Notes on Sample Midterm

This is the midterm I gave in 2013. The course was structured a bit differently then, talking about formal languages first and leaving an introduction to Haskell for later; Prolog wasn't covered at all. As a result, we were further along in covering formal languages by the time of the midterm than we are this year. At this point, I expect that we will finish covering lexical and syntactic analysis in time for the midterm, but not semantic analysis. Thus, Part 3 of the sample midterm will have no equivalent in this year's midterm. You should focus on Parts 1 and 2 in your practice.

Topics potentially covered in this year's midterm:

- Functional and logic programming
  - Differences between programming paradigms (imperative vs functional vs logic)
  - Explain concepts such as tail recursion, lazy evaluation, currying, unification, backtracking
  - Explain what a given simple Haskell or Prolog program does
- Regular languages and lexical analysis
  - Basic definitions: regular language, regular expression, NFA, DFA, ...
  - Provide a regular expression, NFA or DFA for a given regular language
  - Apply the Pumping Lemma to prove that a given language is not regular
  - Demonstrate how to convert between regular expression, NFA, and DFA, and how to minimize a DFA
- Context-free languages and syntactic analysis
  - Basic definitions: Context-free grammar/language, ambiguity, LL(k)/LR(k), leftmost/rightmost derivation, parse tree, push-down automaton, recursive-descent parser...
  - Develop a recursive-descent parser for a simple LL(1) language
  - Demonstrate how to decide whether a language is LL(1)
  - Provide a push-down automaton for a simple context-free language

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).

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

(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.

Question 1.2 (Pumping Lemma) 10 marks

(a) State the Pumping Lemma.

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

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 $011\underline{1}000$ belongs to this language, while the strings $011\underline{0}000$ and 011000 do not (the former has a 0 as its middle letter, the latter has even length).

(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 $101\underline{000}11$ does not.

(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 $0\underline{1}110\underline{1}0$ belongs to this language, while the strings $0\underline{0}110\underline{1}0$ and 011010 do not. Note that the string $0\underline{1}0$ also belongs to this language.

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

(b) Leftmost derivation

(c) Parse tree

(d) Ambiguous grammar

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.

$$\begin{aligned}
\mathit{Function} &\to \mathit{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)}\ \texttt{\{}\ \mathit{Statements}\ \texttt{\}} \\
\mathit{Args} &\to \mathit{Arg}\ \mathit{MoreArgs} \\
\mathit{Arg} &\to \mathit{id} \\
\mathit{Arg} &\to \mathit{number} \\
\mathit{MoreArgs} &\to \varepsilon \\
\mathit{MoreArgs} &\to \texttt{,}\ \mathit{Args} \\
\mathit{Expression} &\to \mathit{Arg} \\
\mathit{Expression} &\to \mathit{FunCall}
\end{aligned}$$

$$\begin{aligned}
\mathit{Statements} &\to \mathit{Statement}\ \mathit{MoreStatements} \\
\mathit{Statement} &\to \mathit{Assignment} \\
\mathit{Statement} &\to \mathit{FunCall} \\
\mathit{MoreStatements} &\to \varepsilon \\
\mathit{MoreStatements} &\to \texttt{;}\ \mathit{Statements} \\
\mathit{Assignment} &\to \mathit{id}\ \texttt{:=}\ \mathit{Expression} \\
\mathit{FunCall} &\to \mathit{id}\ \texttt{(}\ \mathit{Args}\ \texttt{)}
\end{aligned}$$

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.

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

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]

(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.]

(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.]

(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.]

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

(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.]

(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.]

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)