Programming Language Concepts (20013)
Preparation for the Final Exam

This is a collection of 12 questions which are close to what is to be expected within the final exams for PLC. Attached are sample solutions.

List of Questions
Page 1: Regular Expressions
Page 3: BNF Grammar
Page 5: Lambda Calculus
Page 7: Denotational Semantics
Page 9: Scheme / Lists
Page 11: Scheme / Nested Lists
Page 13: Perl / Regular Expressions
Page 15: Perl / Hashes
Page 18: Prolog / Simple Rules and Recursion
Page 20: Prolog / Generation of Primes
Page 22: Prolog / Logical Problem
Page 25: Procedure Calls
Question 1 (Regular Expressions)

Consider following regular expression:
\[ [0-9][0-9]*\left(\text{.\ [0-9]*}\right)?(E[+-]?[0-9][0-9]*)? \]

(a) Specify the associated DFA in graphical form. You are allowed to tag arrows with multiple symbols, e.g. 0-9 for the digits 0 to 9. Please mark the start state and all the final states as such. You do not need to consider error states.

(b) Please check if following sequences are sentences that are recognized by the regular expression above:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Sentence?</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>Yes</td>
</tr>
<tr>
<td>23*5+2</td>
<td>Yes</td>
</tr>
<tr>
<td>123E4</td>
<td>Yes</td>
</tr>
<tr>
<td>623.</td>
<td>Yes</td>
</tr>
<tr>
<td>1.0</td>
<td>Yes</td>
</tr>
<tr>
<td>755.01E-40</td>
<td>Yes</td>
</tr>
<tr>
<td>.234E+10</td>
<td>No</td>
</tr>
</tbody>
</table>
(a) Final states are marked with a double circle. The beginning state is marked with “S”:

(b)

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Sentence?</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>X</td>
</tr>
<tr>
<td>23*5+2</td>
<td>X</td>
</tr>
<tr>
<td>123E4</td>
<td>X</td>
</tr>
<tr>
<td>623.</td>
<td>X</td>
</tr>
<tr>
<td>1.0</td>
<td>X</td>
</tr>
<tr>
<td>755.01E-40</td>
<td>X</td>
</tr>
<tr>
<td>.234E+10</td>
<td>X</td>
</tr>
</tbody>
</table>
Question 2 (BNF Grammar)

Consider following BNF grammar. The start symbol is $\langle$Pedigree$\rangle$:

\[
\begin{align*}
\langle\text{Pedigree}\rangle & \rightarrow \langle\text{Name}\rangle \\
\langle\text{Pedigree}\rangle & \rightarrow \langle\text{Name}\rangle\langle\text{Parents}\rangle \\
\langle\text{Parents}\rangle & \rightarrow \langle\text{Person}\rangle\langle\text{Person}\rangle \\
\langle\text{Person}\rangle & \rightarrow () \mid (\langle\text{Pedigree}\rangle) \\
\langle\text{Name}\rangle & \rightarrow \langle\text{Letter}\rangle \mid \langle\text{Letter}\rangle\langle\text{Name}\rangle \\
\langle\text{Letter}\rangle & \rightarrow A \mid \ldots \mid Z \mid a \mid \ldots \mid z
\end{align*}
\]

Please check for each of the following sequences if it is a sentence that is recognized by this grammar. If yes, a formal derivation is to be given. Whitespace is to be ignored.

(a) Charles

(b) Charles ()

(c) Charles (Elizabeth)

(d) Charles (() Philipp)

(e) Charles () (Philipp)

(f) Charles (Elizabeth) (Philipp)
(a) Is a sentence:

\[
\langle \text{Pedigree} \rangle \rightarrow \langle \text{Name} \rangle \\
\quad \rightarrow \text{Charles}
\]

(b) Is not a sentence.

(c) Is not a sentence.

(d) Is not a sentence.

(e) Is a sentence:

\[
\langle \text{Pedigree} \rangle \rightarrow \langle \text{Name}\rangle\langle\text{Parents}\rangle \\
\quad \rightarrow \text{Charles} \langle\text{Parents}\rangle \\
\quad \rightarrow \text{Charles} \langle\text{Person}\rangle\langle\text{Person}\rangle \\
\quad \rightarrow \text{Charles} () \langle\text{Person}\rangle \\
\quad \rightarrow \text{Charles} () \langle\text{Person}\rangle \\
\quad \rightarrow \text{Charles} () (\langle\text{Pedigree}\rangle) \\
\quad \rightarrow \text{Charles} () (\langle\text{Name}\rangle) \\
\quad \rightarrow \text{Charles} () (\langle\text{Philipp}\rangle)
\]

(f) Is a sentence:

\[
\langle \text{Pedigree} \rangle \rightarrow \langle \text{Name}\rangle\langle\text{Parents}\rangle \\
\quad \rightarrow \text{Charles} \langle\text{Parents}\rangle \\
\quad \rightarrow \text{Charles} \langle\text{Person}\rangle\langle\text{Person}\rangle \\
\quad \rightarrow \text{Charles} (\langle\text{Pedigree}\rangle)\langle\text{Person}\rangle \\
\quad \rightarrow \text{Charles} (\langle\text{Name}\rangle)\langle\text{Person}\rangle \\
\quad \rightarrow \text{Charles} (\text{Elizabeth}) (\langle\text{Pedigree}\rangle) \\
\quad \rightarrow \text{Charles} (\text{Elizabeth}) (\langle\text{Name}\rangle) \\
\quad \rightarrow \text{Charles} (\text{Elizabeth}) (\langle\text{Philipp}\rangle)
\]
Question 3 (Lambda Calculus)

Consider following expression of the $\lambda$ calculus:

$((\lambda x. \lambda y. (x)(x)y)\lambda x.x)a$

(a) Which of the variables are bound? Mark all bound variables with arrows to their corresponding functions. Are there any free variables? If yes, circle them.

(b) Derive the expression as far as possible using $\alpha$ conversions and $\beta$ reductions. Take care that each intermediate step is shown and consistent. $\alpha$ conversions must not be omitted if they are required to preserve the meaning of the $\lambda$ expression.
(b) \( \alpha \) conversions are not necessary in this example, just a series of \( \beta \) reductions:

\[
\begin{align*}
((\lambda x. \lambda y. (x)(x)y)\lambda x.x)a \\
\rightarrow (\lambda y. (\lambda x.x)(\lambda x.x)y)a \\
\rightarrow (\lambda x.x)(\lambda x.x)a \\
\rightarrow (\lambda x.x)a \\
\rightarrow a
\end{align*}
\]
Question 4 (Denotational Semantics)

Consider following language for turtle graphics:

\[
\begin{align*}
\langle \text{Program} \rangle & \rightarrow \langle \text{StatementSequence} \rangle \\
\langle \text{StatementSequence} \rangle & \rightarrow \epsilon \mid \langle \text{Statement} \rangle \langle \text{StatementSequence} \rangle \\
\langle \text{Statement} \rangle & \rightarrow + \mid - \mid F
\end{align*}
\]

The initial position is \((0, 0)\), the initial direction \(\Delta\) is \((0, -1)\). The statements have following meaning:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>add (\Delta) to the current position</td>
</tr>
<tr>
<td>+</td>
<td>set (\Delta) to (L\Delta)</td>
</tr>
<tr>
<td>-</td>
<td>set (\Delta) to (R\Delta)</td>
</tr>
</tbody>
</table>

where \(L := \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}\) and \(R := \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\).

Be \(I\) the domain for whole numbers. The domain for positions and \(\Delta\) is \(P = I \times I\). Hence, the domain for the entire state can be defined as \(S = P \times P\) where the first \(P\) defines the position and the second the current \(\Delta\). The semantic function for statements is of the type \(\text{Statement} \rightarrow S \rightarrow S\).

(a) Write the semantic equation of \(S[ F ]\).

(b) Write the semantic equation of \(S[ + ]\).

(c) Write the semantic equation of \(S[ S_1 S_2 ]\).
(a) 
\[ S[F] = \lambda \sigma. (\sigma \downarrow 1 + \sigma \downarrow 2, \sigma \downarrow 2) \]

(b) 
\[ S[+] = \lambda \sigma. (\sigma \downarrow 1, L\sigma \downarrow 2) \]

(c) 
\[
S[S_1S_2] = S[S_2]S[S_1] \\
= \lambda \sigma. S[S_2]S[S_1] \sigma
\]
Question 5 (Scheme / Lists)

Write a Scheme procedure `compute-sequence` that takes an operator which computes a number from a list, a count, and an initial list. As often as specified by the count, the procedure should invoke the operator on the list which is to be extended *at the front* with the result.

Following example shows how this procedure can be used to compute Fibonacci numbers:

```
guile> (compute-sequence (lambda (x) (+ (car x) (cadr x))) 10 '(1 1))
(144 89 55 34 21 13 8 5 3 2 1 1)
guile>
```
(define (compute-sequence operator count sequence)
  (if (> count 0)
      (compute-sequence operator (- count 1)
        (cons (operator sequence) sequence)
      )
      sequence
  )
)

Question 6 (Scheme / Nested Lists)

Alice travels a lot and keeps her travel plans in following form:

```scheme
(define journeys
  '("June 5" "Stuttgart" "Ulm" "Augsburg" "Munich")
  ("June 6" "Munich" "Salzburg" "Vienna")
  ("June 7" "Vienna" "Bratislava" "Prague")
  ("June 8" "Prague")
  ("June 9" "Prague" "Dresden" "Leipzig")
)
```

Write a Scheme procedure that, given such a list and the name of a town, returns a list of dates on which Alice visits that town. Example: For the above list and “Vienna”, a list of two dates is to be returned: “June 6”, and “June 7”. 
(define (get-dates-of-visits journeys town)
  (if (eq? journeys '())
      '()
    ; else
    (let (
      (more-dates (get-dates-of-visits
                    (cdr journeys) town))
        )
      (if (member town (cdar journeys))
          (cons (caar journeys) more-dates)
        ; else
        more-dates
      )
    )
  )
)

Note that member belongs to the set of standard procedures of R5RS. member uses equal? to check whether its first parameter appears anywhere in the given list. If yes, the list beginning from that element is returned, otherwise #f. member can be easily written:

(define (member object list)
  (if (eq? list '())
      #f
    ; else
    (if (equal? object (car list))
        list
      ; else
      (member object (cdr list))
    )
  )
)
Question 7 (Perl / Regular Expressions)

Write a Perl program that prints all input lines that begin with a CS account name followed by a space. A CS account name consists of three lower case letters, followed by four digits.
#!/usr/local/bin/perl -w

use strict;

while(<>) {
    print if /^\[a-z]\{3\}\d\{4\}\s/;
}

The notation \{3\} or \{4\} gives a count how often the previous expression has to appear. You can also specify ranges like \{3,6\}. Without this notation, the regular expression becomes longer:

#!/usr/local/bin/perl -w

use strict;

while(<>) {
    print if /^\[a-z\][a-z]\{3\}\d\d\d\d\s/;
}

Question 8 (Perl / Hashes)

Assume you have a log of orders that were made using your online shop. This log places each order in a separate line that consists of three colon-separated fields: the name of the customer, the ordered quantity, and the product name. Here is an example:

StarshipHunter:1:Freighter
Rastur:1:Space Spice
ThePlunderers:2:Pedigree Bolts
StarshipHunter:2:Clipper
Rastur:3:Clipper
ThePlunderers:1:Spy Eye

Write a small Perl script that emits recommendations. For a given product name (on the command line) it should print all other products that were bought from a customer who ordered this product. The log is passed via standard input and its correctness may be safely assumed.
Example: For the product name “Clipper”, the products “Freighter” and “Space Spice” should be printed as they both were bought by someone who has ordered “Clipper”. 
I took this question from a final exam I gave once to a class at my German home university and realized too late that it is best solved with hashes of hashes, something which was not covered within this course. Anyway, here is a solution with hashes of hashes:

```perl
#!/usr/local/bin/perl -w

use strict;

my $cmdname = $0; $cmdname =~ s{.*/}{};
my $usage = "Usage: $cmdname product\n";
die $usage unless @ARGV == 1;
my $product = shift;

my %customer_by_product = ();
my %product_by_customer = ();
while(<>) {
    chomp;
    my ($customer, $quantity, $product) = split /:/;
    $customer_by_product{$product}->{$customer} += $quantity;
    $product_by_customer{$customer}->{$product} += $quantity;
}
exit 0 unless defined $customer_by_product{$product};

my %recommendations = ();
foreach my $customer (keys %{$customer_by_product{$product}}) {
    foreach my $product (keys %{$product_by_customer{$customer}}) {
        $recommendations{$product} = 1;
    }
}
delete $recommendations{$product};
print $_, "\n" foreach (sort keys %recommendations);
```

Both hashes, `%customer_by_product` and `%product_by_customer` have as values references to other hashes. Perl is very permissive in constructing these data structures. You can simply reference keys which have never been used before:
$customer_by_product{$product}->{$customer} += $quantity;

This is following in C++ STL terms:

typedef string Customer;
typedef string Product;
map<Product, map<Customer, int> > customer_by_product;

// ...

customer_by_product[product][customer] += quantity;

However, Perl nicely initializes the value to 0, C++ doesn’t.
I promise that the Perl question regarding hashes in the real final exam will
be simpler.
Some final remarks: The product names are taken from the classic board
game “Merchants of Venus” by Avalon Hill. And the recommendations at
Amazon triggered me to come up with this question.
Question 9 (Prolog / Simple Rules and Recursion)

A family has collected a large number of facts in form of following Prolog predicates:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>father(X,Y)</td>
<td>X is the father of Y</td>
</tr>
<tr>
<td>mother(X,Y)</td>
<td>X is the mother of Y</td>
</tr>
<tr>
<td>male(X)</td>
<td>X is male</td>
</tr>
<tr>
<td>female(X)</td>
<td>X is female</td>
</tr>
</tbody>
</table>

(a) Write a predicate \texttt{conflict(X)} that succeeds if \(X\) can be bound to a family member who is father but not male or a mother but not female.

(b) Write a predicate \texttt{brother(X,Y)} that succeeds if \(X\) and \(Y\) are both male and share a parent.

(c) Write a predicate \texttt{relative(X,Y)} if \(X\) and \(Y\) share some ancestor.
(a)

\[
\text{conflict}(X) :- \text{father}(X), \text{female}(X).
\text{conflict}(X) :- \text{mother}(X), \text{male}(X).
\]

(b)

\[
\text{parent}(X,Y) :- \text{father}(X,Y).
\text{parent}(X,Y) :- \text{mother}(X,Y).
\text{brother}(X,Y) :-
    \quad \text{male}(X), \text{male}(Y), X \neq Y, \text{parent}(P,X), \text{parent}(P,Y).
\]

(c)

\[
\text{/* ancestor}(X,Y) : \text{X is an ancestor of Y */}
\text{ancestor}(X,Y) :- \text{parent}(X,Y).
\text{ancestor}(X,Y) :- \text{parent}(Z,Y), \text{ancestor}(X,Z).
\text{relative}(X,Y) :- \text{ancestor}(A,X), \text{ancestor}(A,Y), X \neq Y.
\]
Question 10 (Prolog / Generation of Primes)

Write a Prolog predicate —prime(N)— that generates prime numbers:

?- prime(N).

N = 2 ;
N = 3 ;
N = 5 ;
N = 7 ;
N = 11 ;
N = 13

Yes

?-

Hints: You are allowed to use following predicates:

| natural(N)      | generates a natural number N |
| natural(N,L,U)  | generates a natural number within [L..U] |

The negation may be useful: not(some_predicate(...)) succeeds if some_predicate(...) fails.
prime(2).
prime(N) :- natural(N), N > 2, not(dividable(N)).
dividable(N) :- U is N - 1, natural(D, 2, U), R is N mod D, R = 0.
Question 11 (Prolog / Logical Problem)

Jim, a wealthy broker, was murdered. Inspector Perry comes to the scene and it becomes quickly clear that the murderer had the key to the room Jim was in and knew the safe combination. Only four other persons were on the scene at the time of the crime: Ann, his business partner, Bob, the gardener, Carol, his sister, and Dan, his brother. Following facts were uncovered by Inspector Perry:

- (1) Jim had handed out a key to Carol.
- (2) Jim shared the safe combination with Ann.
- (3) Carol would hand out the key to Bob if she would have it.
- (4) Ann would hand out the safe combination to Dan, and Dan would pass on the key to Ann if he had it.
- (5) Dan would share the safe combination with anyone who is willing to give the key to someone else (not necessarily him).

Following facts can be easily expressed in Prolog:

```prolog
person(ann).
person(bob).
person(carol).
person(dan).

has_key(carol).
has_combination(ann).

/* gives_key(X,Y) :-
   X is willing to give a key to Y if X has it */
gives_key(carol,bob).
gives_key(dan,ann).

/* shares_combination(X,Y) :-
   X is willing to share the safe combination with Y if X has it */
shares_combination(ann,dan).
```

(a) Write a predicate \texttt{pair(X,Y)} that succeeds if \( X \) and \( Y \) are two different persons, i.e. \texttt{pair(bob,carol)} succeeds but \texttt{pair(dan,dan)} does not.

(b) Write the recursive clause of \texttt{has_key(X)} that succeeds if \( X \) is somehow able to obtain a key through others. You can safely assume the absence of cyclic relations which could cause an endless recursion. \texttt{has_combination(X)} has a similar recursive clause that does not need to be written down here.

(c) Write the clause of \texttt{shares_combination} that represents the fact (5).

(d) Write the clause \texttt{suspicious(X)} that succeeds for every \texttt{person(X)} who could have access to the key and to the safe combination.
(a)

\[
\text{pair}(X,Y) :- \text{person}(X), \text{person}(Y), X \neq Y.
\]

(b)

\[
\text{has_key}(X) :- \\
\quad \text{pair}(X,Y), \text{gives_key}(Y,X), \text{has_key}(Y).
\]

(c)

\[
\text{shares_combination}(\text{dan},Y) :- \\
\quad \text{pair}(Y,\text{dan}), \text{pair}(Y,X), \text{gives_key}(Y, X).
\]

(d)

\[
\text{suspicious}(X) :- \text{has_key}(X), \text{has_combination}(X).
\]

And for those who have still not solved it:

2 \text{ ?- suspicious}(X).

X = carol ;

No

3 \text{ ?-}
Question 12 (Procedure Calls)

For the following example, assume Scheme syntax and semantics (including lexical scoping) but with varying parameter passing methods.

```scheme
(define (parameter-demo)
    (define a 1)
    (define b 2)
    (define (f1 a b)
        (display "f1: a = ")(display a)
        (display ", b = ")(display b)
        (newline)
        (set! a b)
    )
    (define (f2 i j)
        (define b 3)
        (define (f3 a)
            (display "f3: a = ")(display a)
            (newline)
            (f1 a b)
            (display "f3: i = ")(display i)
            (newline)
        )
        (display "f2: i = ")(display i)
        (display ", j = ")(display j)
        (newline)
        (f3 i)
        (display "f2: a = ")(display a)
        (display ", b = ")(display b)
        (newline)
    )
    (f2 a b)
)
(parameter-demo)
```
(a) What is the output of the program in case of call-by-value?

<table>
<thead>
<tr>
<th>function</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f3</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>f3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>f2</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

(b) What is the output of the program in case of call-by-reference?

<table>
<thead>
<tr>
<th>function</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f3</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>f3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>f2</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

(c) What is the output of the program in case of call-by-value-return?

<table>
<thead>
<tr>
<th>function</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f3</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>f3</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>f2</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
(a)

<table>
<thead>
<tr>
<th>Function</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2</td>
<td>i = 1, j = 2</td>
</tr>
<tr>
<td>f3</td>
<td>a = 1</td>
</tr>
<tr>
<td>f1</td>
<td>a = 1, b = 3</td>
</tr>
<tr>
<td>f3</td>
<td>i = 1</td>
</tr>
<tr>
<td>f2</td>
<td>a = 1, b = 3</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>Function</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2</td>
<td>i = 1, j = 2</td>
</tr>
<tr>
<td>f3</td>
<td>a = 1</td>
</tr>
<tr>
<td>f1</td>
<td>a = 1, b = 3</td>
</tr>
<tr>
<td>f3</td>
<td>i = 3</td>
</tr>
<tr>
<td>f2</td>
<td>a = 3, b = 3</td>
</tr>
</tbody>
</table>

(c)

<table>
<thead>
<tr>
<th>Function</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2</td>
<td>i = 1, j = 2</td>
</tr>
<tr>
<td>f3</td>
<td>a = 1</td>
</tr>
<tr>
<td>f1</td>
<td>a = 1, b = 3</td>
</tr>
<tr>
<td>f3</td>
<td>i = 1</td>
</tr>
<tr>
<td>f2</td>
<td>a = 1, b = 3</td>
</tr>
</tbody>
</table>