Take-Home Exam in *Implementation of Programming Languages (IPS)*

DIKU, Block 4/2021

Andrzej Filinski Robert Glück

23 June 2021, 10:00 – 15:00

General instructions

This 10-page exam text consists of a number of independent questions, grouped into thematic sections. The exam will be graded as a whole, with the questions weighted as indicated next to each one. However, your answers should also demonstrate a satisfactory mastery of all relevant parts of the course syllabus; therefore, you should aim to answer *at least* one question from each section, rather than concentrating all your efforts on only selected topics.

You are strongly advised to read the entire exam text before starting, and to work on the questions that you find the easiest first. Do not spend excessive time on any one question: you may want to budget with approximately 2–3 minutes/point. This should leave about an hour for the extra overhead due to the all-electronic submission format, such as using diagram-drawing tools, or scanning hand-drawn figures.

In the event of errors or ambiguities in the exam text, you are expected to *state your assumptions* as to the intended meaning, and *proceed according to your chosen interpretation*. The Absalon and Discord discussion fora are *not* to be used during the exam; in serious cases that you cannot resolve yourself, you may inform the course organizer directly at `andrzej@di.ku.dk`, but do not expect an immediate reply. Any significant corrections or clarifications will be announced on Absalon in the first instance.

Your answers must be submitted through the Digital Exam (DE) system as a single PDF file. Do not waste time on fancy typesetting; fixed-width fonts are fine for laying out tables, etc. You are also encouraged to draw figures and diagrams by hand and insert scans or (readable!) photos into the PDF document. After uploading your solutions to DE, remember to also press the "Submit" button on the "Confirm" page, especially if re-uploading a revised version. Note that the submission deadline is strictly enforced by a departmental exam administrator, so be sure to submit on time. Try to set off some time to proofread your solutions against the question text, to avoid losing points on silly mistakes, such as simply forgetting to answer a subquestion.

All unqualified mentions of "the textbook" refer to Torben Mogensen: *Introduction to Compiler Design* (2nd edition). You may answer the questions in English or in Danish.

Academic integrity This take-home exam is to be completed *100% individually*: for the full exam period, you are not allowed to communicate with any other person (whether taking the exam or not) about academic matters related to the course. By submitting a solution for grading you are affirming that you have fully complied with these conditions. Any violations will be handled in accordance with Faculty of Science disciplinary procedures.

Please note that some students may have shifted or extended exam times; thus, the prohibition against discussing the exam in public extends for the *entire day of the exam*.

1 Regular languages and automata

Question 1.1 (6 pts) Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ be the alphabet under consideration.

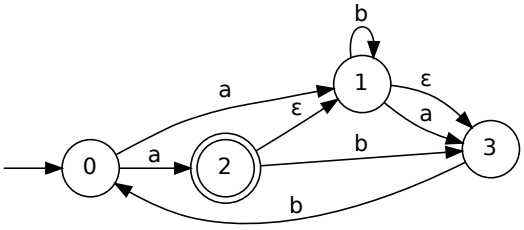
a. Consider the language L consisting of words of unbounded length that contain no identical *adjacent* letters. Examples of words in L: ε, `a`, `acab`, `bab`; non-examples: `cc`, `abba`, `baaa`.

   Is L regular? Explain why (for example, by showing that it can be described by a regular expression, recognized by a DFA, or similar), or why not (for example, by arguing that it cannot be recognized by any finite automaton).

b. Now, let the language L' be defined like L, but with the extra restriction that a word's first and last characters are *also* considered adjacent, in a wrap-around manner, and must hence be different. (In particular, words of length 1 do not satisfy the restriction.) Thus, the examples `a` and `bab` above would be excluded from L', but ε and `acab` are still allowed.

   Is L' regular? Again, explain why or why not.

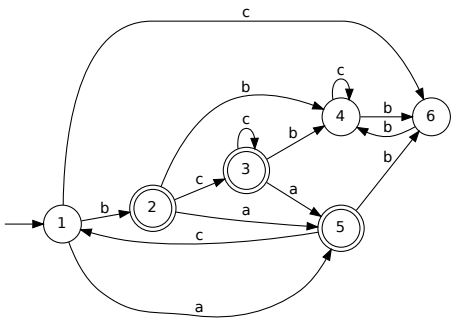
Question 1.2 (7 pts) Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Convert the following nondeterministic finite automaton (NFA) to a deterministic one (DFA), using the subset construction:



NFA for conversion

Show step-by-step how you determine the states and transition function of the DFA, and *draw* the final DFA. (You don't need to show in detail how you compute the ε-closures, but it should be clear in all instances what set you are taking the closure of, and what the result is.)

Question 1.3 (10 pts) Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Minimize the following DFA, using the algorithm from the textbook. Show how the groups are checked and possibly split, and *draw* the final, minimized DFA.



DFA for minimization

Be sure to deal properly with the limitations of the basic algorithm with respect to possible dead states: if you make *move* total, make it clear how you added the new dead state (but you don't need to actually *draw* the corresponding DFA), and how it is treated at the end. Conversely, if you eliminate any dead states before using the minimization algorithm, you must show how you *algorithmically* determined that those are precisely the dead states, not merely postulate it.

2 Context-free grammars and syntax analysis

Question 2.1 (9 pts) Consider the following context-free grammar G:

$S \rightarrow A$
$A \rightarrow A$ `c` A
$A \rightarrow A\ B$
$A \rightarrow B$
$B \rightarrow$ `a`
$B \rightarrow$ `b`

(The terminals are `a`, `b`, and `c`. The start symbol is S.)

a) Can string `caba` be derived from G? If it can, show a rightmost derivation $S \Rightarrow \ldots$.

b) Can string `baca` be derived from G? If it can, show a rightmost derivation $S \Rightarrow \ldots$.

c) Show that G is ambiguous. Find a string and show that it has two different syntax trees.

d) Is $L(G)$ regular? Justify your answer (max. 3 lines).

e) Make an unambiguous grammar for $L(G)$ that is right recursive.

Question 2.2 (14 pts) Consider the following context-free grammar:

$S \rightarrow A$ \$
$A \rightarrow$ `[` B `]`
$B \rightarrow$ `2` C
$B \rightarrow \epsilon$
$C \rightarrow$ `0` B
$C \rightarrow$ `1` B

(The terminals are `0`, `1`, `2`, `[`, and `]`, while \$ is the terminal representing the end of input. The start symbol is S.)

a) Show which of the above nonterminals are nullable. (Show the calculations.)

b) Compute the following first-sets:

$First(S) =$

$First(A) =$

$First(B) =$

$First(C) =$

c) Write down the constraints on follow-sets arising from the above grammar. (You may omit trivial or repeated constraints.)

$\{\$\} \subseteq Follow(A)$ (complete the rest)

d) Find the *least* solution of the constraints from (c). (Show the calculations.)

$Follow(A) =$

$Follow(B) =$

$Follow(C) =$

e) Compute the LL(1) lookahead-sets for all the productions of the grammar. Can one always uniquely choose a production?

$la(S \rightarrow A\ \$) =$

$la(A \rightarrow \texttt{[}\ B\ \texttt{]}) =$

$la(B \rightarrow \texttt{2}\ C) =$

$la(B \rightarrow \epsilon) =$

$la(C \rightarrow \texttt{0}\ B) =$

$la(C \rightarrow \texttt{1}\ B) =$

f) Construct the LL(1) table for the grammar. The table should work with the table-driven LL(1) parser program of the textbook.

g) Show the input and stack at each step during LL(1) parsing of the string `[21]$` using the table-driven LL(1) parser program of the textbook. You can assume that the program initially pushes start symbol S on the stack.

h) What is the derivation order performed by LL(1) parsers?

3 Interpretation and type checking

Question 3.1 (5 pts) This task refers to interpretation. Consider an extension of the functional source language from the textbook's Chapter 4 with the following production:

$$Exp \rightarrow \texttt{natcase}\ Exp\ \texttt{of 0 -> Exp |}\ \textbf{id}\ \texttt{+ 1 ->}\ Exp$$

(Note that the `|` on the RHS of the production is a terminal symbol, not an alternative-separator in the grammar.) The two numeric constants are only allowed to be precisely `0` and `1`, as shown.

The intended semantics of an expression

$$\texttt{natcase}\ Exp_1\ \texttt{of 0 ->}\ Exp_2\ \texttt{|}\ \textbf{id}\ \texttt{+ 1 ->}\ Exp_3$$

is that Exp_1 is evaluated first. If it evaluates to the integer 0, then Exp_2 is evaluated as the result of the `natcase`. On the other hand, if Exp_1 evaluates to an integer $n > 0$, then the result is obtained by evaluating Exp_3 with **id** bound to the integer $n - 1$. If Exp_1 evaluates to a negative integer, or to something that is not an integer at all, then an error is signaled.

For example, using the new construct, the usual factorial function could be written as follows:

```
int fac(int x) = natcase x of 0 -> 1 | y + 1 -> x * fac(y)
```

Extend the interpretation function $Eval_{Exp}$ with a case for the `natcase`-construct. That is, complete the following table entry:

$Eval_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
...	...
`natcase` Exp_1 `of` `0 ->` Exp_2 `|` **id** `+ 1 ->` Exp_3	(Your code would go here)

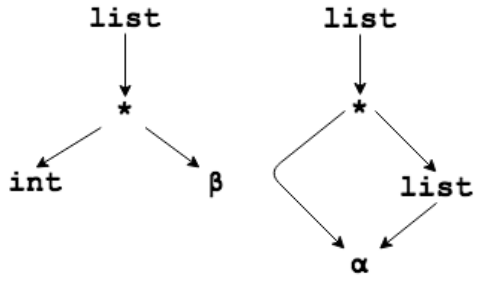
Remember to check for error conditions!

Question 3.2 (5 pts) This task refers to type-expression unification. Do the following types unify? (α and β are universally quantified type variables, and `list` and `*` are type constructors.)

`list(int *` β`)`

`list(`α `* list(`α`))`.

The graph representation of the two types is:



(a) Show the graph representation with the node identifiers (ID) and the final representatives (REP) after applying the most-general unification (MGU) algorithm[1].

(b) What are the type expressions of α and β after most-general unification?

(c) What is the unified type expression?

4 Code generation

Question 4.1 (7 pts) For the source and intermediate languages in the textbook's Chapter 6, generate intermediate code for the following source-code fragment:

```
x := 10;
while (!x) < 1 do
  x := x - a[x]
```

Assume that the symbol table maps `x` to $\texttt{r}_\texttt{x}$, and `a` to $\texttt{r}_\texttt{a}$, which contains the address of the first element of the array `a`. Remember that, like for C, no array-bounds checks are performed.

Use the translation schema for assignment statements from the book (*not* the slides). You don't *need* to reproduce the exact order in which the labels and temporary variables are generated, but we recommend that you still try to do so, to help ensure that you are otherwise following the translation specification exactly.

Question 4.2 (5 pts) This task refers to machine-code generation by the greedy technique that pattern matches the longest available intermediate-language (IL) pattern. Using the intermediate, three-address language (IL) from the textbook (slides) and the IL *patterns*:

t := $\texttt{r}_s$ + k M[$\texttt{t}^{last}$] := $\texttt{r}_t$	sw $\texttt{r}_t$, k($\texttt{r}_s$)
M[$\texttt{r}_s$] := $\texttt{r}_t$	sw $\texttt{r}_t$, 0($\texttt{r}_s$)
$\texttt{r}_d$:= $\texttt{r}_s$ - $\texttt{r}_t$	sub $\texttt{r}_d$, $\texttt{r}_s$, $\texttt{r}_t$
$\texttt{r}_d$:= $\texttt{r}_s$ + k	addi $\texttt{r}_d$, $\texttt{r}_s$, k
IF $\texttt{r}_s$ < $\texttt{r}_t$ THEN $\texttt{lab}_t$ ELSE $\texttt{lab}_f$ LABEL $\texttt{lab}_t$	slt R1, $\texttt{r}_s$, $\texttt{r}_t$ beq R1, R0, $\texttt{lab}_f$ $\texttt{lab}_t$:
IF $\texttt{r}_s$ < $\texttt{r}_t$ THEN $\texttt{lab}_t$ ELSE $\texttt{lab}_f$	slt R1, $\texttt{r}_s$, $\texttt{r}_t$ bne R1, R0, $\texttt{lab}_t$ j $\texttt{lab}_f$
LABEL lab	lab:

[1] Section "Advanced Type Checking" of type-checking slides; Sect. 6.5 of "Compilers: Principles, Techniques, and Tools", 2ed.

Translate the IL code below to MIPS code. (You may use directly, `a`, `x`, `y`, `z` as symbolic registers, or alternatively, use $\texttt{r}_a$, $\texttt{r}_x$, $\texttt{r}_y$, $\texttt{r}_z$, respectively.)

```
x := z^last - y
z := a^last + 5
M[z] := x
IF y < z THEN lab1 ELSE lab2
LABEL lab3
```

Question 4.3 (8 pts) This question concerns code generation in the FASTO compiler. Consider an extension of FASTO with an additional expression form:

$$Exp \rightarrow \texttt{least ID >= } Exp \texttt{ satisfying } Exp$$

The intended semantics is that, in an expression

$$\texttt{least } name \texttt{ >= } Exp_1 \texttt{ satisfying } Exp_2\,,$$

Exp_1 should evaluate to some integer n. Then the `least`-construct repeatedly evaluates Exp_2 (which should return a boolean) with *name* locally bound to n, $n+1$, $n+2$, ..., until Exp_2 returns `true`, at which point the successful value for *name* is returned. (Any other uses of *name* outside of Exp_2 are not affected.) If Exp_2 never becomes true, the expression runs forever. Any errors arising during evaluation of Exp_1 or Exp_2 abort execution as usual.

For example, the FASTO expression

```
least x >= 0 satisfying x*x == 100
```

should evaluate to 10. Had we instead started from, e.g., `12`, not `0`, the program would run forever. On the other hand, had we started from, say, `~20`, the result would have been -10.

In the `AbSyn` module, we extend the type of abstract expressions correspondingly:

```
type Exp<'T> =
...
  | Least of string * Exp<'T> * Exp<'T> * Position
```

Assuming that parsing, type checking, etc. for `Least` have been taken care of, your task is to generate MIPS code for it, according to the above specification. That is, you should complete the following case in the code generator:

```
let rec compileExp (e: Exp<Type>)
                   (vtable: SymTab<Mips.reg>)
                   (place: Mips.reg) : Mips.Instruction list =
  match e with
  ...
  | Least (name, e1, e2, pos) ->
      (* your code would go here *)
```

Try to stick roughly to F# syntax, as used in the compiler, *not* the pseudocode language from the book.

5 Liveness analysis, register allocation, and optimization

Question 5.1 (5 pts) This task refers to liveness analysis. Suppose that, after a liveness analysis, instructions i and j have the following in/out sets:

$$in[i] = \{\texttt{a}, \texttt{b}, \texttt{d}\}$$
$$i : \boxed{\quad ? \quad}$$
$$out[i] = \{\texttt{a}, \texttt{b}, \texttt{d}\}$$

$$in[j] = \{\texttt{c}, \texttt{d}\}$$
$$j : \boxed{\quad ? \quad}$$
$$out[j] = \{\texttt{a}, \texttt{d}\}$$

a) This means that instruction i can be:

(A) `a := b - d`

(B) `GOTO abc`

(C) `IF d < e THEN body ELSE end`

(D) `a := c + 5`

(E) `M[5] := b`

b) This means that instruction j can be:

(A) `M[a] := c`

(B) `RETURN a`

(C) `a := c`

(D) `IF c = d THEN next ELSE end`

(E) `LABEL correct`

Justify why your choice for instruction i and j satisfies the corresponding dataflow equation (not more than 5 lines for each choice). You need *not* justify why other choices for i and j are incorrect. **Note:** Multiple choices may be correct in (a) and (b).

Question 5.2 (12 pts) This task refers to liveness analysis and register allocation. Given the following program:

```
   F(x, y) {
 1:   a := x * y
 2:   b := x
 3:   LABEL loop
 4:   b := b + y
 5:   IF a < b THEN do ELSE end
 6:   LABEL do
 7:   y := b
 8:   b := y + y
 9:   GOTO loop
10:   LABEL end
11:   RETURN b
   }
```

1 Show **succ**, **gen** and **kill** sets for instructions 1–11.

2 Compute **in** and **out** sets for every instruction; stop after two iterations.

3 Show the interference table (Stmt | Kill | Interferes With).

4 Draw the interference graph for `a`, `b`, `x`, and `y`.

5 Color the interference graph with 3 colors; show the stack, i.e., the three-column table:
Node | Neighbors | Color.

6 Explain briefly the main actions that are taken when a register is marked 'spill' (max. 6 lines).
Hint: This task is independent of tasks (1-5).

Question 5.3 (7 pts) This question concerns dead-binding elimination (DBE) in FASTO, as seen in Lecture 6 and the project.

For each of the two FASTO expressions (a) and (b) below, with numbered subexpressions, work out the result of performing DBE on each subexpression. Show both the set of variables ultimately used in the optimized subexpression and the optimized subexpression itself. For subexpressions that are `let`s, also say explicitly whether that `let`-biding can be eliminated or not.

Preferably use the same format for your results as in Task 1 of Weekly 4. However, since none of the expressions evidently contain any I/O, you should just omit the "I/O?" column from the tables.

a. $_{(5)}$`let y = `$_{(3)}$`(let x = `$_{(1)}$`y+1 in `$_{(2)}$`y*y) in `$_{(4)}$`2*x+z`

b. $_{(5)}$`let x = `$_{(1)}$`y+1 in `$_{(4)}$`(let y = `$_{(2)}$`y*y in `$_{(3)}$`2*x+z)`

c. Note that the input expression in (b) is the result of performing the let-flattening transformation from copy/constant propagation on (a), even though no actual propagations are possible in this case. Comment on whether you get the same (or an equivalent) final result for the two expressions in (a) and (b), and briefly (3–5 lines) explain why/why not.

[End of exam text.]