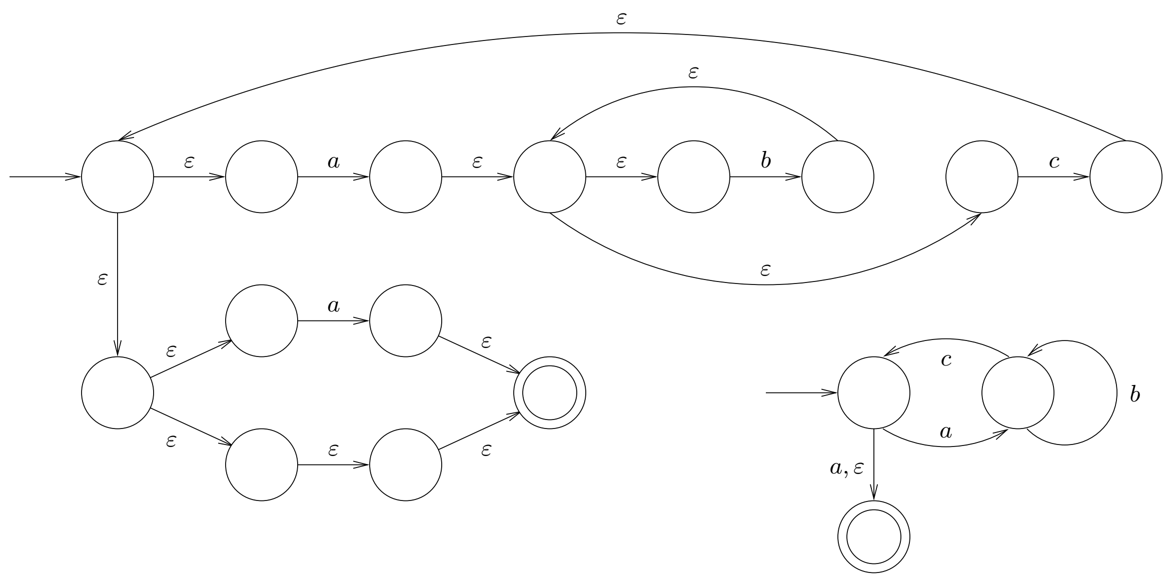
CSCE 531, Spring 2017
Final Exam Answer Key

1. (10 points) Using the method in class or in the text, construct an NFA equivalent to the following regular expression:

   `(ab*c)*a?`

   Answer: Any equivalent NFA between these two extremes is correct:



   The final ε-transition can also be removed by making the start state an accepting state.

2. (20 points) NOTE: Read the entire question before giving your answers.

   Consider the following grammar with start symbol S':

$$\begin{aligned} S' &\to S \\ S &\to iS \\ S &\to iSeS \\ S &\to * \end{aligned}$$

   Using the method described in class or the text, construct the set

$$\{I_0, I_1, I_2, \ldots, I_{11}\}$$

   of states for a canonical LR(1) parser for this grammar, defining the transition function at the same time, including all allowed transitions. Note that in class I denoted the transition function as *trans*. The textbook denotes the same function as *goto*. You may use the letter t to denote this function.

   To ensure a unique correct answer, you must stick to the following rules of order, which mirror the order I used for my example in class:

(a) Give each state as a list of LR(1) items, omitting the brackets.

(b) Give the start state first, and denote it by I_0. Denote other states $I_1, I_2, \ldots$ in the order they are constructed.

(c) List the kernel items first in each state. List additional nonkernal items in the order that they enter the closure.

(d) For each $i \geq 0$, define all transitions out of I_i before defining those out of I_{i+1}.

(e) When finding the transitions out of a state, or computing a closure, consider each item of the state in the order you listed it.

(f) Do not list the empty set as a state.

(g) Collapse LR(1) items with the same core and different lookaheads using the "/" notation, e.g., $S \to i.S \quad \$/e$

Answer: This is the unique correct answer, up to re-ordering of the look-ahead symbols.

$$\begin{array}{ll}
I_0 = \text{start state}: & \\
\quad S' \to .S & \$ \\
\quad S \to .iS & \$ \\
\quad S \to .iSeS & \$ \\
\quad S \to .* & \$
\end{array}$$

$$\begin{array}{ll}
I_1 = t(I_0, S): & \\
\quad S' \to S. & \$
\end{array}$$

$$\begin{array}{ll}
I_2 = t(I_0, i): & \\
\quad S \to i.S & \$ \\
\quad S \to i.SeS & \$ \\
\quad S \to .iS & \$/e \\
\quad S \to .iSeS & \$/e \\
\quad S \to .* & \$/e
\end{array}$$

$$\begin{array}{ll}
I_3 = t(I_0, *): & \\
\quad S \to *. & \$
\end{array}$$

$$\begin{array}{ll}
I_4 = t(I_2, S): & \\
\quad S \to iS. & \$ \\
\quad S \to iS.eS & \$
\end{array}$$

$$\begin{array}{ll}
I_5 = t(I_2, i): & \\
\quad S \to i.S & \$/e \\
\quad S \to i.SeS & \$/e \\
\quad S \to .iS & \$/e \\
\quad S \to .iSeS & \$/e \\
\quad S \to .* & \$/e
\end{array}$$

$$\begin{array}{ll}
I_6 = t(I_2, *): & \\
\quad S \to *. & \$/e
\end{array}$$

$$\begin{array}{ll}
I_7 = t(I_4, e): & \\
\quad S \to iSe.S & \$ \\
\quad S \to .iS & \$ \\
\quad S \to .iSeS & \$ \\
\quad S \to .* & \$
\end{array}$$

$$\begin{array}{ll}
I_8 = t(I_5, S): & \\
\quad S \to iS. & \$/e \\
\quad S \to iS.eS & \$/e
\end{array}$$

$(I_5 = t(I_5, i))$

$(I_6 = t(I_5, *))$

$$\begin{array}{ll}
I_9 = t(I_7, S): & \\
\quad S \to iSeS. & \$
\end{array}$$

$(I_2 = t(I_7, i))$

$(I_3 = t(I_7, *))$

$$\begin{array}{ll}
I_{10} = t(I_8, e): & \\
\quad S \to iSe.S & \$/e \\
\quad S \to .iS & \$/e \\
\quad S \to .iSeS & \$/e \\
\quad S \to .* & \$/e
\end{array}$$

$$\begin{array}{ll}
I_{11} = t(I_{10}, S): & \\
\quad S \to iSeS. & \$/e
\end{array}$$

$(I_5 = t(I_{10}, i))$

$(I_6 = t(I_{10}, *))$

Note that we only list a core item once in each state, regardless of how many lookaheads it has, but as this is a *canonical* LR(1) parser, we do not merge states with common cores. For example, I_2 and I_5 are different states, although they have the same core. Similarly with states I_4 and I_8, etc. If this had been an LALR(1) parser, then these pairs of states would be merged into single states, combining the loohaheads for each core item.

3. (20 points) Consider a bottom-up grammar in BNF for control flow constructs in a typical C-like programming language

$$\begin{array}{rcl}
\langle \textit{start} \rangle & ::= & \langle \textit{stmt} \rangle \\
\langle \textit{stmt} \rangle & ::= & \textbf{if}\ e\ \textbf{then}\ \langle \textit{stmt} \rangle_1 \\
\langle \textit{stmt} \rangle & ::= & \textbf{if}\ e\ \textbf{then}\ \langle \textit{stmt} \rangle_1\ \textbf{else}\ \langle \textit{stmt} \rangle_2 \\
\langle \textit{stmt} \rangle & ::= & \textbf{while}\ e\ \textbf{do}\ \langle \textit{stmt} \rangle_1 \\
\langle \textit{stmt} \rangle & ::= & \textbf{begin}\ \langle \textit{stmt\_list} \rangle\ \textbf{end} \\
\langle \textit{stmt} \rangle & ::= & \textbf{break} \\
\langle \textit{stmt} \rangle & ::= & \textbf{other} \\
\langle \textit{stmt\_list} \rangle & ::= & \texttt{/* empty */} \\
\langle \textit{stmt\_list} \rangle & ::= & \langle \textit{stmt\_list} \rangle_1\ \texttt{';'}\ \langle \textit{stmt} \rangle
\end{array}$$

A **break** statement is meant to unconditionally jump to the end of the closest surrounding **while**-loop, if there is one. (Note that a **break** statement might jump out of an arbitrary number of nested **if** and **begin**–**end** statements.)

Add semantic rules to compute the following inherited attribute: $\langle \textit{stmt} \rangle.\textit{break}$ is a string that is the destination label to which to jump if a break statement is encountered within $\langle \textit{stmt} \rangle$. If the statement is not inside any **while** statement, then use the special value NIL for this attribute. You may assume a function `newSymbol()` is available that returns a fresh label every time it is called.

Also add semantic actions that

- emit an unconditional jump to the proper destination whenever a **break** statement is encountered (if there is no proper destination, issue an error message), and
- emit the corresponding break destination as a label (i.e., followed by a colon) after each **while** statement.

You may assume a function `emit(...)` is available that emits whatever string is passed to it.

You may define any additional attributes that you find helpful, but you must pass all data around on the semantic stack, i.e., you may not declare or use any external (global) variables or data structures. Also, your syntax-directed translation scheme must be L-attributed, and no intermediate actions are allowed.

Answer:

$\langle \mathit{start} \rangle$	::=	$\langle \mathit{stmt} \rangle$	$\langle \mathit{stmt} \rangle.\mathit{break} := \textsc{Nil}$
$\langle \mathit{stmt} \rangle$	::=	**if** e **then** $\langle \mathit{stmt} \rangle_1$	$\langle \mathit{stmt} \rangle_1.\mathit{break} := \langle \mathit{stmt} \rangle.\mathit{break}$
$\langle \mathit{stmt} \rangle$	::=	**if** e **then** $\langle \mathit{stmt} \rangle_1$ **else** $\langle \mathit{stmt} \rangle_2$	$\langle \mathit{stmt} \rangle_1.\mathit{break} := \langle \mathit{stmt} \rangle.\mathit{break}$ $\langle \mathit{stmt} \rangle_2.\mathit{break} := \langle \mathit{stmt} \rangle.\mathit{break}$
$\langle \mathit{stmt} \rangle$	::=	**while** e **do** $\langle \mathit{stmt} \rangle_1$	$\langle \mathit{stmt} \rangle_1.\mathit{break} :=$ `newSymbol()` `emit(`$\langle \mathit{stmt} \rangle_1.\mathit{break} +$ `":")`
$\langle \mathit{stmt} \rangle$	::=	**begin** $\langle \mathit{stmt\_list} \rangle$ **end**	$\langle \mathit{stmt\_list} \rangle.\mathit{break} := \langle \mathit{stmt} \rangle.\mathit{break}$

$\langle \mathit{stmt} \rangle$	::=	**break**	**if** $\langle \mathit{stmt} \rangle.\mathit{break} \neq \textsc{Nil}$ **then**
			   `emit("jump "` $+ \langle \mathit{stmt} \rangle.\mathit{break}$`)`
			else
			   `emit("error:  not inside loop")`

$\langle \mathit{stmt} \rangle$	::=	**other**	
$\langle \mathit{stmt\_list} \rangle$	::=	`/* empty */`	
$\langle \mathit{stmt\_list} \rangle$	::=	$\langle \mathit{stmt\_list} \rangle_1$ `';'` $\langle \mathit{stmt} \rangle$	$\langle \mathit{stmt\_list} \rangle_1.\mathit{break} := \langle \mathit{stmt\_list} \rangle.\mathit{break}$ $\langle \mathit{stmt} \rangle.\mathit{break} := \langle \mathit{stmt\_list} \rangle.\mathit{break}$

Instead of using "+" for string concatenation, successive calls to `emit()` can occur in a row.

4. (10 points) Consider the following C declaration: `double a[9][10][5];`. Assuming that a double is eight bytes and that the base address of a is 1000, find the base address of the double variable `a[2][8][5]`.

Answer: 2160.

By way of explanation (not required), consider the following table:

Datum	`Element Type`	`Size`
`a[9][10][5]`	double	8
`a[9][10]`	double[5]	40
`a[9]`	double[5][10]	400

The 9 is not used (it only determines the total size of the array). So the address is

$$1000 + 2 \cdot 400 + 8 \cdot 40 + 5 \cdot 8 = 2160\ .$$

The fact that the last index is out of range is not relevant; the address is calculated the same way and no range checking is done. The expression `a[2][9][0]` has the same address.