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

DIKU, Block 4/2020

Andrzej Filinski Robert Glück

17 June 2020 9:00 – 18 June 2020 9:00

General instructions

This 11-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 all questions at least partially, rather than concentrating all your efforts on only some of them. 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 (budget with approximately 5 minutes/point).

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 discussion forum is *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 (+ a ZIP appendix for the implementation questions). 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 duration of the entire 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 24 hours beyond the end of the nominal exam period.

1 Regular languages, automata, and lexical analysis

Question 1.1 (5 pts) Let the alphabet $\Sigma = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$.

(a) Give a regular expression (RE) describing precisely the words that contain at least one '`x`' followed (not necessarily immediately) by a '`y`'. Examples: "`xyzzy`", "`yzxzxzy`"; non-examples: "`zyx`", "`yzzy`".

(b) Give an RE for the *complement* of the language in (a), i.e., those words in which no '`x`' is followed (not necessarily immediately) by a '`y`'. (Swap the examples and non-examples above.)

(c) Give a deterministic finite automaton (DFA) for the language in (a).

(d) Give a DFA for the language in (b).

For each part, *briefly* explain how your construction works.

For parts (c) and (d), you may either draw the DFA, or give the same information in textual/tabular form. In either case, be sure to clearly identify the initial and final states of the DFAs. Your DFAs in (c) and (d) *do not* have to be systematically derived from the REs in (a) and (b); and even if they are, you are not asked to show the actual derivations.

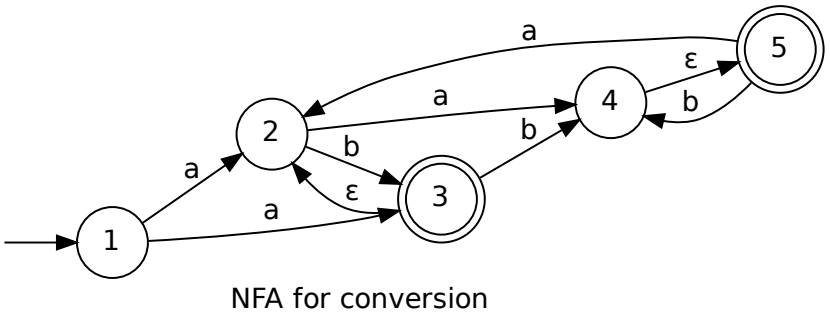
Question 1.2 (4 pts) Draw the non-deterministic finite automaton (NFA) corresponding to the following RE (over the alphabet $\{\mathtt{a}, \mathtt{b}\}$):

$$((\mathtt{a} \mid \mathtt{b}\,?)^*\,(\mathtt{a}\,\mathtt{b})\,?)$$

Recall that $r\,?$ is an abbreviation for $(r \mid \varepsilon)$. By convention, the postfix '?' (like '*') groups tighter than all the infix RE operations.

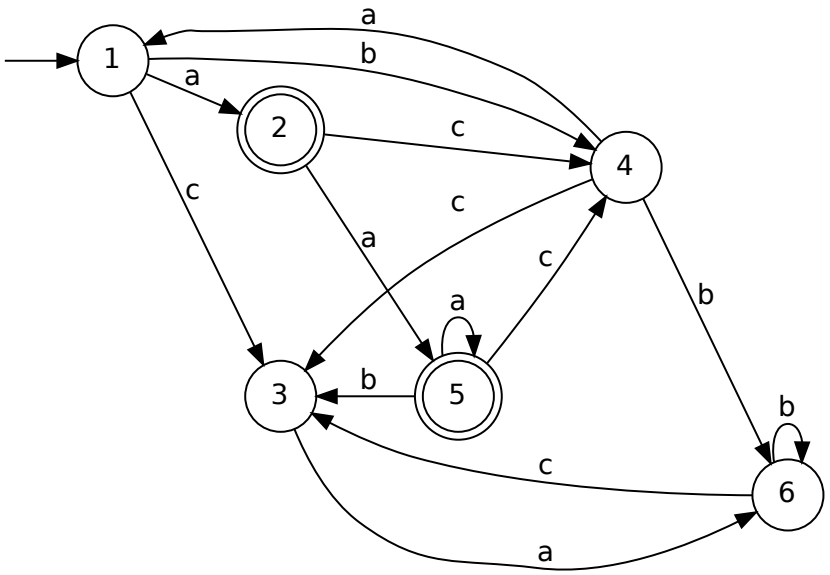
Follow the construction in the textbook as closely as possible; *do not* optimize or simplify the NFA.

Question 1.3 (6 pts) Convert the following NFA to an equivalent DFA:



Show explicitly how the new states and transition functions are derived incrementally. Then *draw* the resulting DFA.

Question 1.4 (8 pts) Minimize the following DFA:



Follow the construction in the textbook as closely as possible; in particular, show how the groups are incrementally refined by the algorithm. If relevant, start by making the *move* function total, by introducing an extra, dead state.

Draw the final, minimized DFA.

2 Context-free grammars and syntax analysis

Question 2.1 (4 pts) This task refers to syntax analysis. Consider the following grammar G with start symbol S:

$S \rightarrow B$
$B \rightarrow A$
$B \rightarrow B$ `[` A `]`
$A \rightarrow A$`a`
$A \rightarrow \epsilon$

a) Is grammar G ambiguous?

b) Describe briefly the three conditions prove that a grammar is unambiguous? (Use not more than 8 lines to answer the question.)

Question 2.2 (12 pts) This task refers to syntax analysis. Consider the following grammar:

$S \rightarrow B\ \$$
$B \rightarrow A\ C$
$C \rightarrow$ `[` A `]` C
$C \rightarrow \epsilon$
$A \rightarrow$ `a` A
$A \rightarrow \epsilon$

(The terminals are `a`, `[`, and `]`, while \$ is the pseudo-terminal representing the end of input.)

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

b) Compute the following first-sets:

$First(S) =$
$First(B) =$
$First(C) =$
$First(A) =$

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

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

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

$Follow(S) =$
$Follow(B) =$
$Follow(C) =$
$Follow(A) =$

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

$la(S \rightarrow B\ \$) =$
$la(B \rightarrow A\ C) =$
$la(C \rightarrow$ `[` A `]` $C) =$
$la(C \rightarrow \epsilon) =$
$la(A \rightarrow$ `a` $A) =$
$la(A \rightarrow \epsilon) =$

f) Write a (checking-only) recursive-descent parser for the grammar in the style of the course book. (Use variable `input`, functions `match` and `reportError`, and the same pseudocode for programming.)

g) What is the result of parsing the word `a[][aa]a$` with the parser in (f)? Explain it briefly (max. 4 lines).

3 Interpretation and type checking

Question 3.1 (7 pts) Consider an extension of the interpreted language in Chapter 4 of the textbook with a vaguely F#-like `match`-expression, given by the following additional nonterminals and productions:

$$
\begin{array}{rcl}
 & \vdots & \\
\mathit{Exp} & \rightarrow & \texttt{match}\ \mathit{Exp}\ \texttt{with}\ \mathit{Branches}\ \texttt{end} \\
\mathit{Branches} & \rightarrow & \texttt{when}\ \mathit{Alts}\ \texttt{:}\ \mathit{Exp} \\
\mathit{Branches} & \rightarrow & \texttt{when}\ \mathit{Alts}\ \texttt{:}\ \mathit{Exp}\ \texttt{;}\ \mathit{Branches} \\
\mathit{Branches} & \rightarrow & \texttt{otherwise}\ \texttt{:}\ \mathit{Exp} \\
\mathit{Alts} & \rightarrow & \mathit{Alt} \\
\mathit{Alts} & \rightarrow & \mathit{Alt}\ \texttt{,}\ \mathit{Alts} \\
\mathit{Alt} & \rightarrow & \textbf{num} \\
\mathit{Alt} & \rightarrow & \textbf{num}\ \texttt{..}\ \textbf{num}
\end{array}
$$

For example, a well formed expression using the new constructs could look like this:

```
match x with
  when 1, 7, 9: x+10 ;
  when 2, 5..8: x+20 ;
  otherwise:    30
end
```

The informal semantics of the `match` construct is that the selector expression is evaluated to a number, which is then compared against the numbers listed in the `when`-branches. If the number matches a branch, the corresponding branch expression is evaluated; if the number doesn't match any branch, but the list ends with an `otherwise`-branch, its expression is evaluated. If there is no `otherwise`, a failed match signals an error. If the number is matched by more than one branch, the first one is chosen.

A `when`-branch lists one or more *alternatives*, each of which may either be a single number, or an inclusive range. For example the range `3 .. 5` is equivalent to the list `3,4,5`. If the nominal upper bound of a range is less than the lower (e.g., `5 .. 3`), the range matches nothing.

Thus, for the example above: if `x` is currently bound to 1, the result is 11; if `x` is 2, the result is 22; if `x` is 3, result result is 30; if `x` is 6, the result is 26; and if `x` is 7, the result

is 17. If we had omitted the `otherwise`-branch, the case where `x` was 3 would instead have signaled an error.

Give definitions for the interpretation functions in the same style as in the textbook, i.e., extend $Eval_{Exp}$ with a case for `match`-expressions, and define suitable auxiliary functions (e.g., $Try_{Branches}$, etc.) to properly express the above semantics.

Question 3.2 (7 pts) This task refers to type checking. Assume we want to extend FASTO with a new second-order array combinator named `upd`, *whose type is*

`upd :` $(\ (\alpha * \beta \rightarrow \gamma)\ *\ \alpha\ *[\beta]\ )\ \rightarrow\ [\gamma]$.

Its semantics is: `upd(f, val, [`a_1`,...,`a_n`]) = [f(val,`a_1`),...,f(val,`a_n`)]`, i.e., it applies the function argument to `val` and each element of the input array.

Your task is to write the type-checking pseudocode for `upd`, i.e., the high-level pseudocode for computing the result type of `upd(f, val_exp, arr_exp)` from the types of `f`, `val_exp` and `arr_exp`, together with whatever other checks are necessary. Assume that the function parameter of `upd` is passed only as a string denoting the function's name, i.e., do *NOT* consider the case of anonymous functions (lambda expressions). Try to stay close to the notation used in the textbook/lecture slides. Give self-explanatory error messages. The result of $Check_{Exp}$ is only the type of the `upd(f, val_exp, arr_exp)` expression. Present the type-checking pseudocode in a way compatible to the course book.

$Check_{Exp}(Exp, vtable, ftable) =$ `match` Exp `with`	

`upd( f`
`, val_exp`
`, arr_exp`
`)`
	(type-checking pseudocode)

Question 3.3 (3 pts) This task refers to type checking. Consider the F# code below:

```
let a = [] in
let z = if List.isEmpty a
        then ([["a"],["b"]], [1], [4,5])
        else ([],            [],  [3])
z
```

a) Write the type of the `then` expression of the `if` branch (before unification with the `else` expression)

b) Write the type of the `else` expression of the `if` branch (before unification with the `then` expression)

c) Write the type of `z`, i.e., the most general unifier of the `then` and `else` expressions.

4 Code generation

Question 4.1 (5 pts) Generate intermediate-language (IL) code for the following program in the source language of the textbook's Chapter 5:

```
x := y;
while x && (y/x < 100) do
  x := x - 1;
y := (x < y)
```

Here, the source variables `x` and `y` hold integers, but remember that any integer can be used as a condition, with a non-zero value interpreted as true, and zero as false.

Assume the variable table contains the mappings $[\texttt{x} \mapsto r_{\texttt{x}}, \texttt{y} \mapsto r_{\texttt{y}}]$, where $r_{\texttt{x}}$ and $r_{\texttt{y}}$ are IL variables. Use the translation schema for assignments (`:=`) given in the textbook (*not* the simplified version in the lecture slides). For *newlabel*()/*newvar*(), just use globally unique names, preferably ending with a digit (to make it evident that they were freshly generated).

Follow the constructions in the textbook as closely as possible; *do not* optimize or simplify the IL code.

Question 4.2 (5 pts) Consider an extension of the source language from the textbook with with C/Java-style *compound assignments*, expressed by the following additional productions for *Stat*:

$$\begin{array}{rcl} & \vdots & \\ \mathit{Stat} & \rightarrow & \textbf{id binop}= \mathit{Exp} \\ \mathit{Stat} & \rightarrow & \mathit{Indexed}\ \textbf{binop}= \mathit{Exp} \end{array}$$

The informal semantics is that, e.g., `x += y` should behave the same as `x := x + y`. However, when the left-hand side of such a compound assignment is an *indexed* variable, e.g, `v[3*i] += y`, the address calculations (in particular, the multiplication) should only happen once in the generated code, not twice.

Extend the definition of the IL generation function $\mathit{Trans}_{\mathit{Stat}}$ with suitable cases for handling the above two new productions in the grammar.

Question 4.3 (4 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*:

q := r_s + k r_t := M[q^{last}]	lw r_t, k(r_s)
r_t := M[r_s]	lw r_t, 0(r_s)
r_d := r_s + r_t	add r_d, r_s, r_t
r_d := r_s + k	addi r_d, r_s, k
IF r_s = r_t THEN lab_t ELSE lab_f lab_t:	bne r_s, r_t, lab_f lab_t:
IF r_s = r_t THEN lab_t ELSE lab_f	beq r_s, r_t, lab_t j lab_f:
lab:	lab:

Translate the IL code below to MIPS code. (You may use directly a, x, y, z as symbolic registers, or alternatively, use r_a, r_x, r_y, r_z, respectively.)

```
a := a + 3
y := a^last + 5
x := M[y^last]
IF z=x THEN lab1 ELSE lab2
lab1:
```

Question 4.4 (10 pts) This task refers to liveness-analysis and register-allocation. Given the following program:

```
   F(a, b, c) {
 1:   x := b
 2:   LABEL begin
 3:   IF c > 0 THEN end ELSE cont
 4:   LABEL cont
 5:   y := x + 1
 6:   b := a + c
 7:   z := x + y
 8:   c := b
 9:   a := a - 1
10:   GOTO begin
11:   LABEL end
12:   RETURN c
   }
```

1 Show **succ**, **gen** and **kill** sets for instructions 1–12.

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, c, x, y, and z.

5 Color the interference graph with 4 colors; show the stack, i.e., the three-column table:
Node — Neighbors — Color.

5 Fasto implementation

In this section, we consider a *generalization* of the `filter` SOAC that you implemented in the Group Project. In the original `filter`, the predicate function determines whether each element of the input array is to be included in the output or not. That is, an element is included either 0 or 1 times, depending on a boolean result.

A more general version of filtering allows the predicate to specify an arbitrary (non-negative) number of repetitions of each element. For example, in plain FASTO, we can write

```
filter(fn bool (int x) => 10 < x, {12,3,5, 26,7,37})
```

which returns {12,26,37}; but with the generalization, we can *also* write

```
filter(fn int (int x) => x / 10, {12,3,5, 26,7,37})
```

which should return {12,26,26, 37,37,37}.

However, in order to simplify the implementation of the compiler, we also specify that the length of the output array can be *at most* the same as the input array; if the output would be longer, the excess elements are simply discarded. For example if the `26` in the example had been `46`, the full output would be {12,46,46, 46,46,37, 37,37}, but this gets truncated to the length-6 result {12,46,46, 46,46,37}.

A *negative* repetition count (for example, if the input array above had replaced `26` with `~26`) should report an error. (You may use same message as for `iota`/`replicate`, or a dedicated one.) This applies even if the output array is already full, so that the element would not be included anyway.

Question 5.1 (4 pts) Modify the *type checker* (`TypeChecker.fs`), to allow the functional argument in `filter` to return either a `bool` or an `int` (but no other type), and give a meaningful error message otherwise.

Question 5.2 (6 pts) Modify the *interpreter* (`Interpreter.fs`), so that it handles both kinds of predicate function, and gives meaningful messages on errors: either if called with inappropriate types, or if an `int`-returning predicate returns a negative number. Note that the interpreter should *also* truncate too-long outputs, as described above.

Question 5.3 (10 pts) Modify the *code generator* (`CodeGen.fs`) to allow general repetition counts. You should assume that the code being compiled has already been properly type checked, but the generated MIPS code should still abort execution if a negative count is encountered at runtime.

In your PDF document, show the source code for your extensions only, i.e., the resulting code for the case

```
| Filter (...) ->
    ...
```

in each of the three affected modules (as well as any auxiliary functions you define). You should also *briefly* explain which lines you modified/added, compared to the original `Filter` code, and why.

Additionally, include the complete (compilable) code for the modules you modify as a separate ZIP archive. The 3 files should be drop-in replacements for the ones in the handout. You may keep any code for the other functionality (`replicate`, `scan`, etc.)

that you implemented for the group project, or start with a fresh copy from the original FASTO handout. (Note, however, that the examples above rely on at least negation and division also being implemented.) *Do not* include the full compiler, and especially not any compiled binaries, in the ZIP file.

In the exam document, you should also demonstrate, by showing suitable sample FASTO programs and their outputs (which may contain error messages where relevant), that your code correctly implements the asked-for functionality, or where it fails to do so. Your demonstrations must be easily *reproducible*: please also include in the above ZIP archive the FASTO source program(s) you used.