CS 320 Principles of Programming Languages **Sample Final**

NAME: (The final will only have six one page questions, not nine pages as provided here!)

READ THESE INSTRUCTIONS VERY CAREFULLY BEFORE YOU BEGIN:

- Do not turn over this page until the test begins.
- Make sure that you have written your name in the space at the top of this page.
- Time allowed is 100 minutes (ONE hour and FORTY minutes).
- You may choose to leave the exam at any time during the first 85 minutes; please make sure that your paper is properly labeled with your name, be sure to hand in your work before leaving, and exit quietly to avoid disturbing other students. To avoid unnecessary disruption, please do not leave the exam in the final 15 minutes of the exam.
- You may only bring basic writing tools (pen, pencil, ruler, . . .) in to this test with you. No other materials—including textbooks, calculators, notes, computers, phones, music players, etc.—may be used.
- Please do all your work on these sheets.
- Keep your answers as clear, as concise, and as legible as possible. None of these problems requires very long answers, but you should make good use of the space provided. (For example, a short answer in a large space is unlikely to score very many points.)
- There will be six questions on the actual final, each taking a single page and worth 8 points. (This sample has 8 questions, one of which spans two pages.) The points for each part of each question are shown in parentheses on the right-hand side; you may use these to judge the relative difficulty and/or level of effort that is expected for each part. Credit will be given for partial answers, and, where appropriate, for correct working or explanation, even if the final answer is incorrect.
- Answer as many questions or parts of questions as you can. You may not be able to answer all of the questions in full.
- Read the questions carefully. You may ask for clarification of questions at any time during the exam. If you feel that you need to make additional assumptions in order to answer a question, be sure to explain that as part of your answer. Beware of trick questions!
- Discussion of this exam with colleagues or classmates is not allowed until all solutions have been turned in.
- Breaches of academic integrity will be treated very seriously.

1) **Describing syntax**: (NOTE: THIS QUESTION SPANS TWO PAGES)

This question is about the syntax of JSON (short for "JavaScript Object Notation"), which, quoting Wikipedia: "...is an open-standard format that uses human-readable text to transmit data objects. It is the most common data format used for asynchronous browser/server communication ...". As such, there is a good chance that you may make use of JSON at some point in your career. For the purposes of this question, however, no prior knowledge of JSON is required or should be assumed. (Some small details of JSON may have been changed for this question.)

(a) What goals should be considered when choosing how to describe the concrete syntax of a programming language? **(2)**

(b) Use the following brief descriptions, taken from a JSON specification, to construct regular expressions that match basic JSON datatypes. Being written in English, the descriptions are informal. Without access to detailed documentation, you will need to make (reasonable) assumptions about what the precise syntax is likely to be. You should use only simple regular expression syntax, but you are encouraged to use named regular expressions where appropriate to make your answers more readable.

i) A *boolean* (in JSON) is "either of the values `true` or `false`". **(1)**

ii) A *string* (in JSON) is "a sequence of zero or more characters, delimited with double-quotation marks and supporting a backslash escaping syntax". **(2)**

iii) A *number* (in JSON) is "a signed decimal number that may contain a fractional part and may use exponential E notation". **(3)**

(c) We can begin to construct a context-free grammar for JSON values using the following three productions for a nonterminal J representing JSON values. The three terminal symbols used here—`BOOLEAN`, `NUMBER`, and `STRING`—correspond in the obvious way to the tokens described by the regular expressions in Part (b) above.

$$J \rightarrow \texttt{BOOLEAN} \qquad J \rightarrow \texttt{STRING} \qquad J \rightarrow \texttt{NUMBER}$$

Explain how this grammar can be extended to support the two other forms of JSON values that are described below. The resulting grammar should be unambiguous. Every production should have a single nonterminal on its left hand side, and a sequence of zero or more terminal or nonterminal symbols on the right hand side; *no additional notation may be used, but you may introduce additional nonterminal symbols as necessary.* As before, you may make reasonable assumptions, documenting them in your answer, in situations where the natural language descriptions do not fully define the JSON syntax.

i) An *array* (in JSON) is "... a list of zero or more values, each of which may be of any type. Arrays use square bracket notation with elements being comma separated." As simple examples, `[1,"hello"]` is a JSON array with two value components, as is `[[],[]]`; in the latter example the two values in the array are themselves (empty) array values. **(4)**

ii) An *object* (in JSON) is "... a list of zero or more name/value pairs where the names are strings. Objects are delimited with curly braces and use commas to separate each pair. Within each pair, a colon character is used to separate the name from its value." As a simple example, the following string

```
{"course":{"dept":"CS", "num":320, "title":"Principles of Prog Lang"}}
```

represents a JSON object with one name/value pair, and the value component in that pair is itself an object with three name/value pairs. **(4)**

Continues on next page ...

2) **Terminology**:

Provide succinct definitions for each of the following terms as they might be used when discussing programming languages:

- static scoping **(1)**

- dynamic dispatch **(2)**

- sum type **(1)**

- abstract class **(1)**

- object lifetime **(2)**

- denotational semantics **(1)**

3) **Type definitions**:

Each of the items listed below begins "X is a . . . ". For each one, you should write down two valid sets of type definitions—one in Java and one in Haskell—that could reasonably be used to describe the *type* of X. You will not be penalized for minor errors in Java or Haskell syntax. For brevity, you should NOT include any constructor definitions or `private` or `public` modifiers in your Java code.

– X is a time of day, comprising two integers representing the hour and minutes, and a flag to distinguish between a.m. and p.m. (You do NOT need to do anything to ensure that the hour and minute integers are in an appropriate range.) **(2)**

– X is a binary tree structure with an integer value at each interior node (each of which has precisely two children) and a string at each leaf node. **(4)**

– X is a predicate on integer values. (i.e., X can be used as a function mapping integers to Booleans.) For example, X could be the predicate for testing whether a given number is even, or for testing whether it is prime, and so on. Note, however, that you are only expected to specify the *type* of X, and should not provide any specific values of that type. **(2)**

Continues on next page . . .

4) **Principles of static analysis**:

(a) Describe one distinct use for static analysis in a compiler for each of the three categories described below. In each case, identify a likely consequence if that kind of static analysis is not performed:

– static analysis focussed on validating input programs: **(2)**

– static analysis focussed on resolving ambiguities: **(2)**

– static analysis focussed on preparing for code generation: **(2)**

(b) Explain why is it necessary, in general, for a static analysis to be a "*conservative*" "*approximation*" of "*dynamic semantics*", including brief descriptions for each of the quoted terms. **(2)**

5) **Safety and language design tradeoffs**:

(a) Describe some of the motivations for the decision to rely on *garbage collection* in the design of the Java programming language: **(3)**

(b) Explain what is meant by a *buffer overflow attack* and why they are important in modern language design. **(2)**

(c) Describe the tradeoffs that the designer of a new language might consider in determining how to mitigate the risks of buffer overflow attacks. **(2)**

(d) What is the significance (with respect to the design of C/C++) of the decision not to allow any unchecked runtime errors in Java? **(1)**

Continues on next page . . .

6) **Encapsulation and scope**:

(a) Explain what is meant by *encapsulation* and identify the language features that are likely to be most important to support the use of encapsulation in the Java programming language. **(2)**

(b) Explain what is meant by an *abstract datatype* (ADT) and describe some possible roles for *environment* data structures in the implementation of a language that supports ADTs. **(2)**

(c) Draw labeled boxes around sections of code in the following program fragment to show the scope of each variable (multiple variables may share the same box). In addition, where possible, draw circles around the binding occurrences of those variables (i.e., the points where bindings are created). **(3)**

```
int f(int x, boolean b) {
  int y = 2*x+1;
  if (b) {
    int t = x;
    x     = v;
    v     = t;
  } else {
    y = x;
  }
  return y;
}
```

(d) Assuming that the code in (c) is part of a larger, valid program, what assumptions can we make about the environment at the point where the code appears? **(1)**

7) **Formalizing programming languages**:

(a) In general terms, what is meant by a *formal description* for a component of a programming language and why might such a definition be important in practice? **(3)**

(b) What additional properties would you expect for a formal description of programming language *syntax* (beyond anything you have already mentioned in Part (a))? **(2)**

(c) Give examples of three distinct ways in which a formal description of programming language *semantics* might be used:

– in work on language implementations: **(1)**

– in program development: **(1)**

– in programming language design: **(1)**

Continues on next page ...

8) **Semantic methods**:

(a) Explain briefly what is meant by the terms *denotational semantics* and *operational semantics*. **(2)**

(b) In the context of *operational semantics*, specifically Hoare Logic, explain what is meant by the following terms:

− *precondition* **(1)**

− *postcondition* **(1)**

− *loop invariant* **(2)**

(c) Briefly, how might preconditions, postconditions, and loop invariants be useful:

i) in the context of algorithm analysis and design? **(1)**

ii) in the context of a large scale program development project? **(1)**

[This page may be used for rough working and notes.]

End