



Exam notes

Adit (hjpg708)

IPS 2023

17. juni 2023

Indhold

Exam22 - Solutions	3
1.1	3
1.2	3
1.3	5
2.1	5
2.2	6
3.1	8
3.2	8
4.1	9
4.2	9
4.3	9
5.1	10
Exam22 - Exam text	12

Exam22 - Solutions

1.1

$start = ec(\{1\}) = \{1, 4\} =: s0$ [accepting]

$move(s0, x) = ec(\{2, 3\}) = \{2, 3, 1, 4\} = \{1, 2, 3, 4\} =: s1$ [accepting]

$move(s0, y) = ec(\{\}) = \text{undefined}$

$move(s0, z) = ec(\{2\}) = \{2, 1, 4\} = \{1, 2, 4\} =: s2$ [accepting]

$move(s1, x) = ec(\{2, 3\}) = \{2, 3, 1, 4\} = \{1, 2, 3, 4\} =: s1$

$move(s1, y) = ec(\{4\}) = \{4\} =: s3$ [accepting]

$move(s1, z) = ec(\{2, 3\}) = \{2, 1, 4\} = \{1, 2, 3, 4\} =: s1$

$move(s2, x) = ec(\{2, 3\}) = \{2, 3, 1, 4\} = \{1, 2, 3, 4\} =: s1$

$move(s2, y) = ec(\{\}) = \text{undefined}$

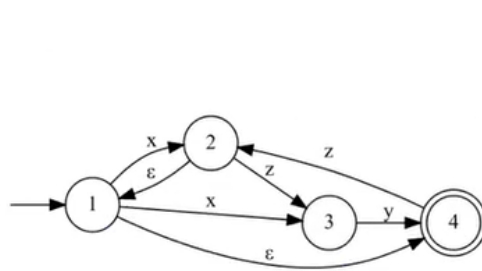
$move(s2, z) = ec(\{3, 2\}) = \{3, 2, 1, 4\} = \{1, 2, 3, 4\} =: s1$

$move(s3, x) = ec(\{\}) = \text{undefined}$

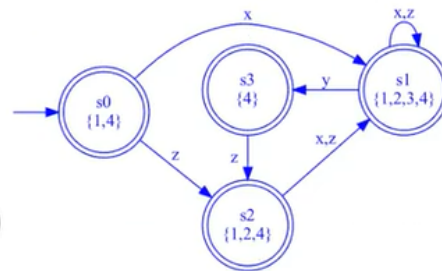
$move(s3, y) = ec(\{\}) = \text{undefined}$

$move(s3, z) = ec(\{2\}) = \{2, 1, 4\} = \{1, 2, 4\} =: s2$

(Note that all the DFA states are accepting, but the transition function is not total, so it's still possible for the DFA to reject some input strings.)

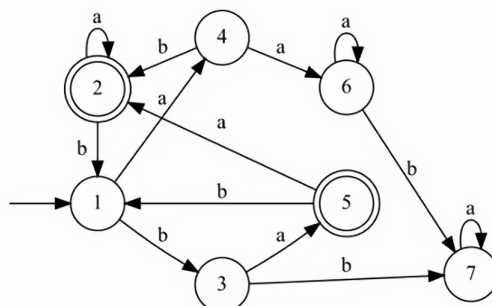


NFA for conversion ($\Sigma = \{x,y,z\}$)

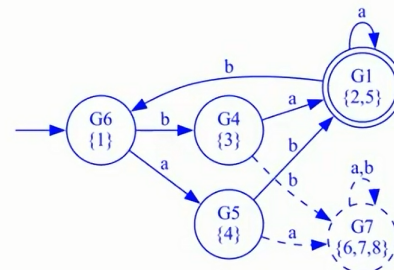


DFA obtained from NFA

1.2



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



Minimized DFA (removed dead state shown dashed)

Preprocessing: add new, rejecting state 8, with b-transition from state 7, and a,b-transition to itself. The main minimization then proceeds as follows:

$G1 := \{2, 5\}$ (accepting)

$G2 := \{1, 3, 4, 6, 7, 8\}$ (rejecting)

<i>G1</i>	<i>a</i>	<i>b</i>		
1	<i>G1</i>	<i>G2</i>	— >	<i>G3</i>
3	<i>G2</i>	<i>G2</i>	— >	<i>G4</i>
4	<i>G2</i>	<i>G1</i>	— >	<i>G5</i>
6	<i>G2</i>	<i>G2</i>	— >	<i>G3</i>
7	<i>G2</i>	<i>G2</i>	— >	<i>G3</i>
8	<i>G2</i>	<i>G2</i>	— >	<i>G3</i>

Split *G2* into $G3 = \{1, 6, 7, 8\}$, $G4 = \{3\}$, and $G5 = \{4\}$

<i>G3</i>	<i>a</i>	<i>b</i>		
1	<i>G5</i>	<i>G4</i>	— >	<i>G6</i>
6	<i>G3</i>	<i>G3</i>	— >	<i>G7</i>
7	<i>G3</i>	<i>G3</i>	— >	<i>G7</i>
8	<i>G3</i>	<i>G3</i>	— >	<i>G7</i>

Split *G3* into $G6 = \{1\}$ and $G7 = \{6, 7, 8\}$

<i>G7</i>	<i>a</i>	<i>b</i>		
6	<i>G7</i>	<i>G7</i>	— >	<i>OK</i>
7	<i>G7</i>	<i>G7</i>	— >	<i>OK</i>
8	<i>G7</i>	<i>G7</i>	— >	<i>OK</i>

<i>G1</i>	<i>a</i>	<i>b</i>		
2	<i>G1</i>	<i>G6</i>	— >	<i>OK</i>
5	<i>G1</i>	<i>G6</i>	— >	<i>OK</i>

Singleton groups:

<i>G4</i>	<i>a</i>	<i>b</i>		
3	<i>G1</i>	<i>G7</i>		

<i>G5</i>	<i>a</i>	<i>b</i>		
4	<i>G7</i>	<i>G1</i>		

<i>G6</i>	<i>a</i>	<i>b</i>		
1	<i>G5</i>	<i>G4</i>		

Postprocessing: *G7* contains the added dead state 8, so it (and transitions to it) are removed as the equivalence class of all the dead states. The final transition table becomes:

<i>G7</i>	<i>a</i>	<i>b</i>		
<i>G1</i>	<i>G1</i>	<i>G6</i>		<i>ACCEPT</i>
<i>G4</i>	<i>G1</i>	—		<i>REJECT</i>
<i>G5</i>	—	<i>G1</i>		<i>REJECT</i>
<i>G6</i>	<i>G5</i>	<i>G4</i>		<i>REJECT, START</i>

(The figure is drawn above, next to the original DFA.)

1.3

a. The set of all strings that are valid Fasto identifiers is regular because we can construct a regular expression or a finite automaton to recognize them.

Regular. It can be characterized as the set difference between the languages of REs "[a-zA-Z][a-zA-Z0-9_]*" and "(if|then|...)" (enumerating all the FASTO keywords)

b. The set of all strings that contain a Fasto keyword is regular because we can construct a regular expression or a finite automaton to recognize any occurrence of a keyword as a substring.

Regular. It is denoted by the RE ".(if|then|...).*", where . = [\NUL - \DEL] matches all ASCII characters.

c. The set of all strings that contain the same keyword (at least) twice, consecutively is regular because we can construct a regular expression or a finite automaton to recognize any occurrence of a keyword followed immediately by the same keyword.

Regular. It is denoted by ".(iff|thenthen|...).*" (doubling all the keywords in the enumeration).

d. The set of all strings that contain the same identifier (at least) twice, consecutively is not regular because it requires unbounded memory, which cannot be represented by a finite automaton.

Not regular. For example, words of the form $\{a0^nba0^mb|n, m \geq 0\}$ (which can never contain FASTO keywords) are in the language if and only if $n = m$, which cannot be decided by any finite automaton.

e. The set of all strings that contain the same identifier (at least) twice, with the two copies non-overlapping, but not necessarily consecutive is regular because we can construct a regular expression or a finite automaton to recognize any occurrence of an identifier followed by the same identifier at a later position in the string.

Regular. Any words containing the same letter twice is included (since all single letters are valid identifiers), and conversely, any word containing the same identifier twice must also contain at least one letter twice. And the language of words with at least one duplicated letter is evidently denoted by RE ".*a.*a.* | .*b.*b.* | ..." (enumerating all letters a-z and A-Z).

2.1

a)

Yes, there is a derivation for $aaa+$. Leftmost derivation:

$S \Rightarrow A \Rightarrow aP \Rightarrow aA+ \Rightarrow aaA+ \Rightarrow aaaP+ \Rightarrow aaa+$

Another leftmost derivation (G is ambiguous):

$S \Rightarrow A \Rightarrow aA \Rightarrow aaP \Rightarrow aaA+ \Rightarrow aaaP+ \Rightarrow aaa+$

b)

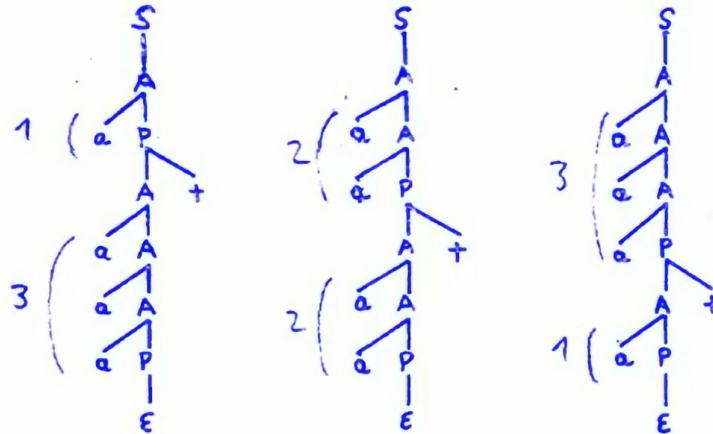
No, there is no derivation for $aa++$.

c)

No, $L(G)$ is not regular. $L(G) = \{a^m +^n | m > n \geq 0\}$ requires a counter.

d)

Strings $aaa+$, $aaaa+$, and others, have several different syntax trees. Thus, G is ambiguous. It is sufficient to show two of the three syntax trees of $aaaa+$:



e)

Considering that $L(G) = \{a^m + a^n \mid m > n \geq 0\} = \{a^n a^m + a^n \mid m \geq 1, n \geq 0\}$:

$S \rightarrow A$

$A \rightarrow aA$

$A \rightarrow aP$

$P \rightarrow aP+$

$P \rightarrow \epsilon$

or

$S \rightarrow P$

$P \rightarrow aP+$

$P \rightarrow A$,

$A \rightarrow aA$

$A \rightarrow a$

and other variants.

2.2

a)

Computing nullable(N) [abbreviated as null(N)] for all N:

```

null(S) = false           [always, because of the $]
null(F) = null(f [ H T ]) = false
null(H) = null(f) || null(g) = false || false = false
null(T) = null(+ H T) || null(eps) = false || true = true

```

b)

Computing first(N) for all N:

```

first(S = first(F $)) = first(F)           [since !null(F)]
first(F) = first(f [ H T ]) = {f}
first(H) = first(f) U first(g) = {f} U {g} = {f, g}
first(T) = first(+ H T) U first(eps) = {+} U {} = {+}

```

Simplifying further:

$\text{first}(S) = \text{first}(F) = \{f\}$

c)

Computing follow(N) for all N:

Generating constraints:

- (1) $S \rightarrow F \$$
1. $\{ \$ \} \leq \text{follow}(F)$ [because $\text{first}(\$) = \{ \$ \}$]
- (2) $F \leftarrow f [H T]$
2. $\{ +, \} \leq \text{follow}(H)$ [because $\text{first}(T \] = \text{first}(T) \cup \text{first}(\] = \{ +, \} \]$
3. $\{ \} \leq \text{follow}(T)$ [because $\text{first}\{ \} = \{ \} \]$
- (3) $H \rightarrow f$
- (4) $H \rightarrow g$
- (5) $T \rightarrow + H T$
4. $\{ + \} \leq \text{follow}(H)$ [because $\text{first}(T) = \{ + \}$]
5. $\text{follow}(T) \leq \text{follow}(H)$ [because $\text{null}(T) \]$
 $\text{follow}(T) \leq \text{follow}(T)$ [because T last in production, trivial]
- (6) $T \rightarrow \text{eps}$

d)

Finding least solution:

Seed rules $\{a_1, \dots, a_n\} \leq \text{follow}(N)$: 1., 2., 3., 4.

Propagation rules $\text{follow}(N_1) \leq \text{follow}(N_2)$: 5.

Set	Seed	Prop 1	Final
follow(F)	\$ [1.]		\$
follow(H)	+ [2.]] [2.]		+]
follow(T)] [3.]]

e)

$\text{la}(S \rightarrow F \$) = \text{first}(F) = \{f\}$

$\text{la}(F \rightarrow f [H T]) = \text{first}(f [H T]) = \{f\}$

$\text{la}(H \rightarrow f) = \text{first}(f) = \{f\}$

$\text{la}(H \rightarrow g) = \text{first}(g) = \{g\}$

[ok: disjoint!]

$\text{la}(T \rightarrow + H T) = \text{first}(+ H T) = \{+\}$

$\text{la}(T \rightarrow \text{eps}) = \text{first}(\text{eps}) \cup \text{follow}(T) = \{ \} \cup \{ \} = \{ \}$

[ok: disjoint!]

e)

Recursive descent parser LL(1). For nonterminals S and T:

```
function parseS() =  
  if input = 'f' then                                (* la(S -> F $) = {f} *)  
    parseF(); match('$')                               (* parse F $ *)  
  else reportError()
```

```

function parseT() =
    if input = '+' then                (* la(T -> + H T) = {+} *)
        match('+'); parseH(); parseT() (* parse + H T *)
    else if input = ']' then           (* la(T -> eps) = {} *)
        (* do nothing, just return *)
    else reportError()

(* Not asked for: nonterminals F and H *)

function parseF() =
    if input = 'f' then
        match('f'); match('['); parseH(); parseT(); match(']') (* parse f [ H T ] *)
    else reportError()

function parseH() =
    if input = 'f' then                (* la(H -> f) = first(f) = {f} *)
        match('f')                    (* parse f *)
    else if input = 'g' then           (* la(H -> g) = first(f) = {g} *)
        match('g')                    (* parse g *)
    else reportError()

```

3.1

```

EvalExp(Exp, vtable, ftable) = case Exp of
...
-----
max(Exp1)      | v1 = EvalExp(Exp1, vtable, ftable)
                | if v1 is an integer then v1 else error()
-----
max(Exp1,Exps) | v1 = EvalExp(Exp1, vtable, ftable)
                | if v1 is an integer then
                |   if v1 = INT_MAX then v1
                |   else
                |     m =EvalExp(max(Exps), vtable, ftable)
                |   else error()
-----

```

3.2

```

CheckExp(Exp, vatble, ftable) = case Exp of
acc(f, a_exp, b_exp) =>
    ta = CheckExp(a_exp, vtable, ftable)
    tb = CheckExp(b_exp, vtable, ftable)
    te = case ta of
        Array(t => t)
        | other => error("array expected")
    tf = lookup(ftable, getname(f))
    case tf of
        unbound => error("function undefined")
    | ((t1 * t2) -> t3) =>
        if (t1 = te) then
            if (t2 = tb) then
                if (t3 = tb) then tb
                else error("type mismatch: fct result and b_exp")
            else error("type mismatch: 2nd fct arg and b_exp")

```



```

        else error("type mismatch: 1st fct arg and a_exp")
    | other => error("binary function expected")

```

4.1

```

t1 := 0
t1 := t1 * 4
t1 := t1 + v2
t0 := M[t1]
v0 := t0
LABEL lab0 // start of loop
t2 := v0
t3 := 0
IF t2 > t3 THEN lab3 ELSE lab2
LABEL lab3 // 2nd conjunct
t5 := v0
t5 := t5 * 4
t5 := t5 + v2
t4 := M[t5]
IF t4 NOT = 0 THEN lab1 ELSE lab2
LABEL lab1 //start of loop body
t7 := v0
t8 := 1
t6 := CALL f0(t7, t8)
v0 := t6
GOTO lab0
LABEL lab2 // after loop
t9 := v0
t9 := t9 * 4
t9 := t9 + v2
t10 := v0;
M[t9] := t10

```

4.2

```

slt R1, a, c
beq R1, R0, lab2
lab1:
addi a, a, 471
add b, a, b
lw x, 0(a)
lw y, 0(b)

```

4.3

```

| Expt (e1, e2, pos) ->
    let lsafe = newLab "expt_safe"
    let lloop = newLab "expt_loop"
    let ldone = newLab "expt_done"
    let t1 = newReg "expt_L"
    let t2 = newReg "expt_R"

    let code1 = compileExp e1 vtable t1
    let code2 = compileExp e2 vtable t2

    [Original MIPS code:]

```

```

code2 @
[ Mips.BGEZ (t2, lsafe);
  Mips.LI (RN5, fst pos);
  Mips.LA (RN6, "_Msg_IllegalExpt_");
  Mips.J "_RuntimeError_";
  Mips.LABEL lsafe;
  Mips.LI (place, 1);
  Mips.BEQ (t2, RZ, ldone) ]
@ code1 @
[ Mips.LABEL lloop;
  Mips.MUL (place, place, t1);
  Mips.ADDI (t2, t2, -1);
  Mips.BNE (t2, RZ, lloop);
  Mips.LABEL ldone ]

```

[Equivalent RISC-V code:]

```

code2 @
[ BGE (t2, Rzero, lsafe);
  LI (Ra0, fst pos);
  LA (Ra1, "m.IllegalExpt");
  J "p.RuntimeError";
  LABEL lsafe;
  LI (place, 1);
  BEQ (t2, Rzero, ldone) ]
@ code1 @
[ LABEL lloop;
  MUL (place, place, t1);
  ADDI (t2, t2, -1);
  BNE (t2, Rzero, lloop);
  LABEL ldone ]

```

5.1

- a From dataflow equation $in[i] = gen[i] \cup (out[i] \setminus kill[i])$ we obtain equation $\{b, c, e\} = gen[i] \cup (c, d, e \setminus kill[i])$, which is only satisfied by

$$(C) \quad gen = \{b\}, \quad kill = \{d\} : \{b, c, e\} = \{b\} \cup (\{c, d, e\} \setminus \{d\}).$$

- b From dataflow equation $in[j] = gen[j] \cup (out[j] \setminus kill[i])$ we obtain equation $\{b, c, e\} = gen[j] \cup (b, c \setminus kill[j])$, which is only satisfied by

$$(A) \quad gen = \{b, e\}, \quad kill = \emptyset : \{b, c, e\} = \{b, e\} \cup (\{b, c\} \setminus \emptyset).$$

$$(D) \quad gen = \{c, e\}, \quad kill = \{c\} : \{b, c, e\} = \{c, e\} \cup (\{b, c\} \setminus \{c\}).$$

5.2

a)

i	succ[i]	gen[i]	kill[i]
1	2		
2	3,9	y	
3	4		
4	5	y,x	s
5	6	y	t

6	7	s,x	x
7	8	t,y	y
8	9		
9	10		
10	-	x	

b)

	iteration 1		iteration 2	
i	out[i]	in[i]	out[i]	in[i]
1	y,x	y,x	y,x	y,x
2	y,x	y,x	y,x	y,x
3	y,x	y,x	y,x	y,x
4	y,s,x	y,x	y,s,x	y,x
5	t,y,s,x	y,s,x	t,y,s,x	y,s,x
6	t,y	t,y,s,x	t,y,x	t,y,s,x
7		t,y	y,x	t,y,x
8			y,x	y,x
9	x	x	x	x
10		x		x

c)

i	out[i]	kill[i]	interferes with
1	y,x		
2	y,x		
3	y,x		
4	y,s,x	s	y,x
5	t,y,s,x	t	y,s,x
6	t,y,x	x	t,y,s
7	y,x	y	t,x
8	y,x		
9	x		
10			

d)

```

s ----- t
|\       /|
| \   / |
| /   \ |
|/     \|
x ----- y

```

e)

Node	Neighbors	Color
t	-	1
x	t	2
y	t,x	3
s	t,x,y	spill

Exam22 - Exam text

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*.

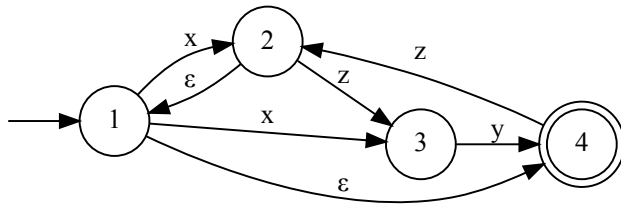
Your answers must be submitted through the ITX system, as directed. Note that editing will lock promptly at the end of the exam, so remember to keep your main document updated, to ensure that all your solutions will be graded. For (sub)questions involving figure drawing, be sure to copy your completed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

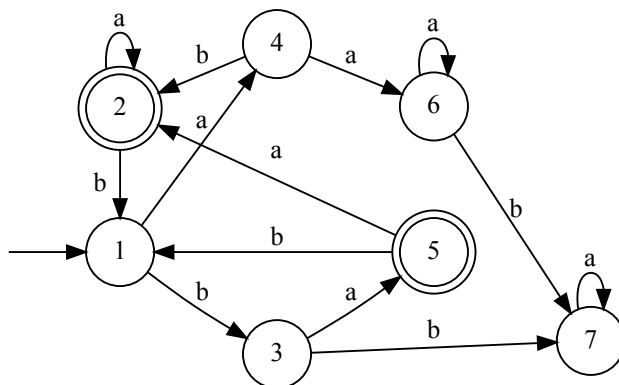
Question 1.1 (8 pts) Convert the following nondeterministic finite automaton (NFA) to a deterministic one (DFA), using the subset construction from the textbook:



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.

- 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 :

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

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

- a) Can string `aaa+` be derived from G ? If it can, show a leftmost derivation $S \Rightarrow \dots$.
- b) Can string `aa++` be derived from G ? If it can, show a leftmost derivation $S \Rightarrow \dots$.
- c) Is $L(G)$ regular? Justify your answer (max. 3 lines).
- d) Show that G is ambiguous. Hint: Show that string `aaaa+` has different syntax trees.
- e) Make an unambiguous grammar for $L(G)$.

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

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

(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:

$$\textit{First}(S) =$$

$$\textit{First}(F) =$$

$$\textit{First}(H) =$$

$$\textit{First}(T) =$$

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

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

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

$$\textit{Follow}(F) =$$

$$\textit{Follow}(H) =$$

$$\textit{Follow}(T) =$$

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

$$\textit{la}(S \rightarrow F \$) =$$

$$\textit{la}(F \rightarrow \mathbf{f} \text{ [} H \text{ T }]) =$$

$$\textit{la}(H \rightarrow \mathbf{f}) =$$

$$\textit{la}(H \rightarrow \mathbf{g}) =$$

$$\textit{la}(T \rightarrow + H \text{ T}) =$$

$$\textit{la}(T \rightarrow \epsilon) =$$

- 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:

$$Exp \rightarrow \mathbf{max} (Exps)$$

(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 $INT_MAX = 2^{31} - 1$, 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 $Eval_{Exp}$. If relevant, you may define additional auxiliary functions (in the same style as, e.g., $Eval_{Exps}$) 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*

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

Its semantics is: $\mathbf{acc}(f, [a_1, \dots, a_n], b) = f(a_n, \dots f(a_2, f(a_1, b)) \dots)$, i.e., it applies the function argument **f** to **a₁** and **b**, then to **a₂** 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 $\mathbf{acc}(f, \mathbf{a_exp}, \mathbf{b_exp})$ 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 $Check_{Exp}$ is the type of the $\mathbf{acc}(f, \mathbf{a_exp}, \mathbf{b_exp})$ expression. Give self-explanatory error messages. Present the type-checking pseudocode in a way compatible to the textbook/lecture slides.

$Check_{Exp}(Exp, vtable, ftable) = \text{case } Exp \text{ of}$	
$\mathbf{acc}(f$, 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 $vtable = [i \mapsto v_0, j \mapsto v_1, x \mapsto v_2]$ and $f table = [sub \mapsto f_0]$. 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*:

$t := r_s + k$ $r_t := M[t^{last}]$	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, a, b, c, x, y , as symbolic registers, or alternatively, use r_a, r_b, r_c, r_x, r_y , respectively.)

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

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 $Exp \rightarrow Exp ** Exp$. 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:

$$\begin{array}{l} in[i] = \{b, c, e\} \\ i : \boxed{\quad ? \quad} \\ out[i] = \{c, d, e\} \end{array}$$

$$\begin{array}{l} in[j] = \{b, c, e\} \\ j : \boxed{\quad ? \quad} \\ out[j] = \{b, c\} \end{array}$$

- | | |
|---|---|
| <p>a) This means that instruction i can be:</p> <p>(A) $M[d] := b$</p> <p>(B) $b := c + e$</p> <p>(C) $d := b$</p> <p>(D) <code>GOTO cde</code></p> <p>(E) None of the above.</p> | <p>b) This means that instruction j can be:</p> <p>(A) $M[b] := e$</p> <p>(B) <code>RETURN a</code></p> <p>(C) <code>IF b = c THEN do ELSE end</code></p> <p>(D) $c := \text{CALL } f(e, c)$</p> <p>(E) None of the above.</p> |
|---|---|

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 interference table:
Instr | Kill | Interferes with.
- d. Draw the interference graph for **s**, **t**, **x**, and **y**.
- e. Color the interference graph with 3 colors; show the stack, i.e., the three-column table:
Node | Neighbors | Color.

[End of exam text.]