Re-Exam in *Implementation of Programming Languages (IPS)*

DIKU, Block 4R/2023

Andrzej Filinski Robert Glück

23 August 2023 (4 hours)

General instructions

This 9-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. (But please put your answers in the correct order in the submission document!) Do not spend excessive time on any one question: you may want to initially budget with about 2 minutes/point; this should leave some time for revisiting questions that you were not able to complete in the first pass.

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*.

Your answers must be submitted through the ITX system, as directed. Note that editing will lock promptly at the end of the exam, so remember to keep your main document updated, to ensure that all your solutions will be graded. For (sub)questions involving figure drawing, be sure to copy your completed figures from the drawing tool into your answer document as you finish them.

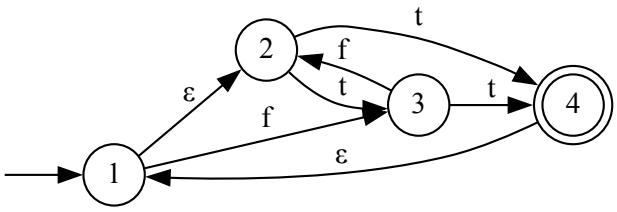
For presenting tabular information, you may use either Word tables, or just visual layout with a fixed-width font. You may also prepare the table as a handwritten figure and copy it in, but that will probably not be the fastest option. When editing program code or similar material, you'll probably want to turn off Word's AutoCorrect features. Also, don't worry about exactly reproducing the fonts, special symbols, subscripts, etc. used in the exam text, as long as the meaning is clear. Finally, don't waste time on copying or paraphrasing the question text into your submission document; it is fine to just state the (sub)question number and give your answer.

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 Æ. Mogensen: *Introduction to Compiler Design* (2nd edition). You may answer the questions in English or in Danish.

1 Regular languages and finite automata

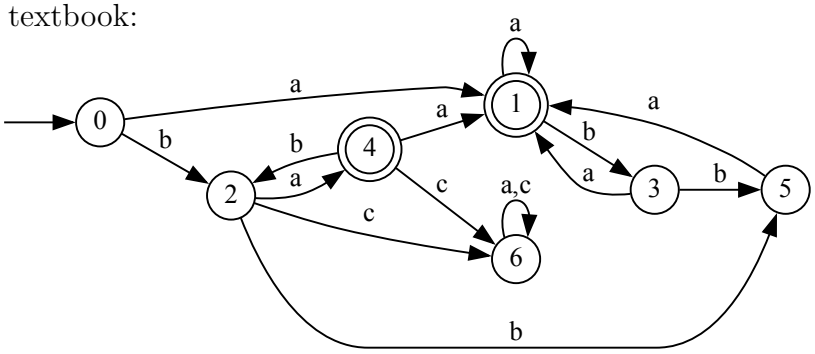
Question 1.1 (8 pts) Convert the following nondeterministic finite automaton (NFA) to a deterministic one (DFA), using the subset construction from the textbook:



NFA for conversion (Σ = {t,f})

Show step-by-step how you determine the states and transition function of the DFA, and *draw* the final DFA. (Remember to indicate the initial and final states!) 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 1.2 (9 pts) Minimize the following DFA, using the algorithm from the textbook:



DFA to minimize (Σ = {a,b,c})

Show how the groups are checked and possibly split; summarize the final, minimized DFA in tabular form, and also *draw* it.

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 explain how you systematically 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:

$$
\begin{array}{lcl}
S & \rightarrow & X \\
X & \rightarrow & X \texttt{ + } Y \\
X & \rightarrow & Y \\
Y & \rightarrow & Y \texttt{ b - -} \\
Y & \rightarrow & \epsilon
\end{array}
$$

(The terminals are `b`, `+`, and `-`. The start symbol is S.)

a. Can string `++` be derived from G? If it can, show a leftmost derivation $S \Rightarrow ...$

b. Can string `b--b--` be derived from G? If it can, show a leftmost derivation $S \Rightarrow ...$

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

d. Eliminate left-recursion from G. Show the transformed grammar.

Question 2.2 (12 pts) Consider the following context-free grammar:

$$
\begin{array}{lcl}
S & \rightarrow & X \texttt{ \$} \\
X & \rightarrow & A\ B \\
A & \rightarrow & \texttt{a} \\
A & \rightarrow & \epsilon \\
B & \rightarrow & \texttt{b}\ A\ B \\
B & \rightarrow & \epsilon
\end{array}
$$

(The terminals are `a` and `b`, 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:

$First(S) =$

$First(X) =$

$First(A) =$

$First(B) =$

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

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

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

$Follow(X) =$

$Follow(A) =$

$Follow(B) =$

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

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

$la(X \rightarrow A\ B) =$

$la(A \rightarrow \texttt{a}) =$

$la(A \rightarrow \epsilon) =$

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

$la(B \rightarrow \epsilon) =$

f. Write a (checking-only) recursive-descent parser for the grammar in the style of the textbook. Show only the parsing functions for the nonterminals S and B. (Use variable `input`, functions `match` and `reportError`, and the same pseudocode for programming.)

3 Interpretation and type checking

Question 3.1 (10 pts) This task refers to interpretation of a simple, C-like expression language, with integers as the only data type, but where expressions can now have *side effects*, and thus modify the contents of variables during evaluation. Its abstract grammar is as follows:

$$\mathit{Exp} \rightarrow \textbf{num} \mid \textbf{var} \mid \mathit{Exp}\ \texttt{+}\ \mathit{Exp} \mid \textbf{var}\ \texttt{=}\ \mathit{Exp} \mid \textbf{var}\ \texttt{++}$$

As usual, parentheses may be used to indicate the intended grouping of concrete expressions with multiple possible parses.

The semantics of all the expression forms is much like in C or C#. For example, the expression `x = 2 + 2` evaluates to 4, but also sets the variable `x` to that result. (Note that, in this question, we follow the C syntactic tradition of using `=`, rather than `:=`, for the assignment operator.) Similarly, the expression `x++` increments `x`, but returns the value `x` had *before* the increment. Accessing a variable before it has been assigned any value is considered an error.

Like in C# (but unlike in C) we explicitly specify that expressions are evaluated strictly from left to right. That is, in Exp_1 `+` Exp_2, the subexpression Exp_1 will always be fully evaluated, and all of its variable updates performed, before evaluation of Exp_2 begins. For example, consider evaluating the expression

```
y = ((x = 3) + x++)
```

in the initial environment (in which all variables are unbound). The subexpression `x = 3` sets `x` to 3 and also returns 3. Then `x++` increments `x` to 4, but returns its previous value of 3, so that the result of the whole expression is 6, which is also stored in `y`.

On the other hand, evaluating

```
y = (x++ + (x = 3))
```

would result in an error, because `x` does not yet have a value when `x++` is evaluated.

In the interpreter, a symbol table *vtable* still maps variable names to (integer) values, and we have the standard functions $\mathit{lookup}(\mathit{vtable}, \mathit{name})$ and $\mathit{bind}(\mathit{vtable}, \mathit{name}, \mathit{value})$. (Since there are no function calls in the expression grammar, no *ftable* is needed.) As usual, we also assume that we have auxiliary functions $\mathit{getvalue}(\textbf{num})$ and $\mathit{getname}(\textbf{var})$ for extracting the numeric value and the variable name from the corresponding grammar terminals.

However, the new interpretation function for expressions,

$$\mathit{Eval}_{\mathit{Exp}}(\mathit{Exp}, \mathit{vtable}) = \cdots$$

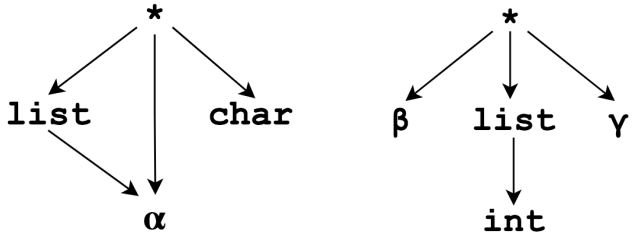
should now return a pair (n, vtable') of the evaluation result itself, together with the possibly modified symbol table resulting from the evaluation. Define $\mathit{Eval}_{\mathit{Exp}}$ using the same pattern-matching notation as in the textbook ("case *Exp* of ..."), but for the new syntax and semantics of the source language.

Question 3.2 (6 pts) This task refers to type-expression unification. Do the following types unify? (α, β, and γ are universally quantified type variables, and `list` and `*` are type constructors.)

`list(`α`) * `α` * char`

β `* list(int) *` γ

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, β, and γ after most-general unification?

c. What is the unified type expression?

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

4 Code generation

Question 4.1 (10 pts) This task refers to intermediate-code generation for the language in Chapter 6 of the textbook. Generate IL code for the code fragment below, in which `x` is an integer variable, `a` is an integer array, `adv` and `tst` are functions defined elsewhere in the source program, and $vtable = [\texttt{x} \mapsto v_1, \texttt{y} \mapsto v_2, \texttt{a} \mapsto v_3]$ and $ftable = [\texttt{adv} \mapsto f_{100}, \texttt{tst} \mapsto f_{200}]$:

```
x := 1;
repeat
  x := adv(x, !(a[x] > 10));
  a[x] := 5
until tst(x)
```

Note that, in the above code, `>` is a **relop** (not a **binop**), and `!` is a logical operator in the grammar of *Cond* (not a **unop**).

Use the translation functions in the textbook as closely as you can, including especially for the assignment statements. The exact *numbering* of the temporary variables and labels is not important, but you should aim to generate exactly as many of each as the formal translation specifies. You only need to show the actual IL code generated, not the details of how it was obtained by nested calls to the various translation functions.

Question 4.2 (6 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 := r_s + k r_t := M[t^{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 LABEL lab_t	slt R1, r_s, r_t beq R1, R0, lab_f lab_t:
IF r_s < r_t THEN lab_t ELSE lab_f	slt R1, r_s, r_t bne R1, R0, lab_t j lab_f
LABEL lab	lab:

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

```
x := x + 51
y := M[x^last]
IF y < z THEN lab1 ELSE lab2
LABEL lab3
IF z < y THEN lab4 ELSE lab5
LABEL lab4
```

5 Liveness analysis and register allocation

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* and *out* sets:

$in[i] = \{\mathsf{a}, \mathsf{b}, \mathsf{d}\}$

i : ?

$out[i] = \{\mathsf{a}, \mathsf{c}, \mathsf{d}\}$

$in[j] = \{\mathsf{b}, \mathsf{d}\}$

j : ?

$out[j] = \{\mathsf{a}\}$

a. This means that instruction i can be:

(A) `c := 7`

(B) `GOTO bad`

(C) `M[c] := b`

(D) `c := b * c`

(E) None of the above.

b. This means that instruction j can be:

(A) `a := d - b`

(B) `RETURN a`

(C) `a := CALL fct(d,b,e)`

(D) `IF b < d THEN now ELSE later`

(E) None of the above.

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:   x := x * 10
2:   LABEL do
3:   a := y - x
4:   y := a
5:   x := x * a
6:   IF 1 < x THEN do ELSE od
7:   LABEL od
8:   RETURN x
   }
```

a. Show **succ**, **gen** and **kill** sets for instructions 1–8.

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

c. Show the interference table:
Instr | Kill | Interferes with.

d. Draw the interference graph for a, x, and y.

e. Color the interference graph with 2 colors; show the stack, i.e., the three-column table:
Node | Neighbors | Color.

6 Fasto implementation

Question 6.1 (13 pts) This task refers to code generation in the FASTO compiler from the group project.

Suppose we want to extend the FASTO language with an operation for *reversing* an array, allowing us to write, e.g., `reverse(iota(3))` (which should evaluate to `{2,1,0}`), or `reverse("Hello")` (which should evaluate to `"olleH"`).

We extend the expression grammar with a new production:

$$Exp \;\rightarrow\; \texttt{reverse}\;(\;Exp\;)$$

We also add a corresponding abstract-syntax constructor:

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

The second argument of `Reverse` contains (after type-checking) the type of the array elements, such as `Int` and `Char` in the two examples above.

Assume that the necessary extensions to the other phases (lexing, parsing, interpretation, type-checking, optimizations) have already been taken care of, so you only have to extend the code generator. That is, give the needed match case(s) for `Reverse` in the main compilation function:

```
let rec compileExp (e      : TypedExp)
                   (vtable : VarTable)
                   (place  : reg) : Instruction list =
  match e with
    ... existing cases ...
    (* your code here *)
```

Your implementation should preferably be optimized in the following sense: if an input array `a` is shorter than two elements, reversing it makes no difference, so `reverse(a)` can simply return the original `a` in that case.

Your code should be directly usable in the FASTO codebase. Don't worry about minor syntactical details, but do aim to write actual F# code, *not* pseudocode.

Note: As a special exemption, if you qualified for the IPS exam based on the 2022 edition of the FASTO compiler, you are *allowed* (but not *required*) to generate MIPS code instead of RISC-V code, and use the corresponding code-generation infrastructure. If you choose this option, you **must** say so explicitly, as the instruction sets are otherwise very similar.

[End of exam text.]