Spring, 1994 Name: ____________________

Com S 541 — Programming Languages 1

# Test on Operational and Denotational Semantics

## Special Directions for this Test

This test has 5 questions and pages numbered 1 through 7.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
Unless we specifically request, λProlog code does not have to run "backwards". You can use cut (`!`) and `not` in your λProlog code.

## General Directions from the "Course Specification"

Short answer questions will be graded on the basis of completeness (did you list all the relevant parts of the answer?) soundness (did you make any errors of fact or contradict yourself?) and clarity. You can help ensure clarity by giving examples where possible.
For essays, critical justifications, and the like there may be no one right answer; so the criteria for judgement will be whether your essay is:

- clear (you use examples where appropriate, you write clear sentences, diagrams, and you avoid excess verbosity),
- sound (you start from facts or reasonable assumptions, your logic is convincing), and
- complete (you consider relevant aspects of the problem and alternatives).

The highest grade will be earned by an answer that is, in addition, concise and elegant; that is, answers that say the most with the least amount of excess. Extra consideration will be given to answers that are especially creative. For programming problems, there is usually more than one right answer, hence it is important to present with your program evidence that it is correct. Answers to programming questions will be scored on the following points:

- elegance and clarity (conciseness, using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution, and how clear and convincing your argument is that the code is correct)
- correctness (whether your program solves the given problem correctly in the given programming language)

You should try to eliminate all syntax errors from your programs, as they will affect both the correctness and clarity of your solution; you may receive no points for programs with major syntax errors. Run-time space and time efficiency of programs is not important, unless the problem says so. Also, checking of external inputs, prompting, labeling output, and other conventions for dealing with humans are not important (for this class), unless the problem says so.
The breakdown of points between these categories is roughly as follows.

- 50% for elegance and clarity (using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution),
- 50% for correctness (whether your program solves the given problem correctly in the given subset of the given language)

You are encouraged to use auxiliary subroutines whenever they are helpful.

1. (5 points) What are the free variables in the following $\lambda$-calculus term? (Don't ask us what a "free variable" is, that's what the question is about.)

   $(\lambda f \,.\, (\lambda y \,.\, x \; (f \; y)))(\lambda z \,.\, (f \; (f \; z)))$

2. (5 points) What parameter passing mechanism is used in Smalltalk? (Recall that one cannot assign to a formal parameter in a Smalltalk method.)

3. (15 points) If the following term in the $\lambda$-calculus has a normal form, give it, otherwise briefly describe why it does not have a normal form. (Don't ask us what a "normal form" is, you're supposed to know that.) Although you don't have to give a formal proof, show each $\beta$ or $\eta$ reduction step.

$$((\lambda k \,.\, (((\lambda r \,.\, (\lambda k \,.\, (k \; z))) \; (\lambda i \,.\, n)) \; k)) \; (\lambda v \,.\, v))$$

4. (40 points) Give an operational semantics for the following denotational semantics. You may use either a computation (little-step) or evaluation (big-step) semantics. There is room on the next page for your answer.

**Abstract Syntax:**
Program ::= `match` Expression
Expression ::= `nothing` | `(quote` Char`)` | `(or` Expression Expression`)`
Char ::= `X` | `Y` | `Z`

**Semantic Domains:**
Truth-Value (see Watt's Appendix D.1.2)
Characters = $\{x, y, z\}$

String-Set = Characters$^*$ $\rightarrow$ Truth-Value
*only-empty-string*: String-Set
$\mathit{only\text{-}empty\text{-}string}(s) = (s = \mathit{nil})$
*one-char*: Characters $\rightarrow$ String-Set
$\mathit{one\text{-}char}(c)(s) = (s = (c \bullet \mathit{nil}))$
*union*: String-Set $\times$ String-Set $\rightarrow$ String-Set
$\mathit{union}(S_1, S_2)(s) = \mathit{either}(S_1(s), S_2(s))$ — see Watt's Appendix D.1.2 for *either*
*member*: Characters$^*\times$ String-Set $\rightarrow$ Truth-Value
$\mathit{member}(s, S) = S(s)$

Value = String-Set

**Semantic Equations:**
*valuation*: Char $\rightarrow$ Character
$\mathit{valuation}\ [\![\mathtt{X}]\!] = x$
$\mathit{valuation}\ [\![\mathtt{Y}]\!] = y$
$\mathit{valuation}\ [\![\mathtt{Z}]\!] = z$

*evaluate*: Expression $\rightarrow$ Value
$\mathit{evaluate}\ [\![\mathtt{nothing}]\!] = \mathit{only\text{-}empty\text{-}string}$
$\mathit{evaluate}\ [\![\mathtt{(quote}\ C\mathtt{)}]\!] = \mathit{one\text{-}char}(\mathit{valuation}\ [\![C]\!])$
$\mathit{evaluate}\ [\![\mathtt{(or}\ E_1\ E_2\mathtt{)}]\!] = \mathit{union}(\mathit{evaluate}\ [\![E_1]\!], \mathit{evaluate}\ [\![E_2]\!])$

*run*: Program $\rightarrow$ Characters$^*$ $\rightarrow$ Truth-Value
$\mathit{run}\ [\![\mathtt{match}\ E]\!] s = \mathit{member}(s, \mathit{evaluate}\ [\![E]\!])$

(space for answer for problem 3)

5. The Hardware-Plus (HP) company has given you a grant to develop a programming language for their next generation reverse-polish notation calculators. They have hired two consultants, Wayne and Garth, who have laid out the initial design for the language, "Way No", which appears below.

**Abstract Syntax:**
Expression ::= Numeral `enter`
    | Expression Expression Operator
    | Expression `not`
    | Identifier `register`
Operator ::= `+` | `-` | `*` | `/` | `=` | `<` | `and` | `or`

**Semantic Domains:**
Truth-Value (see Watt's Appendix D.1.2)
Integer (see Watt's Appendix D.1.3)
Value = *integer* Integer + *truth-value* Truth-Value
Bindable = Value
Environ = Identifier $\rightarrow$ (*bound* Bindable + *unbound*) (see Watt's Appendix D.1.13)

**Semantic Equations:**
Assume that *valuation* is given.
$evaluate$ : Expression $\rightarrow$ (Environ $\rightarrow$ Value)
$evaluate\ [\![N\ \texttt{enter}]\!]\ env = integer(valuation\ [\![N]\!])$
$evaluate\ [\![E_1\ E_2\ op]\!]\ env =\ operator\text{-}value\ [\![op]\!](evaluate\ [\![E_1]\!]\ env,\ evaluate\ [\![E_2]\!]\ env)$
$evaluate\ [\![E\ \texttt{not}]\!]\ env =$
    **let** $truth\text{-}value(b) = evaluate\ [\![E]\!]\ env$ **in**
    $truth\text{-}value(not(b))$
$evaluate\ [\![I\ \texttt{register}]\!]\ env = find(env,I)$

$operator\text{-}value$ : Operator $\rightarrow$ (Value $\times$ Value $\rightarrow$ Value)
$operator\text{-}value\ [\![\texttt{+}]\!] = arithmetic(sum)$
$operator\text{-}value\ [\![\texttt{-}]\!] = arithmetic(difference)$
$operator\text{-}value\ [\![\texttt{*}]\!] = arithmetic(product)$
$operator\text{-}value\ [\![\texttt{/}]\!] = arithmetic(truncated\text{-}quotient)$
$operator\text{-}value\ [\![\texttt{=}]\!] = relational(=)$
$operator\text{-}value\ [\![\texttt{<}]\!] = relational(less)$
$operator\text{-}value\ [\![\texttt{and}]\!] = logical(both)$
$operator\text{-}value\ [\![\texttt{or}]\!] = logical(either)$
See Watt's Appendix D.1.11 for definition of *logical*, *arithmetic*, and *relational*

HP wants customers to be able to use the language "Way No" engineering applications, but they realize that "Way No" has no way to do repeated calculations. To do this they want the language and its semantics adapted for the following expressions.

**Additional Abstract Syntax:**
Expression ::= ...
    | Expression `then` Expression
    | Expression Identifier `assign`
    | Expression `timesRepeat` Expression

Informally, the semantics of these expressions is as follows. An expression of the form "$E_1$ `then` $E_2$" first evaluates $E_1$ then evaluates $E_2$ and returns the value of $E_2$; if $E_1$ fails, so does "$E_1$ `then` $E_2$." An expression of the form "$E$ $I$ `assign`" evaluates $E$, and both assigns that value to $I$ and returns it.

An expression of the form "$E_1$ `timesRepeat` $E_2$" evaluates $E_1$, and if it is an integer strictly greater than 0, evaluates $E_2$ the number of times given by the value of $E_1$ and returns the value of the last evaluation of $E_2$. The value of $E_1$ is only computed once. As an example, the result of the following expression is 7.

```
4 x assign
then
3 timesRepeat (x 1 +) x assign
```

In an expression of the form "$E_1$ `timesRepeat` $E_2$", if the value of $E_1$ is not an integer, then the entire expression fails. If the value of $E_1$ is 0 or negative, then the value of the entire expression is the value of $E_1$.

(a) (20 points) What additions or changes, if any, in the semantic domains are needed to accommodate the additional abstract syntax? List the additions and changes in detail. You may cite parts of Watt's book for the operations of any new or changed semantic domains you may need.

(b) (10 points) What change, if any, should be made to the type of the semantic function *evaluate* to accommodate the additional abstract syntax?

(c) (40 points) Give the semantic equations that define *evaluate* for the additional abstract syntax. Also give the semantic equations for any parts of *evaluate* that might need to change. (If you're running out of time, you can get up to 5 points by describing the changes to the others in words.) You may make any reasonable decisions about the language you find necessary; for maximum credit, note what those decisions are. You may use recursive functions in your semantic equations if necessary. (Hint, be sure to type-check what you write!)