Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Exam 2

Johan Jeuring

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

Preliminaries

- The exam consists of 10 pages (including this page). Please verify that you got all the pages.
- Fill out the answers **on the exam itself**.
- Write your **name** and **student number** here:

- The maximum score is stated at the top of each question. The total amount of points you can get is 100 (10 of which are bonus points).
- Try to give simple and concise answers. Write readable text. Do not use pencils or pens with red ink. You may use Dutch or English.
- When writing grammar and language constructs, you may use any set, sequence, or language operations covered in the lecture notes.
- When writing Haskell code, you may use Prelude functions and functions from the following modules: *Data.Char*, *Data.List*, *Data.Maybe*, and *Control.Monad*. Also, you may use all the parser combinators from the `uu-tc` package. If you are in doubt whether a certain function is allowed, please ask.

Good luck!

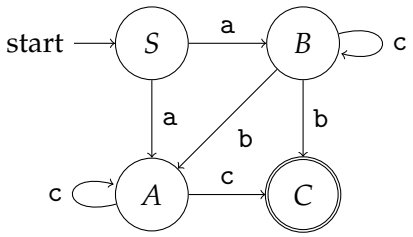
Questions

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

L_1: $S \to \mathtt{a}S \mid \mathtt{b}S \mid \varepsilon$
L_2: $S \to \mathtt{a}A \mid \mathtt{b}S$
$A \to \mathtt{b}S \mid \varepsilon$

Give a regular expression for each language. •

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



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

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

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

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.

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

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

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

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

We augment the grammar above in preparation for LR parsing:

$$S' \to S\$$$

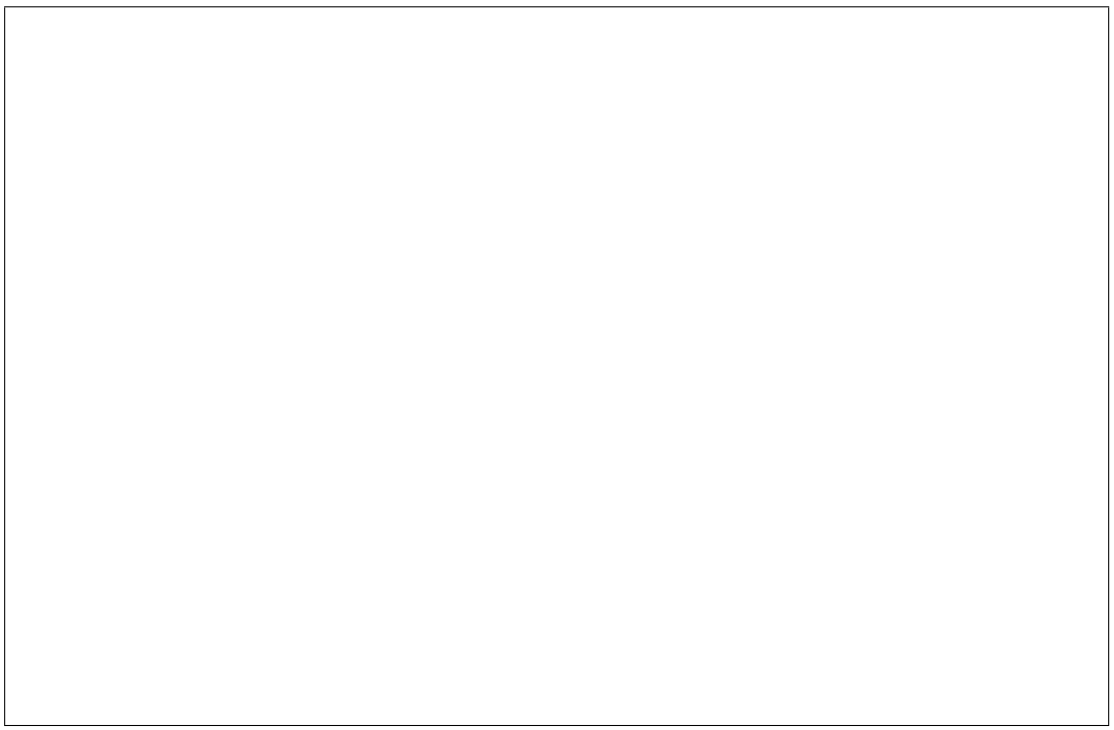
and S' becomes the new start symbol.

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

•

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? •

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. •



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. •

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 . •

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

$$\begin{aligned} \textbf{data}\ \mathit{Regex}\ s = \ & \mathit{Empty} \\ \mid\ & \mathit{Epsilon} \end{aligned}$$

$$
\begin{array}{l}
\mid \mathit{Const}\ s \\
\mid \mathit{Sequ}\ \ (\mathit{Regex}\ s)\ (\mathit{Regex}\ s) \\
\mid \mathit{Plus}\ \ (\mathit{Regex}\ s)\ (\mathit{Regex}\ s) \\
\mid \mathit{Star}\ \ (\mathit{Regex}\ s)
\end{array}
$$

Translate the regular expression

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

into a value of type *Regex Char*. •

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

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

Define a function

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

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*. •

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