



# CSCI 742 - Compiler Construction

---

Lecture 33

Live Variable Analysis

Instructor: Hossein Hojjat

April 16, 2018

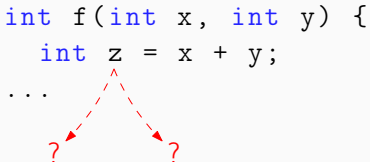
## Recap: Control Flow Graphs

- **Control Flow Graph (CFG):** graph representation of computation and control flow in the program
- Framework to statically analyze program control-flow
- Next: use CFG to statically extract information about program
- Reason at compile-time about run-time values of variables in all program executions
- Data-flow analysis: gather information about the possible set of values of variables at various points in a program

# Liveness

- Liveness is a data-flow property of variables:  
“Is the value of this variable needed?”
- Optimization: eliminate assignments to dead variables  
(i.e. variables that are never used after definitions)

```
int f(int x, int y) {  
    int z = x + y;  
    ...  
    ?  
    ?
```



- Live variable analysis is **undecidable** in general
- We compute a syntactic and conservative approximation of liveness
  - Like many other data-flow analysis techniques

```
int f(int x, int y) {  
    int z = x + y;  
    if (tricky-calculation) x = z;  
    return x;  
}
```

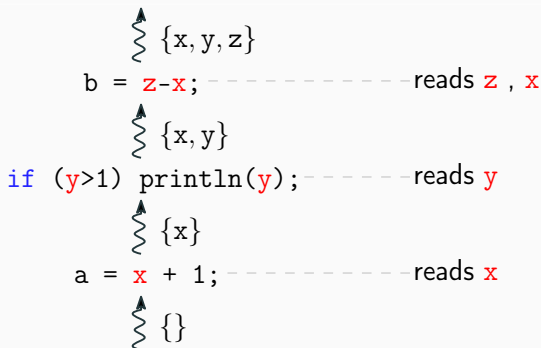
## Live Variable Analysis: Backward

- Liveness is naturally computed using **backward** data-flow analysis
- Usage information from future statements must be propagated backward through the program to discover which variables are live

```
int f(int x, int y) {  
    int z = x + y;  
    ...  
int t=z-2;    println(z);  
              if(z!=2) ...;
```

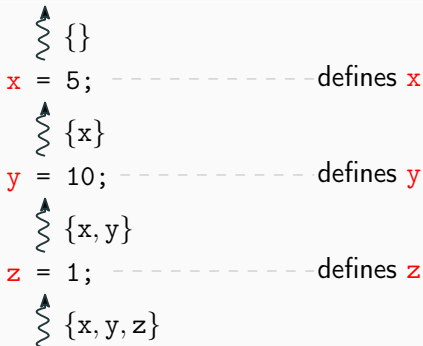
# Live Variable Analysis

- Variable liveness flows backward through the program
- Each statement has an effect on liveness information as it flows past
- A statement makes a variable **live** when it reads it



# Live Variable Analysis

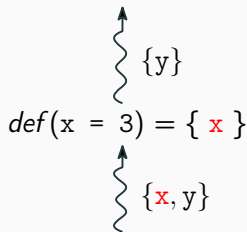
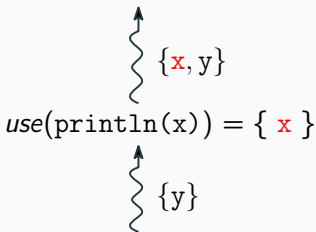
- Variable liveness flows backward through the program
- Each statement has an effect on liveness information as it flows past
- A statement makes a variable **dead** when it defines (assigns to) it



# Live Variable Analysis

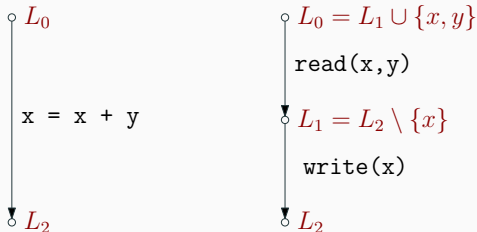
As liveness flows backwards past an statement, we modify liveness information:

- Add any variables which it reads (they become live)
- Remove any variables which it defines (they become dead)



# Live Variable Analysis

- If a statement both references and defines variables, remove the defined variables before adding the read ones
- $L_0$  Initial set of live variables



$$L_0 = (L_2 \setminus \{x\}) \cup \{x, y\}$$

In general:

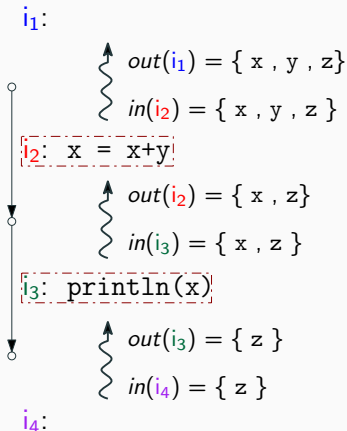
- $in(S)$ : set of live variables immediately before statement  $S$
- $out(S)$ : set of live variables immediately after statement  $S$

$$in(S) = (out(S) \setminus def(S)) \cup use(S)$$



# Straight-Line Code

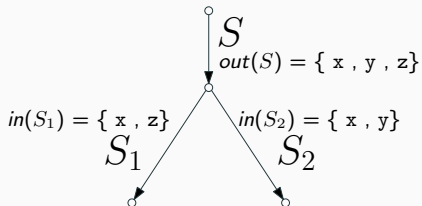
- In straight-line code each node has a unique successor
- Variables live at the exit of a node are exactly those variables live at the entry of its successor



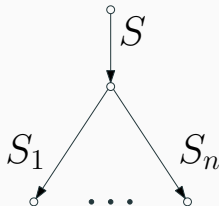
# Multiple Successors

- In general each node has an arbitrary number of successors
- Variables live at the exit of a node are exactly those variables live at the entry of all its successors

**Example:**



**General:**



$$out(S) = \bigcup_{S_i \in succ(S)} in(S_i)$$

# Data-flow Equations

- Start with CFG and derive a system of constraints between live variable sets

$$\begin{aligned}in(S) &= (out(S) \setminus def(S)) \cup use(S) \\ out(S) &= \bigcup_{S_i \in succ(S)} in(S_i)\end{aligned}$$

Solve constraints:

- Start with empty sets of live variables
- Iteratively apply constraints
- Stop when we reach a fixed point

# Constraint Solving Algorithm

**for** all statements  $S$  **do**

$$in(S) = out(S) = \emptyset$$

**repeat**

**select** a statement  $S$  such that

$$in(S) \neq (out(S) \setminus def(S)) \cup use(S)$$

or (respectively)

$$out(S) \neq \bigcup_{S_i \in succ(S)} in(S_i)$$

update  $in(S)$  (or  $out(S)$ ) accordingly

**until** no such change is possible

## Exercise

- Compute the set of live variables at each point of the program

```
x = 5;
y = 10;
z = 0;
while (x > 0) {
  x = x - 1;
  u = y;
  while (u > 0) {
    u = u - 1;
    z = z + 1;
  }
}
```

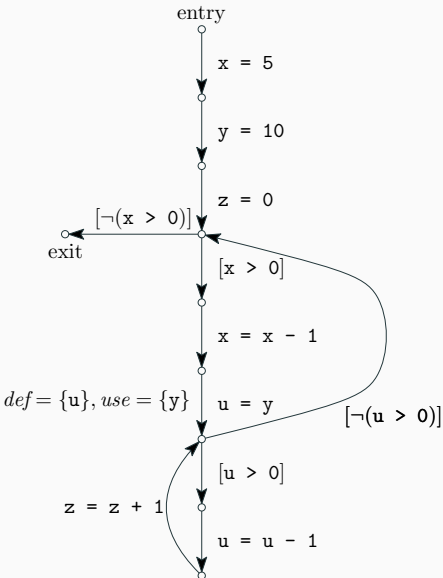
# Exercise

- Compute the set of live variables at each point of the program

```
x = 5;
y = 10;
z = 0;
while (x > 0) {
  x = x - 1;
  u = y;
  while (u > 0) {
    u = u - 1;
    z = z + 1;
  }
}
```

$$in(S) = (out(S) \setminus def(S)) \cup use(S)$$

$$out(S) = \bigcup_{S_i \in succ(S)} in(S_i)$$



# Intra/Inter-Procedural Analysis

- **Intra-procedural** Analysis: analyzing the body of a single procedure
- **Inter-procedural** Analysis: analyzing the whole program with function calls

```
x = 0;  
  // is y live here? (yes iff used in procedure P)  
P ();  
  // is x still equal to 0 here?  
  // (yes iff not changed in P)  
y = x;
```

# Intra/Inter-Procedural Analysis

A naïve and safe approach to inter-procedural analysis:

- Assume any function will read and write all global variables (worst case scenario)
  - Every global variable is live before any function call!
- 
- Leads to over-cautious optimizations
  - There are more accurate inter-procedural analyses that consider the call graph of a program
    - (beyond the scope of the course)