Compilers

Second Test

Spring, 2008

Solution

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

Grade:

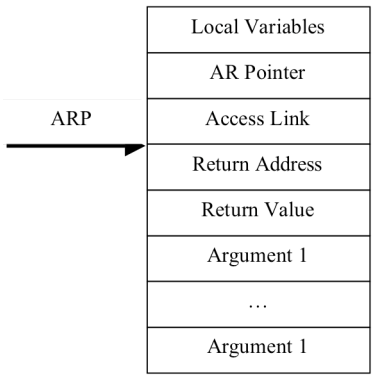
Problem 1 [20 points]:
Problem 2 [40 points]:
Problem 3 [40 points]:

Total:

Problem 1: Run-Time Environment and Organization [20 points]

Under the assumption that the AR are allocated on the stack with the individual layout as shown below, and given the PASCAL code on the right-hand-side answers the following questions:

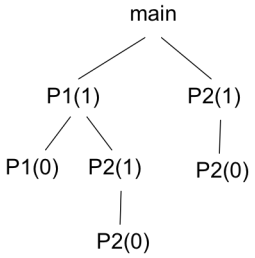
(a) [05 points] Draw the call tree starting with the invocation of the main program.
(b) [10 points] Draw the set of ARs on the stack when the program reaches line 16 in procedure P2. Include all relevant entries in the ARs and use line numbers for the return addresses. Draw direct arcs for the access links and clearly label the values of local variables and parameters in each AR.
(c) [05 points] For this particular example would there be any advantage of using the Display Mechanism?

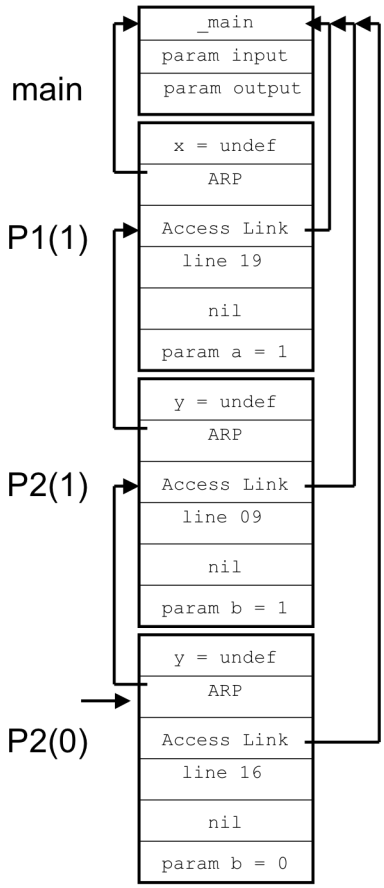



```
01: program main(input,  output);
02:    procedure P1(a: integer));
03:      var x: integer;
04:      begin
05:         if (a <> 0) then
06:           begin
07:             P1(a-1);
08:             P2(a);
09:           end
10:      end;
11:      procedure P2(b: integer);
12:        var y: integer;
13:        begin
14:           if(b <> 0) then
15:              P2(b-1);
16:        end;
17:  begin (* main *)
18:    P1(1);
19:    P2(1);
20:  end.
```

Answers:

(a) and (b) See the call tree on the left and the stack organization on the right below where the values of the access link and activation records pointers are explicit.





(c) None at all. Given that there are no nested procedures the only non-local variables that a given procedure needs to access are the global variables (like in C). So you only need to have the ARP (or Frame-Pointer – FP) and an second GP or Global Pointer registers. Most architectures do support these two registers in hardware.

Problem 2: Basic Transformations and Register Allocation [40 points]

Consider the following three-address format representation of a computation using scalars and arrays A and B and a procedure `F(int a, int b)`. Assume for the purposes of this exercise that no additional registers are needed to access (either writing or reading) the value of an array. As such the access to A[t1] requires no additional registers in addition to the one carrying the value of t1.

```
01:   t1 = i
02:   t2 = A[t1]
03:   t3 = i + 1
04:   t4 = A[t3]
05:   t5 = t2 + t4
06:   t6 = t5 / 2
07:   t7 = i
08:   B[t7] = t6
09:   putparam i
10:   putparam 1
11:   call F,2
12:   t8 = i
13:   t9 = t8 + 2
```

a) [10 points] Apply copy propagation and dead code elimination to the basic block immediately preceeding the function invocation.

b) [20 points] Use the bottom-up register allocation algorithm for 3 registers under the assumption that variable `i` is live outside this basic block and after the call to F. In this section you can simply ignore the instructions on lines 09 through 11. In this part of the code load the value of the scalar variables using the offset of the scalar from the Frame-Pointer (FP) register. So you need to make some changes to the intermediated code.

c) [10 points] Passing the arguments to the procedure F could be done via registers. This would come at the cost of additional registers unless the values required by a given procedure would already reside in registers. For this particular case explian which putparam instruction could you eliminate by using arguments for the procedure that are already in registers.

Answers:

a) [10 points] The statements on lines 01, 07 and 12 are dead once we propagate the corresponding assignments to the instructions on lines 02, 08 and 13.

```
01:   t1 = i
02:   t2 = A[i]
03:   t3 = i + 1
04:   t4 = A[t3]
05:   t5 = t2 + t4
06:   t6 = t5 / 2
07:   t7 = i
08:   B[i] = t6
09:   putparam i
10:   putparam 1
11:   call F,2
12:   t8 = i
13:   t9 = t8 + 2
```

b) [20 points] The code below illustrates the resulting code under the assumption that the variable `i` is initially stored at an negative offset 16 of the Frame-Pointer (FP) register.

```
01:   r0 = FP - 16
02:   r0 = *r0          // loads the value of variable i into r0
02:   r1 = A[r0]        // r0 ← i,t1,   r1 ← t2,   r2 ← empty
03:   r2 = r0 + 1       // r0 ← i,t1, r1 ← t2,   r2 ← t3
04:   r0 = A[r2]        // r0 ← t4,   r1 ← t2,   r2 ← t3
05:   r2 = r1 + r0      // r0 ← t4,   r1 ← t2,   r2 ← t5
06:   r0 = r2 / 2       // r0 ← t6,   r1 ← t2,   r2 ← t5
07:   r1 = FP - 16      // r0 ← t6,   r1 ← i,t7, r2 ← t5
08:   r1 = *r1          // r0 ← t6,   r1 ← i,t7, r2 ← t5
08:   B[r1] = r0        // r0 ← t6,   r1 ← i,t7, r2 ← t5
09:   putparam r1
10:   putparam 1
11:   call F,2
12:   r0 = r1           // r0 ← t8,   r1 ← i,t7,  r2 ← t5
13:   r2 = r1 + 2       // r0 ← t8,   r1 ← i,t7,  r2 ← t9
```

c) [10 points] For this particular case the argument value `i` is already in register `r1`, so we could eliminate the first `putparam` instruction provided now that when doing the register allocation for F we would start with the assumption that the first parameter is already in `r1`.

Problem 3: Control-Flow Graph Analysis and Data-Flow Analysis [40 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 L1
17L4:   return
```

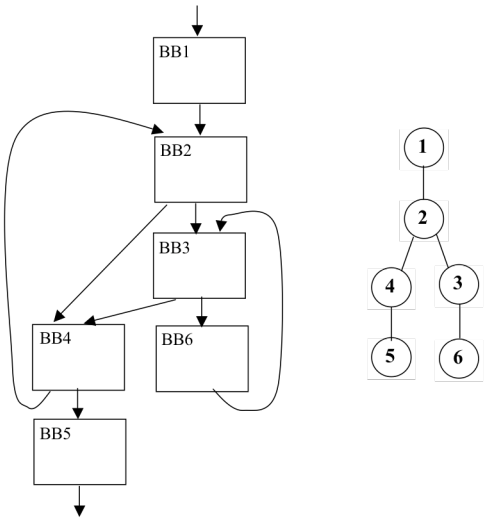
For the code shown above, determine the following:

a) [10 points] The basic blocks of instructions and the control-flow graph (CFG).
b) [10 points] For each variable, its corresponding *du*-chain.
c) [10 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.
d) [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) We indicate the instructions 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" definition 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, u15}
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}
e: {d14, u6}

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, d, e}

For BB6 the live variable solution at the exit of this basic block has {a, b, c, d, e} as there exists a path after BB6 where all the variables are being used. For example variable "e" is used in the basic block following the L1 label and taking the jump to L3 to BB6 again where is is used before being redefined.

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.