Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Solutions for Exam 2

Johan Jeuring

Thursday, 30 January 2014, 08:30–10:30

Please keep in mind that there are often many possible solutions and that these example solutions may contain mistakes.

Questions

1 (5+5 points)**.** Consider the grammars for the regular languages L_1 and L_2:

$$\begin{array}{ll} L_1: & S \to \mathtt{a}S \mid \mathtt{b}S \mid \varepsilon \\ L_2: & S \to \mathtt{a}A \mid \mathtt{b}S \\ & A \to \mathtt{b}S \mid \varepsilon \end{array}$$

Give a regular expression for each language. •

Solution 1.

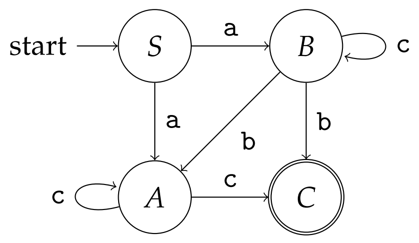
$$\begin{array}{ll} L_1: & (\mathtt{a} + \mathtt{b})\ ^* \\ L_2: & (\mathtt{b}^*\mathtt{ab})^*\mathtt{a} \end{array}$$

Marking

You either get full points or no points for both subquestions.
No penalty for using notation in a novel way (such as ?a instead of a?).
L_1 was answered correctly by almost everybody.
The most common mistakes in L_2 were: empty string part of the language, and accepts a string with a b at the end.

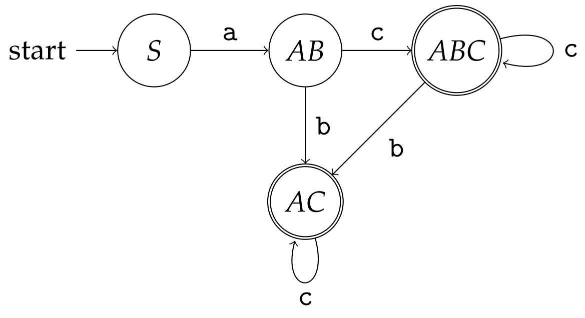
○

2 (10 points)**.** Consider the following NFA:



Transform this non-deterministic automaton into a deterministic automaton (for which you may give a drawing). •

Solution 2.



Marking
The most common mistakes were:
Not a deterministic automaton (in one state): -2
Forgotten an end-state: -2
Also accept `a`: -2
Forgotten a single transition: -2

○

3 (10 points)**.** Consider the following grammar:

$S \to E\{P\} \mid \varepsilon$
$P \to V{=}S \mid \varepsilon$
$V \to \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$
$E \to ! \mid ?D$
$D \to PS$

To use this grammar in an LL(1) parser, we need to determine several properties of this grammar. Fill out the table below by computing the values in the columns for the appropriate rows. Use *True* and *False* for property values and set notation for everything else.

•

Solution 3.

NT	Production	*empty*	*emptyRhs*	*first*	*firstRhs*	*follow*	*lookAhead*
S		True		{!,?}		{{,},!,?}	
	$S \to E\{P\}$		False		{!,?}		{!,?}
	$S \to \varepsilon$		True		{ }		{{,},!,?}
P		True		{a,b,c}		{{,},!,?}	
	$P \to V{=}S$		False		{a,b,c}		{a,b,c}
	$P \to \varepsilon$		True		{ }		{{,},!,?}
V		False		{a,b,c}		{=}	
	$V \to \texttt{a}$		False		{a}		{a}
	$V \to \texttt{b}$		False		{b}		{b}
	$V \to \texttt{c}$		False		{c}		{c}
E		False		{!,?}		{{}	
	$E \to !$		False		{!}		{!}
	$E \to ?D$		False		{?}		{?}
D		True		{a,b,c,!,?}		{{}	
	$D \to PS$		True		{a,b,c,!,?}		{{,a,b,c,!,?}

Marking
One error in a column: -1
More than one error: -2.5
I didn't check the Rhs columns carefully, their function is to help fill out the other columns

○

4 (5 points)**.** Is the above grammar LL(1)? Explain how you arrived at your answer. ●

Solution 4.
The above grammar is not LL(1) because the *lookAhead* sets of the *S* productions have a non-empty intersection.

Marking
No mentioning of lookaheadsets: -2
Incomprehensible argument: -5
Using the wrong terms, but managed to convince that the core is correct: -1

○

5 (10 points)**.** Consider the following grammar, with start symbol S:

$$
\begin{aligned}
S &\to OSS \mid C\\
O &\to * \mid \varepsilon\\
C &\to \mathtt{0} \mid \mathtt{x}
\end{aligned}
$$

We augment the grammar above in preparation for LR parsing:

$$S' \to S\$$$

and S' becomes the new start symbol.

Compute the LR(0) automaton corresponding to the full grammar. Number each state for future reference.

●

Solution 5. I only show the top-level productions ('kernel' items) in a state of the automaton; of course the closure should also be calculated.

$$\begin{tikzcd}
& \boxed{(\mathbf{1})\quad C \to \mathtt{0}\bullet} & \boxed{(\mathbf{2})\quad C \to \mathtt{x}\bullet} & \boxed{(\mathbf{3})\quad O \to *\bullet} \\
\text{start} \arrow[r] & \boxed{(\mathbf{0})\quad S' \to \bullet S\$} \arrow[u, "\mathtt{0}"] \arrow[ur, "\mathtt{x}"] \arrow[urr, "*"] \arrow[r, "O"] \arrow[d, "S"] \arrow[dr, "C"] & \boxed{(\mathbf{4})\quad S \to O\bullet SS} \arrow[ul, "\mathtt{0}"] \arrow[u, "\mathtt{x}"] \arrow[ur, "*"] \arrow[r, shift left, "S"] \arrow[loop below, "O"] \arrow[d, "C"] & \boxed{(\mathbf{5})\quad S \to OS\bullet S} \arrow[ull, "\mathtt{0}"] \arrow[ul, "\mathtt{x}"] \arrow[u, "*"] \arrow[l, shift left, "O"] \arrow[d, "S"] \arrow[dl, "C"] \\
& \boxed{(\mathbf{8})\quad S' \to S\bullet\$} & \boxed{(\mathbf{7})\quad S \to C\bullet} & \boxed{(\mathbf{6})\quad S \to OSS\bullet}
\end{tikzcd}$$

Marking
ε interpreted as a symbol: -4
No closures of item sets: -4

○

6 (5 points)**.** Classify each state in your LR(0) automaton as a shift state, reduce state, or shift-reduce conflict state. Also mark potential reduce-reduce conflicts. If there are conflicts, would applying SLR(1) parsing help to resolve these? ●

Solution 6. The states $(\mathbf{1})$, $(\mathbf{2})$, $(\mathbf{3})$, $(\mathbf{6})$, $(\mathbf{7})$ are all reduce states, and do not gain extra items by computing the closure. The state $(\mathbf{8})$ is the special end state.

The states $(\mathbf{0})$, $(\mathbf{4})$, $(\mathbf{5})$ gain the following items for closure

$S \to \bullet OSS$
$S \to \bullet C$
$O \to \bullet *$
$O \to \bullet$
$C \to \bullet 0$
$C \to \bullet \mathtt{x}$

Therefore, all three states have a shift-reduce conflict. SLR(1) parsing would not be of any help here, because we can shift for all three symbols in the alphabet, and the follow set of O also contains all three symbols.

Marking
Shift/reduce conflicts wrong: -3
Shift/reduce conflicts help, or no mentioning of SLR: -2
SLR(1) doesn't help, but no further explanation: -1

○

7 (5 points)**.** Play through the LR parsing process (no extras, neither SLR nor LALR) for the word `*00`. Resolve potential shift-reduce conflicts by always choosing to shift, and potential reduce-reduce conflicts by picking any of the available reductions. ●

Solution 7.

stack	input	remark
$(\mathbf{0})$	`*00$`	we always shift, thus move to $(\mathbf{3})$
$(\mathbf{0})*(\mathbf{3})$	`00$`	reduce by $O \to *$
$(\mathbf{0})O(\mathbf{4})$	`00$`	we always shift, thus move to $(\mathbf{1})$
$(\mathbf{0})O(\mathbf{4})0(\mathbf{1})$	`0$`	reduce by $C \to 0$
$(\mathbf{0})O(\mathbf{4})C(\mathbf{7})$	`0$`	reduce by $S \to C$
$(\mathbf{0})O(\mathbf{4})S(\mathbf{5})$	`0$`	we always shift, thus move to $(\mathbf{1})$
$(\mathbf{0})O(\mathbf{4})S(\mathbf{5})0(\mathbf{1})$	`$`	reduce by $C \to 0$
$(\mathbf{0})O(\mathbf{4})S(\mathbf{5})C(\mathbf{7})$	`$`	reduce by $S \to C$
$(\mathbf{0})O(\mathbf{4})S(\mathbf{5})S(\mathbf{6})$	`$`	reduce by $S \to OSS$
$(\mathbf{0})S(\mathbf{8})$	`$`	success

Marking
Not completely finsihed the derivation: -1 or -2
Wrong derivation strategy: -5

○

8 (5 points)**.** Are there words in the language that cannot be parsed successfully using the simplistic strategy from the previous task? If so, give an example. ●

Solution 8. Yes, one example is the word `00` which can be derived as follows

$$S \Rightarrow OSS \Rightarrow^* \varepsilon CC \Rightarrow^* \texttt{00}$$

Trying to parse this word yields

stack	input	remark
$(\mathbf{0})$	`00`	we always shift, thus move to $(\mathbf{1})$
$(\mathbf{0})\texttt{0}(\mathbf{1})$	`0`	reduce by $C \to 0$
$(\mathbf{0})C(\mathbf{7})$	`0`	reduce by $S \to C$
$(\mathbf{0})S(\mathbf{8})$	`0`	failure

Marking
Correct, but inconsistent with answers to previous exercises: -2
Correct, but with strange arguments: -2
Yes, but wrong example: -4

○

9 (15 points)**.** In the 'additional task' 8 of the third lab exercise you have to include a `for` statement in the source language of (simplified) C#, and add functionality to compile a `for` statement. Here is an example of a `for` statement:

```
for (n=0; n<10; n++)
{ do something }
```

You can assume that the three components between parentheses are expressions, and that doing something is achieved by means of a block of statements.

Sketch how you would translate a `for` statement into SSM instructions. Give an explanation similar to the explanations of translating statements on the slides on the Simple Stack Machine. Is your translation optimal? See the SSM instruction set reference in Section **??**. ●

Solution 9.
See your own solution to additional task 8 in the solution to the labs.

Marking
No discussion of optimality: -4
No arguments for why the presented solution is optimal or not: -2
No abstraction from the example: -1
No use whatsoever of SSM instructions in the solution: -5
Confusing arguments: -1 to -10
No initialisation: -2
No code for "do something": -3

○

10 (5 points)**.** Consider the following Haskell datatype that describes regular expressions over an alphabet type *s*:

```
data Regex s = Empty
             | Epsilon
             | Const s
             | Sequ  (Regex s) (Regex s)
             | Plus  (Regex s) (Regex s)
             | Star  (Regex s)
```

Translate the regular expression

$(\texttt{aa} + \texttt{b})^*$

into a value of type *Regex Char*. •

Solution 10.

```
Star (Plus (Sequ (Const 'a') (Const 'a')) (Const 'b'))
```

Marking
Most people answered this exercise correctly. Some errors:
Sequencing star using *Sequ* to the end: -3
$+$ binds stronger than sequencing: -2

○

11 (10 points)**.** Define an algebra type *RegexAlgebra* and a fold function *foldRegex* for the *Regex* type. •

Solution 11. The algebra needs two parameters, because *Regex* already has one:

```
type RegexAlgebra s r = (r, r, s → r, r → r → r, r → r → r, r → r)
```

The *fold* function is straightforward to define:

```
foldRegex :: RegexAlgebra s r → Regex s → r
foldRegex (empty, epsilon, const, sequ, plus, star) r = fold r
  where
    fold Empty         = empty
    fold Epsilon       = epsilon
    fold (Const s)     = const s
    fold (Sequ r₁ r₂)  = sequ (fold r₁) (fold r₂)
    fold (Plus r₁ r₂)  = plus (fold r₁) (fold r₂)
    fold (Star r)      = star (fold r)
```

Marking
The algebra contributes 4 points to the grade, the fold 6 points.
Not providing the *s* type as an argument in the algebra: -1
Not providing the *s* type as an argument in the algebra, and not using it: -2
Recursive occurrences in algebra constant type *s* instead of recursive type *r*: -2
Using *RegEx* itself instead of *r* at recursive positions: -3
Using *Const* instead of a type variable in the algebra: -2

fold not applied recursively: -4
Algebra not passed on in recursive calls: -2

○

12 (10 points)**.** This is a *bonus* exercise. If you answer the previous exercises correctly, your grade will be a 10.

Define a function

$$regexParser :: Eq\ s \Rightarrow Regex\ s \rightarrow Parser\ s\ [s]$$

using the parser combinators such that *regexParser r* is a parser for the language described by the regular expression *r*. The parser should return the list of symbols recognized. You should define the function in terms of *foldRegex*. ●

Solution 12.

```
regexParserAlgebra :: Eq s ⇒ RegexAlgebra s (Parser s [s])
regexParserAlgebra = (empty,
                      succeed [],
                      λs → (:[]) <$> symbol s,
                      λx₁ x₂ → (++) <$> x₁ <*> x₂,
                      (<|>),
                      λx → concat <$> many x)
regexParser′ :: Eq s ⇒ Regex s → Parser s [s]
regexParser′ = foldRegex regexParserAlgebra
```

Handling the end of input was not required, but ok if done.

Marking
Using a *fold* but no algebra component correct: +2
Per correct algebra component: about +1 or +2

○