Problem Set B

Due Thursday, October 2, 2025 at 6:00 PM

Page limit: 3 pages (single-spaced)

Name: (Tripp) Milton Lamb

Date: 10/01/2025

# [25 points] This question deals with short-circuit boolean evaluation (SSBE).

## What is short-circuit boolean evaluation?

Short-circuit boolean evaluation is when run-time optimization leads to avoiding evaluating unnecessary expressions when determining the evaluation of a parent conditional expression. For example, if the first expression of an AND conditional evaluates to false the other side of the AND expression is unnecessary to evaluate to know the value of the whole parent expression since for an AND expression both sides must be true. An OR conditional is similar except that it short circuits if the first expression evaluates to true.

## What are the advantages of short-circuit boolean evaluation?

The advantage of short-circuit boolean evaluation is that unnecessary evaluations can be avoided which speeds up execution time.

## What are the disadvantages of short-circuit boolean evaluation?

The disadvantage of short-circuit boolean evaluation is that unnecessary evaluations (in the context of the conditional expression) will be avoided which means any side effects of the short-circuited evaluations will not occur.

## How would you show whether a language supports SSBE? Choose one of C++, Java or Python. Submit a short program that shows, unambiguously, whether it does or does not support it. (Be sure to run the program!) Explain your answer.

# [25 points] This question deals with the concept of a variable.

## What does it mean to de-reference a variable?

This is strange wording, so it is a little difficult to answer. Normally dereferencing is used to talk specifically about pointers (a type of variable). In actuality, all variables redirect an executing program to a value at a memory location, so de-referencing a variable is actually getting the value at the variable’s stored location. This is what programming languages abstract away.

But dereferencing a pointer variable contains another step. A pointer variable stores a reference to another variable which in turn stores a reference to a memory location. De-referencing a pointer means to access the value that the pointer points to rather than the pointer reference itself.

## What’s the difference between a variable and an expression?

A variable stores data and the information necessary to parse that data as a value. A variable is also used to store the result of an expression. An expression is a set of operands (variables, literals, and other expressions) and operators that are to be evaluated. A variable is a reference to a value at a memory location while an expression will evaluate to a value.

## We’ve introduced six things that comprise a variable. Choose three of them and explain what they are in your own words.

Identifier – the name of a variable. It is used by the program to determine which memory location to retrieve data from.

Value – the value associated with the variable. It exists in memory at a specific memory location. It exists in memory as 1’s and 0’s but is given context by the associated type.

Type – the kind of data stored in a memory location. It tells the program how to treat the data being retrieved by the address associated with an identifier.

# [25 points] For each of the following pairs of concepts, briefly describe an important difference between the two with respect to programming languages. Do NOT merely define each term – differentiate them!

## Operator vs. operand

An operator is an action while an operand is a thing (value) that action (operator) will be applied to. Common operators are actions (mappings/functions) such as ‘add’, ‘subtract’, ‘negate’, and ‘or’. Operators are not the symbols used to represent such actions. An operand is a literal, variable, or expression to be acted upon.

## **break** statement vs **continue** statement

The **break** and **continue** statement are both used to control loop statements. **Break** is used to early exit from the loop while the **continue** statement exists to short-circuit the current iteration of the loop and move immediately on to the next iteration.

## Copy semantics vs. reference semantics

Copy semantics and reference semantics both refer to ways an assignment statement can behave. A copy assignment literally creates a duplicate of the data at one memory location and places it in another. The two memory locations can be changed without affecting the other. A reference assignment creates a pointer to the same memory location. There is only one memory location and a change to it by one reference affects the value referred to by another.

# [25 points] We have looked at three different strategies for precisely representing semantics in a programming language.

## Name each strategy and (in your own words) describe how it represents semantics.

Operational semantics - describes the meaning of a program by describing step by step the effect of each statement on the complete state of the machine it is running on. This is done by breaking down a high level language into some intermediary language. This language must be low level enough to have completely obvious and unambiguous meaning (at least at the level of investigation); however, it must be high enough level to be easily understood.

Denotational semantics - while operational semantics focuses on the steps that cause a change of state, denotational semantics focuses on correctness and the absolute state of the machine at each step of the program. Denotational semantics is closely related to proof by induction, and offers a strong mathematical framework which allows proofs of algorithms to be possible. Denotational semantics breaks a program down recursively and treats state as a list of tuples of id-value pairs of every variable in the state of the program.

Axiomatic semantics - deals with defining pre- and post- conditions to every statement in an algorithm effectively creating a complete description on the bounds of the state. All of this can be done without actually running a program. A combination of pre-condition and statement define the post-condition of that statement which in turn becomes the pre-condition of the next statement.

## Of the three, which one do you think is most useful to you in understanding semantics and using that understanding while creating programs? Be sure to identify your choice, and explain why you think it will be most useful to you.

I would say the axiomatic semantics is the most useful for understanding the semantics of a program. While I believe that all are useful to different degrees during the defining of a language; I would also say axiomatic semantics although limited would also be the one most likely to offer practical support. As of now I find it dubious the other two would offer much practical use when writing real world programs.

From the examples in class; in our textbook Concepts of Programming Languages; and another text book Programming Languages Design and Implementation, I remain unconvinced that rewriting a program into a ‘medium’ level language offers any benefit to understanding over natural language or another more well understood high-level language. The concept of rewriting a program so that you are only focused on the level of abstraction of interest is important in good code design; however, this is a more general concept than that of operational semantics and so far find operational semantics unnecessary for practical coding.

Denotational semantics shines for specific algorithmic solutions to well-characterized problems and their rigorous proofs. However, for software design as a whole it seems rather impractical. The idea that one would break down 100,000+ line programs dealing with user interfaces, network communication, and data format conversions into denotational semantics (particularly on the company’s dime) seems absurd. An exception to this worth mentioning is life-critical software (such as certain aircraft subsystems and hospital equipment).

A more informal version of axiomatic semantics is something I use on a near-daily basis. Axiomatic semantics is not so different from well-defined interfaces to functions and programs. A well-defined interface is more than just the type definitions it is also the valid ranges of the data within those types. Focusing axiomatic semantics at the level of a single procedure rather than a whole program can provide a good indication of when refactoring needs to take place. A ballooning post-condition is a good indication that something is wrong in the structure of your code. It is an indication that patterns like dependency injection or rewriting expressions should be considered. Finally, axiomatic semantics can be a very useful tool when trying to sort out overcomplicated conditional nesting in old code.