Compilers

Second Test

Spring 2009

June 9, 2009

Solution

Problem 1: Code Generation and Storage Organization [20 points]
Assume that your language directly supports the primitive `vector` data type declared as "`vector a [n]:type`" where n defines its run-time length and type is the declared type of the elements of the vector, say an integer or a double for example.

a) [10 points] Given that the size of the vector is only known at run-time, outline the placement of two vector variables in the snippet of code shown below for a procedure P.

```
procedure P(int a, int n)
  vector x[n], y[n] : integer;
  begin
    …
  end
```

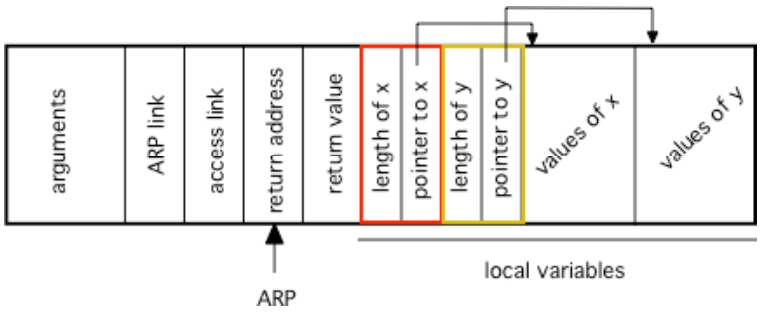
b) [10 points] Present a SDT scheme that performs the initialization of the elements of a vector to be executed as part of the prologue code of the procedure where the vector is declared with the generic production:

decl → `vector` id '[' Exp ']' ':' type ';'

Assume that you can access the length of the vector at the position indicated by the symbolic value of `offsetVector(v)` with respect to the ARP and that you can access the address of the first element of the vector using a specific code generation macro named `loadAddress(v)` where v is the symbolic name of the vector. Also, do not forget that you need to loop through all the elements of a vector and set them to a specific null value. Assume also that this null value can be found using the auxiliary function `ZeroType(t)` for a given type value t and that the size of the individual elements of a vector can be found using the auxiliary function `baseSize(t)`. For the purposes of determining the addresses where the code is located assume there is a global variable named "nextAddress" that is properly set up when you start your code generation.

Solution:

a) The layout is depicted in the figure below where each vector has a fixed part, the length and the pointer to the first element of the vector itself. The variable portion is allocated at "the end" of the AR whose size depends on the size of the vector base elements.



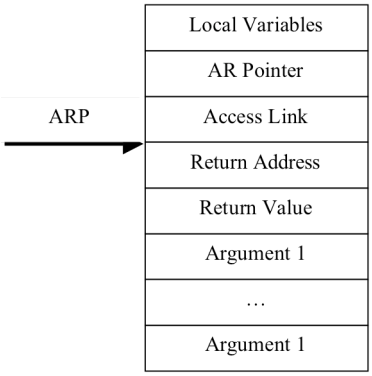
b) The SDT scheme needs to load initialize all the elements of the vector with the symbolic ZERO value. In the case of an integer base type the ZERO value is the integer value 0 whereas in the case of a double it would be 0.0. The SDT below accomplishes the generation of code using a simple loop code construct where we have assumed the length or size of the vector is positive. A possible SDT is as shown below:

```
decl → vector id '[' Exp ']' ':' type ';'        {
                    tn = newtemp();
                    emit("tn = ARP + offsetVector(id.name)"); // gets the address of the length
                    emit("tn = *tn");                         // gets the length of vector
                    tv = newtemp();                           // to hold the value to be initialized
                    tv = ZeroType(type.value);                // typically a zero binary pattern
                    tp = newtemp();                           // a pointer to the addresses in memory
                    emit("tp = loadAddress(id.name)");        // loads the first address of vector
                    emit("L: if(tn = 0) goto nextaddress+9"); // go to exit
                    emit(" *tp = tv; ");                      // write zero value to memory
                    emit("  tp = tp + baseSize(type.value)"; // advances the pointer in memory
                    emit("  tn = tn – 1 ");                   // decrement counter
                    emit("  goto L");                         // go around again
}
```

Problem 2: Run-Time Environment [30 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:

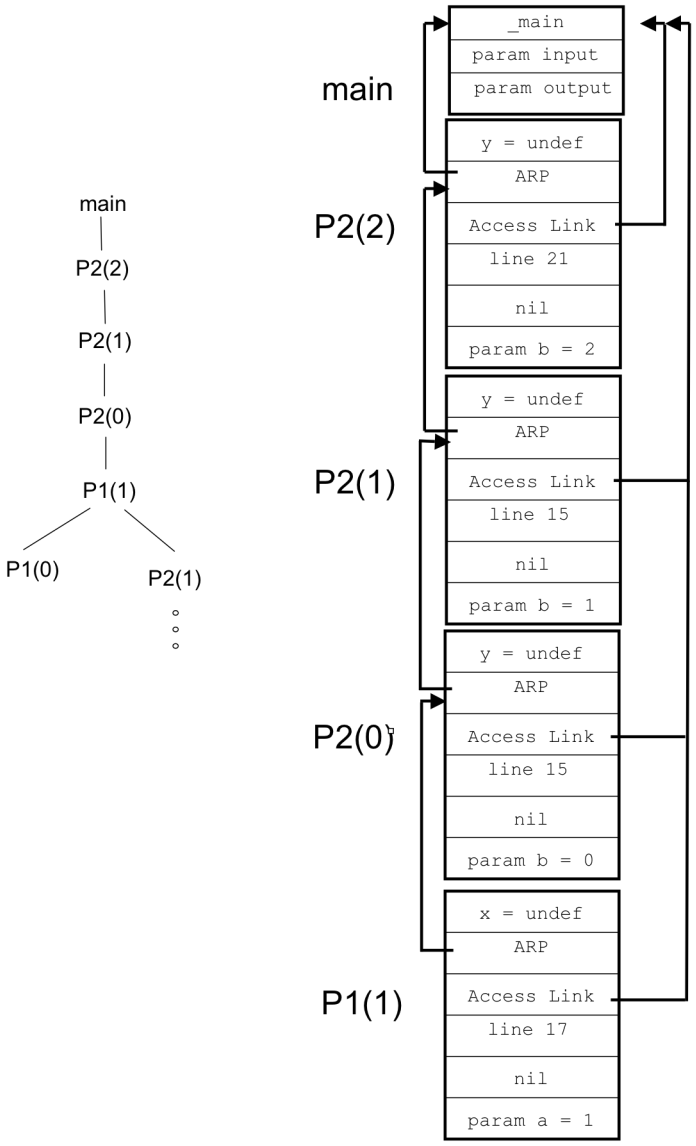
(a) [05 points] Draw the call tree starting with the invocation of the main program.
(b) [10 points] Draw the set of ARs on the stack when the program reaches line 07 in procedure P1. Include all relevant entries in the ARs and use line numbers for the return addresses. Draw direct arcs for the access 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? Explain.
(d) [10 points] Indicate in detail the AR organization for procedure P2 assuming each integer uses 4 bytes and local variables are located with positive offset relatively to the ARP pointer.




```
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, z[10],w[b]: integer;
13:        begin
14:           if(b <> 0) then
15:              P2(b-1);
16:           else
17:              P1(1)
18:        end;
19:  begin (* main *)
20:    P1(0);
21:    P2(2);
22:  end.
```

Solution:

(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. Notice that by accident here the code can enter an infinite loop and hence the call tree in unbounded.



(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 a second GP or Global Pointer registers. Most processor architectures do support these two registers in hardware.

Problem 3: Register Allocation [25 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)`.

```
01:   t1 = i
02:   t2 = A[t1]
03:   t3 = j
04:   t4 = B[t3]
05:   putparam t2
06:   putparam t4
07:   call F,2
08:   t5 = i
09:   t6 = t5 + 1
```

a) [05 points] Assume for the purposes of this exercise that the arrays A and B are global arrays located at a specific offset of a Global Register (GP). To access the arrays in instructions in lines 02 and 04 you need to use GP. As such modify the code to make the access to the arrays explicit taking into account the fact that each integer uses 4 bytes of memory. Also load the value of the scalar variables using the offset of the scalar from the Frame-Pointer (FP)

b) [15 points] Show the assignment of variables to registers before the execution of each instruction, and using the bottom-up register allocation algorithm for 3 registers, rewrite the code in this procedure. Assume that variable `i` is live outside this basic block and after the call to F.

c) [05 points] Do you think a global register allocation algorithm would do better than the bottom-up allocator for this particular segment of code? Why or why not?

Solution:

a) [05 points] The modified code has to take into account the fact that you need to access the local variables via the frame-Pointer (FP) and the arrays via the Global Pointer (GP). A revised version of the code is shown below. We use lines numbers and comments to help you understand the mapping.

```
01:   t1 = FP + offset_i      // getting the address of i
      t1 = *t1                // loading the value of i in t1
02:   t1 = t1 * 4             // computing the offset of A[i]
      t2 = GP + offset_A      // getting the base address of A
      t2 = t2 + t1            // computing the offset of address of A[i]
      t2 = *t2                // loading of A[i]
03:   t3 = FP + offset_j      // getting the address of j
      t3 = *t3                // loading the value of j in t3
      t3 = t3 * 4             // computing the offset of B[i]
      t4 = GP + offset_B      // getting the base address of A
04:   t4 = t3 + t4            // computing the offset of address of B[j]
      t4 = *t4                // loading of B[j]
05:   putparam t2
06:   putparam t4
07:   call F,2
08:   t5 = FP + offset_i      // getting the address of i
      t5 = *t5                // loading the value of i in t5
09:   t6 = t5 + 1             // calculation in line 09
```

b) [15 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 + offset_i      // r0 = t1;    r1 = empty; r3 = empty
      r0 = *r0                // r0 = t1;    r1 = empty; r3 = empty
02:   r0 = r0 * 4             // r0 = t1;    r1 = empty; r3 = empty
      r1 = GP + offset_A      // r0 = t1;    r1 = t2;    r3 = empty
      r1 = r1 + r0            // r0 = empty; r1 = t2;    r3 = empty
      r1 = *r1                // r0 = empty; r1 = t2;    r3 = empty
03:   r0 = FP + offset_j      // r0 = t3;    r1 = t2;    r3 = empty
      r0 = *r0                // r0 = t3;    r1 = t2;    r3 = empty
      r0 = r0 * 4             // r0 = t3;    r1 = t2;    r3 = empty
      r3 = GP + offset_B      // r0 = t3;    r1 = t2;    r3 = t4
04:   r3 = r0 + r3            // r0 = t3;    r1 = t2;    r3 = t4
      r3 = *r3                // r0 = empty; r1 = t2;    r3 = t4
05:   putparam r1             // r0 = empty; r1 = t2;    r3 = t4
06:   putparam r3             // r0 = empty; r1 = empty; r3 = t4
07:   call F,2                // r0 = empty; r1 = empty; r3 = empty
08:   r0 = FP + offset_i      // r0 = t5;    r1 = empty; r3 = empty
      r0 = *r0                // r0 = t5;    r1 = empty; r3 = empty
09:   r1 = r0 + 1             // r0 = t5;    r1 = t6;   r3 = empty
```

Notice that in the last instruction we kept the value of t5 in r0 as the corresponding program variable, I, is alive and hence its value is used next in addition to the use on line 09.

(c) [05 points] For this particular code the number of register is 3 which is strictly necessary given the calculations of the array addresses and the fact that we need to hold the values in register for parameters while we compute the value of the second argument. If we were to compute the inference web we would find out that there is a clique of dimension 3 and hence we would have to have at least 3 registers. The global graph-based algorithm would find we need at least 3 registers without spilling which is exactly what we got with this bottom-up allocator.

Problem 4: Control-Flow Graph Analysis and Live-Variable Analysis [25 points]

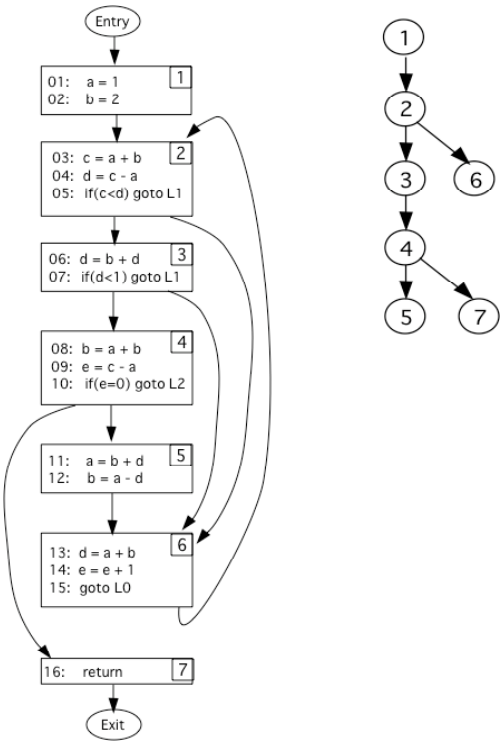
For the code shown above, determine the following:

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

a) [10 points] The basic blocks of instructions and the control-flow graph (CFG) and the CFG dominator tree.
b) [05 points] Identify the natural loops and determine if they are nested or not (explain why or why not).
c) [10 points] The live variables at the end of each basic block. A variable is said to be live at the end of a given basic block or instruction if its value is used beyond that basic block or instructions. 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 14 and 15.

***Solution**:*
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}
BB6: {13, 14, 15}
BB7: {16}



b) The only back edge in this CFG is the edge (6,2) as the node 2 dominates the node 6. Tracing back the nodes of the CFG backwards we get the natural loop composed by the basic blocks {2,3,4,5,6} as the only loop in this code.

c) 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}
BB4: {a, b, c, e}
BB5: {a, b, e}
BB6: {a, b, e}
BB7: { }

For BB6 the live variable solution at the exit of this basic block has {a, b, 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 L0 label and taking a path back to BB6 again where is used before being redefined. As to variables a and b they are used at the beginning of the basic block 2. Variables c and d, however, are not live at the end as they are immediately redefined at the entrance of basic block 2.