1. Small-Step Semantics Warm-up

In the class, we have seen a simple boolean language as shown below.

T ::= true | false | if t then t else t

v ::= true | false

All the below sub-questions pertain to the above language.

1. What is the single-step evaluation of the term **if true then if false then false else false else true.** Write the rule name as well the result of single-step evaluation in the form of t → t ′.

To evaluate the given term using single-step evaluation, we need to apply the appropriate reduction rule. Let's break down the term and apply the correct rule:

The term is: if true then (if false then false else false) else true

We can use the if-true rule on this term: That rule says that if the condition of an if-expression is true, we can reduce it to its "then" branch.

The name of the rule and the result of the single-step evaluation are:

if-true: if true then (if false then false else false) else true −→ if false then false else false

In this one step, we've simplified the outer if-expression by removing the true condition and the else branch, and we are left with the inner if-expression in the "then" branch.

1. What is the final result of evaluation of the term **if true then if false then false else false else true**? Represent it using multiple-step relation.

To interpret the expression "**if true then if false then false else false else true**" according to the multi-step relation,

the following sequence of steps should be taken by applying appropriate reduction rules: Here is the breakdown:

1. We first apply the if-true rule to the outer if-expression:

if-true: if true then (if false then false else false) else true −→ if false then false else false

2. Now we got "**if false then false else false**". We apply the if-false rule:

if-false: if false then false else false −→ false

3. We have obtained a value, so the evaluation here stops.

Representing this using the multiple-step relation, we get:

if true then if false then false else false else true −→\* false

Herein, the symbol −→\* denotes the reflexive and transitive closure of the single-step relation, so that means the term on the left reduces to the term on the right in zero or more steps. The final result of the evaluation is false.

1. Write all intermediate steps for the evaluation of the term **if true then if false then false else false else true.** That is, how many steps does it take to evaluate to a value? Write the name of the rule used in each step. Your answer should be of the form

t rulename −−−−−−→ t ′ rulename −−−−−−→ t ′′ rulename −−−−−−→ . . . .

Certainly, I will write out all the intermediate steps for the evaluation of the term "if true then if false then false else false else true", including the rule name used in each step. Here is the detailed evaluation:

if true then if false then false else false else true

whether-

if true––––––→ if false then false else false

if-false−−−−−−→ false

Let's break this down:

1. In the first step, we apply the if-true rule to the outer if-expression. This rule states that when the condition of an if-expression is true, we can reduce it to its "then" branch.

2. We now apply the if-false rule to the remaining if-expression: the if-false rule says that an if-expression is reducible to its "else" branch when its condition is false.

3. Since the second step returns a value condition, that is (false), the evaluation stops here. In all, this term is evaluated in 2 steps into a value.

2. **Boolean Arithmetic Language**

Suppose, I extend the above language as follows and refer to it as boolean and arithmetic language:

n ∈ Z

t::= · · · | n | t1 + t2

v::= · · · | n

where . . . refers to extending terms and values, respectively, defined in the original boolean language. That is, it extends the set of terms to include integers, denoted by n, and addition operation for two terms. For example, 1 + 2, 10 + 43 + 56 are valid terms of the extended language. The following questions use the above extended language.

1. Is 1 + true a valid term of the boolean and arithmetic language?

No, 1 + true is not a well-formed term in the boolean and arithmetic language.

1. Type mismatch: Most programming languages and formal systems combining boolean and arithmetic typically introduce clear separation between boolean values (true/false) and numeric values (integers, floats etc.)

2. Incompatible Operands: The addition operator (+) is usually defined for numeric operands, not for the combination of a number with a boolean value. It expects both operands to be of the same type, usually numeric.

3. Implicit Type Conversions: Some languages may automatically convert true to a number, usually 1, when used in some numerical operations, but this is generally bad practice and not supported everywhere. Even in those languages where such conversions might be possible, it is generally avoided because of the confusion and errors it may cause.

4. Formal definition of language: In the formal definition of boolean and arithmetic languages, usually, operations are defined separately for boolean and arithmetic expressions. Additions are supposed to be defined for arithmetic terms, and logical operations for boolean terms.

5. Strong typing: If this expression were written in, or presented to, a strongly typed language or formal system, it would be rejected immediately on account of the type incompatibility between the numeric value 1 and the boolean value true.

Instead of 1 + true you would normally have to do either of:

1. Use only arithmetic on numeric values: 1 + 1

2. Use only boolean operations on boolean values: true AND false

3. When one needs to mix types-which is usually discouraged-one should perform a type conversion using a type conversion function: 1 + to Number(true).

In formal language theory, and indeed, in most conventions of programming, logical consistency demands a clear distinction between the types of values and operations that may be performed on them to prevent errors.

1. Intuitively, the term t1 + t2 should evaluate t1 and t2 to some integers n1 and n2 and return the result of n1 + n2. How many small-step rules exist for term addition? Write the inference rules by giving them suitable names.

In order to add the terms together,

t1+t2 --> t1​ +t2

Next, we define the small-step operational semantics for this operation. The intuition of small-step semantics is that the evaluation proceeds by a series of small steps until it reaches a final value, that is, a normal form (in this case, an integer). What we want to specify in this respect is how each term of an addition expression can be reduced to a simpler one.

For term addition in the extended boolean and arithmetic language, there are three small-step evaluation rules. These rules dictate how the addition term t1 + t2 should be evaluated, eventually yielding a result of n1 + n2, where n1 and n2 are integers.

1. E-Add1 (Evaluate the Left Operand)

If the left operand t1 is not a value, you need to evaluate t1 first. This rule applies to cases where t1 itself is not fully evaluated yet.

t1🡪 t1' / t1 + t2 🡪t1' + t2’

2.E-Add2 (Evaluate the Right Operand)

If the left operand t1 is already a value (an integer), but the right operand t2 is not, then you evaluate t2. This rule applies when the left term has been fully evaluated but the right term has not.

\frac{t\_2 \xrightarrow{\text{step}} t\_2'}{n + t\_2 \xrightarrow{\text{E-Add2}} n + t\_2'}

Where n is an integer.

3. E-Add (Addition of Two Integers)

Once both t1 and t2 have been fully evaluated to integers, you can perform the actual addition and return the result.

\[

\frac{}{n\_1 + n\_2 \xrightarrow{\text{E-Add}} n\_1 + n\_2}

\]

Summary of Small-Step Rules:

E-Add1: Evaluate the left operand if it’s not yet a value.

E-Add2: Evaluate the right operand once the left one is a value.

E-Add: Once both operands are integers, perform the addition and return the result.

These rules ensure that each operand is evaluated individually before the final addition is performed.

1. Using the small-step rules in the above subquestion, evaluate the term 1 + 2 + 3 until it yields a value. Your answer should be of the form 1 + 2 + 3 rulename −−−−−−→?? rulename −−−−−−→?? . . . . (Note that you should fill in ?? and . . . until your evaluation stops, i.e., your final term is a value from the catgeory v).