Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Solutions for Exam 2

Sean Leather, Johan Jeuring

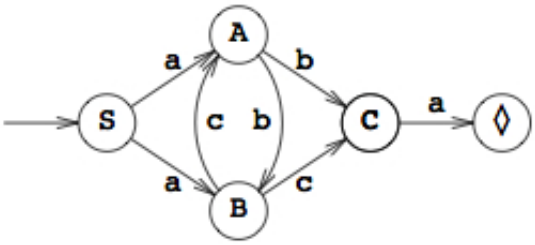
Thursday, 31 January 2013, 17:00–20:00

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

Questions

Regular grammars, NFAs, DFAs, Pumping Lemmas

Consider the following NFA, with start state S, and accepting state $\diamond$.



1 (5 points)**.** Construct a regular grammar with the same language. •

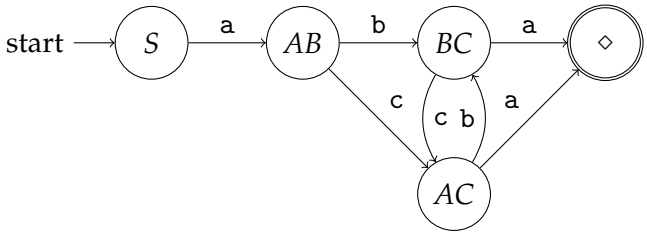
Solution 1.

$$
\begin{aligned}
S &\to \mathtt{a}\ A \mid \mathtt{a}\ B \\
A &\to \mathtt{b}\ B \mid \mathtt{b}\ C \\
B &\to \mathtt{c}\ A \mid \mathtt{c}\ C \\
C &\to \mathtt{a}
\end{aligned}
$$

○

2 (5 points)**.** Construct a DFA (Deterministic Finite-state Automaton) with the same language (you may draw a DFA). •

Solution 2.



○

A palindrome is a string which reads the same when reversed. In the following task you will look at palindromes in DNA strings. A DNA string is a list of 'C', 'A', 'T', 'G' symbols. The male DNA on the Y chromosome contains some huge palindromes, some more than a million DNA symbols long. These palindromes have a small non-palindromic gap in the middle, which may contain an arbitrary sequence of DNA symbols. A small example of a DNA palindrome with a gap is the string "ATACGTATA". This string has a non-palindromic gap of length three ("CGT") in the middle.

3 (8 points)**.** Give a context-free grammar specifying the language of palindromes in DNA with gaps of at most length 3 in the middle. •

Solution 3.

$$P \to \mathtt{A}\ P\ \mathtt{A} \mid \mathtt{C}\ P \mid \mathtt{C} \mid \mathtt{G}\ P\ \mathtt{G} \mid \mathtt{T}\ P\ \mathtt{T} \mid D\ D\ D$$
$$D \to \mathtt{A} \mid \mathtt{C} \mid \mathtt{G} \mid \mathtt{T} \mid \varepsilon$$

○

4 (8 points)**.** Is the language of palindromes in DNA with gaps of at most length 3 in the middle specified in the previous exercise regular? If so, give a regular grammar or a DFA for the language. If not, use the regular pumping lemma to prove this. •

Solution 4. The language of palindromes in DNA with gaps in the middle is not regular. We use the regular pumping lemma to prove this.
Let $n \in \mathbb{N}$.
Take $s = \mathtt{A}^n\ \mathtt{C}\ \mathtt{G}\ \mathtt{T}\ \mathtt{A}^n$, with $x = \varepsilon$, $y = \mathtt{A}^n$, and $z = \mathtt{C}\ \mathtt{G}\ \mathtt{T}\ \mathtt{A}^n$. The sentence s is an element of the language.
Let u, v, w be such that $y = uvw$ with $v \not\equiv \varepsilon$, that is, $u = \mathtt{A}^p$, $v = \mathtt{A}^q$ and $w = \mathtt{A}^r$ with $p + q + r = n$ and $q > 0$.
Take $i = 2$, then $xuv^2wz = \mathtt{A}^{p+2q+r}\ \mathtt{T}\ \mathtt{A}^n = \mathtt{A}^{n+q}\ \mathtt{C}\ \mathtt{G}\ \mathtt{T}\ \mathtt{A}^n$.
Since $q > 0$, this is not a sentence in the language of palindromes with gaps of at most length 3.
Using the negative version of the regular pumping lemma, we conclude that this language is not regular. ○

LL parsing

Consider the following context-free grammar:

Session → *Facts Question* | `(` *Session* `)` *Session*
Facts → *Fact Facts* | ε
Fact → `! x`
Question → `? x`

This grammar describes a simple language that could be used as the input language for a rudimentary consulting system: the user enters some facts, and then asks a question. There is also a facility for sub-sessions. The contents of the facts and questions are of no concern here. They are represented by the word `x`, which is regarded as a terminal symbol.

5 (8 points)**.** Determine the empty property, and the first and follow sets for each of the nonterminals of the above grammar. •

Solution 5.

	empty	first	follow
Session	False	{`!`, `?`, `(`}	{`)`}
Facts	True	{`!`}	{`?`}
Fact	False	{`!`}	{`!`, `?`}
Question	False	{`?`}	{`)`}

○

6 (8 points)**.** Using empty, first, and follow, determine the lookahead set of each production in the above grammar. •

Solution 6.

Session → *Facts Question*	{`!`, `?`}
Session → `(` *Session* `)` *Session*	{`(`}
Facts → *Fact Facts*	{`!`}
Facts → ε	{`?`}
Fact → `! x`	{`!`}
Question → `? x`	{`?`}

○

7 (4 points)**.** Is the above grammar LL(1)? Explain how you can determine this using the lookahead sets of the productions. •

Solution 7. Since the intersection of the lookahead sets for any pair of productions for the same non-terminal is empty, the above grammar is LL(1). ○

8 (6 points)**.** The string `( ? x ) ! x ? x` is a sentence of the above grammar. Show how an LL(1) parser recognizes this string by using a stack. Show step by step the contents of the stack, the part of the input that has not been consumed yet, and which action you perform. If the above grammar is not LL(1), point at the step where different choices can be made. •

Solution 8.

Stack	input	action
Session	`( ? x ) ! x ? x`	Expand
`(` *Session* `)` *Session*	`( ? x ) ! x ? x`	Match
Session `)` *Session*	`? x ) ! x ? x`	Expand
Facts Question `)` *Session*	`? x ) ! x ? x`	Expand
Question `)` *Session*	`? x ) ! x ? x`	Expand
`? x )` *Session*	`? x ) ! x ? x`	Match
`x )` *Session*	`x ) ! x ? x`	Match
`)` *Session*	`) ! x ? x`	Match
Session	`! x ? x`	Expand ○
Facts Question	`! x ? x`	Expand
Fact Facts Question	`! x ? x`	Expand
`!` *Facts* `x` *Question*	`! x ? x`	Match
Facts `x` *Question*	`x ? x`	Expand
`x` *Question*	`x ? x`	Match
Question	`? x`	Expand
`? x`	`? x`	Match
`x`	`x`	Match
—	—	Succeed

LR parsing

Consider the context-free grammar:

$E \to E$ `-` T
$E \to T$
$T \to$ `n`
$T \to$ `(` E `)`

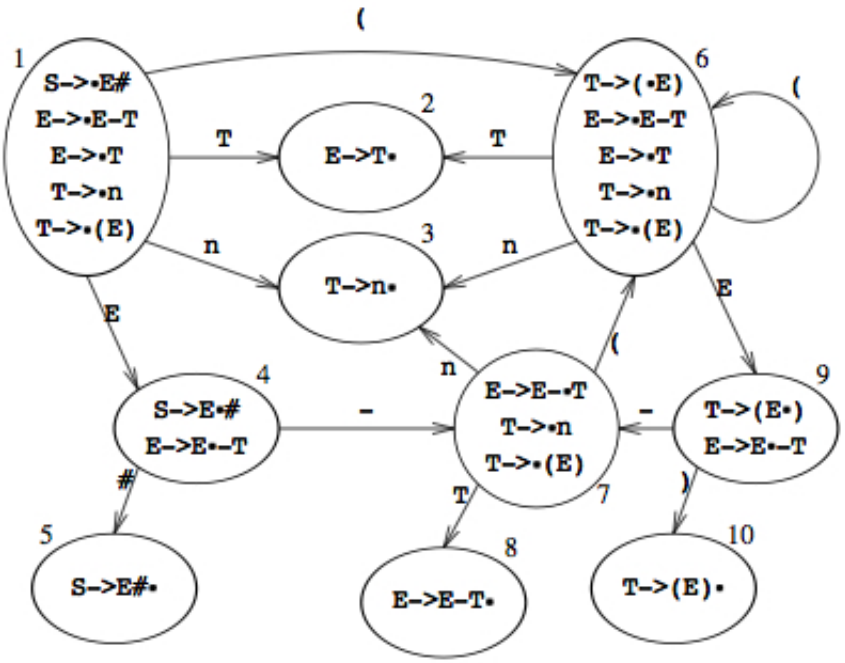
We want to use an LR parsing algorithm to parse sentences from this grammar. We start with extending the grammar with a new start-symbol S, and a production

$S \to E$ \$

where \$ is a terminal symbol denoting the end of input.

9 (8 points)**.** Construct the LR(0) automaton for the extended grammar. •

Solution 9. The LR(0) automaton corresponding to the full grammar looks as follows (each state is numbered before the production for future reference):



○

10 (4 points)**.** Is this grammar LR(0)? Explain why or why not. •

Solution 10. The grammar is LR(0), because there are no shift/reduce or reduce/reduce conflicts. ○

11 (8 points)**.** The string `n - ( n - n ) $` is a sentence of the above grammar. Show how an LR(0)-based parser recognizes this string by using a stack. Show step by step the contents of the stack mixed with the states in the LR(0) automaton you pass through, the part of the input that has not been consumed yet, and which action you perform. •

Solution 11.

Stack	input	action
1	n - (n - n) $	Shift
1 n 3	- (n - n) $	Reduce
1 T 2	- (n - n) $	Reduce
1 E 4	- (n - n) $	Shift
1 E 4 - 7	(n - n) $	Shift
1 E 4 - 7 (6	n - n) $	Shift
1 E 4 - 7 (6 n 3	- n) $	Reduce
1 E 4 - 7 (6 T 2	- n) $	Reduce
1 E 4 - 7 (6 E 9	- n) $	Shift
1 E 4 - 7 (6 E 9 - 7	n) $	Shift
1 E 4 - 7 (6 E 9 - 7 n 3	) $	Reduce
1 E 4 - 7 (6 E 9 - 7 T 8	) $	Reduce
1 E 4 - 7 (6 E 9	) $	Shift
1 E 4 - 7 (6 E 9) 10	$	Reduce
1 E 4 - 7 T 8	$	Reduce
1 E 4	$	Shift
1 E 4 $ 5		Accept

○

12 (3 points)**.** Suppose we take the same grammar, but replace the productions for E by:

$E \to T$ - E
$E \to T$

This grammar is not LR(0). Explain why. •

Solution 12. This grammar is not LR(0): there will be a shift/reduce conflict in the state that contains the two items $E \to T \bullet$ - E and $E \to T \bullet$. ○

13 (3 points)**.** The grammar is SLR(1). Explain why. •

Solution 13. The nonterminal E can be followed by the symbols $\{\$,)\}$. So when the next symbol in the input is one of these symbols, we reduce by means of the production $E \to T$, and when the next input symbol is - we shift. ○

Code generation

For mysterious reasons, the Maya culture received a surge of interest by the end of 2012. As a consequence, computers now perform many calculations based on Mayan input. In this exercise you will develop a code generator for generating stack machine code to calculate Mayan numbers.

On a separate sheet of paper you can find the Wikipedia description of Mayan numbers. To process Mayan numbers, I use the following abstract syntax:

```
type MayaNumber = [MayaBase]
data MayaBase   = Shell
                | DotsLines Int Int deriving Show
```

where a *MayaBase* value corresponds to one of the vertical levels (the 1s, 20s, 400s, etc, where the higher levels, with the higher values, come first). *DotsLines* 3 4 means 3 dots above 4 lines. For example, the Mayan number for 429 (given as example in the Wikipedia document), is represented as

$$mn429 = [DotsLines\ 1\ 0, DotsLines\ 1\ 0, DotsLines\ 4\ 1]$$

14 (12 points)**.** Write a function

$$maya2Code :: MayaNumber \to Code$$

which takes a Mayan number, and produces stackmachine code, which when run on the SSM, produces the integer corresponding to the Mayan number. All multiplications and additions have to be performed on the stack machine, so calculating the resulting integer i in Haskell, and then outputting $[LDC\ i, TRAP\ 0]$ is not sufficient. The type *Code* is the same as in the third lab, and repeated below. •

Solution 14.

$$\begin{aligned}
&maya2Code\ maya &&= maya2Code'\ (reverse\ maya) +\!\!+ [TRAP\ 0]\\
&maya2Code'\ [] &&= [LDC\ 0]\\
&maya2Code'\ [x] &&= codeBase\ x\\
&maya2Code'\ (x : y : ys) &&= codeBase\ x +\!\!+\\
&&&\quad [LDC\ 20] +\!\!+\\
&&&\quad maya2Code'\ (y : ys) +\!\!+\\
&&&\quad [MUL, ADD]\\
&codeBase\ Shell &&= [LDC\ 0]\\
&codeBase\ (DotsLines\ m\ n) &&= [LDC\ m, LDC\ n, LDC\ 5, MUL, ADD]\\
&main &&= putStrLn\ \$\ formatCode\ (maya2Code\ mn429)
\end{aligned}$$

∘

type *Code* = [*Instr*]

data *Instr*

```
  = STR Reg | STL Int | STS Int | STA Int      — Store from stack
  | LDR Reg | LDL Int | LDS Int | LDA Int      — Load on stack
  | LDC Int | LDLA Int | LDSA Int | LDAA Int  — Load on stack
  | BRA Int | Bra String                       — Branch always (relative/to label)
  | BRF Int | Brf String                       — Branch on false
  | BRT Int | Brt String                       — Branch on true
  | BSR Int | Bsr String                       — Branch to subroutine
  | ADD | SUB | MUL | DIV | MOD                — Arithmetical operations on 2 stack operands
  | EQ | NE | LT | LE | GT | GE                — Relational operations on 2 stack operands
  | AND | OR | XOR                             — Bitwise operations on 2 stack operands
```

```
  | NEG | NOT                          — operations on 1 stack operand
  | RET | UNLINK | LINK Int | AJS Int  — Procedure utilities
  | SWP | SWPR Reg | SWPRR Reg Reg | LDRR Reg Reg— Various swaps
  | JSR | TRAP Int | NOP | HALT        — Other instructions
  | LABEL String                       — Pseudo-instruction for generating a label
  deriving Show
```