Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Solutions for Exam 2

Sean Leather

Monday, 30 January 2012, 17:00 – 20:00

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

1 Questions

1.1 Regular Languages

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

$$
\begin{array}{ll}
L_1: & S \rightarrow \mathtt{ab} \mid \mathtt{cd}S \\
L_2: & S \rightarrow S\ A \mid \varepsilon \\
 & A \rightarrow \mathtt{abcd} \mid \mathtt{dcba}
\end{array}
$$

(a) Give a regular expression for each language.

(b) Define a parser for each regular expression using the Haskell *Regex* combinator library described in Section 2.1. The parsers should produce an appropriate representation of the input.

•

Solution 1.

(a) The regular expressions:

$$
\begin{array}{ll}
L_1: & (\mathtt{cd})^{*}\mathtt{ab} \\
L_2: & (\mathtt{abcd} + \mathtt{dcba})^{*}
\end{array}
$$

(b) The parsers:

```
l1 = (,,) <$> many ((,) <$> symbol 'c' <*> symbol 'd')
          <*> symbol 'a'
          <*> symbol 'b'
l2 = many (r 'a' 'b' 'c' 'd' <|> r 'd' 'c' 'b' 'a')
  where r w x y z = f <$> s w <*> s x <*> s y <*> s z
        f a b c d = [a,b,c,d]
        s = symbol
```

○

2 (15+15 points)**.** For each language definition below, show whether or not the language is regular. If it is regular, give one of the following:

(a) a regular grammar in an acceptable form,

(b) a regular expression, or

(c) a finite state automaton.

If the language is not regular, prove that using the pumping lemma for regular languages.

(a) $\{\, \mathsf{o}^m \mathsf{p}^n \mid n = m + 1 \,\}$

(b) $\{\, \mathsf{3}^j \mathsf{7}^k \mid j > 2, k < 5 \,\}$

●

Solution 2.

(a) The language $L = \{\, \mathsf{o}^m \mathsf{p}^n \mid n = m + 1 \,\}$ is not regular. To prove it, we must assume it is regular and find a contradiction with the pumping lemma.

Let $x = \varepsilon$, $y = \mathsf{o}^m$, $z = \mathsf{p}^{m+1}$.

Then, $xyz = \mathsf{o}^m \mathsf{p}^{m+1} \in L$ and $|y| \geqslant m$.

From the pumping lemma, we know there must be a loop in y, i.e. $y = uvw$ with $q = |v\,| > 0$ such that $xuv^i wz \in L$ for all $i \in \mathbb{N}$.

Let $i = 2$. We expect $xuv^2wz \in L$. If $u = \mathsf{o}^s$, $v = \mathsf{o}^q$, $w = \mathsf{o}^r$, then we expect $\mathsf{o}^s \mathsf{o}^{2\,q} \mathsf{o}^r \mathsf{p}^{m+1} \in L$. But it does not, because $s + 2\ q + r > m$ Therefore, L is not regular.

(b) Two possible options:

$\mathsf{333}^+(\varepsilon + \mathsf{7} + \mathsf{77} + \mathsf{777} + \mathsf{7777})$
$\mathsf{333}^+\mathsf{7?7?7?7?}$

○

1.2 Simple Stack Machine

3 (15 points)**.** Translate this program into SSM instructions. See the SSM instruction set reference in Section 2.2.

```
void main() {
  int x = fib(4);
  fib(x);
}

int fib(int n) {
  if (n < 2)
    return 1;
  else
    return fib(n - 1) + fib(n - 2);
}
```

•

Solution 3.

```
        bsr     main
        halt
main:   link    1       ; int x
        ldc     4
        bsr     fib     ; fib(4)
        stl     1
        bsr     fib     ; fib(x)
        unlink
        sts     -1
        ret
fib:    ldl     -2      ; if (n < 2)
        ldc     2
        lt
        brt     true
false:  ldl     -2      ; else ... fib(n-1)
        ldc     1
        sub
        bsr     fib
        ldl     -2      ; fib(n-2)
        ldc     2
        sub
        bsr     fib
        add             ; return fib(n-1) + fib(n-2)
        bra     cont
```

```
true:   ldc     1        ; return 1
cont:   ret
```

○

4 (10 points)**.** Given the initial SSM register state below, show the final (relative) state after the above instructions have been executed (and just before the program finishes). You may assume that the code and stack memory do not share address space.

Register	Initial Value	Description of Initial Value
PC	i	First instruction address
SP	s	Current head of stack
MP	m	Unknown
RR	r	Unknown

●

Solution 4.

Register	Value	Note
PC	$i+2$	After `bsr`
SP	s	Back to head of stack
MP	m	Unchanged
RR	r	Unchanged

○

1.3 LL Parsing

5 (30 points)**.** Copy the table below and complete it by computing the values in the columns for the appropriate rows. Use *True* and *False* for property values and set notation for everything else.

NT	Production	*empty*	*emptyRhs*	*first*	*firstRhs*	*follow*	*lookAhead*
M							
	$M \to$ `<`E`>`M						
	$M \to \varepsilon$						
E							
	$E \to Q$						
	$E \to Q$ `;` E						
Q							
	$Q \to$ `0`						
	$Q \to$ `1`						
	$Q \to M$						

●

Solution 5.

NT	Production	*empty*	*emptyRhs*	*first*	*firstRhs*	*follow*	*lookAhead*
M		True		$\{\texttt{<}\}$		$\{\texttt{;},\texttt{>}\}$	
	$M \rightarrow \texttt{<}E\texttt{>}M$		False		$\{\texttt{<}\}$		$\{\texttt{<}\}$
	$M \rightarrow \varepsilon$		True		$\{\}$		$\{\texttt{;},\texttt{>}\}$
E		True		$\{\texttt{0},\texttt{1},\texttt{<}\}$		$\{\texttt{>}\}$	
	$E \rightarrow Q$		True		$\{\texttt{0},\texttt{1},\texttt{<}\}$		$\{\texttt{0},\texttt{1},\texttt{<},\texttt{>}\}$
	$E \rightarrow Q\texttt{;}E$		False		$\{\texttt{0},\texttt{1},\texttt{<},\texttt{;}\}$		$\{\texttt{0},\texttt{1},\texttt{<}\}$
Q		True		$\{\texttt{0},\texttt{1},\texttt{<}\}$		$\{\texttt{;},\texttt{>}\}$	
	$Q \rightarrow \texttt{0}$		False		$\{\texttt{0}\}$		$\{\texttt{0}\}$
	$Q \rightarrow \texttt{1}$		False		$\{\texttt{1}\}$		$\{\texttt{1}\}$
	$Q \rightarrow M$		True		$\{\texttt{<}\}$		$\{\texttt{<},\texttt{;},\texttt{>}\}$

○

6 (15 points)**.** Is the above grammar LL(1)? Explain how you arrived at your answer. If the grammar is not LL(1), transform the grammar such that is LL(1) and complete a new table with only the rows that differ from the old table. •

Solution 6.

The above grammar is not LL(1) because the *lookAhead* sets of the E productions have a non-empty intersection. To make this grammar LL(1), we only need to left-factor E.

NT	Production	*empty*	*emptyRhs*	*first*	*firstRhs*	*follow*	*lookAhead*
	$E \rightarrow QF$		True		$\{\texttt{0},\texttt{1},\texttt{<}\}$		$\{\texttt{0},\texttt{1},\texttt{<},\texttt{>}\}$
F		True		$\{\texttt{;}\}$		$\{\texttt{>}\}$	
	$F \rightarrow \texttt{;}E$		False		$\{\texttt{;}\}$		$\{\texttt{;}\}$
	$F \rightarrow \varepsilon$		True		$\{\}$		$\{\texttt{>}\}$

○

7 (15 points)**.** Show the steps that a parser for the above LL(1) grammar (after transformation if necessary) goes through to recognize the following input sequence:

$\texttt{<0;<1>>}$

For each step (one per line), show the stack, the remaining input, and the action (followed by the relevant symbol or production) performed. If you reach a step in which you cannot proceed, note the action as “error.“ •

Solution 7.

stack	input	action
M	<0;<1>>	initial state
<E>M	<0;<1>>	expand M
E>M	0;<1>>	match <
QF>M	0;<1>>	expand E
0F>M	0;<1>>	expand Q
F>M	;<1>>	match 0
;E>M	;<1>>	expand F
E>M	<1>>	match ;
QF>M	<1>>	expand E
MF>M	<1>>	expand Q
<E>MF>M	<1>>	expand M
E>MF>M	1>>	match <
QF>MF>M	1>>	expand E
1F>MF>M	1>>	expand Q
F>MF>M	>>	match 1
>MF>M	>>	expand F
MF>M	>	match >
F>M	>	expand M
>M	>	expand F
M	ε	match >
ε	ε	expand M

○

1.4 LR Parsing

8 (25 points)**.** Copy the table below and complete it by computing the values in the columns for the appropriate rows. Where the label "(set)" is given, use set notation. Where "(RE)" is given, use regular expression notation. A set may reference other sets – using $[X]$ as the notation for the left context set of X – but a regular expression must not reference other regular expressions.

NT	Production	Left Context (set)	Left Context (RE)	LR(0) Context (RE)
S				
	$S \to A$s			
A				
	$A \to A\%B$			
	$A \to B$			
B				
	$B \to$ b			
	$B \to$ aA			

●

Solution 8.

NT	Production	Left Context (set)	Left Context (RE)	LR(0) Context (RE)
Z		$\{\varepsilon\}$	ε	
	$Z \to S\$$			$S\$$
S		$[Z]$	ε	
	$S \to A\mathtt{s}$			$A\mathtt{s}$
A		$[S] \cup [A] \cup [B] \cdot \{\mathtt{a}\}$	$((A\%)?\mathtt{a})^*$	
	$A \to A\%B$			$((A\%)?\mathtt{a})^* A\ \%B$
	$A \to B$			$((A\%)?\mathtt{a})^* B$
B		$[A] \cup [A] \cdot \{A\%\}$	$((A\%)?\mathtt{a})^* (A\%)?$	
	$B \to \mathtt{b}$			$((A\%)?\mathtt{a})^* (A\%)?\mathtt{b}$
	$B \to \mathtt{a}A$			$((A\%)?\mathtt{a})^* (A\%)?\mathtt{a}A$

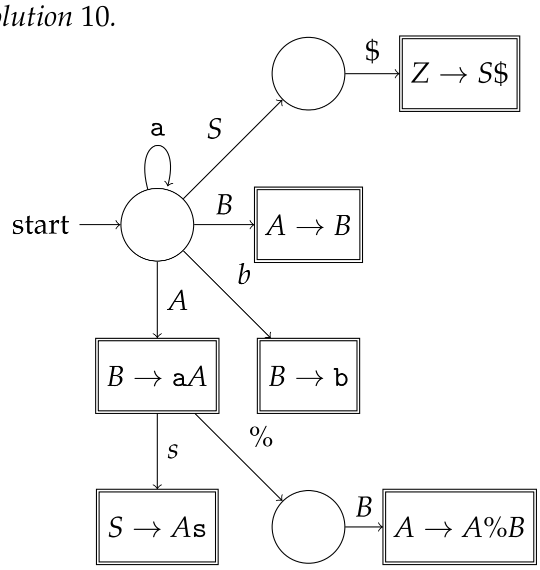
○

9 (15 points)**.** Is the above grammar LR(0)? Explain how you arrived at your answer. If the grammar is not LR(0), transform the grammar such that is LR(0) and complete a new table with only the rows that differ from the old table. ●

Solution 9. The above grammar is LR(0) because it satisfies the LR(0) condition, i.e. no LR(0) context is a prefix of another context. Therefore, we do not need to transform the grammar. ○

10 (15 points)**.** Construct the deterministic LR(0) automaton (characteristic machine) for the above LR(0) grammar (after transformation if necessary). Clearly label the start state, transitions, and accepting states. ●

Solution 10.



○

11 (15 points)**.** Show the steps that a parser for the above LR(0) grammar (after transformation if necessary) goes through to recognize the following input sequence:

`ab%abas`

For each step (one per line), show the stack, the remaining input, and the action (followed by the relevant symbol or production) performed. If you reach a step in which you cannot proceed, note the action as "error."

Note: You may use either symbols alone or symbols along with states from your DFA above, as you like. •

Solution 11.

stack	**input**	**action**
ε	`ab%abas$`	initial state
`a`	`b%abas$`	shift `a`
`ab`	`%abas$`	shift `b`
`a`B	`%abas$`	reduce $B \to$ `b`
`a`A	`%abas$`	reduce $A \to B$
B	`%abas$`	reduce $B \to$ `a`A
A	`%abas$`	reduce $A \to B$
A%	`abas$`	shift %
A%`a`	`bas$`	shift `a`
A%`ab`	`as$`	shift `b`
A%`a`B	`as$`	reduce $B \to$ `b`
A%`a`A	`as$`	reduce $A \to B$
A%B	`as$`	reduce $B \to$ `a`A
A	`as$`	reduce $A \to A\%B$
A`a`	`s$`	shift `a`
A`a`	`s$`	error

○

2 Appendix

2.1 Regular Expression Combinators

The following is the interface to a small regular expression parser combinator library. It centers around the abstract *Regex* datatype. The combinators are built from standard Haskell library type classes: *Functor*, *Applicative*, and *Alternative*. The semantics of each function should be clear from its name, type, and your experience with similar parser combinator libraries.

```haskell
data Regex s a = ...
instance Functor (Regex s) where ...
instance Applicative (Regex s) where ...
instance Alternative (Regex s) where ...
```

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Functor}\ f) \Rightarrow \mathit{Applicative}\ f\ \textbf{where} \\
\quad \mathit{pure} \quad :: a \rightarrow f\ a \\
\quad (<*>) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \\
\quad (*>) \quad :: f\ a \rightarrow f\ b \rightarrow f\ b \\
\quad (<*) \quad :: f\ a \rightarrow f\ b \rightarrow f\ a \\
\textbf{class}\ (\mathit{Applicative}\ f) \Rightarrow \mathit{Alternative}\ f\ \textbf{where} \\
\quad \mathit{empty} :: f\ a \\
\quad (<|>) :: f\ a \rightarrow f\ a \rightarrow f\ a \\
\quad \mathit{some} \quad :: f\ a \rightarrow f\ [a] \\
\quad \mathit{many} \quad :: f\ a \rightarrow f\ [a]
\end{array}
$$

$$
\begin{array}{l}
\mathit{satisfy} :: (s \rightarrow \mathit{Bool}) \rightarrow \mathit{Regex}\ s\ s \\
\mathit{symbol} :: (\mathit{Eq}\ s) \Rightarrow s \rightarrow \mathit{Regex}\ s\ s \\
\mathit{run} \quad :: \mathit{Regex}\ s\ a \rightarrow [s] \rightarrow \mathit{Maybe}\ a
\end{array}
$$

2.2 SSM Reference

SSM instructions are given in textual form, called assembler notation. For this exam, a program is a sequence of instructions with each instruction on a separate line, optionally proceed by a label and a colon (e.g. `main:`). A label (e.g. `main`) may be used as an argument to an instruction.

Copying Instructions

Instructions	Description
`ldc`	Load a constant
`lds`	Load a value relative to the SP
`ldh`	Load a value relative to the HP
`ldl`	Load a value relative to the MP
`lda`	Load a value pointed to by the value on top of the stack
`ldr`	Load a register value
`ldrr`	Load a register with a value from another register
`ldsa`	Load address of value relative to the SP
`ldla`	Load address of value relative to the MP
`ldaa`	Load address of value relative to the address on top of the stack
`sts`	Store a value relative to the SP
`sth`	Store a value relative to the HP
`stl`	Store a value relative to the MP
`sta`	Store a value pointed to by a value on the stack
`str`	Store a value in a register

Convenience Instructions For the Stack

Instructions	Description
`ajs`	Adjust the SP
`link`	Save the MP, adjust the MP and SP suitable for programming language function entry
`unlink`	Reverse of link

Arithmetic Instructions

Instructions	Description
`add`, `sub`, `mul`, `div`, `mod`, `neg`, `and`, `or`, `xor`	Binary operations
`not`	Unary operation
`cmp`	Put an int value on the stack which is interpreted as a status register value containing condition code to be used by a branch instruction
`eq`, `ne`, `lt`, `gt`, `le`, `ge`	Put true value on the stack if comparison is true

Control Instructions

Instructions	Description
`beq`, `bne`, `blt`, `bgt`, `ble`, `bge`	Branch on equality, unequality, less than, greater than, less or equal, greater or equal. These instructions pop the stack, interpret it as a condition code and jump accordingly
`bra`	Branch always, no popping of the stack
`brf` (`brt`)	Branch if top of stack is false (true)
`bsr`	Branch to subroutine. Like bra, but pushes the previous PC before jumping
`jsr`	Jump to subroutine. Like bsr, but pops its destination from the stack
`ret`	Return from subroutine. Pops a previously pushed PC from the stack and jumps to it
`halt`	Halt execution