Compiler Construction

Take-home test 2, 2018–2019

(original version)

Submission deadline: Monday, 17 June 2019, 23:59 CEST

The take-home tests are to be done individually. You may discuss problems *and exchange* ideas *freely, but you must not share* solutions *(algorithms, code, text, drawings, etc.): do not* show *your solutions to anyone and do not* look *at other students' solutions.*

In packaging and submitting your solutions, please adhere to the following rules (which are meant to lighten the task of assessment). Failure to do so will result in the deduction of up to 4 points.

- Submit your solution on CANVAS (also if you are late). You may submit multiple times, but *only the last submission will be assessed*, and its submission time will be used to deduct points if you are late (as described in the module guide).
- Put all your files into a single .zip archive (use the .zip format, not .tar, .rar, or any other format). Loose files will not be assessed.
- Combine your textual answers into *one* text or PDF file named `answers.txt` or `answers.pdf` located at the top level, i.e. not in any subfolders, of your .zip file. Your name and student number must be included in the first line of the text of this answers file. Do not use any other formats.
- You can create illustrations or drawings by (a) drawing them using a drawing program (such as Inkscape or GraphViz) or a drawing package for LaTeX (such as TikZ), or (b) drawing them by hand on paper. You can include your drawings in your submission by (c) making them part of your PDF or (d) submitting them as separate (scanned or generated) PDF or JPEG files (but not in any other format) that you reference from your answers file. Any combination of (a) or (b) with (c) or (d) is allowed.
- Put all your classes, grammars and other programming resources into Java packages `pp.s1234567.q2_X`, replacing `1234567` by your own student number and `X` by the question number. The folder hierarchy for your packages must start at the top level, i.e. not in any other subfolders, of your .zip file. You may reuse predefined classes from the lab sessions, provided they are completely unchanged; otherwise, copy them into your package. Reference the electronic resources from the text or PDF file with your answers.
- Any resources in separate files (inside the submitted .zip file) that are not mentioned in your answers file will not be assessed.

Question 1 *(25 points)* Consider the following JAVA code snippet, which uses binary search to find a position of the target element `t` in the sorted array `a`, or reports that `t` is not in `a`:

```
1  int l = 0;
2  int r = a.length - 1;
3  int m = 0;
4  while(l <= r)
5  {
6    m = l + r;
7    m = m / 2;
8    if(a[m] < t)
9      l = m + 1;
10   else if(a[m] > t)
11     r = m - 1;
12   else
13     break;
14 }
15 if(l <= r)
16   print("Found␣t␣at␣index␣", m);
17 else
18   print("Did␣not␣find␣t");
```

This can be parsed using the grammar `Fragment.g4` (provided on CANVAS).

1. *(6 points)* Draw an *abstract* syntax tree (AST) for lines 4 to 14 this code snippet, in which you show exactly those nodes that correspond to non-terminals of type `stat` or `expr`. Annotate the nodes of your AST with the corresponding rule tag (e.g. `whileStat` or `addExpr`) according to `Fragment.g4` and the (starting) line number in the code snippet.
2. *(7 points)* Draw a control flow graph of this code snippet, in which every line appears as a control flow node, except for lines 5 and 14. Use the line numbers to label the nodes.
3. *(5 points)* Referring to your control flow graph, list the sets of control flow nodes that form basic blocks. Explain your answer.
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 two additional nodes, one for the target element `t`, and one for the array `a`. Only draw dependency arrows between two nodes if the effect of the source node affects the result of the target node. □

Question 2 *(15 points)* Again regard the program fragment in the previous question. Assume that the target value `t` (4 bytes) is provided in global memory at address `@t`. Likewise, 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.

1. Give ILOC code for the program fragment, and give JUNIT tests using arrays with at least four elements that cover both `print`-statements. Use the ILOC instruction **`out`** to mimic the effect of the `print` statements (using an arbitrary register value for the second `print`).

□

Question 3 *(20 points)* Consider the following PYTHON program `iter_fib.py` (also on CANVAS):

```
1  n = 4
2  def iter_fib():
3    a = 0
4    b = 1
5    def fib_step():
6      nonlocal a, b
7      c = a
8      a = b
9      b = b + c
10   def iterate(n):
11     if n > 0:
12       fib_step()
13       iterate(n - 1)
14   iterate(n - 1)
15   return b
16 print("fib(4) =", iter_fib())
```

This recursively computes the n-th Fibonacci number for $n \geq 1$; here, n is set to 4. The `nonlocal` statement is a relatively recent addition to PYTHON that finally provided the language with full support for nested functions with lexical scoping as discussed in the lecture; see

https://www.python.org/dev/peps/pep-3104/

for a detailed description of this statement and the justification for adding it to PYTHON in this way.

1. Give diagrams (as used in the lectures) displaying the activation records (ARs) during execution, for every case where the program reaches the end of line 9. Indicate the values of the ARP for each case. Make your ARs heap-based. You can assume that all variables are 4-byte integers and that all parameters are passed by value. Annotate your ARs such that the meaning of all of their elements is clear. You may use line numbers for return addresses. If you omit or do not use some part of the "standard" ARs that we discussed in the lecture, say why. □

Question 4 *(40 points)* Give hand-written ILOC code for `iter_fib`, without optimisations, following the memory layout of the previous question. Allocate activation records by manipulating `r_arp` and do not use **push**- and **pop**-instructions.

1. *(15 points)* Give your solution as an ILOC file.
2. *(5 points)* Give a (JUNIT) test class for your solution.
3. *(5 points)* Explain, by comments in your ILOC code, exactly what instructions correspond to the *precall*, *prologue*, *epilogue* and *postcall* phases, and what happens there.
4. *(5 points)* Implementations of `iterate` can use an optimisation called *tail call elimination*. Find out what this optimisation is and answer the following questions: When can it be applied? How does it work? What are its benefits? What are its drawbacks?
   Bonus (2 points): Name one reasonably popular programming language implementation (for a language that does contain recursive procedures) that does tail call elimination, and one that does not. Provide references (e.g. to web pages).
5. *(5 points)* Draw the activation record diagrams from Question 3, but for the case where tail call elimination is used.
6. *(5 points)* Change your ILOC code from subquestion 1 above to use tail call elimination. Use your (JUNIT) test class to check that the new code still works correctly. □