CST Part IB Supervisions

Example Sheet 4

Petar Veličković

Lent Term 2016

Compiler Construction

By the time we reach this supervision, the Compiler Construction course should be completed. This means that we may begin to shift our focus to Databases. For this supervision there will be more questions than usual, as well as an assortment of exam questions. We do not need to do them all next week; it would be useful if you would coordinate your efforts within a single group and hand in the same group of questions for this week, so everyone is in sync. I recommend attempting at least 60-70% of them now. The primary focus are assorted topics of interest, such as advanced stack frames, tail recursion optimisation, object-oriented representations etc.

1. Describe the benefits obtained when allowing *first-class functions* and *nested functions* in a programming language. What are the potential issues that need to be addressed when designing a runtime environment with these constructs? Provide an example of a language that does not support first-class functions, and a language that does support first-class functions but does not support nested functions.

2. Define and explain the purpose of a *static link*. Considering the following code:

   ```
   let main() =
   (   let a, b = 1, 2
       let f(x, y) = a*x + b*y
       let c = 3
       c := f(4, 5)
   )
   ```

   Sketch the state of the stack just before and just after starting to execute `f` in the final command. Demonstrate the usefulness of a static link in this case.

3. If a language supports certain constructs, simple stack-based evaluation as discussed in the lectures will not work (it potentially could result in unexpected run-time errors). Could you think of an example of this?

4. Consider the following code extract:

```
let a = 1;
let f() = a;
let g(a) = f();
print g(2);
```

What would you expect this code to print? If we were using a *dynamically-scoped* language (e.g. older dialects of LISP), what will actually be printed, and why?

5. By example of three different source codes of the same (mathematical) function, demonstrate the notion of a *tail-recursive function* and the *continuation passing style*. Then explain why a tail-recursive function is beneficial to an optimising compiler, and how CPS can be used to convert any function into a tail-recursive version.

6. Continuing from the previous problem, consider the standard functions `foldl` and `foldr`. Which one of these is tail-recursive? Why would the other one still be useful (given that they accomplish ∼the same goal)? Present a tail-recursive variant of the other with the aid of CPS.

7. Attempt the Compiler Construction exam questions from 2014 (2014p3q4, 2014p3q5).

8. Explain the "agreements" established between the caller and the callee of a particular method, if computation is to be performed on a register machine.

9. (**OPTIONAL***) Register spilling is mentioned very briefly in the lectures. Given that accessing memory locations is much more expensive than accessing registers, the ideal would be to spill as few registers into memory as possible. Suggest an algorithm which we could use, by examining a program, to determine which variables' contents should be spilled.
   [*Hint:* Consider a graph where the nodes are the variables of the program.]

10. Explain, by way of simple examples, *function inlining*, *constant propagation* and *peephole optimisation*. Could you give a rough idea of when inlining should be (in)appropriate?

11. Explain, with simple examples, how *objects* are represented in memory, and how *inheritance* between objects is handled (emphasising *method overriding*).

12. Considering the following C++ code:

    ```
    class A { int a1, a2; };
    class B : A { int b; };
    class C : A { int b; };
    class D : B, C { int d; };
    ```

    Suggest a possible storage layout for object `D`. What if `B` and `C` were extending `A` virtually (e.g. `class B : virtual A { ... }; class C : virtual A ...`)?

13. Explain, with a simple example, how support for *exception handling* can be implemented in a language?

*Steps into Part II territory (*Optimising Compilers*), but should be solvable.

14. Attempt the Compiler Construction exam questions from 2011 (2011p3q4, 2011p3q5).

Databases

The questions presented here cover the basics of what was covered so far; essential relational model theory, basic SQL and relational calculi. Feel free to let me know if any of the concepts are unclear (preferably in advance).

1. Attempt the 2012p4q6 exam question.

2. Define what it means for a database query to be *safe*.

3. Let $R(\mathbf{A}, \mathbf{B})$ and $S(\mathbf{B}, \mathbf{C})$ be schemas that contain r and s non-duplicate entries, respectively. For each of the following RA queries, state the *minimal* and *maximal* possible number of entries in the result, and the circumstances under which they occur.

   a) $\sigma_p(R \times S)$

   b) $\pi_{\mathbf{A},\mathbf{C}}(R \times S)$

   c) $\pi_{\mathbf{B}}(R) - (\pi_{\mathbf{B}}(R) - \pi_{\mathbf{B}}(S))$

4. With the same assumptions as in the previous problem, let b be any value from $\mathbf{B}$. Consider the following three RA queries:

   a) $\pi_{\mathbf{A},\mathbf{C}}(R \bowtie \sigma_{\mathbf{B}=b}(S))$

   b) $\pi_{\mathbf{A}}(\sigma_{\mathbf{B}=b}(R)) \times \pi_{\mathbf{C}}(\sigma_{\mathbf{B}=b}(S))$

   c) $\pi_{\mathbf{A},\mathbf{C}}(\pi_{\mathbf{A}}(R) \times \sigma_{\mathbf{B}=b}(S))$

   Will all of these queries always return the same results? If yes, provide an intuitive proof for why this is the case. If not, demonstrate an instance of the schemas where the results will differ.

5. Attempt the 2004p5q8 exam question.

Practical work

Please submit the parser exercise for this week if you haven't already. Once you've done so, you may look at the final step in our compiler—the implementation of a *code generator* for a virtual stack machine.

The virtual stack machine supports arbitrarily-named variables, and has the following commands:

`NOP`	empty instruction (no-op)
`PUSH n`	pushes a constant n to the stack
`POP`	pops a value off the stack
`LOAD s`	pushes the value of variable s to the stack
`STORE s`	pops a value x off the stack and assigns a variable s to x
`ADD`	pops two values x and y off the stack, pushes $x + y$
`SUB`	pops two values x and y off the stack, pushes $x - y$
`MUL`	pops two values x and y off the stack, pushes $x * y$
`OR`	pops two values x and y off the stack, pushes $x \mid y$
`AND`	pops two values x and y off the stack, pushes x & y
`NOT`	pops a value x off the stack, pushes $!x$
`JMP n`	jumps to the nth instruction in the code
`JZ n`	pops a value x off the stack, then jumps to n if $x = 0$
`JP n`	pops a value x off the stack, then jumps to n if $x > 0$
`JM n`	pops a value x off the stack, then jumps to n if $x < 0$
`HALT`	stops the execution

1. Define a data structure/type in your language of choice which you consider to be most suitable for storing a single instruction.
2. Implement a function `codegen`, which takes as its input an abstract syntax tree (as generated by `parse`) and produces a corresponding list of instructions.
3. Demonstrate that your function works as expected by presenting the result of lexing, parsing and generating (i.e. *compiling!!!*) the program given in *Example Sheet 2*.
4. Suggest any simple optimisations which could be performed to code produced by this method.