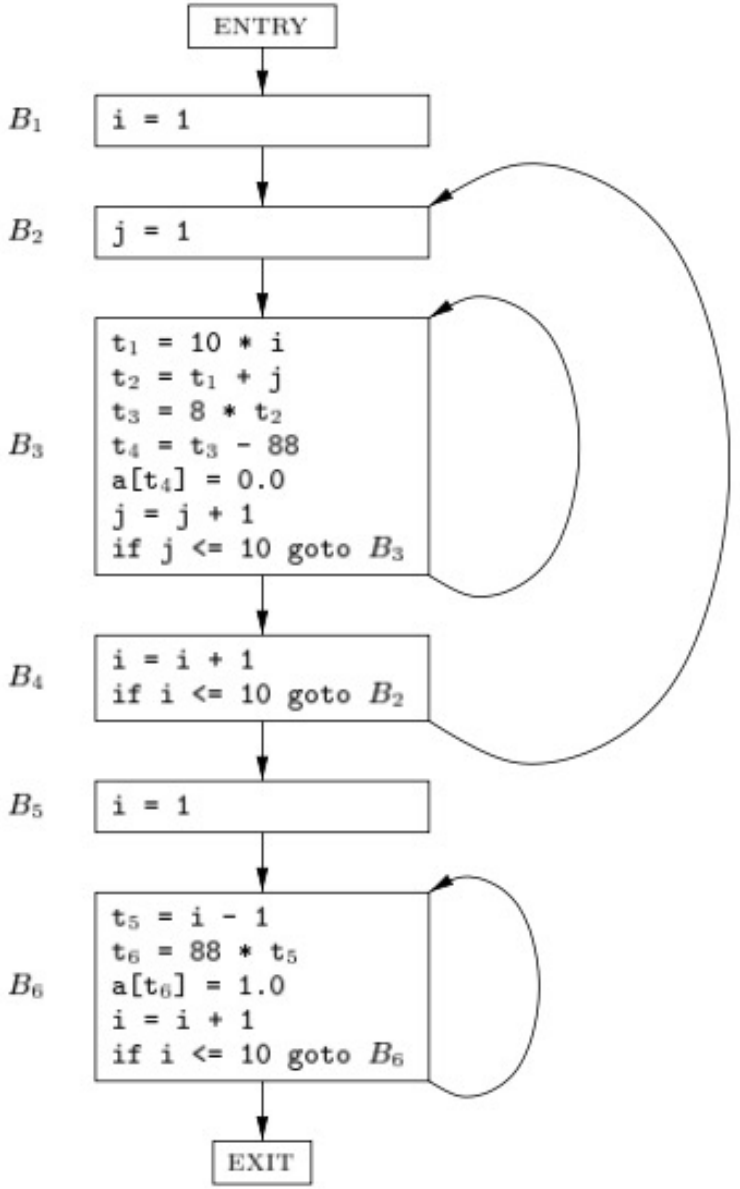
Example: CFG

```
int main() {
    int x = 0, n = 0;
    scanf("%d", &x);
   while (x != 1) {
        if (x % 2) {
            x = 3 * x + 1;
            ++n;
        } else {
            x /= 2;
            n = n+1;
        }
    }
    printf("%d\n", n);
}
```



Example with IR

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```



Data-flow Analysis Preliminaries

- **Lattice**: Algebraic structure with elements on which a data-flow analysis is performed.
- **Meet**($\sqcap$) **and Join**($\sqcup$): Operations on elements of lattice L
- **Unique elements of *L*:** bottom ($\bot$), and top ($\top$); $\forall x \in L$, $x \sqcap \bot = \bot$ and $x \sqcup \top = \top$.
- **Flow Function**: Abstracts the effect of a program construct to its effect on the corresponding lattice elements.
- **Monotonicity**: A function f: $L \rightarrow L$ is a monotone, if for all $x,y \in L$, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.
- **Fixed Point**: A fixed point of a function f: $L \rightarrow L$ is an element $z \in L$, such that $f(z) = z$.
- **Termination**: Finite lattice height combined with monotonicity of flow functions guarantee termination of data-flow analysis algorithms

Constant Propagation: Example

- Aims at determining the variables that have constant values on all executions of a program

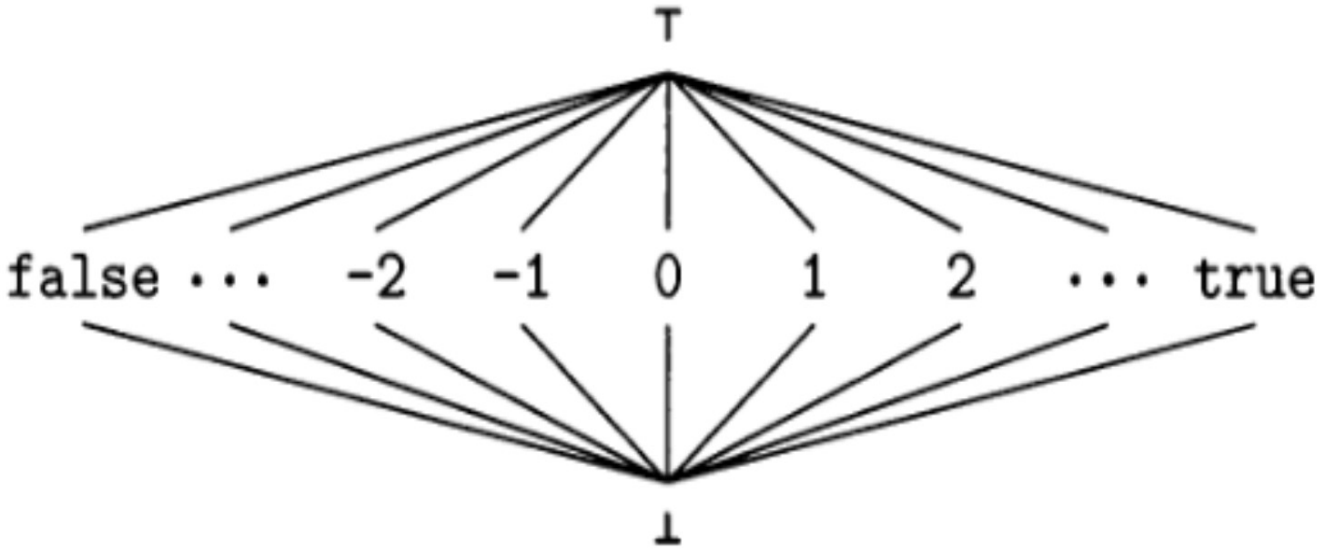
```
1.  a = 1;
2.  b = 4;
3.  while (a > 0) {
4.          if (a == 1) {
5.                  c = 9;
6.                  a = 4;
7.          } else {
8.                  c = b + 5;
9.                  a = a – b;
10.         }
11.         print c;
12. }
```

1. Is the value of ‘b’ constant at Line 8?
2. Is the value of ‘a’ constant at Line 4?
3. Is the value of ‘c’ constant at Line 11?
4. What is the output of the program?

Uses of constant propagation

- Expressions can be evaluated at compile time (such expression in loops save many evaluations at execution time)
- Dead code elimination: code that is never executed can be deleted
- Procedure integration or function inlining

Lattice for constant propagation



Constant Propagation: Lattice Meet Rules

- $\bot$ = Constant value cannot be guaranteed
- $\top$ = May be a constant, not yet determined (init values)

- $\boldsymbol{x} \sqcap \top = \boldsymbol{x}$ and $\boldsymbol{x} \sqcap \bot = \bot$, $\forall x \in L$

- $\boldsymbol{c1} \sqcap \boldsymbol{c1} = \boldsymbol{c1}$, $c1$ is a constant

- $\boldsymbol{c2} \sqcap \boldsymbol{c1} = \bot$, $c1$ and $c2$ are constants

Iterative Data-Flow Analysis

- Input:
  - A flow graph $G = (N,E)$ with entry and exit
  - A lattice L
- Output:
  - *in*(B) for each B $\in N$
  - *out*(B) for each B $\in N$
- Computation: Solves the following data-flow equations iteratively

$$in(B) = \begin{cases} Init & \text{for } B = \texttt{entry} \\ \bigsqcap_{P \in Pred(B)} out(P) & \text{otherwise} \end{cases}$$

$$out(B) = F_B(in(B))$$

- *'Init'* represents appropriate initial value for the data-flow information

Iterative Data-Flow Analysis (Kildall's worklist-based algorithm)

```
procedure Worklist_Iterate(N,entry,F,dfin,Init)
   N: in set of Node
   entry: in Node
   F: in Node × L ⟶ L
   dfin: out Node ⟶ L
   Init: in L
begin
   B, P: Node
   Worklist: set of Node
   effect, totaleffect: L
   dfin(entry) := Init
   Worklist := N - {entry}
   for each B ∈ N do
      dfin(B) := ⊤
   od
   repeat
      B := ♦Worklist
      Worklist -= {B}
      totaleffect := ⊤
      for each P ∈ Pred(B) do
         effect := F(P,dfin(P))
         totaleffect ⊓= effect
      od
      if dfin(B) ≠ totaleffect then
         dfin(B) := totaleffect
         Worklist ∪= Succ(B)
      fi
   until Worklist = ∅
end    || Worklist_Iterate
```

Initialize worklist and *in*() of all the nodes in CFG

Apply flow function

Propagate data-flow facts to successors

Constant Propagation: Statements of Interest

Assignment Statements:

- **y = c**, y ∈ *Var*, c ∈ *Const*
  - y = 5;
- **y = input(),** y ∈ *Var*
  - scanf(“%d”,&y);
- **y = z**, y ∈ *Var*, z ∈ *Var*
- **y = e**, y ∈ *Var*, e ∈ *Expr*
  - ‘y = z + 5;’, ‘y = z * w;’, ‘y = y/z;’ etc.

IDFA Specific to Constant Propagation

- Assumption: one basic block per statement and start with the entry node
- Data flow values are given by a map m: $Var \rightarrow Vals$ (from Lattice L)
- Effect of a flow function f_s for a statement s: $m' = f_s(m)$
- Effect of a flow function on statements other than assignment statements is: $m' = m \Rightarrow f_s$ is identity function
- At a merge point, get a meet of the flow maps

More on Flow Function f_s

If s is an assignment statement to variable v, then $f_s(m) = m'$, where:

- For all $v' \neq v$, $m'(v') = m(v')$
- If the RHS of the statement is a constant c, then $m'(v) = c$
- If the RHS is an expression (say y *op* z),

$$m'(v) = \begin{cases} m(y) \; op \; m(z), & \text{if m(y) and m(z) are constant values} \\ \bot, & \text{if either of } m(y) \text{ and } m(z) \text{ is } \bot \\ \top, & \text{Otherwise} \end{cases}$$

- If the RHS is an expression that cannot be evaluated, then then $m'(v) = \bot$

Working Example

```
1.      x = 10;
2.      y = 1;
3.      z = 5;
4.      if (cond) {
5.          y = y/x;
6.          x = x - 1;
7.          z = z + 1;
8.      } else {
9.          z = z + y;
10.         y = 0;
11.     }
12.     print x + y + z;
```

	Map
0. Entry	x → T, y → T, z → T
1. x = 10;	
2. y = 1;	
3. z = 5;	
4. **if** (cond) {	
5. y = y/x;	
6. x = x - 1;	
7. z = z + 1;	
8. } **else** {	
9. z = z + y;	
10. y = 0;	
12. **print** x + y + z;	

Working Example

```
1.      x = 10;
2.      y = 1;
3.      z = 5;
4.      if (cond) {
5.          y = y/x;
6.          x = x - 1;
7.          z = z + 1;
8.      } else {
9.          z = z + y;
10.         y = 0;
11.     }
12.     print x + y + z;
```

	Map
0. Entry	x → T, y → T, z → T
1. x = 10;	x → 10, y → T, z → T
2. y = 1;	x → 10, y → 1, z → T
3. z = 5;	x → 10, y → 1, z → 5
4. **if** (cond) {	x → 10, y → 1, z → 5
5. y = y/x;	
6. x = x - 1;	
7. z = z + 1;	
8. } **else** {	x → 10, y → 1, z → 5
9. z = z + y;	
10. y = 0;	
12. **print** x + y + z;	

Working Example

```
1.      x = 10;
2.      y = 1;
3.      z = 5;
4.      if (cond) {
5.          y = y/x;
6.          x = x - 1;
7.          z = z + 1;
8.      } else {
9.          z = z + y;
10.         y = 0;
11.     }
12.     print x + y + z;
```

	Map
0. Entry	x → T, y → T, z → T
1. x = 10;	x → 10, y → T, z → T
2. y = 1;	x → 10, y → 1, z → T
3. z = 5;	x → 10, y → 1, z → 5
4. **if** (cond) {	x → 10, y → 1, z → 5
5. y = y/x;	x → 10, y → 0, z → 5
6. x = x - 1;	x → 9, y → 0, z → 5
7. z = z + 1;	x → 9, y → 0, z → 6
8. } **else** {	x → 10, y → 1, z → 5
9. z = z + y;	
10. y = 0;	
12. **print** x + y + z;	

Working Example

```
1.      x = 10;
2.      y = 1;
3.      z = 5;
4.      if (cond) {
5.          y = y/x;
6.          x = x - 1;
7.          z = z + 1;
8.      } else {
9.          z = z + y;
10.         y = 0;
11.     }
12.     print x + y + z;
```

	Map
0. Entry	x → T, y → T, z → T
1. x = 10;	x → 10, y → T, z → T
2. y = 1;	x → 10, y → 1, z → T
3. z = 5;	x → 10, y → 1, z → 5
4. **if** (cond) {	x → 10, y → 1, z → 5
5. y = y/x;	x → 10, y → 0, z → 5
6. x = x - 1;	x → 9, y → 0, z → 5
7. z = z + 1;	x → 9, y → 0, z → 6
8. } **else** {	x → 10, y → 1, z → 5
9. z = z + y;	x → 10, y → 1, z → 6
10. y = 0;	x → 10, y → 0, z → 6
12. **print** x + y + z;	

Working Example

```
1.      x = 10;
2.      y = 1;
3.      z = 5;
4.      if (cond) {
5.          y = y/x;
6.          x = x - 1;
7.          z = z + 1;
8.      } else {
9.          z = z + y;
10.         y = 0;
11.     }
12.     print x + y + z;
```

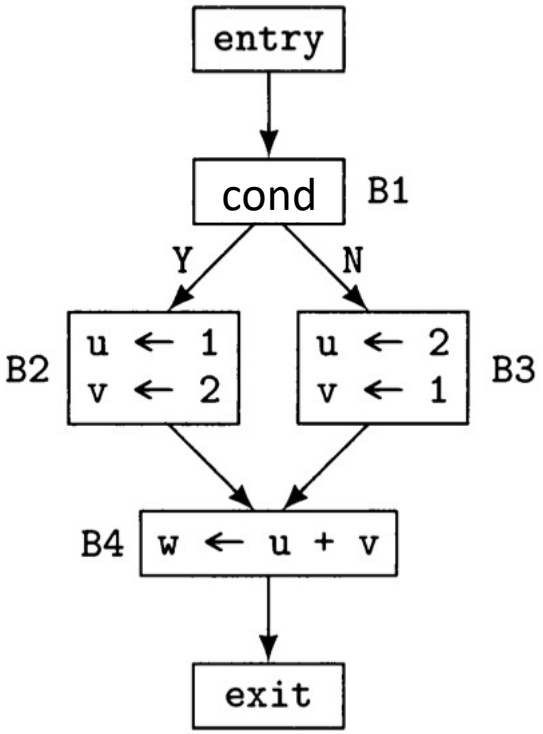
	Map
0. Entry	x → T, y → T, z → T
1. x = 10;	x → 10, y → T, z → T
2. y = 1;	x → 10, y → 1, z → T
3. z = 5;	x → 10, y → 1, z → 5
4. **if** (cond) {	x → 10, y → 1, z → 5
5. y = y/x;	x → 10, y → 0, z → 5
6. x = x - 1;	x → 9, y → 0, z → 5
7. z = z + 1;	x → 9, y → 0, z → 6
8. } **else** {	x → 10, y → 1, z → 5
9. z = z + y;	x → 10, y → 1, z → 6
10. y = 0;	x → 10, y → 0, z → 6
12. **print** x + y + z;	[x → 9, y → 0, z → 6] ⊓ [x → 10, y → 0, z → 6] = [x → ⊥, y → 0, z → 6]

Example for you to try

```
1.  x = 10;
2.  y = 1;
3.  z = 1;
4.  while (x > 1) {
5       y = x * y;
6.      x = x – 1;
7.      z = z * z; }
8.  p = x + y + z;
```

	Iteration1	Iteration2	Iteration3
Entry	x → T, y → T, z → T, p → T	{T, T, T, T}	{T, T, T, T}
x = 10;	{10, T, T, T}	{10, T, T, T}	{10, T, T, T}
y = 1;	{10, 1, T, T}	{10, 1, T, T}	{10, 1, T, T}
z = 1;	{10, 1, 1, T}	{10, 1, 1, T}	{10, 1, 1, T}
while (x > 1) {	{10, 1, 1, T}	{10, 1, 1, T} ⊓ {9, 10, 1, T} = {⊥, ⊥, 1, T}	{⊥, ⊥, 1, T}
y = x * y;	{10, 10, 1, T}	{⊥, ⊥, 1, T}	{⊥, ⊥, 1, T}
x = x – 1;	{9, 10, 1, T}	{⊥, ⊥, 1, T}	{⊥, ⊥, 1, T}
z = z * z; }	{9, 10, 1, T}	{⊥, ⊥, 1, T}	{⊥, ⊥, 1, T}
p = x + y + z;	{10, 1, 1, 12}	{⊥, ⊥, 1, ⊥}	{⊥, ⊥, 1, ⊥}

Distributive property



$[u \to 1, v \to 2, w \to T] \sqcap [u \to 2, v \to 1, w \to T] = [u \to \bot, v \to \bot, w \to T]$

Effect f4 $\Rightarrow [u \to \bot, v \to \bot, w \to \bot]$

Effect f4: $[u \to 1, v \to 2, w \to T] \Rightarrow [u \to 1, v \to 2, w \to 3]$,
Effect f4: $[u \to 2, v \to 1, w \to T] \Rightarrow [u \to 2, v \to 1, w \to 3]$

$[u \to 1, v \to 2, w \to 3] \sqcap [u \to 2, v \to 1, w \to 3] = [u \to \bot, v \to \bot, w \to 3]$

$f4(f2(m1) \sqcap f3(m1)) \sqsubseteq f4(f2(m1)) \sqcap f4(f3(m1))$

Conditional Constant Propagation

Idea: Process a block only if it is found executable.

```
x = 1;

…
if (x == 1) {
      y = 1;
} else {
      y = 10;
}
print x + y ;
```

- Mark the “entry” node executable.
- If a node has only one successor, then mark the successor “executable”.
- If a conditional branch node has more than one successor: evaluate the condition (based on the abstract values) and appropriately mark the successors as executable or not.
- Iterate till there is no change (in the flow maps and the list of executable nodes).