CS164 Programming Languages and Compilers **Spring 2024**

Discussion Worksheet 8: Week of 4/3

1 CFG Generation

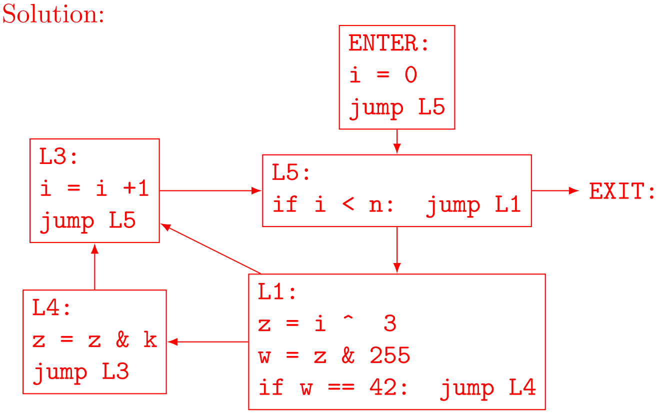
Before many local or global optimizations, we must rewrite our program's code as a Control Flow Graph. A CFG is a directed graph in which each node is a basic block, and each edge represents the ability to jump from the end of one basic block to the beginning of another.
A basic block is a maximal sequence of instructions where only the first instruction may be a label, and only the last instruction may be a branch/jump. This enforces that the instructions within a basic block always execute sequentially, and that non-linear control flow only occurs in between basic blocks.

Exercise 1. Convert the following IL code fragments to CFGs.

1.1

```
ENTER:
    i = 0
    jump L5
L1:
    z = i ^ 3
    w = z & 255
    if w == 42: jump L4
L3:
    i = i + 1
    jump L5
L4:
    z = z & k
    jump L3
L5:
    if i < n: jump L1
EXIT:
```

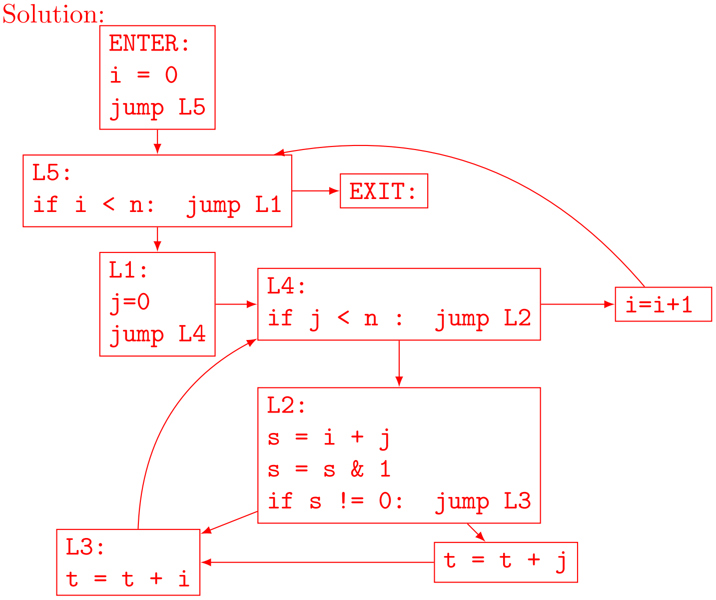
Solution:



1.2

```
ENTER:
    i = 0
    jump L5
L1:
    j = 0
    jump L4
L2:
    s = i + j
    s = s & 1
    if s != 0: jump L3
    t = t + j
L3:
    t = t + i
L4:
    if j < n: jump L2
    i = i + 1
L5:
    if i < n: jump L1
EXIT:
```

Solution:



2 Local Optimization

After algebraic simplification, there are three primary methods of local optimization we will focus on:

1. **Dead Code Elimination**: If `w := rhs` appears in a basic block and `w` does not appear anywhere else in the program, then the statement is dead and can be eliminated.
2. **Common Subexpression Elimination**: If two different expressions have the same rhs, we may change the rhs of the second expression to be the temporary saved from the first expression. (Requires Single Assignment Form)
3. **Copy Propagation**: If w := x appears in a basic block (where x is a variable), all subsequent uses of w can be replaced by x. (Requires Single Assignment Form)

Exercise 2 . Consider the following basic block. Suppose only y and z are live (y and z are being used somewhere else in the program) at the end of the basic block.

```
x := a + b
t := a
u := b
z := t + u
y := a + b
```

Perform the following sequences of optimizations on this code. *When local optimization requires Single Assignment Form, you may assume we will provide it.*

1. Perform Common Subexpression Elimination, followed by Dead Code Elimination, followed by Copy Propagation.

   Solution:
   ```
   x := a + b
   t := a
   u := b
   z := a + b
   y := x
   ```

2. Perform Dead Code Elimination, followed by Copy Propagation, followed by Common Subexpression Elimination.

   Solution:
   ```
   t := a
   u := b
   z := a + b
   y := z
   ```

3. Perform Copy Propagation, followed by Common Subexpression Elimination, followed by Dead Code Elimination.

   Solution:
   ```
   x := a + b
   z := x
   y := x
   ```

4. Is there any other ordering which would produce more optimal code? If yes, give the ordering and the resultant code.

   Solution: Copy propagation → Dead code elimination → Common subexpression elimination
   ```
   z := a + b
   y := z
   ```

3 Global Optimization

Global optimization generally focuses on flow analysis. We will first focus on constant propagation, which flows forwards.

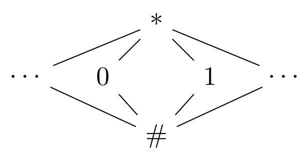
Exercise 3. What is the difference between constant folding, copy propagation, and constant propagation?

Solution: Constant folding and copy propagation are local analyses. Both of them apply solely within a single basic block. Constant folding is the pre-computation of certain constant expressions locally, e.g. replacing $2+2$ with 4. Copy propagation is the replacement of w with x after the expression $w := x$ in a basic block. Constant propagation is propagation of constant values across the entire program. It requires some form of reasoning about the consequences of branching logic in the program.

For constant propagation, each x can be associated with one of 3 values at each program point:

- #, meaning "this statement is not reachable"
- c, meaning "x has constant value c
- *, meaning "don't know if x is constant"

Recall these values are arranged in a lattice where $\# < c < *$, and all c's are not comparable, i.e.



For each statement, we define a transfer function. We define this by specifying the relation between $C_{in}(x,s)$ (the dataflow value of x before the statement s), and $C_{out}(x,s)$ (the dataflow value of x after the statement s).

The rules for constant propagation are as follows:

1. $C_{in}(x,s) = lub(C_{out}(x,p)|p$ is a predecessor of s) (What does it mean to be a predecessor of s?

   Solution: p is a predecessor of s if s can be executed directly after p)

2. $C_{out}(x, y := e) = C_{in}(x, y := e)$ if $y \neq x$

3. $C_{out}(x, x := e) = eval(e, C_{in})$

Perform a constant propogation analysis on the below CFG. The rules are reproduced here for your reference.

1. $C_{in}(x, s) = lub(C_{out}(x, p) | p$ is a predecessor of $s)$
2. $C_{out}(x, y := e) = C_{in}(x, y := e)$ if $y \neq x$
3. $C_{out}(x, x := e) = eval(e, C_{in})$