Compilers

Second Test

June 12, 2007

Duration: 2 hours

Please label all pages you turn in with your name and student number.

Grade:

Problem 1 [30 points]:
Problem 2 [20 points]:
Problem 3 [50 points]:

Total:

Problem 1: Run-Time Environment and Storage Allocation [30 points]

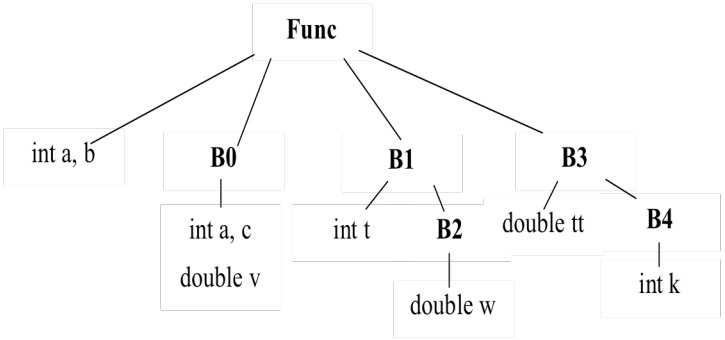
a) [5 points] Describe the basic structure of an Activation Record (AR).
b) [5 points] Where are AR stored at run-time. Why?
c) [5 points] Why do you need an AR for? What features of a language requires explicit AR and why?
d) [5 points] What is the purpose of the access link? What language mechanism requires its use? Why?
e) [10 points] For the code structure shown below determine the location in the AR of each local variable in each block. Assume that the first local variable begins at the zero offset and that integer values take 4 bytes and double values use 8 bytes.

```
void func(){
 int a, b;
 B0:    {
                int a, c;
                double v;
        }
 B1:    {
                int t;
                B2:     {
                        double v;
                }

        }
 B3:    {
                double tt;
                BB4: {
                  int k;
                }
        }
}
```

Answers:

a) The AR has a field for each function/procedure parameter, return address, return value, the ARP link and access link as well as local variables.
b) The AR are allocated and maintained on the stack. This is because their life span is connected to the activation or life-times of the corresponding procedures and function, which exist or form a stack over time following the caller-callee relationship.
c) To capture the execution context of a procedure/function. Recursion is the basic language meachnism that requires an AR for each invocation, as there can be multiply active instances of the local variables.
d) The access link is used to access non-local variables in languages supporting lexical scoping.
e) The key issue in allocating the sapce in the AR is to determine which scope blocks are simultaneously active. To capture this we can organize the nesting of these scopers in a tree as shown in the figure below and them it is obvious what space can be reuse when each of the scope are no longer active.



int a			4 bytes
int b			4 bytes
int a	int t	double tt(H)	4 bytes
int c	double w(H)	double tt(L)	4 bytes
double v(H)	double w(L)	int k	4 bytes
double v(L)			4 bytes

Problem 2: Register Allocation [20 points]

```
 1:       i = 0;
 2:       a = 1;
 3:       b = i * 4;
 4:       c = a + b;
 5: L1:  if (i > n) goto L2
 6:       c = a + 1;
 7:       i = i + 1;
 8:       b = i * 4;
 9:       if (c <= p) goto L3
10:       e = 1;
11:       c = e - b;
12:       a = e;
13:       goto L4
14: L3:  d = 2;
15:       c = d + b;
16:       a = d;
17: L4:  goto L1
18: L2:  return;
```

For the code shown above, determine the following:

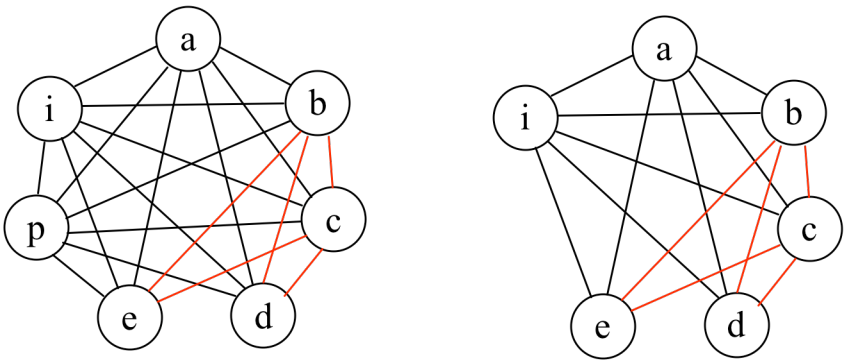
a) [5 points] The live ranges for the variables i, a, b, c, d, e, p.
b) [5 points] The interference graph using the simplest definition of interference where the webs include the live range in terms of the line numbers where an access to a variable (either a read or a write operation) occurs.
c) [5 points] Determine the minimum number of registers needed (without spilling, of course) using the graph coloring algorithm described in class.
d) [5 points] Assuming that you were short of one register, which register(s) would you spill and at which points in the program would you insert the spill code? Explain.

Answers:

a) The basic observation is that one variable is live at a specific program point p if its value is still going to possibly be used in the future. The best way to figure this out is to begin by constructing the control-flow graph (CFG) for this code. For instance variable `i` is live at program point 11 as there is a possibility that the value defined for this variable at point 7 is going to be used along the path that traverses the basic block that contains the instructions 10 through 13 which then jumps to line 5 which reads this `i` variable to evaluate the test `(i > n)`. Tracing the definitions and uses for each variable we can arrive at the following live ranges where the number indication the source program line numbers:

i: {1 – 17}
p: {1-17}
a: {1-17}
b: {3,4}, {8,9,10,11}, {8,9,14,15}
c: {4}, {6},{11},{16}
d: {14,15,16}
e: {10,11,12}

b) The variables {i, a, p} interfere will all the other variables (using bloack edges). The complete interference graph is as shown below on the left.



c) Using the algorithm we have studied in class for graph coloring for N = 6 given that that is the highest degree in the graph it is highly likely that for N = 6 we will be able to find such coloring of the interference graph. In this case only nodes `d` and `e` go into the stack as they have degree 5 less than 6. If that is the case we now remove one node from the graph, say node `p` that interferes with all the other remaining nodes in the graph. We are left with the graph above in the middle. Then we must push all the remaining nodes on the stack again, this time all of them have degree less than 6. We then pop the nodes one by one and assign colors. One such color is to assign the same color, say red to {e,d} as they do not interfere and any other 4 colors to the remaining nodes. This results in 5 colors leaving the node corresponding to p uncolored.

d) If you had one less register say 4 registers, you could leave the node with the shortest live range, say node `d` or `e` out of registers.

Problem 3: Control-Flow Graph Analysis and Data-Flow Analysis [60 points]

```
01        a = 1;
02        b = 2;
03 L0:    c = a + b;
04        d = c - a;
05        if (c < d) goto L2
06 L1:    d = b + d;
07        if( (d < 1) goto L3
08 L2:    b = a + b;
09        e = c – a;
10        if(e == 0) goto L0
11        a = b + d;
12        b = a – d;
13        goto L4;
14 L3:    d = a + b;
15        e = e + 1;
16        goto L3:
17L4:     exit
```

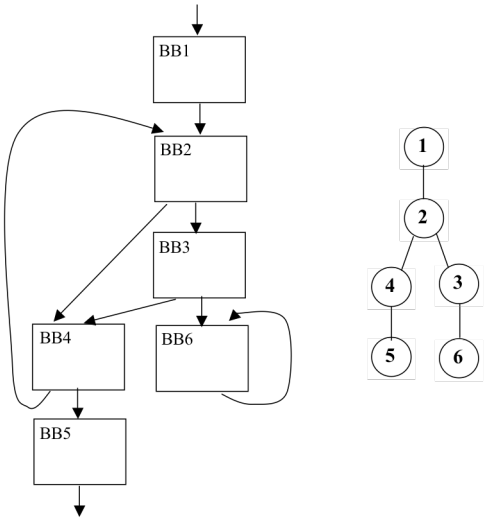
For the code shown above, determine the following:

a) [10 points] The basic blocks of instructions.
b) [10 points] The control-flow graph (CFG)
c) [10 points] For each variable, its corresponding *du*-chain.
d) [20 points] The live variables at the end of each basic block. You do not need to determine the live variables before and after each basic block and justify your answer for the value presented for the basic block containing instructions at line 6 and 7.
e) [10 points] Is the live variable analysis a forward or backward data-flow analysis problem? Why and what does guarantee its termination when formulated as a data-flow analysis itereative problem?

Answers:

a) and b) We indicate the instruction in each basic block and the CFG and dominator tree on the RHS.

BB1: {01, 02}
BB2: {03, 04, 05}
BB3: {06, 07}
BB4: {08, 09, 10}
BB5: {11, 12, 13}
BB6: {14, 15, 16}



c) The def-use chains can be obtained by inspection of which definitions are not killed along all path from its definition point to the use points. For example for variable “a” its defintinion at the instruction 1, denoted by a1 reaches the use points at the instructions 3, 4, 8, 9 and 15, hence the notation {d1, u3, u4, u8, u9, 15}. Definition a1 does not reach use u12 because there is another defintion at 11 that masks it, hence another du chain for “a” denoted as {d11, u12}. The complete list is shown below.

a: {d1, u3, u4, u8, u9, 15}
a: {d11, u12}
b: {d2, u3, u4, u6, u14, u8}
b: {d8, u11, u8, u14}
b: {d12}
c: {d3, u4, u5, u9}
d: {d4, u5, u6}
d: {d6, u7}
d: {d14, u6}
e: {d9, u10, u15}
e: {d14, u15}

d) At the end of each basic block we can inspect the code and ask if a given variable is still used after the control flow leaves that basic block. The results of this inspection is depicted below:

BB1: {a, b}
BB2: {a, b, c, d, e}
BB3: {a, b, c, d, e}
BB4: {a, b, d, e}
BB5: { }
BB6: {a, b, c, e}

For BB6 the live varaible solution at the exit of this basic block has {a, b, c, e} as for variable “d” there is no path out of this basic block where the current value of the variable is used. For variable “d” the value is immediately redefined in BB3 without any possibility of being used.

e) The live varaible analysis is a backward data-flow problem as we propagate the information about a future use of a variable backward to specific points of the program. If there is a definition at a specific point backward the solution kills all other uses and resets the information associated with that variable. As with many other iterative formulations of data-flow analysis problems termination is guaranteed by the fact that the lattice, in this case the set of variables, has finite cardinality or length. The flow-function, in this case set-union is monotonic.