ITX Exam in *Implementation of Programming Languages (IPS)*

DIKU, Block 4/2022

Andrzej Filinski Robert Glück

22 June 2022 (4 hours)

General instructions

This 8-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 initially budget with about 2 minutes/point; this should leave some time for coming back to 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.*

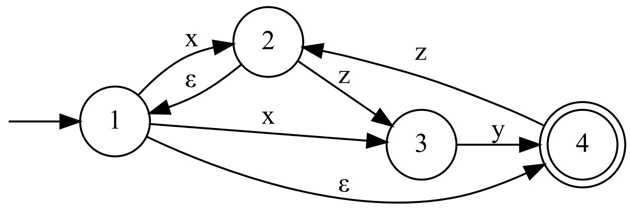
Your answers must be submitted through the ITX system, as directed. Note that editing will lock promptly at the end of the exam, so remember to keep your main document updated, to ensure that all your solutions will be graded. For (sub)questions involving figure drawing, be sure to copy your completed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

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, automata, and lexical analysis

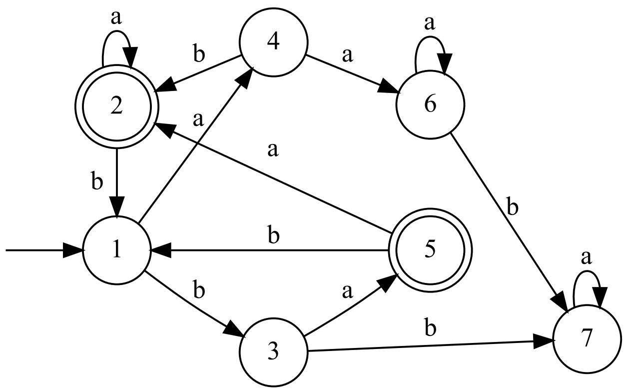
Question 1.1 (8 pts) Convert the following nondeterministic finite automaton (NFA) to a deterministic one (DFA), using the subset construction from the textbook:



NFA for conversion ($\Sigma = \{x,y,z\}$)

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 (10 pts) Minimize the following DFA, using the algorithm from the textbook:



DFA to minimize ($\Sigma = \{a,b\}$)

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

If relevant for satisfying the requirements of the basic minimization algorithm, preprocess the DFA by adding a new dead state (*not* by removing existing states), and also postprocess the minimized DFA to account for dead states. You should make it clear how the outputs from this pre- and post-processing differ from the inputs, but you don't need to draw any of the intermediate DFAs, just the final one.

Question 1.3 (5 pts) Let the alphabet Σ consist of all ASCII characters (character codes 0 through 127). Are the following languages regular? For each, *briefly* (2–5 lines) explain why or why not.

a. The set of all strings that are valid FASTO identifiers. (Recall that an identifier consists of letters, digits, and underscores, but must start with a letter, and must not be a keyword, as determined in the `keyword` function of the lexer.)

b. The set of all strings that *contain* a Fasto keyword. ("Contains" here simply means that the keyword occurs as a substring, not necessarily as a separate token. For example, the word `mint` contains the keywords `in` and `int`.)

c. The set of all strings that contain the *same* keyword (at least) twice, consecutively. For example `tintin` is not included (because the two occurrences of `in` are not consecutive), but `tintintin` is (because it contains two consecutive occurrences of the keyword `int`).

d. The set of all strings that contain the same identifier (at least) twice, consecutively. For example, `tintinx` is included (because of the identifier `tin`), but `tinti` is not.

e. The set of all strings that contain the same identifier (at least) twice, with the two copies *non-overlapping*, but *not necessarily consecutive*. For example, `mintmin` is included (because it contains, e.g., the identifier `min` twice).

2 Context-free grammars and syntax analysis

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

$$
\begin{aligned}
S &\rightarrow A \\
A &\rightarrow \texttt{a}\ A \\
A &\rightarrow \texttt{a}\ P \\
P &\rightarrow A\ \texttt{+} \\
P &\rightarrow \epsilon
\end{aligned}
$$

(The terminals are `a` and `+`. The start symbol is S.)

a) Can string `aaa+` be derived from G? If it can, show a <u>leftmost</u> derivation $S \Rightarrow \ldots$.

b) Can string `aa++` be derived from G? If it can, show a <u>leftmost</u> derivation $S \Rightarrow \ldots$.

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

d) Show that G is <u>ambiguous</u>. Hint: Show that string `aaaa+` has different syntax trees.

e) Make an <u>unambiguous</u> grammar for $L(G)$.

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

$$
\begin{aligned}
S &\rightarrow F\ \$ \\
F &\rightarrow \texttt{f}\ \texttt{[}\ H\ T\ \texttt{]} \\
H &\rightarrow \texttt{f} \\
H &\rightarrow \texttt{g} \\
T &\rightarrow \texttt{+}\ H\ T \\
T &\rightarrow \epsilon
\end{aligned}
$$

(The terminals are `f`, `g`, `+`, `[`, 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:

$First(S) =$

$First(F) =$

$First(H) =$

$First(T) =$

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

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

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

$Follow(F) =$

$Follow(H) =$

$Follow(T) =$

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

$la(S \;\rightarrow\; F\ \$) =$

$la(F \;\rightarrow\; \texttt{f [}\ H\ T\ \texttt{]}) =$

$la(H \;\rightarrow\; \texttt{f}) =$

$la(H \;\rightarrow\; \texttt{g}) =$

$la(T \;\rightarrow\; \texttt{+}\ H\ T) =$

$la(T \;\rightarrow\; \epsilon) =$

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

3 Interpretation and type checking

Question 3.1 (7 pts) This task refers to interpretation of the language in Chapter 4 of the textbook. We want to extend this language with a simple `max`-expression, which returns the largest number from among its arguments. Its syntax is given by the following additional production:

$$\mathit{Exp} \rightarrow \texttt{max} \; (\; \mathit{Exps} \;)$$

(where the nonterminal *Exps* has already been introduced as a non-empty, comma-separated list of expressions). For example the expression `let x=2 in max(5, x+8, 2*3)` should evaluate to 10 (the value of the second argument to `max`).

As usual, the argument expressions are evaluated from left to right. Unlike in a function call, however, only enough arguments are evaluated to determine the result. Specifically, if one of the argument expressions evaluates to the number $\mathit{INT\_MAX} = 2^{31} - 1$, that will necessarily be the maximum of the list, and none of the remaining expressions should be evaluated.

Extend the interpreter in Fig. 4.2 of the textbook to handle `max`-expressions, by giving the necessary new case(s) for the function $\mathit{Eval}_{\mathit{Exp}}$. If relevant, you may define additional auxiliary functions (in the same style as, e.g., $\mathit{Eval}_{\mathit{Exps}}$) to express the desired behavior. Remember to check for error conditions, as the expressions to be evaluated may not have been type-checked.

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 `acc`, *whose type is*

$$\texttt{acc} \; : \; (\; (\alpha * \beta \rightarrow \beta) \; * \; [\alpha] \; * \; \beta \;) \; \rightarrow \; \beta.$$

Its semantics is: $\texttt{acc(f, [a}_1\texttt{,...,a}_n\texttt{], b) = f(a}_n\texttt{, ...f(a}_2\texttt{,f(a}_1\texttt{,b))...)}$, i.e., it applies the function argument `f` to $\texttt{a}_1$ and `b`, then to $\texttt{a}_2$ and the previous result, and so on.

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

$\mathit{Check}_{\mathit{Exp}}(\mathit{Exp}, \mathit{vtable}, \mathit{ftable})$ = `case` *Exp* `of`	

`acc( f`
`, a_exp`
`, b_exp`
`)`
	(type-checking pseudocode)

4 Code generation

Question 4.1 (10 pts) This task is about intermediate-code generation for the source language in Chapter 6 of the textbook. Generate IL code for the following code fragment, where `x` is an integer array, and `sub` is a function with two arguments:

```
i := x[0];
while i > 0 && x[i] do
  i := sub(i, 1);
x[i] := i
```

Assume $vtable = [\mathtt{i} \mapsto v_0, \mathtt{j} \mapsto v_1, \mathtt{x} \mapsto v_2]$ and $ftable = [\mathtt{sub} \mapsto f_0]$. Follow the translation schema in the book as closely as you can, including especially for 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 (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 := `$\mathtt{r}_s$` + k` $\mathtt{r}_t$` := M[`$\mathtt{t}^{last}$`]`	`lw `$\mathtt{r}_t$`, k(`$\mathtt{r}_s$`)`
$\mathtt{r}_t$` := M[`$\mathtt{r}_s$`]`	`lw `$\mathtt{r}_t$`, 0(`$\mathtt{r}_s$`)`
$\mathtt{r}_d$` := `$\mathtt{r}_s$` + `$\mathtt{r}_t$	`add `$\mathtt{r}_d$`, `$\mathtt{r}_s$`, `$\mathtt{r}_t$
$\mathtt{r}_d$` := `$\mathtt{r}_s$` + k`	`addi `$\mathtt{r}_d$`, `$\mathtt{r}_s$`, k`
`IF `$\mathtt{r}_s$` < `$\mathtt{r}_t$` THEN `$\mathtt{lab}_t$` ELSE `$\mathtt{lab}_f$ `LABEL `$\mathtt{lab}_t$	`slt R1, `$\mathtt{r}_s$`, `$\mathtt{r}_t$ `beq R1, R0, `$\mathtt{lab}_f$ $\mathtt{lab}_t$`:`
`IF `$\mathtt{r}_s$` < `$\mathtt{r}_t$` THEN `$\mathtt{lab}_t$` ELSE `$\mathtt{lab}_f$	`slt R1, `$\mathtt{r}_s$`, `$\mathtt{r}_t$ `bne R1, R0, `$\mathtt{lab}_t$ `j `$\mathtt{lab}_f$
`LABEL lab`	`lab:`

<u>Translate the IL code below to Mips code.</u> (You may use directly, `a`, `b`, `c`, `x`, `y`, as symbolic registers, or alternatively, use $\mathtt{r}_a$, $\mathtt{r}_b$, $\mathtt{r}_c$, $\mathtt{r}_x$, $\mathtt{r}_y$, respectively.)

$\mathtt{IF\ a < c}^{last}\ \mathtt{THEN\ lab_1\ ELSE\ lab_2}$
$\mathtt{LABEL\ lab_1}$
`a := a + 471`
`b := a + b`
$\mathtt{x := M[a}^{last}\mathtt{]}$
$\mathtt{y := M[b}^{last}\mathtt{]}$

Question 4.3 (10 pts) This question is about MIPS code generation in the FASTO compiler.

Suppose we have extended FASTO with an additional arithmetic operator for *exponentiation*, with a new production *Exp* → *Exp* `**` *Exp*. Both operands must be of type `int`; the left-hand one (the *base*) can have any sign, but the right-hand one (the *exponent*) must be non-negative. The expression then returns the base raised to the power of the exponent. For example, the expression `(1+1) ** 3` should evaluate to 8, while `2 ** (0-3)` should give an error. If the exponent is 0, the result is always 1, regardless of the base.

Assume that the abstract syntax has been extended with an `Expt` constructor (analogous to `Plus`, `Times`, etc.), and that lexing, parsing, interpretation, type checking, and optimizations for `Expt` have already been taken care of, so that only code generation remains. Fill in the following case of the main compilation function:

```
let rec compileExp  (e      : TypedExp)
                    (vtable : VarTable)
                    (place  : Mips.reg)
                  : Mips.Instruction list =
  match e with
  ...
  | Expt (e1, e2, pos) -> (* Your code here *)
```

To get full points (but partial solutions are better than nothing!), the evaluation of Exp_1 `**` Exp_2 should be short-circuiting in the following sense:

- If Exp_2 evaluates to a negative number, the program should stop with an error, and not even attempt to evaluate Exp_1.
- If Exp_2 evaluates to 0, the result should be 1, and again, Exp_1 should not be evaluated.
- If Exp_2 evaluates to a positive number, Exp_1 should be evaluated (only once!), and the result of the exponentiation calculated by straightforward repeated multiplications. (As usual in FASTO arithmetic, you can ignore the possibility of overflow.)

Recall that, to abort the program with an error message, the code should jump to the label `"_RuntimeError_"`, with MIPS register `$5` containing the source-line number, and `$6` containing a pointer to a (null-terminated) error-message string. You may assume that the label `"_Msg_IllegalExpt_"` points to a suitable such message.

Don't worry about minor syntactic issues, but your code should be as close to actual F# code, suitable for inclusion in the compiler, as you can make it.

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*/*out* sets:

$$in[i] = \{\mathtt{b}, \mathtt{c}, \mathtt{e}\} \qquad\qquad in[j] = \{\mathtt{b}, \mathtt{c}, \mathtt{e}\}$$

$$i : \boxed{\quad ? \quad} \qquad\qquad j : \boxed{\quad ? \quad}$$

$$out[i] = \{\mathtt{c}, \mathtt{d}, \mathtt{e}\} \qquad\qquad out[j] = \{\mathtt{b}, \mathtt{c}\}$$

a) This means that instruction i can be:

(A) `M[d] := b`

(B) `b := c + e`

(C) `d := b`

(D) `GOTO cde`

(E) None of the above.

b) This means that instruction j can be:

(A) `M[b] := e`

(B) `RETURN a`

(C) `IF b = c THEN do ELSE end`

(D) `c := CALL f(e,c)`

(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:   LABEL loop
 2:   IF y > 0 THEN do ELSE od
 3:   LABEL do
 4:   s := y - x
 5:   t := y * y
 6:   x := s - x
 7:   y := t / y
 8:   GOTO loop
 9:   LABEL od
10:   RETURN x
    }
```

a. Show **succ**, **gen** and **kill** sets for instructions 1–10.

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

c. Show the <u>interference table</u>: Instr | Kill | Interferes with.

d. Draw the <u>interference graph</u> for `s`, `t`, `x`, and `y`.

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

[End of exam text.]