Banner number: Name:

Midterm Exam

CSCI 3136: Principles of Programming Languages

February 20, 2013

Group 1		Group 2		Group 3		$\sum$
Question 1.1		Question 2.1		Question 3.1		
Question 1.2		Question 2.2		Question 3.2		
Question 1.3		Question 2.3				
$\sum$		$\sum$		$\sum$		

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. Keey your answers short and to the point.
- **You are not allowed to use a cheat sheet.**
- **Read every question carefully before answering. In particular, do not waste time on a proof if none is asked for, and do not forget to provide one if it is required.**
- **Do not forget to write your banner number and name on the top of this page.**
- **This exam has 10 pages, including this title page. Notify me immediately if your copy has fewer than 10 pages.**
- The total number of marks in this exam is 100.

1 Regular Languages and Finite Automata

Question 1.1 (Finite automata) — 15 marks

(a) Provide the formal definition of a *non-deterministic* finite automaton (NFA).

An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$*, where*

- Q *is a set of* states*,*
- Σ *is an* alphabet*,*
- $q_0 \in Q$ *is the* start state*,*
- $F \subseteq Q$ *is a set of* accepting states*, and*
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ *is the* transition function*.*

(b) When does an NFA accept a string σ?

Given an NFA $M = (Q, \Sigma, \delta, q_0, F)$*, a string* $\sigma = x_1 x_2 \dots x_n \in \Sigma^*$ traces *a sequence of states* $q_0 q_1 \dots q_m$ *if there exist indices* $0 < i_1 < i_2 < \dots < i_n \leq m$ *such that* $q_{i_j} \in \delta(q_{i_j - 1}, x_j)$*, for all* $1 \leq j \leq n$*, and* $q_i \in \delta(q_{i-1}, \varepsilon)$*, for all* $i \notin \{i_1, i_2, \dots, i_n\}$*. The NFA accepts a string* σ *if and only if there exists a sequence of states* $q_0 q_1 \dots q_m$ *traced by* σ *such that* $q_m \in F$*.*

(c) Are DFA and NFA equally powerful, that is, can every language that can be recognized by a DFA also be recognized by an NFA and vice versa? If you answer no, provide a language that can be recognized by one of these two types of machines but not by the other. If you answer yes, *sketch* a proof that shows that every language recognizable by an NFA is also recognizable by a DFA and vice versa.

They are equally powerful.

A DFA is *an NFA, so every language that can be recognized by a DFA can be recognized by an NFA.*

To prove that a language that can be recognized by an NFA M *can also be recognized by a DFA* M'*, we construct the state set of* M' *as the set of all subsets of states of* M*. Assuming such a subset* Q' *is the set of states the NFA could be in after consuming a string* σ*, it can be in a subset of states* Q'' *after consuming the string* σx*, where* $x \in \Sigma$*, exactly if for every state* $q \in Q''$*, there exists a state* $q' \in Q'$ *such that* q *can be reached from a state in* $\delta(q', x)$ *using only* ε*-transitions. In this case, we set* $\delta(Q', x) = Q''$ *for the DFA.*

The start state of the DFA is the set of all NFA states that can be reached from the start state of the NFA using only ε*-transitions.*

To obtain the final DFA, we remove all those states that cannot be reached from the DFA's start state, and we define the set of final states of the DFA to be all those sets of NFA states that include a final state of the NFA. Indeed, if such a set Q' *is the set of states that can be reached after consuming a string* σ*, then* σ *is accepted by the NFA as per the definition in part (b) and, thus, needs to be accepted by the DFA. If the set* Q' *does not include a final state of the NFA, the NFA rejects* σ*, so the DFA should do the same.*

Question 1.2 (Pumping Lemma) — 10 marks

(a) State the Pumping Lemma.

For every regular language $\mathscr{L}$, there exists an integer $n_{\mathscr{L}}$ with the following property: Every string $\sigma \in \mathscr{L}$ of length at least $n_{\mathscr{L}}$ can be divided into three substrings α, β, and γ such that $|\alpha\beta| \leq n_{\mathscr{L}}$, $|\beta| > 0$, and, for all $k \geq 0$, the string $\alpha\beta^k\gamma$ is also in $\mathscr{L}$.

(b) Sketch the proof of the Pumping Lemma. (*Hint:* Remember how regular languages and DFA relate to each other.)

If $\mathscr{L}$ is regular, there exists a DFA M that recognizes $\mathscr{L}$. We choose $n_{\mathscr{L}}$ to be one more than the number of states in M. The sequence of states of M traced by a string σ of length at least $n_{\mathscr{L}}$ must contain at least one state more than once because M has less than $n_{\mathscr{L}}$ states. Let q be the first state that is encountered twice, let α be the prefix of σ that leads to the first occurrence of q, let β be the part of σ following α and which traces the sequence of states between the first and the second occurrence of q, and let γ be the remainder of σ. Since reading β starting from state q leads back to state q, every string $\alpha\beta^k\gamma$, for any $k \geq 0$, leads to the same state as σ. Thus, if M accepts σ, that is, if $\sigma \in \mathscr{L}$, it also accepts every string $\alpha\beta^k\gamma$, that is, every such string is in $\mathscr{L}$.

Finally, observe that $|\beta| > 0$ because it is a string that leads M from one occurence of q to the next occurrence of q in the sequence of states traced by σ. $|\alpha\beta| \leq n_{\mathscr{L}}$ because every state of M, except q, is contained exactly once in the sequence of states traced by $\alpha\beta$, q is contained twice in this sequence, and M has less than $n_{\mathscr{L}}$ states.

Question 1.3 (Deciding whether a language is regular) **15 marks**

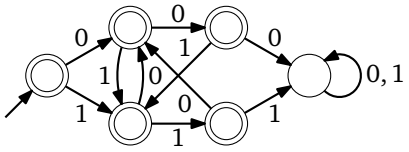
For each of the following languages, state whether it is regular or not. If it is, prove your claim by providing a graphical representation of a (deterministic or non-deterministic) finite automaton that recognizes it. If it is not, prove your claim using the Pumping Lemma.

(a) The language $\mathscr{L}$ of all binary strings of odd length whose middle letter is a 1. In other words, the string 0111000 belongs to this language, while the strings 0110000 and 011000 do not (the former has a 0 as its middle letter, the latter has even length).

This language is not regular. Assume it is, and let $n_{\mathscr{L}}$ be the integer associated with $\mathscr{L}$ by the Pumping Lemma. Now let $\sigma = 0^{n_{\mathscr{L}}}10^{n_{\mathscr{L}}} \in \mathscr{L}$. By the Pumping Lemma, we can divide σ into three substrings α, β, and γ such that $|\alpha\beta| \leq n_{\mathscr{L}}$, $|\beta| > 0$, and $\alpha\beta^2\gamma \in \mathscr{L}$. Since the first $n_{\mathscr{L}}$ letters in σ are 0s, we have $\beta = 0^k$, for some $k > 0$, and, hence, $\sigma' = \alpha\beta^2\gamma = 0^{n_{\mathscr{L}}+k}10^{n_{\mathscr{L}}}$. This is a contradiction because it implies that either σ' is of even length or its middle letter is a 0; in either case, we have $\sigma' \notin \mathscr{L}$ contrary to what the Pumping Lemma states.

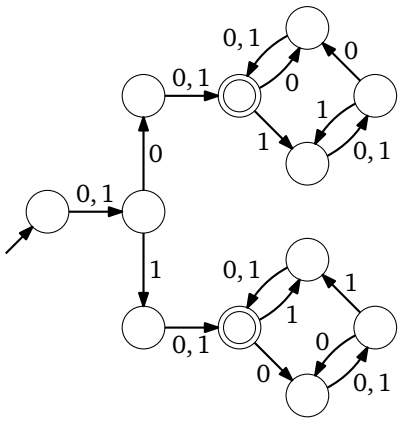
(b) The language $\mathscr{L}$ of all binary strings with no three consecutive letters that are the same. In other words, the string 1010011001 belongs to this language, the string 1010011 does not.

This language is regular. Here is a DFA that recognizes it:



(c) The language $\mathscr{L}$ of all binary strings of odd length whose second and second-last letters are the same. In other words, the string 0111010 belongs to this language, while the strings 0011010 and 011010 do not. Note that the string 010 also belongs to this language.

This language is regular. Here is a DFA that recognizes it:



2 Context-Free Languages and Parsing

Question 2.1 (Derivations, parse trees, and ambiguity) — 10 marks

Given a context-free grammar, give formal definitions for the following concepts.

(a) Derivation

A derivation is a sequence of strings $\sigma_0, \sigma_1, \ldots, \sigma_n$ over the alphabet $V \cup \Sigma$ (terminals and non-terminals), where $\sigma_0 = S$ (the start symbol), $\sigma_n \in \Sigma^*$, and for every $1 \leq i \leq n$, $\sigma_{i-1} = \alpha_i X_i \gamma_i$ and $\sigma_i = \alpha_i \beta_i \gamma_i$, where X_i is a non-terminal and $X_i \to \beta_i$ is a production in the grammar.

(b) Leftmost derivation

A leftmost derivation is a derivation as defined in part (a) where $\alpha_i \in \Sigma^*$, for all $1 \leq i \leq n$. In other words, for every string σ_{i-1}, $1 \leq i \leq n$, we obtain the next string σ_i in the derivation by replacing the leftmost non-terminal X_i in σ_{i-1} with the right-hand side of a production for X_i.

(c) Parse tree

A parse tree is a rooted tree with root S (the start symbol of the grammar) and with leaves labelled with letters in $\Sigma \cup \{\varepsilon\}$. All internal nodes are labelled with non-terminals. For every internal node X with children $Y_1, Y_2, \ldots, Y_k$ from left to right, the grammar contains a production $X \to Y_1 Y_2 \ldots Y_k$.

(d) Ambiguous grammar

A grammar is ambiguous if there exist two different parse trees defined by this grammar and such that listing the leaves of theses two trees from left to right yields the same string in Σ^*.

Question 2.2 (Recognizing LL(1) grammars) **15 marks**

Is the following grammar LL(1)? *Function* is the start symbol. Non-terminals are *CamelCase* italic strings. All other *strings* and `symbols` are terminals. Prove that your answer is correct.

Function → *id* `(` *Args* `)` `{` *Statements* `}`
Args → *Arg MoreArgs*
Arg → *id*
Arg → *number*
MoreArgs → *ε*
MoreArgs → `,` *Args*
Expression → *Arg*
Expression → *FunCall*
Statements → *Statement MoreStatements*
Statement → *Assignment*
Statement → *FunCall*
MoreStatements → *ε*
MoreStatements → `;` *Statements*
Assignment → *id* `:=` *Expression*
FunCall → *id* `(` *Args* `)`

This grammar is not LL(1). Observe that both Assignment and FunCall have id in their FIRST sets. Thus, id is in the PREDICT set of both productions for Statement: Statement → Assignment and Statement → FunCall. Since the PREDICT sets of all productions for the same non-terminal must be disjoint for the grammar to be LL(1), this shows that the grammar is not LL(1).

Question 2.3 (Parsing LL(1) languages) — 10 marks

Provide the pseudo-code of a recursive-descent parser that recognizes the language given by the following grammar with start symbol S. (This is the language of arithmetic expressions in Polish notation.) The PREDICT set of each production in the grammar is listed after the production.

$S \to E\ \$$	$\{+, -, *, /, id\}$
$E \to O\ E\ E$	$\{+, -, *, /\}$
$E \to id$	$\{id\}$
$O \to +$	$\{+\}$
$O \to -$	$\{-\}$
$O \to *$	$\{*\}$
$O \to /$	$\{/\}$

```
procedure parseS
   case next input symbol of
      +, -, *, /, id → parseE; match($);
      otherwise      → reject the input;

procedure parseE
   case next input symbol of
      +, -, *, /     → parseO; parseE; parseE;
      id             → match(id);
      otherwise      → reject the input;

procedure parseO
   case next input symbol of
      +              → match(+);
      -              → match(-);
      *              → match(*);
      /              → match(/);
      otherwise      → reject the input;

procedure match(x)
   if next input symbol = x
      then consume this input symbol;
      else reject the input;
```

3 Semantic Analysis and More

Question 3.1 (Is it lexical, syntactic or semantic?) 10 marks

For each of the following conditions on a valid program, state whether it is a lexical, syntactic or semantic constraint. (I have given in square brackets examples of programming languages that use these rules.)

(a) Statements need to be separated by semicolons or line breaks. [Haskell]

Syntactic

(b) A variable that appears in an expression has to be declared in one of the lexical units containing this expression. (The nesting of lexical units makes it possible that this expression is contained in multiple lexical units.) [Every strongly typed language with lexical scoping.]

Semantic

(c) When assigning the result of a function call to a variable, the return value of the function and the variable must have the same type. [Every strongly typed imperative programming language.]

Semantic

(d) A floating point number consists of a mantissa and an optional exponent. The mantissa starts with an optional sign, followed by a non-empty sequence of digits, optionally followed by a period and another non-empty sequence of digits. The exponent starts with a lowercase or uppercase 'E', followed by an optional sign, followed by a non-empty sequence of digits. [Common to most programming languages.]

Lexical

(e) Every '(' has to be matched by a ')' and vice versa. [Common to most programming languages.]

Syntactic

(f) A type name starts with an uppercase letter, optionally followed by a sequence of lowercase and uppercase letters. An identifier starts with a lowercase letter, optionally followed by a sequence of lowercase and uppercase letters. [Similar to Haskell's rules for type names and identifiers.]

Lexical

(g) The left-hand side of an assignment must denote an L-value (i.e., a reference to a memory location). [Every imperative programming language.]

Semantic

Question 3.2 (Action routines) — 15 marks

Consider a simple "programming language" that can be used to describe programs consisting entirely of variable declarations and assignments. The variables can be of two types: `string` and `int`. The language provides two conversion operations: `toStr` expects an `int` as argument and returns a single-character string whose only character is the one whose ASCII code is the given `int`; `toInt` expects a `string` as argument and returns the ASCII code of the first character. The `int` type supports addition, subtraction, multiplication, and division with the usual semantics, except that we ignore precedence to keep things simpler. The `string` type supports addition, which concatenates the two given strings. Every variable needs to be declared before it is first assigned to; every variable can be declared only once. A variable can be used in the right hand side of an assignment only after assigning a value to it. The grammar specifying the syntax of this language is as follows:

Program → *Statements* `$`
Statements → *ε*
Statements → *Statement* `;` *Statements*
Statement → *Declaration*
Statement → *Assignment*
Declaration → *Type identifier*
Type → `int`
Type → `string`
Assignment → *identifier* `:=` *Expression*
Expression → *Term ExpressionTail*
ExpressionTail → *ε*
ExpressionTail → *Operator Expression*
Term → *identifier*
Term → *string_literal*
Term → *int_literal*
Term → `(` *Expression* `)`
Term → `toInt (` *Expression* `)`
Term → `toStr (` *Expression* `)`
Operator → `+`
Operator → `-`
Operator → `*`
Operator → `/`

Your goal is to obtain an interpreter that executes the program. You can assume that the interpreter maintains a symbol table `T` that maps each identifier to its type and value. In other words, `T[id]` equals `(t, v)` if identifier `id` is defined; in this case `t` is its type and `v` its value. If `id` is undefined, then `T[id] = nil`. Every terminal stores its textual representation in its `.val` attribute (provided by the scanner). Augment the grammar with action routines written in C, Java, Python or Ruby that perform the following tasks:

- Check the above type constraints and ordering constraints of statements. If any of these constraints is violated, the interpreter may call a function `error()` to terminate the execution without a meaningful error message.
- Update the symbol table `T` to reflect the effects of the statements in the program assuming there are no semantic errors in the program. In other words, a declaration "`int x;`", for example, should lead to adding an entry with key `x` to the symbol table, and an assigment "`x = 2;`" should store the value 2 in `x`'s slot in the symbol table.

Your action routines may use the symbol table and define attributes of non-terminals as needed. The runtime system available to the action routines provides functions `to_int`, `to_string`, `concat`, `add`, `subtract`, `multiply`, and `divide` which implement string-to-integer conversion, integer-to-string conversion, string concatenation, integer addition, integer subtraction, integer multiplication, and integer division, respectively. Provide your answer on the next page.

Question 3.2 (Continued)

Action routines are Ruby code here. Non-terminals are abbreviated.

```
P  -> SS '$'
SS ->
SS -> S ';' SS
S  -> D
S  -> A
D  -> T id { if T[$2.val] != nil then error() else T[$2.val] = [$1.val, nil] end }
T  -> 'int' { $0.val = 'int' }
T  -> 'string' { $0.val = 'string' }
A  -> id ':=' { $3.in_val = nil }
      E { if T[$1.val] == nil or T[$1.val][0] != $3.val[0] then error()
          else T[$1.val] = $3.val end }
E  -> TT { if $0.in_val == nil then $2.in_val = $1.val
           elsif $0.in_val[0] == $1.val[0] then
             $2.in_val = [$1.val[0], $0.in_val[1]($0.in_val[2], $1.val[1])]
           else error() end }
      EE { $0.val = $2.val }
EE -> { $0.val = $0.in_val }
EE ->   { $1.in_val = $0.in_val[0] }
      O { $2.in_val = [$0.in_val[0], $1.val, $0.in_val[1]] }
      E { $0.val = $2.val }
TT -> id { if T[$1.val] == nil or T[$1.val][1] == nil then error()
           else $0.val = T[$1.val] end }
TT -> string_literal { $0.val = ['string', $1.val] }
TT -> int_literal { $0.val = ['int', $1.val] }
TT -> { $2.in_val = nil } '(' E ')' { $0.val = $2.val }
TT -> { $3.in_val = nil } 'toInt' '(' E ')'
      { if $3.val[0] != 'string' then error()
        else $0.val = ['int', to_int($3.val[1])] end }
TT -> { $3.in_val = nil } 'toStr' '(' E ')'
      { if $3.val[0] != 'int' then error()
        else $0.val = ['string', to_string($1.val[1])] end }
O  -> '+' { if $0.in_val == 'int' then $0.val = add else $0.val = concat end }
O  -> '-' { if $0.in_val == 'string' then error() else $0.val = subtract end }
O  -> '*' { if $0.in_val == 'string' then error() else $0.val = multiply end }
O  -> '/' { if $0.in_val == 'string' then error() else $0.val = divide end }
```