Technical University of Denmark

Written examination, May 23, 2017

Example solution.

Course name: Computer Science Modelling

Course number: 02141

Aids allowed: All written aids are permitted

Exam duration: 4 hours

Weighting: 7-step scale

Old exam sets are not indicative of new exam sets.

Proofs

Exercise 1a:

w	ϵ	bbc	aac	$abbc$	$abbbc$	$aabbcc$
L_1	*yes*	*yes*	*yes*	*yes*	*no*	*yes*
L_2	*no*	*yes*	*no*	*yes*	*yes*	*yes*
L_3	*no*	*yes*	*no*	*yes*	*yes*	*yes*
L_4	*no*	*yes*	*yes*	*yes*	*no*	*yes*

Exercise 1b: The subset construction gives the following:

	a	b	c	
$\{S\}$	$\{A\}$	$\{B\}$	$\emptyset$	1
$\{A\}$	$\{A\}$	$\{B, D\}$	$\emptyset$	2
$\{B\}$	$\emptyset$	$\{B, C, D\}$	$\emptyset$	5
$\{B, D\}$	$\emptyset$	$\{B, C, D\}$	$\emptyset$	3
$\{B, C, D\}$	$\emptyset$	$\{B, C, D\}$	$\{C, H\}$	6
$\{C, H\}$	$\emptyset$	$\emptyset$	$\{C, H\}$	7
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	4

The initial state is $\{S\}$ and the final state is $\{C, H\}$.

This is indeed the DFA for L_2; the last column shows the translation of state names.

Exercise 1c:

$\subseteq$	L_1	L_2	L_3	L_4
L_1	*yes*	**no**[2]	**no**[2]	**no**[2]
L_2	**no**[4]	*yes*	**yes**[1]	**no**[4]
L_3	**no**[4]	**yes**[1]	*yes*	**no**[4]
L_4	**yes**[3]	**no**[5]	**no**[5]	*yes*

1: The DFA constructed in Exercise 1b for L_3 equals the one given for L_2 so the two languages are equal. And indeed they equals the regular expression $a^*bbb^*cc^*$

2: ϵ is in L_1 but not in any of the other languages.

3: Any string in L_4 is also in L_1 as the b's come in pairs in both languages and they start with a sequence of a's and end with a sequence of c's.

4: $abbbc$ is in L_2 (and L_3) but not in L_1 – nor in L_4.

5: acc is in L_4 but not in L_2 (and L_3).

Exercise 1d: Assume that L_4 is regular; then the Pumping Lemma gives that there exists a number n such that for all strings $w \in L_4$, if $|w| \geq n$ then there exists strings x, y and z such that $w = xyz$, $|xy| \leq n$ and $y \neq \epsilon$ and $xy^kz \in L_4$ for all $k \geq 0$. Consider $w = a^nb^{2(n+1)}c \in L_4$; then $w = xyz$ as suggested by the Pumping Lemma. From $0 < |xy| \leq n$ we see that xy only can contain a's. Consider now $xz = a^mb^{2(n+1)}c$ obtained by removing the substring y; here

$m = n - |y|$ and since $|y| \geq 1$ we have $m \leq n - 1$. We also have $xz \in L_4$ so $n + 1 \leq m + 1$ must be the case. But then $n \leq m \leq n - 1$ and we have a contradiction. Thus L_4 cannot be regular.

Exercise 2 If L is regular it is defined by a regular expression E; we define a regular expression for $\mathsf{pre}(L)$ as follows:

$$\begin{array}{rcl}
\mathsf{pre}(\emptyset) & = & \emptyset \\
\mathsf{pre}(\epsilon) & = & \epsilon \\
\mathsf{pre}(a) & = & \epsilon + a \\
\mathsf{pre}(E_1 + E_2) & = & \mathsf{pre}(E_1) + \mathsf{pre}(E_2) \\
\mathsf{pre}(E_1\, E_2) & = & \mathsf{pre}(E_1) + E_1\, \mathsf{pre}(E_2) \\
\mathsf{pre}(E^*) & = & E^*\, \mathsf{pre}(E)
\end{array}$$

An alternative solution is as follows: If L is regular then it is accepted by a DFA $M = (Q, \Sigma, \delta, q_0, F)$. Construct the NFA $N = (Q, \Sigma, \delta, q_0, F')$ where F' is the set of states from which a final state can be reached. More precisely define the set of states $reach_i \subseteq Q$ that can reach a state in F in at most i steps as follows: $reach_0 = F$ and $reach_{i+1} = \{q \in Q \mid \exists a \in \Sigma : \delta(q, a) \in reach_i \vee q \in reach_i\}$.

Exercise 3

(a) AA $\langle q_\triangleright, \sigma\rangle \xRightarrow{x \leq y} \langle q_2, \sigma\rangle \xRightarrow{z := y} \langle q_\blacktriangleleft, \begin{bmatrix} x \mapsto 5 \\ y \mapsto 9 \\ z \mapsto 9 \end{bmatrix}\rangle$

BB $\langle q_\triangleright, \sigma\rangle \xRightarrow{z := x} \langle q_1, \begin{bmatrix} x \mapsto 5 \\ y \mapsto 9 \\ z \mapsto 5 \end{bmatrix}\rangle \xRightarrow{z \leq y} \langle q_2, \begin{bmatrix} x \mapsto 5 \\ y \mapsto 9 \\ z \mapsto 5 \end{bmatrix}\rangle \xRightarrow{z := y} \langle q_1, \begin{bmatrix} x \mapsto 5 \\ y \mapsto 9 \\ z \mapsto 9 \end{bmatrix}\rangle$

$\xRightarrow{z \geq y} \langle q_\blacktriangleleft, \begin{bmatrix} x \mapsto 5 \\ y \mapsto 9 \\ z \mapsto 9 \end{bmatrix}\rangle$

(b) AA is not deterministic because when $x = y$ one can go from $q_\triangleright$ to q_1 as well as q_2 so the execution sequences differ

BB is not deterministic because when $x = y$ one can go from q_1 to $q_\blacktriangleleft$ as well as q_2 so the execution sequences differ

(c) AA cannot run forever

BB can run forever in case $x = y$

(d) AA is given by if $x \geq y \rightarrow z := x$ [] $x \leq y \rightarrow z := y$ fi

BB has no program because $z \geq y$ differs from done$(z \leq y)$, (in particular when $z = y$)

1

(e) $\{-+0,\ 0+0,\ ++0\}$ and is written $\begin{matrix}-+0\\0+0\\++0\end{matrix}$

(f)

	q_0	q_1	q_2	$q_{\blacktriangleleft}$
AA	$-+0$ $0+0$ $++0$	$++0$	$-+0$ $0+0$ $++0$	$+++$ $-++$ $0++$
BB	$-+0$ $0+0$ $++0$	$-+-$ $0+0$ $+++$ $-++$ $0++$	$-+-$ $0+0$ $+++$ $-++$ $0++$	$+++$ $-++$ $0++$

(g) AA it does: all triples have + in the end

BB — " ——————

2

(h) AA $4 \cdot 3^3 = 108$ as there are 4 program points and three variables taking three values each

BB similarly

(i) AA we must check in all states of the form $(q_\triangleright, xyz)$
it is vacuously true except in the state $(q_\triangleright, 321)$
the path $(q_\triangleright, 321) \longrightarrow (q_1, 321) \longrightarrow (q_\blacktriangleleft, 323)$
establishes that $\#_{321} \Rightarrow \exists\Diamond(\#_{323} \wedge \blacktriangleleft)$ hold in all initial states

BB much as before
the path $(q_\triangleright, 321) \longrightarrow (q_1, 323) \longrightarrow (q_\blacktriangleleft, 323)$
establishes the result

(j) AA the answer is still true
because there is no other path than $(q_\triangleright, 321) \rightarrow (q_1, 321) \rightarrow (q_\blacktriangleleft, 323)$

BB the answer is still true
because there is no other path than $(q_\triangleright, 321) \rightarrow (q_1, 323) \rightarrow (q_\blacktriangleleft, 323)$

Exercise 4

(a) $\langle x := a, \sigma \rangle \to \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$

$\langle \text{skip}, \sigma \rangle \to \sigma$

$$\frac{\langle c_1, \sigma \rangle \to \sigma' \qquad \langle c_2, \sigma' \rangle \to \sigma''}{\langle c_1 ; c_2, \sigma \rangle \to \sigma''}$$

$$\frac{\langle GC, \sigma \rangle \to \sigma'}{\langle \text{if } GC \text{ fi}, \sigma \rangle \to \sigma'}$$

$$\frac{\langle GC, \sigma \rangle \to \sigma' \qquad \langle \text{do } GC \text{ od}, \sigma' \rangle \to \sigma''}{\langle \text{do } GC \text{ od}, \sigma \rangle \to \sigma''}$$

$$\frac{\langle c, \sigma \rangle \to \sigma'}{\langle b \to c, \sigma \rangle \to \sigma'} \quad \text{if } \mathcal{B}[\![b]\!]\sigma = tt$$

$$\frac{\langle GC_1, \sigma \rangle \to \sigma'}{\langle GC_1 \,[\!]\, GC_2, \sigma \rangle \to \sigma'}$$

$$\frac{\langle GC_2, \sigma \rangle \to \sigma'}{\langle GC_1 \,[\!]\, GC_2, \sigma \rangle \to \sigma'}$$

$\langle \text{do } GC \text{ od}, \sigma \rangle \to \sigma \quad \text{if } \mathcal{B}[\![\text{done}(GC)]\!]\sigma = tt$

11 (b) All complete execution sequences $\langle q_{\triangleright}, \sigma \rangle \Rightarrow^{*} \langle q_{\triangleleft}, \sigma' \rangle$
give rise to some $\langle C, \sigma \rangle \to \sigma'$ and vice versa

There may be infinite execution sequences $\langle q_{\triangleright}, \sigma \rangle \Rightarrow^{n} \langle q_n, \sigma_n \rangle \Rightarrow \ldots$
in which case both $\langle C, \sigma \rangle \not\to$ and $\langle C, \sigma \rangle \to \sigma'$ for some σ'

Looping is masked in NS but not in FM (or SOS)

5

Exercises on Context-free Languages

Exercise 5 (30%)

Go is a programming language created at Google to easily build concurrent programs. This exercise is based on a small subset of the language that we call here tinyGo, whose syntax is given by the following grammar:

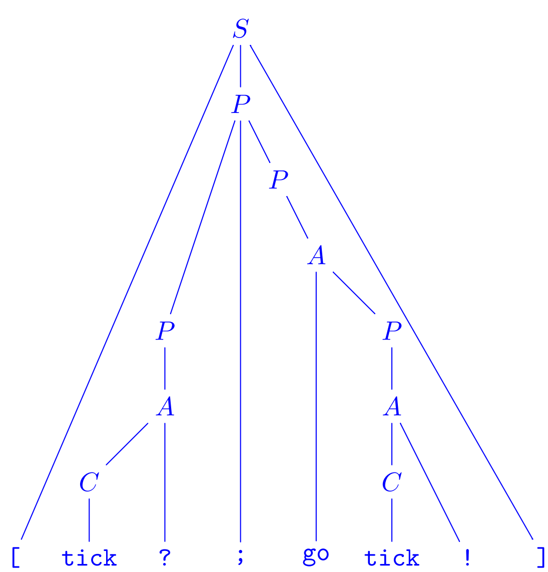
S	$\rightarrow$	`[`P`]`	(scoped program)
P	$\rightarrow$	A	(actions)
	$\mid$	P `;` P	(sequential composition of programs)
	$\mid$	P `+` P	(non-deterministic choice between programs)
A	$\rightarrow$	skip	(do nothing)
	$\mid$	C`!`	(output on a channel)
	$\mid$	C`?`	(input on a channel)
	$\mid$	go P	(spawn a parallel program)
	$\mid$	S	(scoped program)
C	$\rightarrow$	`tick` $\mid$ `tack`	(finite set of channels)

where the set of non-terminal symbols (or variables) is $\{S, P, A, C\}$, the set of terminal symbols (or tokens) is {`[`, `]`, `;`, `+`, `skip`, `!`, `?`, `go`, `tick`, `tack`}, and the initial symbol is S.

(a) Show that the following program is accepted by the grammar of tinyGo by providing a parse tree for it:

`[ tick? ; go tick! ]`

Solution: the solution is straightforward. The parse tree is unique:



(b) Show that the grammar is ambiguous by providing a tinyGo program with two distinct parse trees.

Solution: a simple solution is to reverse the order execution in the program of exercise (a):

`[ go tick! ; tick? ]`

Two distinct parse trees can be given, each encoding different precedence orders between sequential composition and asynchronous execution. Textually:

`[ go (tick! ; tick?) ]` and `[ (go tick!) ; tick? ]`

(c) Enumerate all sources of ambiguities very briefly (1-3 sentences).

Solution: The sources of ambiguity are: (i) unspecified associativity of `;`, (ii) unspecified associativity of `+`, and (iii) unspecified precedence between operators `go` , `;` and `+`.

(d) Provide an unambiguous grammar for tinyGo. Your grammar should accept exactly those programs accepted by the original grammar. Explain how you transformed the grammar to obtain a new one. Hint: You can obtain the new grammar by applying the transformations seen during the course and in the mandatory assignment. Show that the program provided as a solution to (b) has a unique parse tree in the new grammar.

Solution: The following grammar is obtained by applying standard transformations seen in class: (i) left/right-associativity is imposed by admitting left/right-recursion only and (ii) operator precedence is obtained by stratifying the grammar into layers (in the following solution actions A have the highest precedence, choices come next (Q), and sequences (P) have lowest precedence:

$$\begin{array}{lcl} S & \rightarrow & \texttt{[}P\texttt{]} \\ P & \rightarrow & P \ \texttt{;} \ Q \mid Q \\ Q & \rightarrow & Q \ \texttt{+} \ A \mid A \\ A & \rightarrow & \textsf{skip} \mid C\texttt{!} \mid C\texttt{?} \mid \textsf{go}\ A \mid S \\ C & \rightarrow & \texttt{tick} \mid \texttt{tack} \end{array}$$

The unique parse tree of the program in the solution (b) corresponds to `[ (go tick!) ; tick? ]`

(e) Consider again the original grammar of tinyGo described at the beginning of the exercise. Show that the grammar is not an LL(1) by providing an example of a valid program where an LL(1) parser would not be able to choose a production based on the lookahead token.

Solution: the grammar is not LL(1) and the table below shows several cases in which several productions may be chosen for a given token. For the sake of brevity the cell contains the right-hand side of the productions only.

	;	+	skip	!	?	go	[	]	tick	tack
S							[P]			
P			A P ; P P + P			A P ; P P + P	A P ; P P + P		A P ; P P + P	A P ; P P + P
A			skip			go P	[P]		C! C!	C? C!
C									tick	tack

(f) Consider the following alternative grammar for tinyGo

$$
\begin{array}{lcl}
S & \to & [P] \\
P & \to & A \mid A\ ;\ P \mid A\ +\ P \\
A & \to & \mathsf{skip} \mid C! \mid C? \mid \mathsf{go}\ P \mid S \\
C & \to & \mathtt{tick} \mid \mathtt{tack}
\end{array}
$$

Apply the transformations seen in class (e.g. left-factorisation) to obtain an LL(1) grammar. Show that the grammar is LL(1) by providing a deterministic parsing table for it, where rows correspond to non-terminal symbols and columns correspond to terminal symbols. In the cell corresponding to non-terminal X and terminal y you should write the production that parser should choose if it is trying to parse an expression generated by X and the lookahead symbol is y. You can use a copy of the following template for providing your solution:

	;	+	skip	!	?	go	[	]	tick	tack
S										
P										
A										
C										

Solution: The following solution can be obtained starting from the original grammar and applying left-factorisation on the productions of P and (a subset of those in A):

$$
\begin{array}{lcl}
S & \to & [P] \\
P & \to & AQ \\
Q & \to & \epsilon \mid\ ;\ P \mid\ +\ P \\
A & \to & \mathsf{skip} \mid CO \mid \mathsf{go}\ P \mid S \\
O & \to & ! \mid ? \\
C & \to & \mathtt{tick} \mid \mathtt{tack}
\end{array}
$$

Applying the algorithm seen in class (based on the computation of *First*() and *Follow*() lookahead symbols) we obtain the below parsing table. The tricky part is to deal with the cases in which ϵ can be produced.

	`;`	`+`	`skip`	`!`	`?`	`go`	`[`	`]`	`tick`	`tack`
S							`[` P `]`			
P			AQ			AQ	AQ		AQ	AQ
Q	`;` P	`;` Q						ϵ		
A			`skip`			`go` P	`[` P `]`		CO	CO
O				`!`	`?`					
C									`tick`	`tack`

(g) Consider the subset of `tinyGo` programs that are *deterministic* and *sequential*. These are programs generated by P in our original grammar which do not contain the tokens `+`, `go`, `[` and `]`. Such language can be described with the following grammar (with start symbol P):

$$\begin{array}{lcl} P & \rightarrow & A \mid P \;\texttt{;}\; P \\ A & \rightarrow & \texttt{skip} \mid C\texttt{!} \mid C\texttt{?} \\ C & \rightarrow & \texttt{tick} \mid \texttt{tack} \end{array}$$

We are now interested in a subset of such language that we call *balanced programs*. Those are programs that contain the same number of input and output operations on each channel. For example, the program

```
tick! ; tack? ; tick? ; tack! ; tick! ; tick?
```

is balanced the number of times `tick` is used as input and output is the same (2) and the number of times `tack` is used as input and output is also the same (1). Instead program

```
tick! ; tick? ; tick?
```

is *not* a balanced program since the program performs two read operations on channel `tick` but it performs only one output on the same channel. Show that the set of balanced programs is a context-free language by providing a context-free grammar for such programs.

Solution: The key observation here is that the language is similar to the language of balanced parenthesis and the language of equal number of 0's an 1's, which have been seen in class in several variations. The following grammar is inspired by those examples:

$$\begin{array}{lcl} P & \rightarrow & \texttt{skip} \mid P \;\texttt{;}\; P \\ & \mid & \texttt{tick?}\, Q \;\texttt{;}\; \texttt{tick!} \mid \texttt{tick!}\, Q \;\texttt{;}\; \texttt{tick?} \\ & \mid & \texttt{tack?}\, Q \;\texttt{;}\; \texttt{tack!} \mid \texttt{tack!}\, Q \;\texttt{;}\; \texttt{tack?} \\ Q & \rightarrow & \epsilon \mid \;\texttt{;}\; P \end{array}$$