Banner number: Name:

Final Exam

CSCI 3136: Principles of Programming Languages

April 17, 2019

Group 1		Group 2		Group 3		
Q1.1 (9)		Q2.1 (5)		Q3.1 (5)		$\sum$ (70)
Q1.2 (6)		Q2.2 (10)		Q3.2 (5)		
Q1.3 (10)		Q2.3 (12)		Q3.3 (8)		
$\sum$		$\sum$		$\sum$		

Instructions:

- Provide your answer in the box after each question. If you absolutely need extra space, use the backs of the pages; but try to avoid it. Keep your answers short and to the point.
- You are not allowed to use a cheat sheet.
- Make sure your answers are clear and legible. If I can't decipher an answer or follow your train of thought with reasonable effort, you'll receive 0 marks for your answer.
- Read every question carefully before answering.
- Do not forget to write your banner number and name on the top of this page.
- This exam has 13 pages, including this title page. Notify me immediately if your copy has fewer than 13 pages.

Question 1.1: Formal languages and automata — 9 marks

(a) Formally define what a deterministic finite automaton is.

A deterministic finite automaton is a tuple $M = (\Sigma, S, \delta, s_0, F)$. Σ is a finite alphabet. S is a finite set of states. $\delta : S \times \Sigma \to S$ is a transition function. $s_0 \in S$ is the start state. $F \subseteq S$ is a set of accepting states.

(b) Formally define the language decided by a deterministic finite automaton.

A DFA $M = (\Sigma, S, \delta, s_0, F)$ decides the language $\mathcal{L} = \{\sigma \in \Sigma^* \mid \delta^*(s_0, \sigma) \in F\}$, where $\delta^*(s, \varepsilon) = s$ and $\delta^*(s, x\sigma) = \delta^*(\delta(s, x), \sigma)$ for all $s \in S$.

(c) Formally define what a context-free grammar is.

A context-free grammar is a quadruple $G = (\Sigma, V, s_0, R)$. Σ is a finite alphabet of terminals. V is a finite set of non-terminals. $s_0 \in V$ is a start symbol. R is a set of productions of the form $s \to \sigma$, where $s \in V$ and $\sigma \in (V \cup \Sigma)^*$.

(d) Formally define the language defined by a context-free grammar.

A context-free grammar $G = (\Sigma, V, s_0, R)$ defines the language $\mathcal{L} = \{\sigma \in \Sigma^* \mid s_0 \Rightarrow_G^* \sigma\}$, where $\alpha x \beta \Rightarrow_G \alpha \sigma \beta$ if $(x, \sigma) \in R$ and $\sigma \Rightarrow_G^* \sigma'$ if either $\sigma = \sigma'$ or there exists a sentential form $\sigma'' \in (V \cup \Sigma)^*$ such that $\sigma \Rightarrow_G \sigma''$ and $\sigma'' \Rightarrow_G^* \sigma'$.

Question 1.2: Strong, static, and dynamic typing — 6 marks

(a) Explain the difference between a strongly typed language and a weakly typed language.

A strongly typed language specifies the set of operations that may be applied to objects of a certain type. A weakly typed language may allow operations defined for one type A to be applied to objects of a different type B (by silently re-interpreting these values to be of type B.)

(b) Consider a strongly typed language. Explain the difference between this language being statically typed or dynamically typed.

In a statically typed language, variables have type constraints and can store only values of types that satisfy these constraints. This ensures in particular that operations are applied only to values that support them. In a dynamically typed language, variables do not have any type constraints. Any variable can store values of any type. As a result, since the language is strongly typed, it is the responsibility of the runtime system to check whether the values stored in variables support the operations the program applies to them.

Question 1.3: Paramater passing and array representations — 10 marks

(a) Consider a class `Counter` that holds an integer counter. Upon creation of a `Counter` object, its value is 0. The `Counter` class implements two methods: `increment()` increases the value held by the `Counter` object by one; `getValue()` returns the current value of the counter. Now consider the following piece of pseudo-code (**the syntax is C++-like, the semantics are not (completely)**). Without any knowledge about the function `f`, determine the range of possible values the following code may print (`print()` prints the value of its argument):

```
void main() {
    Counter ctr;
    for (int i = 0; i < 10; ++i) {
        ctr.increment();
    }
    f(ctr);
    print(ctr.getValue());
}
```

given function parameters are passed by

Value:	*10*
Reference:	*Any value* $x \geq 0$
Sharing:	*Any value* $x \geq 10$

(b) Give two reasons why it may be important to know how different programming languages store multi-dimensional arrays in memory.

1. *For operations that need to access all elements in the array, knowledge of the layout of the array in memory may allow us to order the accesses to the array so that consecutive accesses access array elements that are stored close to each other. This helps minimize the number of cache misses and thus to improve the efficiency of our program.*
2. *When passing arrays between parts of a program written in different programming languages, it is important to make sure that the layout of the array used by both programming languages is compatible.*

Question 2.1: Implementation of static scoping — 5 marks

Languages that support recursion represent the local variables of each function call in a stack frame. Now assume a given language uses static scoping, functions can be nested arbitrarily, and each function introduces a new scope. How can a compiler for such a language support efficient variable access? In other words, variable accesses should be implemented by following (few) pointers and address arithmetic; no lookup tables or any other form of search should be required at runtime.

The stack frame corresponding to each function holds the variables defined in that function. The compiler can determine the arrangement of the variables in the stack frame at compile time. Thus, given a frame pointer to a reference location in the stack frame, variables can be located by adding a fixed offset, determined at compile time, to the location referenced by the frame pointer.

This leaves the problem of finding the stack frame that contains a variable referenced by a particular variable access. This variable can be located in the current scope or any surrounding scope. The runtime system of the programming language can equip every stack frame with a static link to the most recent stack frame corresponding to the immediately surrounding function. The sequence of these static links defines the static chain of the stack frame. If a variable `x` *in a given function* `f` *is accessed from a function* `g` *nested k levels deep inside f , then k hops along the static chain of* `g`*'s stack frame locate the most recent stack frame corresponding to* `f` *and adding the appropriate offset to the resulting frame pointer into* `f`*'s stack frame, as discussed above, locates* `x`*.*

Since the nesting depths of functions can be determined at compile time, the compiler can generate the code necessary to follow the correct number of static links and then add the correct offset to the resulting frame pointer to access `x`*.*

Question 2.2: Implementation of control constructs — 10 marks

(a) Explain how the following `switch`-statement on the left can be implemented more efficiently than the corresponding `if`-statement on the right. Explain how the correct branch to be used based on `x`'s value is found.

```
int x;
switch (x) {
    case 2, 5:    case1();
    case 3, 4:    case2();
    case 6, 7, 8: case3();
    default:      case4();
}
```

```
int x;
if (x == 2 || x == 5) {
    case1();
} else if (x == 3 || x == 4) {
    case2();
} else if (x == 6 || x == 7 || x == 8) {
    case3();
} else {
    case4();
}
```

The only way to implement the `if`*-statement is to linearly try each condition until the first one that matches is found. In this case, this could take up to 7 comparisons. The reason this cannot be avoided is that, in general,* `if`*-statements allow arbitrary conditions to be used.*

A `switch`*-statement is more restrictive in that it matches the value of a variable* `x` *to the first branch labelled with* `x`*'s value. This can be implemented using a jump table: In this case, we would first check whether* `x < 2` *or* `x > 8`*, in which case the default branch applies. For each of the values* 2–8*, we store the corresponding branch in a lookup table. Finding the branch corresponding to* `x`*'s value translates into a single table lookup. Thus, the* `switch`*-statement locates the correct branch using only two comparisons and one table lookup compared to up to 7 comparisons for the* `if`*-statement.*

(b) Assume you run the following C program and the following Scheme program on a computer with 1GB of memory. (**This restriction is important!**) What happens in each case? Explain. (Assume that the integer type in both cases is wide enough to represent the sum to be computed.)

```
unsigned long long calculateSum(
        unsigned long long s,
        unsigned long long n) {
    if (n == 0) {
        return s;
    } else {
        return calculateSum(s + n, n - 1);
    }
}

int main() {
    unsigned long long sum =
        calculateSum(0, 1000000000);
    printf("sum = %lld\n", sum)
    return 0;
}
```

```
(define (calculateSum s n)
  (if (= n 0)
      s
      (calculateSum
         (+ s n) (- n 1))))

(define (main)
  (let ([sum (calculateSum
                0 1000000000)])
    (display sum)
    (newline)))

(main)
```

C:

The important observation is that the function `calculateSum` *makes* $1,000,000,000$ *recursive calls. Since C does not perform tail call optimization for tail-recursive functions, this requires* $1,000,000,000$ *stack frames. Since we have only 1GB of memory, the program runs out of memory and crashes or terminates with some form of runtime exception.*

Scheme:

Scheme performs tail call optimization, that is, it unrolls the sequence of recursive calls made by `calculateSum` *into a simple loop. Since this loop uses only constant space, the program does not run out of space and eventually prints the value of* $\sum_{i=1}^{1,000,000,000} i$.

Question 2.3: Garbage collection — 12 marks

(a) Explain how reference counting works as a method to implement garbage collection. Explain what happens when you allocate a new object, when performing pointer assignment, and when a reference to an allocated object goes out of scope.

Each heap-allocated object stores the number of pointers that point to it.

When a new object is allocated on the heap and the pointer to the object is assigned to a variable, the object's reference count is initialized to 1.

When a pointer p *to an object is assigned to a different pointer variable* q*, the reference count of the object reference by* p *is increased by one (because* q *now also points to it) and the reference count of the object previously referenced by* q *is decreased by one (because* q *no longer points to it).*

Similarly, when a pointer variable p *goes out of scope, the reference count of the object referenced by* p *is decreased by one because* p *no longer points to it.*

When the reference count of an object becomes 0*, then there is no more pointer to it, so the object is unreachable and can be deallocated.*

(b) Under what circumstances does reference counting fail to work correctly? Explain why.

Any type of circular structure poses problems. For example, when eliminating the last static or local variable that points to any node of a doubly-linked list, the list cannot be reached anymore and its nodes should be freed. However, since the nodes point to each other, they all have a positive reference count and thus are not freed.

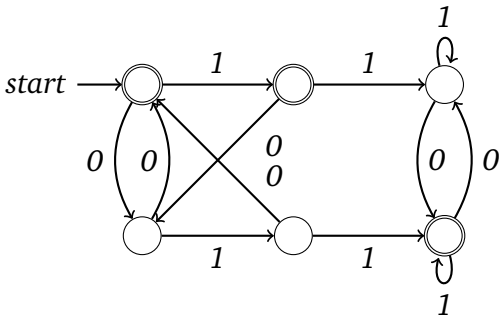
(c) Briefly describe a garbage collection strategy that does not have this shortcoming.

Mark-and-sweep is a general garbage collection strategy that also works in the presence of circular structures. Heap memory is not reclaimed until the available heap memory runs low. At this point, the garbage collector determines which objects are still reachable from static variables and variables on the stack and frees the objects that are not. To find the objects that are reachable from static variables and variables on the stack, the garbage collector first marks every heap object as unreachable. It then views the set of static variables, variables on the stack, and objects on the heap as the vertices of a graph with a directed edge from one object to another if the object stores a pointer to the other object. It performs depth-first search starting from the the static variables and variables on the stack and marks all heap objects visited by the search as reachable. These are all the reachable objects. All other objects can be reclaimed.

Question 3.1: Regular languages **5 marks**

Construct a DFA for the language $\mathcal{L}$ of binary strings that satisfy exactly one of the following two conditions:

- The number of 0s is even and the string contains no two consecutive 1s.
- The number of 0s is odd and the string does contain two consecutive 1s.



Question 3.2: Non-regular languages **5 marks**

Prove that the language of all strings that contain at least twice as many 0s as 1s is not regular.

We use $\mathcal{L}$ to denote this language. Assume the language is regular. Then, by the Pumping Lemma, there exists an integer $n_{\mathcal{L}}$ such that every string σ with $|\sigma| \geq n_{\mathcal{L}}$ can be written as the concatenation of three substrings $\sigma = \alpha\beta\gamma$ such that $|\alpha\beta| \leq n_{\mathcal{L}}$, $|\beta| > 0$, and $\alpha\beta^k\gamma \in \mathcal{L}$ for all $k \geq 0$.

So consider the string $\sigma = 1^{n_{\mathcal{L}}}0^{2n_{\mathcal{L}}} \in \mathcal{L}$. The decomposition of σ into three substrings α, β, and γ as above satisfies $\alpha = 1^a$, $\beta = 1^b$, and $\gamma = 1^c0^{2n_{\mathcal{L}}}$, where $a+b+c = n_{\mathcal{L}}$. According to the Pumping Lemma, the string $\alpha\beta\beta\gamma$ should also be in $\mathcal{L}$ but it contains $a + 2b + c = n_{\mathcal{L}} + b > n_{\mathcal{L}}$ 1s and only $2n_{\mathcal{L}}$ 0s, so it is not in the language.

This proves that $\mathcal{L}$ is not regular.

Question 3.3: Context-free languages — 8 marks

Construct a recursive-descent parser for the language described by the following grammar:

$$
\begin{aligned}
\textit{Expression} &\rightarrow \textit{Subexpr}\ \texttt{\$} \\
\textit{Subexpr} &\rightarrow \textit{number} \\
\textit{Subexpr} &\rightarrow \texttt{+}\ \textit{Subexpr}\ \textit{Subexpr} \\
\textit{Subexpr} &\rightarrow \texttt{-}\ \textit{Subexpr}\ \textit{Subexpr} \\
\textit{Subexpr} &\rightarrow \texttt{*}\ \textit{Subexpr}\ \textit{Subexpr} \\
\textit{Subexpr} &\rightarrow \texttt{/}\ \textit{Subexpr}\ \textit{Subexpr}
\end{aligned}
$$

+, -, *, /, $, and *number* are terminals. Provide your parser in pseudo-code. The pseudo-code should make clear which recursive calls are made depending on the next terminal in the input.

```
def parseExpression():
    if next token in [number, +, -, *, /]:
        parseSubexpr()
        expect($)
    else:
        throw SyntaxError()

def parseSubexpr():
    if next token == number:
        expect(number)
    elif next token == +:
        expect(+)
        parseSubexpr()
        parseSubexpr()
    elif next token == -:
        expect(-)
        parseSubexpr()
        parseSubexpr()
    elif next token == *:
        expect(*)
        parseSubexpr()
        parseSubexpr()
    elif next token == /:
        expect(/)
        parseSubexpr()
        parseSubexpr()
    else:
        throw SyntaxError()
```

```
def expect(tok):
    if next token == tok:
        advance to the next token
        in the input
    else:
        throw SyntaxError()
```

Provide a parse tree that represents the recursive calls made by the parser on the input
`/ + * 3 2 - 1 2 4 $`.