CS 320 Principles of Programming Languages **SAMPLE MIDTERM**

NAME:

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 are six questions on this paper, each taking a single page and worth 8 points. 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) **Syntax and Semantics**:

(a) Explain in broad terms what is meant by the terms *syntax* and *semantics*. **(2)**

(b) Comment briefly on the role of syntax and semantics in the following scenarios:

- You press the button at a crosswalk to walk safely across a busy intersection. **(2)**

- You show a young child a piece of paper with the word "tiger" on it—but the child cannot read and does not respond. When you show a *picture* of a tiger instead, the child smiles, shouts "tiger" and, imitating the animal, pretends to growl! **(2)**

(c) From a practical perspective, why is it important to ensure that each of the syntax and the semantics of a programming language are well-defined? [Hint: provide separate answers for each one!] **(2)**

2) **Programming Language Fundamentals**:

This question assumes the basic `Prop` language for digital circuits that was presented in Week 1.

(a) For a pair of expressions in the `Prop` language, how would you determine whether: **(2)**

– they are *equivalent*?

– they are *equal*, ignoring irrelevant details of syntax?

(b) Why is it useful to enumerate each of the different forms of expression in the `Prop` language using the constructors of a Haskell `data` definition? **(2)**

(c) Although the `Prop` language is intended to be used in a very specific setting, not as a general purpose programming language, it is still missing some important features. Briefly describe two of the most prominent omissions, and explain why those features might be useful in a practical language. **(4)**

Continues on next page ...

3) **Regular Expressions and Context Free Grammars**:

(a) Give a regular expression for the language containing all positive integers, in decimal notation, with no leading zeros. **(1)**

(b) Give a regular expression for the language containing all positive integers, in decimal notation, with no leading zeros, that are greater than 15 and less than 88. **(2)**

(c) Consider the simple language of Boolean expressions that is described by the grammar $P \rightarrow$ `TRUE`; $P \rightarrow$ `FALSE`; $P \rightarrow$ `NOT` P; $P \rightarrow P$ `AND` P. How many different derivations are there with at most two steps that begin with the nonterminal P? [Hint: To save time and writing, and to justify your answer, you are encouraged to show the set of all derivations in the form of a tree.] **(4)**

(d) Why is there a difference between the number of distinct derivations and the number of distinct strings that are generated in your answer to Part (c)? **(1)**

4) **Interpreters and Compilers**:

(a) State, as carefully as you can, what it means to say that a compiler is *correct*. **(2)**

(b) Should you feel more comfortable about traveling in an autonomous (driverless) car if you are told that all the code for the computers in the vehicle was produced using a compiler that had been proved correct? [Hint: This requires a more thoughtful answer than just "yes" or "no"!] **(2)**

(c) What is the primary distinction between an *interpreter* and a *compiler*? What practical factors might be considered in a team that is choosing which of these tools to use for a new software development project (assume that both support the same source language)? **(2)**

(d) What are the general goals of *modularity* in programming? Describe a specific example that demonstrates the benefits of modularity in the construction of a practical compiler system. **(2)**

Continues on next page ...

5) **Compiler Phase Structure**:

(a) Draw a carefully labeled diagram to illustrate the structure of a typical compiler pipeline with six phases. Include a brief (single sentence) description for each phase. **(6)**

(b) In general terms, describe the conditions under which the code implementing the final phase in your compiler pipeline might report an error to the user? **(2)**

6) **Programming Paradigms**:

(a) In very brief terms, describe and distinguish between the *procedural*, *functional*, and *object-oriented* programming paradigms. **(3)**

(b) Consider the following Haskell function definition:

```
> getNumber             :: [Char] -> Int -> Int
> getNumber []     n     = n
> getNumber (c:cs) n
>            | isDigit c = getNumber cs (10*n+digitToInt c)
>            | otherwise = n
```

Using the syntax of Python (a close approximation will suffice), explain how the same algorithm might be coded up in a procedural language where the input string is represented by Python string value and there are no function calls, except for basic functions corresponding to `isDigit` and `digitToInt` in the Haskell code. **(3)**

(c) Based on your experience with Haskell this term (or on your most recent experience learning a new language or paradigm), what do you consider to be the primary benefits and the biggest obstacles in any attempt to learn a new language or paradigm? [You may use space on the next page if you need it, but answers should still be brief.] **(2)**

Continues on next page ...

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