CS614: Advanced Compilers

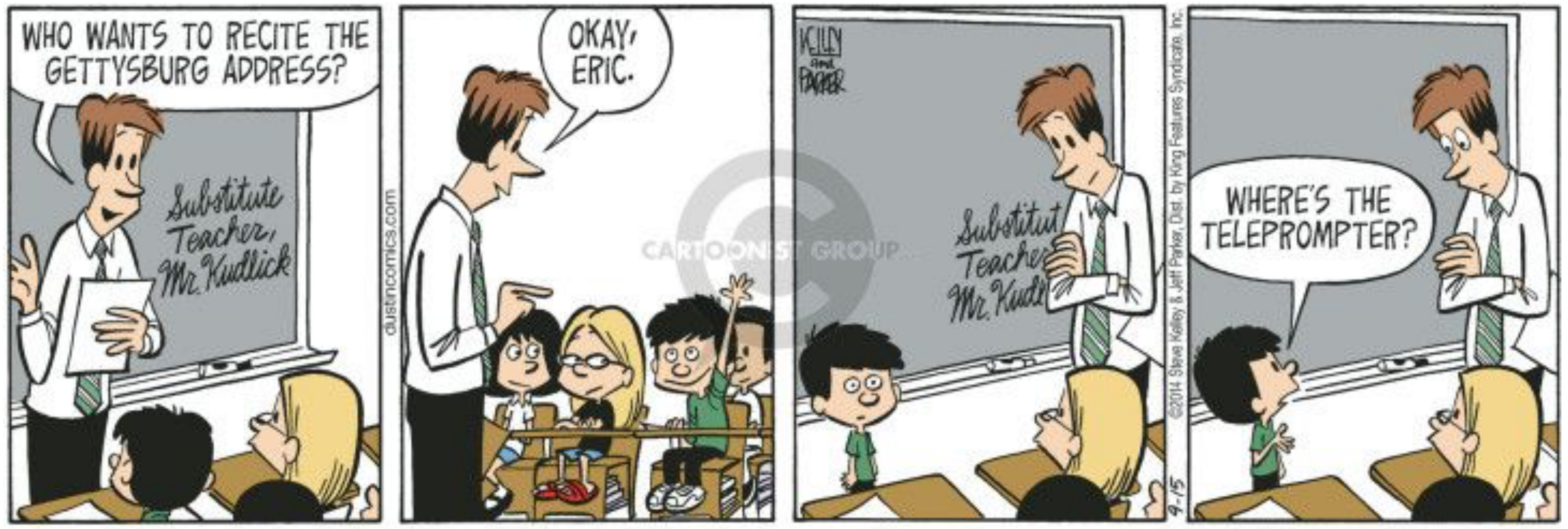
Constant Propagation. SSA Form

Manas Thakur
CSE, IIT Bombay

Spring 2025

Things we learnt in the last class

- Control-flow graphs as a program representation for compiler backends
  - Make the flow of control explicit
  - Can work with a small set of jump instructions
- Performing iterative dataflow analysis while modeling the control flow
  - Fixed points, monotonicity, finiteness of dataflow values
- Example analyses:
  - Forward: Common subexpressions
  - Backward: Live variables



Simple Constant Propagation (+Folding)

```
a = 10;
b = 20;
c = a + b;
```

➡

```
a = 10;
b = 20;
c = 30;
```

```
a = 10;
if (i > j)
   b = a;
else
   c = a;
```

➡

```
a = 10;
if (i > j)
   b = 10;
else
   c = 10;
```

Flow-Insensitive Constant Propagation

```
a = 10;
b = 20;
c = a + b;
a = 30;
d = a + 5;
```

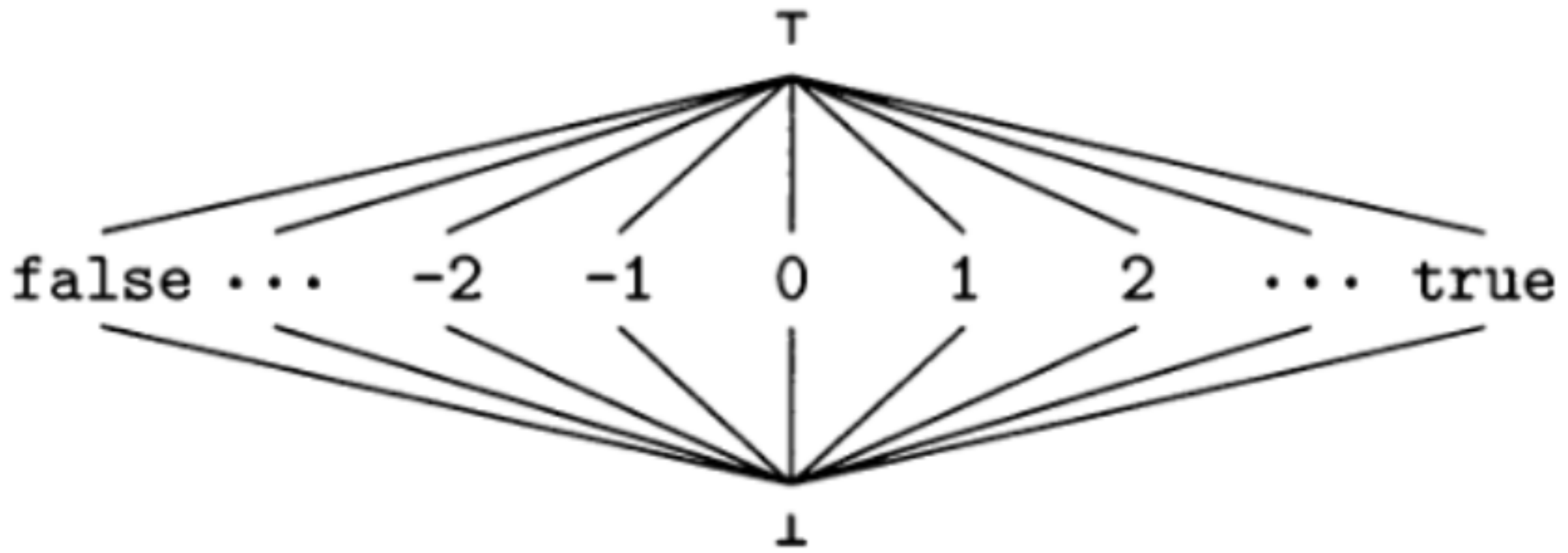
```
a = 10;
b = 20;
c = a + 20;
a = 30;
d = a + 5;
```

```
a = 10;
b = 20;
c = 30;
a = 30;
d = 35;
```

Flow-Sensitive CP

Simple Constant Propagation IDFA

- Dataflow value:
  - A map from variables to constants
- Direction:
  - Forward
- Which all constants:
  - There are an infinite of them!
  - Let's assume a **lattice** formed by the set of constants
    - where ⊤ (\top) means "not yet known" and ⊥ (\bot) means "not a constant"



Simple Constant Propagation IDFA

```
for each n {
    for each v:
        IN[n,v] = \top
    for each v:
        OUT[n,v] = \top
}
repeat
    for each n {
        save older values of IN and OUT
        for each v in USE[n] {
            IN[n,v] = IN[n,v] \meet OUT[p,v] for each predecessor p of n
        }
        OUT[n,v] = copy(IN[n,v])
        for each v in DEF[n] {
            switch (n) {
                case "v = \cons":
                    OUT[n,v] = \cons
                case "v = w":
                    OUT[n,v] = IN[n,w]
                case "v = w1 op w2":
                    OUT[n,v] = IN[n,w1] op IN[n,w2]
            }
        }
    }
until fixed-point
```

Initialization

Iterate over all nodes until fixed point

Dataflow computation

Worklist-based Simple Constant Propagation

Initialization

```
worklist = {All stmts of the form "v = \cons"}
while !worklist.isEmpty() {
    n = worklist.removeOne()
    save older values of IN and OUT
```

Dataflow computation

```
    if OUT[n] changed:
        worklist.addAll(succ[n])
}
```

Add only dependents to worklist;
stop when no more work is left

Simple Flow-Insensitive Constant Propagation

```
for each variable v:
    VAL(v) = \top
```

Single global information about all variables

```
repeat
    for each n {
        for each v in DEF[n] {
            switch (n) {
                case "v = \cons":
                    VAL[v] = VAL[v] \meet \cons
                case "v = w":
                    VAL[v] = VAL[v] \meet VAL[w]
                case "v = w1 op w2":
                    VAL[v] = VAL[v] \meet (VAL[w1] op VAL[w2])
            }
        }
    }
until fixed-point
```

Dataflow computation until fixed point

Usually very fast and memory efficient, but very imprecise

Achieving efficiency for *flow-sensitive* analyses

```
S1: y = 1;
S2: y = 2;
S3: x = y;
```

- What's the advantage if the above program is rewritten as follows?

```
S1: y1 = 1;
S2: y2 = 2;
S3: x = y2;
```

- Def-use becomes explicit; analysis can become faster (when and when not?)

Can we always just rename variables to get to this form?

Static Single Assignment (SSA) Form

- A form of IR in which each use can be mapped to a single definition.
- Achieved using variable renaming and phi nodes.

```
if (flag)
    x = -1;
else
    x = 1;
y = x * a;
```

→

$$
\begin{array}{l}
\texttt{if (flag)} \\
\quad x_1 = -1; \\
\texttt{else} \\
\quad x_2 = 1; \\
x_3 = \Phi(x_1, x_2) \\
y = x_3 * a;
\end{array}
$$

- Many (most!) compilers use SSA form in their IRs.

SSA Classwork

- Convert the following program to SSA form:
  - (Hint: First convert to 3AC)

```
x = 0;
for (i=0; i<N; ++i) {
   x += i;
   i = i + 1;
   x--;
}
x = x + i;
```

```
x = 0
i = 0
L1: if i >= N goto L2
x = x + i
i = i + 1
x = x − 1
i = i + 1
goto L1
L2: x = x + i;
```

$x_1 = 0;$
$i_1 = 0;$
$L_1\text{:}\ i_{13} = \Phi(i_1, i_3);$
$\text{if } i_{13} >= N \text{ goto } L_2$
$x_{13} = \Phi(x_1, x_3);$
$x_2 = x_{13} + i_{13};$
$i_2 = i_{13} + 1;$
$x_3 = x_2 - 1;$
$i_3 = i_2 + 1;$
$\text{goto } L_1;$
$L_2\text{:}\ x_4 = \Phi(x_1, x_3);$
$x_5 = x_4 + i_{13};$

Constructing SSA Form

SSA Construction

- Three steps:
  - Rename variables that are assigned more than once.
  - Replace uses of renamed variables based on reaching definitions.
  - In case of multiple reaching definitions, insert a Φ (phi) function that gathers all the definitions of the variable into a new variable.

- Notes:
  - Φ functions have no equivalence in hardware.
  - They need to be removed after performing enabled optimizations.

Insertion of Φ functions

- Intuitively:
  - If two paths in a CFG with a definition of a variable v converge at a node n, then we need a Φ function at node n.
  - The number of arguments of the Φ function is the same as the *in-degree* of n.

- A Φ function is also an assignment to the variable being addressed, so a Φ-insertion may lead to the insertion of more Φ-assignments at other nodes.

- We need an algorithm to convert a given CFG to SSA form with appropriate insertion of Φ functions.