Examination 2021-06-12

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).** Suppose we have a language with persistent objects that can be located anywhere on the network. The name of an object is similar to names used on the WWW, with the form:

   `[<class>:]  [//<address>/] <path>`

   The notation above is interpreted as follows:

   - square brackets `[]` delimit optional parts of a name;
   - angle brackets `<>` delimit patterns, described below;
   - an identifier is a sequence of one or more lower case letters, capital letters, digits, and the characters '_', '$', and '-', not beginning with a digit or '-';
   - `<class>` is an identifier;
   - `<address>` is a list of one or more identifiers, separated by '.';
   - `<path>` is a list of one or more identifiers, separated by '/';

   Write a regular expression describing object names. You may use the notation of regular expressions given in the lecture slides (or some other notation provided you make it clear what its constructs mean).

2. **Grammars (3 + 3 = 6 pts total).**

   (a) Explain how to write a context-free grammar for a programming language where each function must contain at least one `return` statement.

   (b) Consider the grammar:

$$\begin{aligned} E &\rightarrow T \mid T + E \\ T &\rightarrow int \mid int * E \end{aligned}$$

   This grammar is ambiguous. Write an unambiguous grammar that recognizes the same language as the grammar above. Briefly describe why your grammar recognizes the same language and why it is unambiguous.

---

3. **LL Parsing (1 + 1 + 2 + 2 + 1 + 1 = 8 pts total).** Consider the following grammar:

$$\begin{aligned} S &\rightarrow NP\ VP \mid NP\ VP\ NP \\ NP &\rightarrow \texttt{the}\ AP\ \texttt{noun} \\ AP &\rightarrow AP\ \texttt{adjective} \mid \epsilon \\ VP &\rightarrow \texttt{verb} \end{aligned}$$

   The eventual goal is to obtain an LL(1) grammar that generates the same language as the grammar above.

   (a) Show the grammar after performing left-factoring.

   (b) Eliminate left-recursion from the grammar of part (a).

   (c) Compute *First* and *Follow* sets for the grammar of part (b).

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

	`the`	`noun`	`adjective`	`verb`	$
S					
$\vdots$					
$\vdots$					
VP					

   Is this grammar LL(1)?

   (e) Show the parser moves for the input sentence:

```
the adjective noun verb the adjective adjective noun
```

   (f) Show the parse tree for the above sentence.

---

4. **Activation Records (5 pts).** Instead of (or in addition to) using static links, there are other ways of getting access to non-local variables. One way is just to leave the variable in a register! Consider the program given at the top of the next page:

```
function f() : int =
  let var a := 42
      function g() : int = (a+1)
   in g() + g()
end
```

If `a` is left in, say register r_7, while `g` is called, then `g` can just access it from there.

What properties must a local variable, the function in which it is defined, and the functions in which it is used, have for this trick to work?

---

5. **Runtime Organization and Code Generation (5 pts total).** Recall the MIPS instructions:

   `sw $a0 n($sp)` store the value of the accumulator at address `n+$sp`

   `lw $ra n($sp)` load the return address register with the value stored at address `n+$sp`

   `addiu $sp $sp n` adjust the value of the stack pointer by `n`

   `move $a0 $t1` move the contents of `$t1` into `$a0`

   `jr $ra` jump to address stored in register `$ra`

   Imagine a compiler that uses a variation of the calling convention that we used in the lectures. We give below the code that this compiler generates to setup the activation record for a function and also the code that is used to access the i^{th} $(i = 1, \ldots, n)$ formal argument of a function with n arguments. The caller pops the arguments off the stack in this calling convention. On entry to a function the return address is in `$ra` and on exit the return value is in `$a0`.

```
cgen(f(x1,...,xn) begin e end) =
   sw    $fp 0($sp)
   addiu $sp $sp -4   ; push the old FP on the stack
   move  $fp $sp      ; set the new FP
   cgen(e)            ; evaluate the body of the function
   ...                ; code to return

cgen(xi) =            ; get the ith argument
   lw    $a0 z($fp)   ; where z = 8 + 4 * (n - i)
```

   (a) **(2.5pts)** Draw the stack contents at the point where the function `f` has been invoked and its execution is just before the code that evaluates the function body (i.e., the `cgen(e)` above). Put larger addresses at the top. Clearly mark on the stack the location of the frame pointer, the stack pointer, and the positions where the arguments and the old frame pointer are stored.

   (b) **(2.5pts)** Write the code for return (the `...` in the above code). To return you must use the instruction `jr $ra`. Recall that the return value must be in `$a0` and that the calling function pops the arguments.

---

6. **Parameter Passing Mechanisms (6 pts total).** Computer scientists working for SSA (the Swedish Space Agency) recently discovered that extraterrestrials are using an imperative programming language called Plan9 which has C-like syntax. They already know quite a lot about this language, but they are not really sure whether it uses call by value, call by reference, or call by name for parameter passing. Help them to discover which of these three mechanisms Plan9 uses by writing a (preferably simple) *single* program (in Plan9) that when executed prints "call by value", "call by reference", or "call by name" depending on the parameter passing mechanism that Plan9 uses. Briefly describe how your program works.

   Note: If you cannot think of a program that distinguishes between all three of the parameter passing mechanisms, answer this question by writing a program that distinguishes between two of them. A correct such program will get only **2 pts**.

---

7. **Basic Blocks, Liveness Analysis, and Register Allocation (7 pts total).** Consider the following fragment of intermediate code:

```
L0: x := y + x
    w := 2 * x
    if s = u goto L1
    x := w + u
    u := u - 1
    goto L2
L1: s := w + 1
L2: y := s + x
    if y > 1000 goto L0
L3:
```

   (a) **(1 pts)** Break the above piece of code into basic blocks and draw its control-flow graph. Place each basic block in a single node.

   (b) **(2 pts)** Annotate your control-flow graph with the set of variables live before and after each statement (not just before and after every block!), assuming that only `s` and `u` are live at label `L3`. Make sure it is clear where your annotations are placed.

   (c) **(4 pts)** Draw the register inference graph for this intermediate code and assign each temporary to a register for a machine that has only 3 registers. If you need to spill some temporary into memory, rewrite the program using psuedoinstructions `load x` and `store x` to load and spill the value of `x` from and to memory.

Good luck !