Department of Computer Science, Australian National University
COMP2600 — Formal Methods for Software Engineering
Semester 2, 2011

— Assignment 3 —
Lambda Calculus, Grammars, PDAs and Turing Machines — Solutions

1 Regular Languages and Grammars [4 marks]

NOTE - the question paper somehow left out the production $D \to d\ B$, so the corresponding solution to (1) should leave out the edge labelled d, and the solution to (2) should leave out the production $B \to D\ d$

Consider this right-linear grammar which generates a language L.

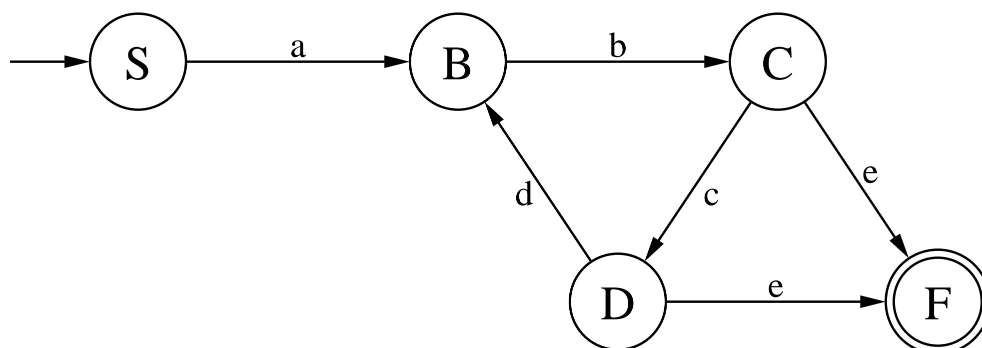
$S \to a\ B$ $\qquad D \to d\ B$
$B \to b\ C$ $\qquad D \to e$
$C \to c\ D$ $\qquad C \to e$

(upper case letters are non-terminals, lower-case letters are terminals, and the start symbol is S).

(1) [2 marks] Give a NFA which accepts the language L.

Solution: The natural way of doing this is where a non-terminal can be expanded to the strings which would be accepted by the machine starting at the corresponding state.

This gives the following NFA:



Note that the grammar could contain $D \to e\ F$ and $F \to \epsilon$ instead of $D \to e$, and it would give the same NFA.

(2) [2 marks] Give a left-linear grammar which generates the same language L.

Solution: The natural way to do this is where a non-terminal can be expanded to the strings which would be accepted by the machine if the corresponding state were a final state.

This gives the grammar (with starting symbol F)

$$
\begin{array}{ll}
F \to C\ e & C \to B\ b \\
F \to D\ e & B \to D\ d \\
D \to C\ c & B \to a
\end{array}
$$

Alternatively, instead of $B \to a$, you could have productions $B \to S\ a$ and $S \to \epsilon$.

2 Context-free Languages and Pushdown Automata [7 marks]

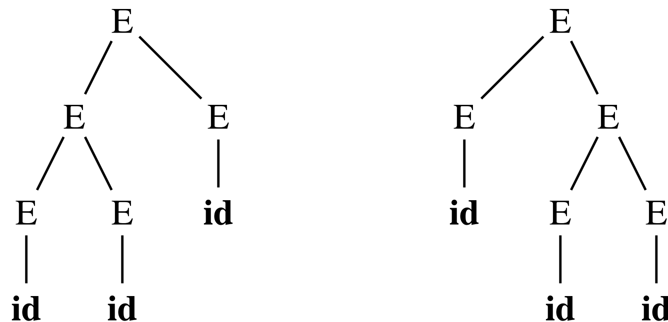
Consider the following grammar which generates a language of simple expressions.

$$
\begin{array}{l}
E \to E\ E \\
E \to \langle E \rangle \\
E \to \mathbf{id}
\end{array}
$$

Here E is the start symbol, and you treat **id** (which denotes an identifier) as a terminal symbol, and as a token of the language. So there are three terminal symbols, **id**, '$\langle$' and '$\rangle$'. (Think of '$\langle$' and '$\rangle$' as parentheses — we just use these symbols to make writing PDA transitions like $\delta(\ldots)\ldots$ less confusing).

(1) [1 mark] This grammar is ambiguous. Give an example of a string which can be generated two different ways. Which one corresponds to the interpretation of expressions in Haskell, or in the λ-calculus? (That is, if the angle brackets were in fact parentheses).

Solution: The sentence "**id id id**" can be generated as either of the following:



The diagram on the left corresponds to parsing in Haskell or the λ-calculus.

(2) [2 marks] Give the (non-deterministic) pushdown automaton which corresponds naturally to this grammar, and give a trace show how it runs on the string "**id** $\langle$**id id**$\rangle$"

Solution: With q_0 as start state, and q_2 as final state:

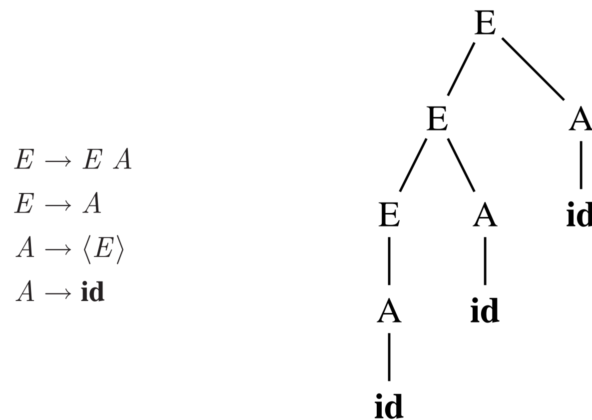
$$
\begin{array}{rclrcl}
\delta(q_0, \epsilon, Z) & \mapsto & q_1/EZ & \delta(q_1, \epsilon, E) & \mapsto & q_1/E\ E \\
\delta(q_1, \langle, \langle) & \mapsto & q_1/\epsilon & \delta(q_1, \epsilon, E) & \mapsto & q_1/\langle E \rangle \\
\delta(q_1, \rangle, \rangle) & \mapsto & q_1/\epsilon & \delta(q_1, \epsilon, E) & \mapsto & q_1/\mathbf{id} \\
\delta(q_1, \mathbf{id}, \mathbf{id}) & \mapsto & q_1/\epsilon & \delta(q_1, \epsilon, Z) & \mapsto & \underline{q_2}/\epsilon
\end{array}
$$

$$
\begin{array}{llll}
(q_0, \textbf{id}\ \langle \textbf{id id} \rangle, Z) & \Rightarrow (q_1, \textbf{id}\ \langle \textbf{id id} \rangle, EZ) & \Rightarrow (q_1, \textbf{id}\ \langle \textbf{id id} \rangle, EEZ) & \\
 & \Rightarrow (q_1, \textbf{id}\ \langle \textbf{id id} \rangle, \textbf{id}EZ) & \Rightarrow (q_1, \langle \textbf{id id} \rangle, EZ) & \\
 & \Rightarrow (q_1, \langle \textbf{id id} \rangle, \langle E \rangle Z) & \Rightarrow (q_1, \textbf{id id} \rangle, E \rangle Z) & \\
 & \Rightarrow (q_1, \textbf{id id} \rangle, EE \rangle Z) & \Rightarrow (q_1, \textbf{id id} \rangle, \textbf{id}E \rangle Z) & \\
 & \Rightarrow (q_1, \textbf{id} \rangle, E \rangle Z) & \Rightarrow (q_1, \textbf{id} \rangle, \textbf{id} \rangle Z) & \\
 & \Rightarrow (q_1, \rangle, \rangle Z) & \Rightarrow (q_1, \epsilon, Z) & \Rightarrow (q_2, \epsilon, \epsilon)
\end{array}
$$

(3) [2 marks] Modify the grammar so that it accepts the same language, and is not ambiguous; make sure that it leads to parsing expressions as in Haskell, or in the λ-calculus.

Solution: We can use a new non-terminal A to mean an "atomic" expression, with the following grammar, and we show a parse tree for **id id id**.

$$
\begin{array}{l}
E \rightarrow E\ A \\
E \rightarrow A \\
A \rightarrow \langle E \rangle \\
A \rightarrow \textbf{id}
\end{array}
$$



(4) [2 marks] The original grammar is left-recursive. Modify it so as to give a grammar which generates L but is not left-recursive, and where the corresponding pushdown automaton requires only one lookahead symbol to choose the correct transition. Note: lookahead can detect end-of-input, use the symbol '$\dashv$' for end-of-input. For each non-terminal which is the left-hand side of more than one production, state what are the appropriate lookahead symbols for each of the productions.

Solution: We can avoid left-recursion using the grammar on the left. Note the similarity to part (3), and that L has the property that the reverse of any sentence in L is also in L. But that grammar won't have the one-symbol lookahead property because two productions for E start with A. The grammar on the right solves this problem also. The lookahead symbols are shown in square brackets. The lookahead symbol for $E' \rightarrow \epsilon$ is '$\rangle$' or '$\dashv$' because only those can follow an E'.

$$
\begin{array}{l}
E \rightarrow A\ E \\
E \rightarrow A \\
A \rightarrow \langle E \rangle \\
A \rightarrow \textbf{id}
\end{array}
\qquad\qquad
\begin{array}{ll}
E \rightarrow A\ E' & \\
E' \rightarrow A\ E' & [\ \langle,\ \textbf{id}\ ] \\
E' \rightarrow \epsilon & [\ \rangle, \dashv\ ] \\
A \rightarrow \langle E \rangle & [\ \langle\ ] \\
A \rightarrow \textbf{id} & [\ \textbf{id}\ ]
\end{array}
$$

3 Lambda Calculus [4 marks]

Given the following definitions

$$S = \lambda\, x\; y\; z.\; x\; z\; (y\; z) \qquad K = \lambda\, x\; y.\; x \qquad I = \lambda\, x.\; x$$
$$B = \lambda\, f\; g\; x.\; f\; (g\; x) \qquad C = \lambda\, f\; x\; y.\; f\; x\; y$$

evaluate the following (by substituting for S, K, I, B, C and performing all possible β-reductions)

(1) $C\; S\; I$

(2) $\lambda\, f\; x.\; S\; f\; (K\; x)$

(3) $B\; (S\; B)\; K$

The file `solution.hs` shows these intermediate steps in Haskell; you can check that all the steps for each problem have the same type.

Version according to the definition of C in the question paper

$$\begin{aligned}
C\; S\; I &= (\lambda\, f\; x\; y.\; f\; x\; y)\; S\; I \\
&= (\lambda\, x\; y.\; S\; x\; y)\; I \\
&= \lambda\, y.\; S\; I\; y \\
&= \lambda\, y.\; (\lambda\, x\; y\; z.\; x\; z\; (y\; z))\; I\; y \\
&= \lambda\, y.\; (\lambda\, x\; y'\; z.\; x\; z\; (y'\; z))\; I\; y \\
&= \lambda\, y.\; (\lambda\, y'\; z.\; I\; z\; (y'\; z))\; y \\
&= \lambda\, y.\; (\lambda\, z.\; I\; z\; (y\; z)) \\
&= \lambda\, y.\; (\lambda\, z.\; (\lambda\, x.\; x)\; z\; (y\; z)) \\
&= \lambda\, y.\; (\lambda\, z.\; z\; (y\; z))
\end{aligned}$$

Version according to the definition of C in `comb.hs`

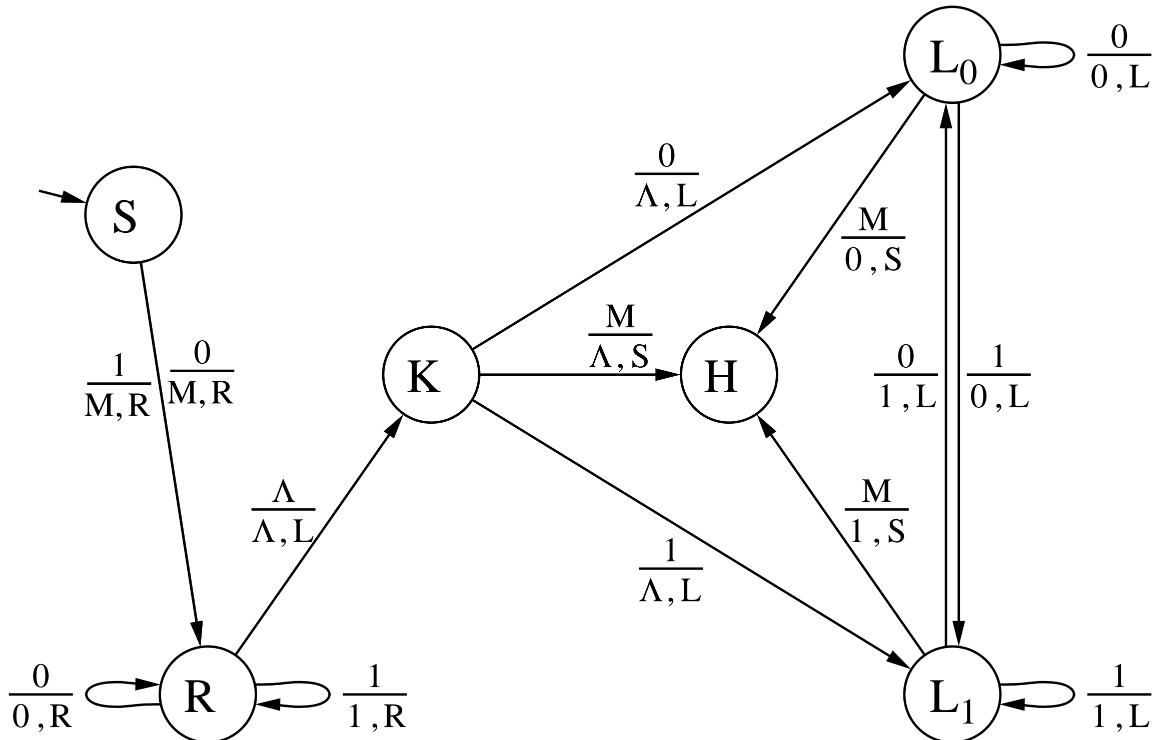
$$\begin{aligned}
C\; S\; I &= (\lambda\, f\; x\; y.\; f\; y\; x)\; S\; I \\
&= (\lambda\, x\; y.\; S\; y\; x)\; I \\
&= \lambda\, y.\; S\; y\; I \\
&= \lambda\, y.\; (\lambda\, x\; y\; z.\; x\; z\; (y\; z))\; y\; I \\
&= \lambda\, y.\; (\lambda\, x\; y'\; z.\; x\; z\; (y'\; z))\; y\; I \\
&= \lambda\, y.\; (\lambda\, y'\; z.\; y\; z\; (y'\; z))\; I \\
&= \lambda\, y.\; (\lambda\, z.\; y\; z\; (I\; z)) \\
&= \lambda\, y.\; (\lambda\, z.\; y\; z\; ((\lambda\, x.\; x)\; z)) \\
&= \lambda\, y.\; (\lambda\, z.\; y\; z\; z) \\
&= \lambda\, y\; z.\; y\; z\; z
\end{aligned}$$

$$
\begin{aligned}
\lambda f\ x.\ S\ f\ (K\ x) &= \lambda f\ x.\ (\lambda\, x\ y\ z.\ x\ z\ (y\ z))\ f\ (K\ x) \\
&= \lambda f\ x.\ (\lambda\, y\ z.\ f\ z\ (y\ z))\ (K\ x) \\
&= \lambda f\ x.\ (\lambda\, z.\ f\ z\ (K\ x\ z)) \\
&= \lambda f\ x.\ (\lambda\, z.\ f\ z\ ((\lambda\, x'\ y.\ x')\ x\ z)) \\
&= \lambda f\ x.\ (\lambda\, z.\ f\ z\ ((\lambda\, y.\ x)\ z)) \\
&= \lambda f\ x.\ (\lambda\, z.\ f\ z\ x) \\
&= \lambda f\ x\ z.\ f\ z\ x
\end{aligned}
$$

$$
\begin{aligned}
B\ (S\ B)\ K &= (\lambda f\ g\ x.\ f\ (g\ x))\ (S\ B)\ K \\
&= (\lambda\, g\ x.\ (S\ B)\ (g\ x))\ K \\
&= \lambda\, x.\ S\ B\ (K\ x) \\
&= \lambda\, x.\ (\lambda\, x'\ y\ z.\ x'\ z\ (y\ z))\ B\ (K\ x) \\
&= \lambda\, x.\ (\lambda\, y\ z.\ B\ z\ (y\ z))\ (K\ x) \\
&= \lambda\, x.\ (\lambda\, z.\ B\ z\ (K\ x\ z)) \\
&= \lambda\, x.\ (\lambda\, z.\ (\lambda f\ g\ x'.\ f\ (g\ x'))\ z\ (K\ x\ z)) \\
&= \lambda\, x.\ (\lambda\, z.\ (\lambda\, g\ x'.\ z\ (g\ x'))\ (K\ x\ z)) \\
&= \lambda\, x.\ (\lambda\, z.\ (\lambda\, x'.\ z\ (K\ x\ z\ x'))) \\
&= \lambda\, x.\ (\lambda\, z.\ (\lambda\, x'.\ z\ ((\lambda\, x''\ y.\ x'')\ x\ z\ x'))) \\
&= \lambda\, x.\ (\lambda\, z.\ (\lambda\, x'.\ z\ ((\lambda\, y.\ x)\ z\ x'))) \\
&= \lambda\, x.\ (\lambda\, z.\ (\lambda\, x'.\ z\ (x\ x'))) \\
&= \lambda\, x\ z\ x'.\ z\ (x\ x')
\end{aligned}
$$

4 Turing Machines [5 marks]

Construct a Turing Machine that deletes the symbol at the current head position, by moving all the symbols to the right of this position one tape square to the left, and returns the head to the original position.



Overview:
- Overwrite current symbol with M
- Move to right-hand end of string (state R)
- Move left, until reaching M; each step remember the symbol seen by going to state L_0 or L_1; write the symbol previously seen (which is 0 or 1 if you are in state L_0 or L_1).

State S: Initial: write M

State R: Go to right hand end of string

State K: If still at symbol M then you started at the right hand end of the string, write blank and halt. Otherwise write blank (you are now beyond the new right hand end of the string) and remember symbol seen by going to L_0 or L_1.

State L_i: Write symbol remembered (i), remember current symbol j by going to L_j. Repeat this until you see M; then halt.

State H: Having seen M, and overwritten it, halt.