Compiler Construction

Homework Series 2

2016–2017; for submission deadline see BLACKBOARD

The homework exercises are to be done individually. You may discuss and exchange ideas freely, but not share code — except insofar this was developed during the lab sessions — or copy solutions.

In packaging and submitting your solutions, please stick to the following guidelines (which are meant to lighten the task of assessment). Failure to do so may reduce your grade.

- Submit your solution on BLACKBOARD (also if you are late). You may submit multiple times, but *only the last submission will be rated* (and its submission time will be used to deduct any points if you are late!)
- Put all your files into a single archive (zipped, tarred or however combined). Loose files will not be graded.
- Combine your textual answers into *one* pdf file, clearly recognizable from its file name, and *with your name and student number included in the text*. No other formats! If you must use Word, convert to pdf! If you dare use LaTeX, submit only the pdf even so! If you want to include drawings or hand-written text, scan them in and combine them into a single pdf!
- Put all your classes, grammars and other electronic resources into a package `pp.s1234567.q2_X`, *replacing the `1234567` by your own student number and the `X` by the question number*. You may reuse classes from the lab sessions, provided they are completely unchanged; otherwise, copy them into your package. Include a reference to the electronic resources in the pdf file of the previous item.

In case of late submission, the grading rules described on BLACKBOARD apply.

Question 1 *(30 points)* Consider the following JAVA code snippet, which omputes the product of all non-negative numbers in the array `a`:

```
int result = 1;
for (int i = 0; i < a.length-1; i = i + 1) {
    if (a[i] < 0) {
        continue;
    }
    result = result * a[i];
}
printf("Result:␣", result);
```

This can be parsed using the grammar `Fragment.g4` (provided in the files for this homework series). *Make sure you understand what the* JAVA *statement* **`continue`** *achieves in the context of a* **`for`** *statement.*

1. *(5 points)* Draw an *abstract* syntax tree (AST) of this code snippet, in which you only show (all) nodes that correspond to non-terminals of type `program`, `stat` or `expr`; i.e., omit all terminals and `decl`-, `target`- and `type`-nodes from the tree. Label the nodes of the AST with a combination of their tag according to `Fragment.g4` and a unique number.
2. *(7 points)* Draw a control flow graph of this code snippet, in which all `expr`-nodes and non-composite `stat`-nodes (i.e., all `stat`-nodes that do not have a `stat` as a child) of your AST appear as control flow nodes. Use the labelling of the previous sub-question to identify the control flow nodes.
3. *(5 points)* Explain *(do not just list)* which sets of control flow nodes form basic blocks. *(To answer this question, make sure you fully understand the concept of basic blocks.)*
4. *(7 points)* Draw a data dependency graph for this code snippet, using the same nodes as for the control flow graph, as well as a single additional node for the array `a`. Only draw dependency arrows between two nodes if the result of the source node affects the result of the target node.
5. *(6 points)* Assume that the array `a` is provided in global memory at address `@a`; it consists of a sequence of 4-byte slots, the first of which contains the length of the array, whereas the next slots contain its values. What optimisation(s) can a compiler carry out on the code generated for this program fragment? For each optimisation, how can the compiler detect that it is applicable? □

Question 2 *(20 points)* Consider the following statement, which assigns to `p` the percentage of `a` in the sum `a+3*b`. Assume that all values are integers.

```
p = a*af / (a*af + b*bf);
```

Translate this to ILOC in an optimal way (using as few instructions as possible), assuming that `a`, `af`, `b`, `bf`, and `p` are stored at offset `@a`, `@af`, `@b`, `@bf` and `@p` with respect to `r_arp`.

1. *(6 points)* Give a stack-based solution, i.e., in which there are only two registers and values of sub-expressions are pushed onto and popped from the stack (using the **`push`** and **`pop`** instructions in the ILOC library on BLACKBOARD).
2. *(6 points)* Give a register-based solution, i.e., in which values of sub-expressions are stored in registers.
3. *(8 points)* Compare these two kinds of solution in general: what are their advantages and disadvantages? □

Question 3 *(20 points)* Consider the following JAVA class (also provided as source file):

```
public class BinSearch {
    private static int[] a;

    public static void main(String[] args) {
        int length = args.length - 1;
        a = new int[length];
        for (int i = 0; i < a.length; i++) {
            a[i] = Integer.parseInt(args[i]);
            if (i > 0 && a[i] < a[i - 1]) {
                System.err.println("Input␣array␣is␣not␣sorted");
            }
        }
        depth(Integer.parseInt(args[length]));
    }

    public static void depth(int val) {
        System.out.println("Depth:␣" + depth3(val, 0, a.length));
    }

    public static int depth3(int val, int low, int high) {
        int result;
        if (high < low) {
            result = -1;
        } else {
            int mid = (low + high) / 2;
            if (a[mid] == val) {
                result = 1;
            } else if (a[mid] < val) {
                result = depth3(val, mid + 1, high) + 1;
            } else {
                result = depth3(val, low, mid - 1) + 1;
            }
        }
        return result;
    }
}
```

This computes the recursive depth at which a given array element is found in a binary search. Assume the program is called with parameters `-1 3 5 10 15 3`, and give diagrams displaying the memory content at every moment the program reaches the **`return`** statement of `depth3`. In your diagrams, use the following assumptions:

- The code is compiled exactly as given, without optimisation.
- The activation records should be included. They should *not* be stack-based, and only contain the elements necessary for this example.
- The method `main` (and its argument `args`) should not be included; hence, the first activation record is that of `depth`.
- The array `a` should also be included in your diagrams. It starts with a memory slot that contains its length, followed by slots for each of its values.

□

Question 4 *(30 points)* In this exercise, you have to write an ILOC program that mimics the `depth` and `depth3` methods of the previous question. Assume that the (absolute) address of the array `a` is given in a symbolic constant `@a`, and that the value of the argument `val` is stored at the activation record pointer `r_arp` when the program starts.

1. *(15 points)* Program `depth` and `depth3`, without any optimisations, following the memory layout of the previous question. Do not use the ILOC **`push`** and **`pop`** instructions, and build allocate activation records by manipulating `r_arp`. You do not have to worry about overlap with `@a`. Include your solution as an ILOC file.
2. *(5 points)* Explain exactly what parts of your ILOC code correspond to the precall, prologue, epilogue and postcall phases (see the book for the definition of these concepts), and what happens there.
3. *(5 points)* Explain if you have used caller-saves ot callee-saves registers, and why.
4. *(5 points)* What optimisations can you make to your code? Do not give the optimised code; instead, describe your optimisations in words. □