CS591-5 Final

IIT Mandi

Total marks = 60, Time = 120 min

30 May 2018

Read the instructions and questions carefully. You may make any reasonably assumptions that you think are necessary; but state them clearly. Answer briefly. For questions with sub-parts, the division for the sub-parts are given in square brackets.

1. [20] **End-to-end**: Consider the following snippet of C code.

```
// Assume: i, num, val are integer variables.
//         Array A is a one dimensional array of integers.
//         print is an method that takes two integers as args.
//         Size of integer = 32 bits.
//         Number of available registers = 4; each of size 32 bits.

    for (i=0;i<num;i=i+1)
        if (A[i] == 0)
            break;
    print(i,0);
```

List ten unique tokens identified by the scanner [4]. Generate the corresponding three-address-codes [6]. Show the live-range interference graph [6]. Using the Kempe's heuristic do the register allocation (state the variables that gets registers/spilled) and show the generated code [4]. [Bonus] Use a hypothetical architecture and its assembly code to generate the final code [2].

Ans:

Tokens:

```
[kw, for]
[kw, if]
[kw, break]
[LSQBR, []
[RSQBR, ]]
[RBR, )]
[LBR, (]
[RBR, )]
[ID, i]
[OP, = ]
[INT, 0 ]
[DELIM, ;]
[OP, < ]
[ID, num]
[ID, A]
[OP, + ]
[OP, == ]
[ID, print]

Three address code:
L0: i = 0
L1: t1 = i >= num
L2: if t1 goto L8
L3: t2 = A[i]
L4: t3 = t2 == 0
L5: if t3 goto L2
L6: i = i + 1
L7: goto L1
L8: t4 = i + 1
L9: param t4
L10: param 0
L11: call print 2

Live ranges:
i: L1-L9,
num: L0-L7,
A: L0-L7
t1: L2
```

```
t2: L4
t3: L5
t4: L9

Generated code
L0: R0 = 0
L1: R3 = R0 >= R1
L2: if R3 goto L8
L3: R3 = R2[R0]
L4: R3 = R3 == 0
L5: if R3 goto L2
L6: R0 = R0 + 1
L7: goto L1
L8: R3 = R0 + 1
L9: param R3
L10: param 0
L11: call print 2
```

2. [20] **Reaching definitions**. (a) Define reaching definition [2].
(b) Give two example program snippets (smaller the better) to show how reaching definitions can be used by the compiler to do something meaningful [4].
Ans:

```
(a)
A particular definition of a variable is said to reach a given point if
there is an execution path from the definition to that point
the variable might may have the value assigned by the definition.

(b1)
    int i, j;
    j = i; // uninitialized var

(b2)
{
    int i;
    i = 2 // this def does not reach any use.
    ...   // code that does not use i.
}
```

(c) Define IN and OUT functions for any basic-block n in terms of (other) IN, OUT, GEN and $KILL$ maps [3].
Ans:

$$OUT(n) = GEN(n) \cup (IN(n) - KILL(n))$$
$$IN(n) = \cup_{m \in Pred(n)} OUT(m)$$

(d) Argue that the worklist based algorithm (discussed in the class) that computes reaching definitions terminates [3].
Ans:
The set of reaching definitions at any point expand monotonically (the transfer function is a monotone). The lattice is a finite height lattice - can only change a finite number of times.

(e) What is the complexity of the algorithm in big O notation? [3]. Briefly discuss.
Ans:
For any node, computing the 'totaleffect' = $O(N^2)$ – assuming union/intersection is $O(N)$.
A node's reaching definitions may change at most $O(N)$ times and hence each time its successors ($O(N)$) may be added to the worklist. Thus each node may be added to the worklist $O(N^2)$ times. Max entries read from the worklist = $O(N^3)$. Total cost $O(N^5)$.

(f) To compute reaching definitions, in the algorithm, we initialize the OUT maps to the empty set. If it was initialized to the set of all the definitions then will we get the desired solution? Yes, No, Sometime? Explain [3]. Feel free to explain using examples.
Ans:
For programs with no-loop, it will make no difference.
For programs with loops, we will get conservative results.

(g) What would happen if IN maps (for all the nodes except *entry*) were initialized to the empty set? To the set of all the definitions? [1+1]

Ans:
Empty-set: Works as per expectation.
All Definitions: Makes no difference - if the nodes are processed from the entry-point (and there is a single starting point) and we process a node only after, the out of atleast one of the predecessors has been computed. Otherwise, if we process the nodes in random order, it will lead to imprecision - we may get OUT as set of all-definitions.

3. [20] **Mixed-Bag** (a) Consider the following grammar to derive while loops:

```
S -> WhileStmt
     | SeqStmt
     | BreakStmt
WhileStmt -> while (cond) { S }
SeqStmt -> S  S
BreakStmt -> break;
```

For each of the possible nodes in the syntax tree, specify how you will process them to generate three-address codes [10].
Ans:

```
S -> WhileStmt {S.code = WhileStmt.code}
S -> SeqStmt {S.code = SeqStmt.code}
S -> BreakStmt {S.code = BreakStmt.code}
WhileStmt -> while (cond) { S }
             {S.target=L1;
          WhileStmt.code = L: cond.code || t=cond.addr || S.code || goto L; L1}
SeqStmt -> S1  S2 {S1.target = S2.target = SeqStmt.target;
                   SeqStmt.code = S1.code || S2.code }
BreakStmt -> break; {BreakStmt.code = goto BreakStmt.target}
```

(b) Draw the constant-propagation lattice, and define the meet and join rules [2].
(c) Which all C language constructs lead to merge points where we take meet of the constants, during our constant propagation algorithm? [3]
Ans:

```
if (cond) S
if (cond) S else S
while (cond) S
switch (cond) S
for (;;) S
do S while (cond)
goto L1;..;goto L1; .. L1:..
```

(d) Present an extension to the discussed constant propagation algorithm to handle conditional constants [5]. That way, your algorithm should be able to identify conditional constants in codes of the following form:

```
a = 2; c = a;
...
if (c == 2){
    ...
    b = 3
    ...
}else {
    ...
    b = 4
    ...
}
print a, b, c; // a, c are simple constants.
               // b is a conditional constant that evaluates to 3. Since it can
               // proven that only one branch is ever taken - since we know
               // the value of the condition is always the same (true, in this case).
```