Spring 2019 CSE Qualifying Exam: Compilers

1. **LR-Parsing.** Consider the following augmented grammar G with start symbol S':

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & iS \\ S & \rightarrow & iSeS \\ S & \rightarrow & \{L\} \\ S & \rightarrow & a \\ L & \rightarrow & LS \\ L & \rightarrow & \varepsilon \end{array}$$

(Here, ε denotes the empty string.)

   (a) For the grammar G above generate all of the LR(1) sets of items $I_0, I_1, I_2, \ldots$ along with complete transition information for an LALR (LALR(1)) parser. [Note: This is NOT a canonical LR(1) parser; states with common cores are merged.]

   (b) Using your answer to part (a), construct the action table and describe any conflicts. Assume the productions are numbered in order: 0–6.

2. **Syntax-Directed Definition: Variable scope.** Consider the BNF grammar for expression syntax, below:

$$\begin{array}{rcl} \langle \textit{expr} \rangle & ::= & \langle \textit{expr} \rangle_1 + \langle \textit{term} \rangle \\ & \mid & \langle \textit{term} \rangle \\ \langle \textit{term} \rangle & ::= & \langle \textit{term} \rangle_1 * \langle \textit{factor} \rangle \\ & \mid & \langle \textit{factor} \rangle \\ \langle \textit{factor} \rangle & ::= & \textbf{id} \\ & \mid & \textbf{const} \\ & \mid & (\ \langle \textit{expr} \rangle \ ) \\ & \mid & \textbf{let id} := \langle \textit{expr} \rangle_1 \ \textbf{in} \ \langle \textit{expr} \rangle_2 \ \textbf{end} \end{array}$$

- Here, **id** has a lexical attribute **id**.*text* that is the text of the identifier (i.e., the name of the variable) returned by the lexical analyzer.
- In a **let** ... **end** expression, the variable before the := sign is called a *binding occurrence* of that variable, and the *scope* of this occurrence is the corresponding $\langle \mathit{expr} \rangle_2$ (note: not $\langle \mathit{expr} \rangle_1$).
- We say that an occurrence of a variable v in a (sub)expression E is *bound in E* if that occurrence is in the scope of a binding occurrence of v inside E. Otherwise, the occurrence is *free in E*.

For example, consider the expression

```
x + let x := 4 in
        let y := x+w in
            x+y+z
        end
    end
```

The occurrences of the variables `x`, `y`, `z` are all free in the subexpression `x+y+z`, but these occurrences of `x` and `y` are bound in the entire expression. The `x` occurring in the subexpression `x+w` is bound in the expression as a whole, but the occurrences of `w` and of `z` are both free in the expression as a whole, as is the occurrence of `x` before the first **let**.

(a) (60% credit) Add semantic rules to the grammar above to compute as a synthesized attribute $\langle \mathit{expr} \rangle$.*freevars* the set of variables that occur free at least once in $\langle \mathit{expr} \rangle$. (Compute the same attribute for $\langle \mathit{term} \rangle$ and $\langle \mathit{factor} \rangle$.)

(b) (40% credit) Add semantic rules that compute as an inherited attribute a Boolean value **id**.*isfree* to each non-binding occurrence of a variable, indicating whether that occurrence is free in the expression as a whole. (You do not need to set this attribute to binding occurrences.)

In both parts, you may use set-valued attributes with some standard basic set operations: union ($A \cup B$ for sets A and B), the empty set $\emptyset$, as well as the following pure functions (with no side effects) for any object x and set A:

add(x, A)**:** returns $A \cup \{x\}$.

remove(x, A)**:** returns $A \setminus \{x\}$.

in(x, A)**:** returns TRUE if $x \in A$ and FALSE otherwise.

You may also test equality of objects with =.

3. **Control Flow and Liveness Analysis.** The following fragment of 3-address code was produced by a non-optimizing compiler:

```
 0   start: sum = 0
 1          i = 0
 2   L0:    j = 0
 3          if i >= 16 goto L6
 4          if j >= 64 goto L6
 5   L1:    t1 = b
 6          t2 = i * w
 7          t1 = t1 + t2
 8          t3 = j * 8
 9          t1 = t1 + t3
10          t4 = t1
11          f = a[t4]
12          if f > 0 goto L2
13          goto L3
14   L2:    sum = sum + f
15          goto L4
16   L3:    sum = sum - 1
17   L4:    j = j + 1
18          if j < 32 goto L1
19          i = i + 1
20          if i < 8 goto L0
21          goto L5
22   L5:    goto L0
23          sum = 0
24   L6:    no-op
```

Assume that there are no entry points into the code from outside other than at `start`.

(a) (20% credit) Decompose the code into basic blocks $B_1, B_2, \ldots$, giving a range of line numbers for each.

(b) (30% credit) Draw the control flow graph, describe any unreachable code, and coalesce any nodes if possible. Also give all the sets of blocks that constitute loops, marking inner loops as such.

(c) (30% credit) Give a table with 25 rows saying which variables are live immediately before each line number. Assume that `n` and `sum` are the only live variables immediately after line 24.

(d) (20% credit) Describe any simplifying transformations that can be performed on the code (i.e., transformations that preserve the semantics but reduce (i) the complexity of an instruction, (ii) the number of instructions, (iii) the number of branches, or (iv) the number of variables).

Algorithms

1. **(Solving a Recurrence)** Find tight asymptotic bounds on any positive real-valued function $T(n)$ satisfying the following recurrence for all sufficiently large n:

$$T(n) = T\left(\frac{3}{5}n\right) + 4T\left(\frac{2}{5}n\right) + n^2$$

   That is, find an expression $f(n)$, as simple as possible, such that $T(n) = \Theta(f(n))$. Use the substitution method to **prove** that your answer is correct. (Note: Implicit floors or ceilings in the recurrence do not affect the answer.)

2. **(Selection)** You are given two sorted arrays $X[1 \ldots n]$ and $Y[1 \ldots n]$ of n numbers each, where n is a positive integer. You may assume that there are no duplicates anywhere in the combined arrays. You are also given an integer k such that $1 \le k \le n$.

   Describe an algorithm to find the k^{th} order statistic (i.e., the k^{th} smallest) of the $2n$ numbers in the combined arrays. (For example, the n^{th} order statistic is just the median.)

   Your algorithm should run in time $O(\lg k)$, assuming (as usual) that each array access, comparison, and arithmetic operation on array indices takes constant time. (Note that you cannot merge the two arrays into a single sorted array, because this takes too long.)

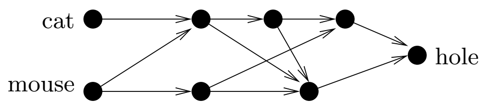
   Explain why your algorithm works, in enough detail to convince an intelligent but skeptical reader that it is correct.

3. **(Dynamic Programming)** The game of *Cat and Mouse* is played on a directed graph G, which we will assume is acyclic. There is a unique vertex h of G with outdegree 0, and we call this vertex the *hole*.[1] Two players, the *cat* and the *mouse*, start on any two respective vertices of G. They then alternate turns, where the player whose turn it is traverses an outgoing edge to an adjacent vertex.

   The game ends when either

   - one or the other of the players reaches the hole, in which case the mouse wins and the cat loses (presumably, the mouse can escape down the hole but the cat would get stuck there), or else
   - the two players occupy the same non-hole vertex, in which case the cat wins and the mouse loses (for obvious reasons).

   For example, in the following game, the cat can win if she moves first, but the mouse can win if he moves first:



   In the next paragraph, CF stands for "cat first" and MF stands for "mouse first."

   Describe an algorithm that takes as input a directed acyclic graph G with n vertices, represented as an adjacency list where $G.V = \{v_1, \ldots, v_n\}$, and outputs an $n \times n \times 2$ table

$$Win[1 \ldots n,\ 1 \ldots n,\ \{\text{CF}, \text{MF}\}]$$

   such that, for each $1 \le i, j \le n$ and each $t \in \{\text{CF}, \text{MF}\}$, $Win[i, j, t]$ contains either "C" (for cat) or "M" (for mouse) depending on who wins the game when both play optimally with perfect information, the cat starting on vertex v_i, the mouse starting on vertex v_j, and the player making the first move given by t.[2]

   You may assume that the adjacency list of G has been topologically pre-sorted, so that $(v_i, v_j) \in G.E$ implies $i < j$. The hole is then the last vertex v_n.

   Explain how your algorithm works, in enough detail to allow an intelligent non-expert to implement it correctly. For full credit, your algorithm should run in $O(mn)$ time, where $m = |G.E|$.

[1] Since h is the only vertex with outdegree 0, it is reachable from every other vertex in the graph.

[2] A non-ending game position is winning for the next-to-move player exactly when there exists a move the player can make that results in a new winning position for that same player; otherwise, the position is winning for the other player.