CS3040

Compiler Design.

Notes

112101033
Neeraj Krishna.N.

## Code Generation:

→ generate code for a single basic block

→ Basic Block, a block of code in which all statements are executed ~~during~~ ~~ev~~ in every run of the program.

→ Registers - used to store values, addresses, and intermediate expression values.

→ No. of registers in every architecture is limited

→ thus allocation of registers is ~~an~~ ~~an~~ ~~important~~ one of the most important part of code generation.

## Register Descriptor & Address descriptor

register descriptor - a datastructure used to keep track of current value in the registers.

address - descriptor - a data structure which keeps track of the locations of the current value of a particular variable.

Consider 3-address operations,

Example: $a = b + c$.

i) first load the values ~~to~~ of b and c ~~using~~ to appropriate registers say $R_b$ and $R_c$ ~~using~~

ii) Decide the register to store the value of a, say $R_a$.

iii) addu $R_a, R_b, R_c$.

iv) ~~load~~ store value of $R_a$ into appropriate address.

~~Manageme~~ Updation of Register Descriptor and address descriptor

1. for the instruction `load $R_a$, a`.
   i) change register descriptor for register $R_a$ to 'a'
   ii) ~~change~~ update address descriptor for 'a' by adding register R as an additional location.

2. for instruction `store $R_a$, a`
   change address descriptor for a to include its memory location.

3. Suppose instruction is `add $R_a, R_b, R_c$`
   i) Change register descriptor for a ~~so~~ so that it only holds 'a'. Change address ~~des~~ descriptor of 'a' so that it's only location is $R_x$

ii) ~~remove~~ ~~&~~ for all other variables, remove 'Ra' from its address - descriptor, if it has.

Similar is the case for other instructions, like sub, mul, etc.

## Live Variable Analysis

A variable is said to be live at a particular point if its current value is used in a future point

This is a method used for register allocation.

We use CFG (Control Flow ~~of~~ Graph) of the intermediate code

Use: an occurrence of the variable to the right of an assignment statement, or occurrence of variable in any other expression which denotes the use of that variable.

Def: An assignment to a variable.

Live on edge: A variable is live on edge if there is a directed path from edge to a node that has a use of that variable with 'no def' of that variable ~~along the~~ in the path.

Live-in : → if variable is live on any ~~in-edge~~ in-edges

Live-out → if variable is live on any out-edges.

Computing Live-in & Live-out for basic blocks

Backward Analysis

$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

$s \in succ[n]$ ↓ successors of 'n'

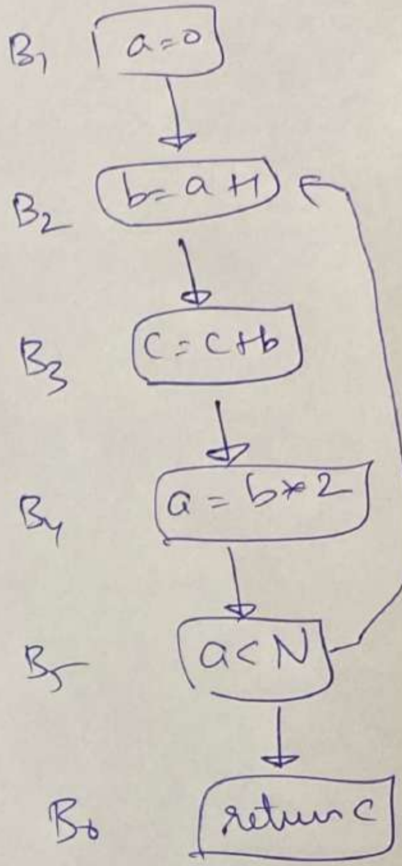
Algorithm for Live Variable Analysis

```
for each 'n' {
    in[n] ← {}
    out[n] ← {}
}
repeat {
    for each n {
        in'[n] ← in[n]; out'[n] ← out[n]
        in[n] ← use[n] ∪ (out[n] - def[n])
        out[n] ← ∪ in[s]
              s∈succ[n]
    }
} until (in'[n] = in[n] and out'[n] = out[n] for all n)
```

example

a = 0
b = a+1
c = c+b
a = b * 2
if (a < N) goto L1
return c

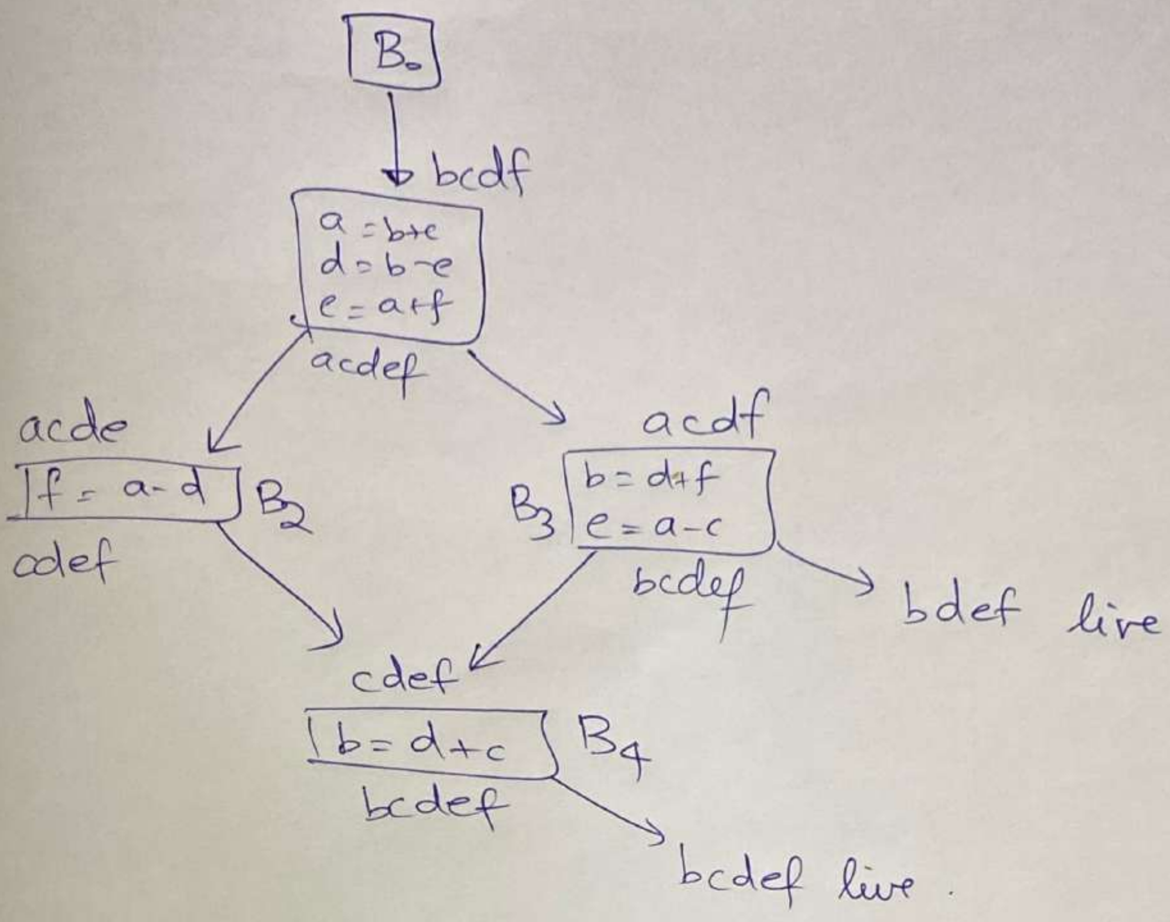
$\xrightarrow{\text{CFG}}$




| | | | Iteration ① | | Iteration ② | |
|---|---|---|---|---|---|---|
| | Use | Def | in | out | in | out |
| B6 | c | | c | | c | |
| B5 | a | | ac | c | ac | ca |
| B4 | b | a | bc | ac | bc | ac |
| B3 | (c,b) | c | bc | bc | bc | bc |
| B2 | a | b | ac | bc | ac | bc |
| B1 | | a | c | ac | c | ac |

Iteration ③

| | in | out |
|---|---|---|
| B6 | c | |
| B5 | ac | ac |
| B4 | bc | ac |
| B3 | bc | bc |
| B2 | ac | bc |
| B1 | c | ac |




| | | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|
| $B_1$ | use | 0 | 2 | 1 | 1 | 0 | 1 |
| | live | 1 | 0 | 1 | 1 | 1 | 1 |
| $B_2$ | use | 1 | 0 | 0 | 1 | 0 | 0 |
| | live | 0 | 0 | 1 | 1 | 1 | 1 |
| $B_3$ | use | 1 | 0 | 1 | 1 | 0 | 1 |
| | live | 0 | 1 | 1 | 1 | 1 | 1 |
| $B_4$ | use | 0 | 0 | 1 | 1 | 0 | 0 |
| | live | 0 | 1 | 1 | 1 | 1 | 1 |
| $\sum_B use_B + 2 * live_B$ | | 4 | 6 | 11 | 12 | 8 | 10 |

The above algorithm is ~~too~~ basically a fixpoint kind of algorithm.

## Data Flow Analysis (DFA)

→ used in compiler optimization.

→ low level intermediate code is modified to new optimized code in one or more stages.

→ semantics of program must not change ~~in the process of~~

→ Examples of

- → Machine independent Optimization -
  - → Global Common SubExpression Elimination
  - → Constant Folding, Dead code elimination
  - → Code Motion, Induction Variable Elimination.
- → Machine dependent Optimization
  - → Register Allocation, Instruction Scheduling.

→ Refers to techniques that derive information about the flow of data along the execution paths of the program.

→ Ex: common subexpression
→ find if 2 identical expressions evaluate to same value along all possible execution paths.

- Dead Code Elimination
Checking if a definition of variable is not used later.

→ Important things to consider.
→ should consider <u>all possible execution sequences</u>.
→ Some places → interprocedural paths will be required

→ Local Analysis
Intraprocedural Analysis
Interprocedural Analysis.

## <u>Forward DFA</u>

$$out[B] = F_B(In[B])$$

$$in[B] = \overset{\text{intersection}}{\bigcap} out[P] \quad \forall P \in \text{predecessors}(B)$$

## <u>Backward DFA</u>

~~out [~~

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

$$out[B] = \bigcap in[S] \quad \forall S \in \text{successors}(B).$$

# Available Expression

## Compilation:

~~init [Entry] = $\phi$~~

OUT [Entry] = $\phi$

universal set

For each (Basic block B other than Entry) OUT [B] = $U$

while (changes to any OUT occur) {

for each (basic block B other than Entry) {

$$IN[B] = \cap (OUT[P]) \ \forall P \in pred[B]$$

$$OUT[B] = Gen[B] \cup (IN[B] - KILL(B))$$

}

}

## Other Examples of DFA.

| | Reaching Definition | Live Variable. |
|---|---|---|
| Domain | Set of all definitions | Set of variables. |
| Direction | Forward Analysis | Backward Analysis. |
| ~~Transit~~ Trans function | $Gen_B \cup (X - Kill_B)$ | $use_B \cup (X - def_B)$ |
| Boundary | OUT [Entry] = $\phi$ | IN [EXIT] = $\phi$ |
| Meet | union | union. |
| Equation | ~~$OUT_B$~~ $OUT[B] = F_B(IN[B])$<br>$IN[B] = \bigwedge_{P \in Pred[B]} OUT[P]$ | $IN[B] = F_B(OUT[B])$<br>$OUT[B] = \bigwedge_{S \in succ(B)} IN(S)$ |
| Initialization | OUT [B] = $\phi$ | IN [B] = $\phi$ |