Homework I

CS 426 — Compiler Construction
Fall Semester 2019

Handed out October 31, 2019. Due by 5 PM, Thursday November 7, 2019
Turn in electronically to vadve@illinois.edu or slide under Professor Adve's door

---

1. *(30 points) Intermediate representations*:

   (a) (15 points) Construct a Control-Flow Graph (CFG) for code fragments 2 and 3. Use maximal-length basic blocks, i.e., each basic block should include as many statements as possible. You do not need to show the statement code within the basic blocks; just label each block with the line number of the first statement. If more than one basic blocks start on the same line L, you may label the first statement of each basic block and use the label in conjuction with L. You may use any other notation, as long as it is clear.

   (b) (15 points) Convert code fragments 1 and 2 into a Static Single Assignment (SSA) representation. Assume SSA form uses minimal (but not pruned) ϕ functions. You don't need to show the CFG, but remember the cardinal rule: always draw the CFG when converting non-trivial code to SSA form

```
(1)   // Code Fragment 1 (C)
(2)   X = 1;
(3)   Y = 0;
(4)   Z = 0;
(5)   do {
(6)       Z = Z + 1;
(7)       Y = Y + Z;
(8)       if (Z < X)
(9)           continue;
(10)      do {
(11)          Z = Z - 1;
(12)          if (Z > 0)
(13)              continue;
(14)          break;          // exits innermost do ··· while loop
(15)      }
(16)      while ( Z <= 65535);
(17)      if (X > 1024)
(18)          break;          // exits innermost do ··· while loop
(19)      X = X * 2;
(20)  }
(21)  while (Y <= 65535);
(22)  printf("%d\n", Y);
```

```
(1)   // Code Fragment 2 (C)
(2)   i = 0;
(3)   j = 1024;
(4)   k = 0;
(5)   for(i = 0; i < j; ++i){
(6)       switch(i%3) {
(7)           case 0:
(8)               k += i;
(9)               break;
(10)          case 1:
(11)              k += i*2;
(12)              break;
(13)          default:
(14)              printf("%d\n", i);
(15)              break;
(16)      }
(17)      continue;
(18)      i *= 4;
(19)  }
```

```
(1)   // Code Fragment 3 (Java)
(2)   int[] array = new int[3];
(3)   try{
(4)       for(int i=0;i<4;++i){
(5)           array[i] = i;
(6)       }
(7)       System.out.println(array);
(8)   }
(9)   catch(ArrayIndexOutOfBoundsException e){
(10)      System.out.println("Array out of bounds!");
(11)  }
(12)  finally{
(13)      System.out.println(array);
(14)  }
```

2. *(20 points) Intermediate Representations*:

   Requirements of SSA Form: Determine whether each of the following code fragments represents valid, minimal SSA form. (Draw the CFG if it will help, but we do not need to see it.) Then circle *Minimal*, *Invalid*, or *Not Minimal*, where "*Not Minimal*" means "*Valid but not Minimal*." Explain your answer precisely in a brief phrase or sentence in each case (e.g., "Assignment to X does not dominate use." or "Unnecessary Phi function for variable Y.", etc.).

   (a)
```
        x1 = 0;
        y1 = 1
    L1: x2 = phi(x1,x3);
        y2 = phi(y1,y3);
        y3 = y2 * 2;
        x3 = x2 + 1;
        if (y3 < 10)
           goto L1;
        exit();
```

   This code fragment is: Minimal, Invalid, Not-Minimal

   Explain:

(b)
```
    x1 = user_input();
    if(x1 % 2)
       x2 = x1 + 1;
    output(x1+x2);
    exit();
```

This code fragment is: Minimal, Invalid, Not-Minimal
Explain:

(c)
```
    x1 = 0;
L1: x2 = phi(x1,y1);
    y1 = phi(x1,x2);
    if (y1 < 10)
       goto L1;
    exit();
```

This code fragment is: Minimal, Invalid, Not-Minimal
Explain:

(d)
```
    x1 = 0;
    y1 = 0;
L1: x2 = phi(x1, x3);
    y3 = y2 + x2;
    y2 = y3;
    if (x2 < 10)
       goto L1;
    exit();
```

This code fragment is: Minimal, Invalid, Not-Minimal
Explain:

(e)
```
    A = user_input();
    if (A == 0)
       B = 5;
    else
       C = 8;
    D = phi(A, C);
    return D;
```

This code fragment is: Minimal, Invalid, Not-Minimal
Explain:

3. *(20 points) Runtime environments*:

   For this question, you must identify the *user-visible* memory objects created by the code below, along with their storage classes and creation times. The *user-visible* memory objects in a C program include functions, variables, and dynamically allocated memory blocks created by `alloca()` or `malloc()`. *Some user-visible objects that may not be obvious are noted in the comments below.* You should list every memory object created when compiling, linking, loading, or executing this program. **Omit any objects outside the code that is shown.**

   Give your answer in the form of a table with one row per object, and with the following 3 columns:

   (1) ***Objects*** : The name of the object. Make sure you distinguish a pointer variable from an object it may point to, e.g., `ptr` vs. `Objects2`.

   (2) ***Memory Region*** : The memory region in which the object is allocated. The choices:

   { *Code, Initialized Globals, Uninitialized Globals, Stack, Heap* }

   (3) ***Creation Time*** : The step in the compilation/execution process at which the object is created and initialized. Any object that must be included in the executable file is created at "link time". Distinguish objects created at function entry from those created when executing a particular statement within the function. Therefore, the choices are:

   { *Link time, Load time, Function entry, Statement runtime* }.

   Note #1: Ignore all optimizations performed by the compiler that may affect the answers.

   Note #2: Provide answers for all user-visible objects, not just the ones identified in comments.

```
#define   FixedSize 100
int       Size  = FixedSize;
double    GlobalStore[FixedSize];

 void dummy(struct Bag* B, int N) {                               /* dummy   */
     static double SaveSum = 0.0;
     int i;
     double x;
     double(*fp)(double);
     struct Contents *listS, *nextSPtr, *nextHPtr;

     fp = B->F;
     x  = fp(GlobalStore[N]);

     listS = (struct Contents*) alloca(sizeof(struct Contents));    /* Object1 */
     nextHPtr = 0;
     nextSPtr = 0;
     ...
     for (i=0; i < N; ++i) {
         struct Contents *ptr;                                      /* ptr     */
         ptr = (struct Contents*) malloc(sizeof(struct Contents));  /* Objects2 */
         ptr->next = nextHPtr;
         nextHPtr = ptr;

         ptr = (struct Contents*) alloca(sizeof(struct Contents));  /* Objects3 */
         ptr->next = nextSPtr;
         nextSPtr = ptr;
     }
     ...
     B->list   = nextHPtr;
     ...
 }
```

4. *(18 points) Code generation and OO dispatch*:

   During *dynamic method dispatch* in COOL, several operations are performed. Some properties of these operations depend on the *static type* of the receiver reference, some depend on the dynamic type of the object, and some are *independent of the type* (i.e., identical for all classes). Ignore SELF_TYPE throughout this question; it actually does not affect your answers. Assume no special optimizations are performed for method dispatch, i.e., focus on the code generated by Intermediate Code Generation.

   For each of the following properties, state which it depends on:

   Static Type, Dynamic Type, Neither

   (a) The numbers and types of arguments for a method

   (b) The address of the function to call

   (c) The vtable to use for the call

   (d) The offset (or field index in LLVM) of the vtable pointer within the object struct

   (e) The offset (or field index in LLVM) of the function pointer entry within the vtable struct

   (f) The number of load instructions that must be executed before making the call

5. *(12 points) Compiler Construction*:

   For each of the following actions, state whether it happens
   at *compiler construction time* (i.e., when developing the compiler),
   at *compile time* (i.e., when compiling or linking a program), or
   at *run-time* (i.e., when executing the code generated by the compiler).

   In some cases, multiple answers are possible (i.e,. an action can happen at either of two different times in different situations). For such cases, state why in the space provided. For example, *Evaluating an arithmetic expression* can happen:

   at compile-time, if all operands have known constant values,

   at runtime otherwise.

   There are no trick questions here. Consider only normal scenarios discussed in class.

   (a) Assign address range for a global variable.

   (b) Compute address of a local variable.

(c) Encounter a shift-reduce conflict.

(d) Choose in which area of memory (static data, heap, or stack) a particular variable will be stored.

(e) In a language with strong type safety (like Java), check that the type of an operand can be correctly converted (i.e., type-cast) to the type of the result variable for a particular operation.

(f) Allocate storage for an activation record.