Examination 2021-08-23

Kompilatorteknik 1 (1DL321) – 5 hours

This is an examination **done at home**, so you can have your books and notes open when you take it. Please use an editor or word processor for all your answers in text. For questions that require drawing some picture or graph, you can either use an appropriate program or draw them on paper and attach a (good quality) picture of your solution to your answers.

When you finish, you need to either **upload a (preferably single) PDF** to your group in Studium or, in case you have trouble with uploading it, submit your PDF to the teacher. In either case, **you are supposed to stop answering questions at 13:00**, and your submission **must be uploaded/received by 13:30**. Remember to **write your anonymous code** in all PDF documents you submit.

The exam contains **40** points in total and their distribution between sub-questions is clearly identifiable. To get a final grade of **5** you must score at least **34**. To get a grade of **4** you must score at least **29** and to get a final grade of **3** you must score at least **20**.

Whenever in *real doubt* for what a particular question might mean, make sure to **state your assumptions clearly**. During the duration of the exam, you can ask questions to the teacher either **by sending him a mail**, or **by calling him on the mobile phone** that you can find in his homepage.

DON'T PANIC!

1. **Regular Expressions (3 pts total).** Consider a language where real constants are defined as follows: A real constant contains a decimal point or E notation or both. For example, 0.01, 2.71821828, ~1.2E12, and 7E~5 are real constants. The symbol "~" denotes unary minus and may appear before the number or on the exponent. There is *no* unary + operation. There must be at least one digit to the left of the decimal point, but there might be no digits to the right of the decimal point. The exponent following the "E" is a (possibly negative) integer. Write a regular expression for such real constants. You may define names (e.g., `digit`) for regular expressions you want to use in more than one place.

2. **Grammars & Ambiguity (4 pts total).** Show with an example and briefly explain why the following attempt to solve the dangling else ambiguity is still ambiguous.

$$\begin{aligned} \textit{stmt} &\rightarrow \texttt{if}\ (\ \textit{expr}\ )\ \textit{stmt} \mid \textit{matched\_stmt} \\ \textit{matched\_stmt} &\rightarrow \texttt{if}\ (\ \textit{expr}\ )\ \textit{matched\_stmt}\ \texttt{else}\ \textit{stmt} \mid \textbf{other} \\ \textit{expr} &\rightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

3. **Grammars (2 pts total).** Describe two common idioms in context free grammars that can *not* be parsed top-down.

4. **LL parsing (6 pts total; 1 for each row).** Consider the following grammar:

1	$E \to FE'$
2	$E' \to AFE'$
3	$E' \to \epsilon$
4	$A \to a$
5	$A \to \epsilon$
6	$F \to GF'$
7	$F' \to *F$
8	$F' \to \epsilon$
9	$G \to (E)$
10	$G \to fG$
11	$G \to c$

Construct an LL(1) parsing table for this grammar. Your table should look as follows:

	\$	a	f	c	(	)	$*$
E							
E'							
F							
F'							
G							
A							

5. **Calling Conventions (4 pts total; 1pt each).** Give the output of the following program, written in C syntax, when the parameter passing method is: (a) call-by-value, (b) call-by-reference, (c) call-by-value-result, and (d) call-by-name.

```
int i = 0;

void swap(int x, int y)
{  x = x+y; y = x-y; x = x-y;  }

main()
{ int a[3] = {1,2,0};
  swap(i, a[1]);
  printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```

NOTE: Your answer should look like this:

(a) With call-by-value the program prints:
(b) With call-by-reference the program prints:
(c) With call-by-value-result the program prints:
(d) With call-by-name the program prints:

6. **Symbol Tables (4 pts).** Describe a symbol table organization for a language with arbitrarily nested scopes; i.e., functions may be nested in other functions) that allows for:

   (a) Quick lookup of every name;

   (b) Ability to print a global list of all names in alphabetical order.

   Justify your decisions. There is not a unique correct answer, but the reasoning behind each decision will count towards "correctness."

---

7. **Activation Records & Tail Recursion (7 pts total).** Suppose `f` is a function with a call to `g` somewhere in its body:

```
f(...) {
   ... g(...) ...
}
```

   We say that this particular call to `g` is a *tail call* if the call is the last thing `f` does before returning. For example, consider the following functions for computing positive powers of 2:

```
f(x:Int, acc:Int) : Int {
   if x > 0 then return f(x-1, acc*2) else return acc
};
g(x:Int) : Int {
   if x > 0 then return 2*g(x-1) else return 1
}
```

   Here `f(x,1) = g(x) =` 2^x for all $x \geq 0$. The recursive call to `f` is a tail call, while the recursive call to `g` is not, because after the call to `g` there is still a multiplication by 2 to be performed. A function in which all recursive calls are tail calls is called *tail recursive.*

   (a) **(1 pt)** Below is a non tail recursive function for computing factorial.

```
fact(n:Int) : Int {
   if n > 1 then return n*fact(n-1) else return 1
}
```

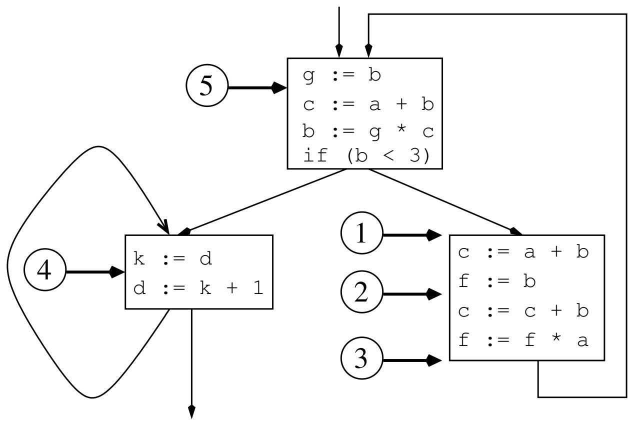
   Write a tail recursive function `fact2` that computes the same result as `fact`.

   (b) **(3 pts)** Recall that function calls are usually implemented using a stack of activation records. Trace through the execution of `fact` and `fact2` both computing the factorial of 4, writing out the stack of activation records at each step (i.e., draw the tree of activation records). Next to each stack frame, indicate the number of arithmetic operations (including comparisons) done before, during and after each record is created or destroyed.

   (c) **(3 pts)** Is there any place where you can see potential for making the execution of the tail-recursive `fact2` more time- or space-efficient than `fact`? You may not change `fact2`'s source code, but you may compile it differently. Briefly justify your answer.

---

8. **Register allocation (10 pts total; 2 pts each).** Consider the function whose control-flow graph is given below:



(a) Write down the set of live temporaries at the points 1–5 in the program. Assume that there are no live variables on exit. (It is a good idea to compute the live variables at all points in the program even though this part does not require it).

(b) Draw the register interference graph. Show a coloring for the graph using a minimal number of colors. Use color names such as $R_1, R_2, \ldots$ Write the color next to each node in the graph.

(c) Consider that we swap the instructions `f := b` and `c := c + b` so that the basic block containing them is as follows:

```
c := a + b
c := c + b
f := b
f := f * a
```

What is now the set of live registers at point 2 in the program (right before the instruction `f := b` in the new program)? What is now the minimum number of colors necessary to color the register interference graph?

(d) Consider again the original control-flow graph and assume that we have only 3 available registers for allocation. We will have to spill some temporaries to memory. Explain why it is not a good idea to spill temporary `k` into memory.

(e) While performing register allocation it is possible to eliminate some move instructions (instructions of the form `x := y`). If the register allocator assigns the same register to temporaries `x` and `y`, then the move instruction can be eliminated.

A simple way to force the register allocator to place two temporaries `x` and `y` in the same register is to combine the nodes in the register interference graph for `x` and `y` into one node (the combined node has all the edges of both original nodes). When is it legal to combine two nodes in the register interference graph? Please be precise and careful in your answer.

Good luck !