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~~ ~~at~~ 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~~ ~~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 of b and c to appropriate registers say R_b and R_c

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

iii) add R_a, R_b, R_c.

iv) store value of R_a into appropriate address.

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) 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 that it only holds 'a'. Change address descriptor of 'a' so that it's only location is R_x

ii) ~~remove~~ & for all other variables, remove 'R'a 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 registes 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-edges

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

Computing Live-in & Live-out for basic Blocks Backward Analysis

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

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

succ[n] ↓ Successors of 'n'.

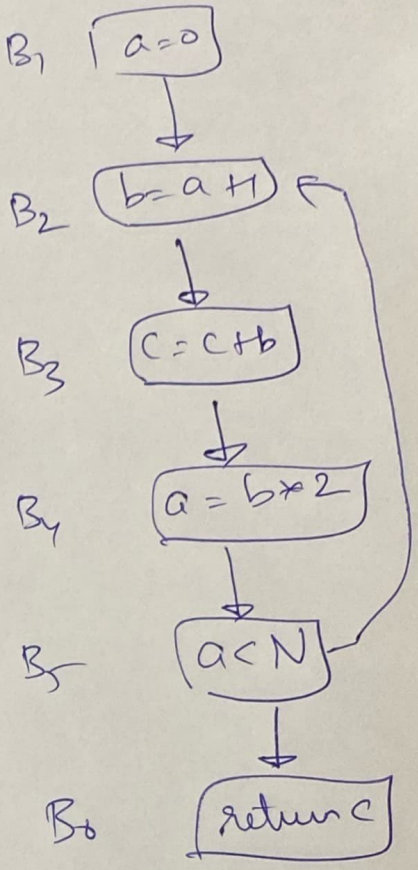
Algorithm for Live Variable Analysis

```
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
```

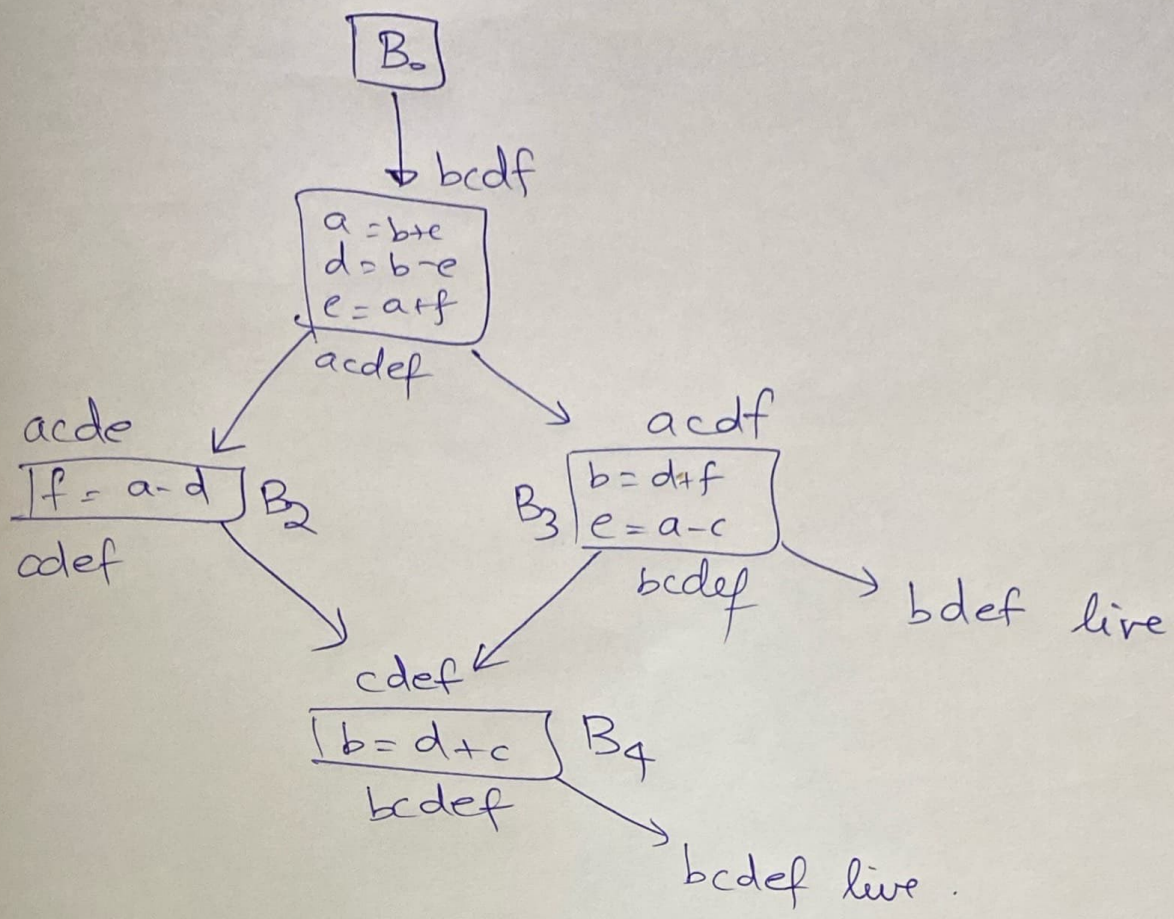
CFG →



	Use	Def	Iteration ① in	Iteration ① out	Iteration ② in	Iteration ② out
B_6	c		c		c	
B_5	a		ac	c	ac	ca
B_4	b	a	bc	ac	bc	ac
B_3	(c,b)	c	bc	bc	bc	bc
B_2	a	b	ac	bc	ac	bc
B_1		a	c	ac	c	ac

Iteration ③

	in	out
B_6	c	
B_5	ac	ac
B_4	bc	ac
B_3	bc	bc
B_2	ac	bc
B_1	c	ac



		a	b	c	d	e	f
	use	0	2	1	1	0	1
B_1	live	1	0	1	1	1	1
	use	1	0	0	1	0	0
B_2	live	0	0	1	1	1	1
	use	1	0	1	1	0	1
B_3	live	0	1	1	1	1	1
	use	0	0	1	1	0	0
B_4	live	0	1	1	1	1	1
$\sum_B (use_B + 2 \times live_B)$		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 all possible execution sequences.
→ Some places → interprocedural paths will be required

→ Local Analysis
Intraprocedural Analysis
Interprocedural Analysis.

Forward DFA

$$out[B] = F_B(In[B])$$
$$in[B] = \bigcap_{\text{intersection}} out[P] \quad \forall P \in predecessors(B)$$

Backward DFA

~~out f~~
$$in[B] = F_B(out[B])$$
$$out[B] = \bigcap in[S] \quad \forall S \in successors(B).$$

Available Expression

Compilation:

~~out [Entry] = ∅~~

OUT [Entry] = ∅

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) {

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

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

}

}

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	$OUT[Entry] = \emptyset$	$IN[EXIT] = \emptyset$
Meet	union	union.
Equation	~~OUT~~ $_B$ $OUT[B] = F_B(IN[B])$ $IN[B] = \bigwedge_{P \in Pred[B]} OUT[P]$	$IN[B] = F_B(OUT[B])$ $OUT[B] = \bigwedge_{S \in succ(B)} IN(S)$
Initialization	$OUT[B] = \emptyset$	$IN[B] = \emptyset$