University of Massachusetts Lowell — Comp 3010: Organization of Programming Languages

Assignment 1

Name: ***[[TODO: HARISHWAR REDDY ERRI]]***

UML ID: ***[[TODO: 02148304]]***

Collaborators: *[[TODO: Put your collaborators here, .]]*

Make sure that the remaining pages of this assignment do not contain any identifying information.

1 Small-Step Semantics Warm-up (15 points)

$$t ::= \textsf{true} \mid \textsf{false} \mid \textsf{if}\ t\ \textsf{then}\ t\ \textsf{else}\ t$$
$$v ::= \textsf{true} \mid \textsf{false}$$

$$\textsc{Name}\ \frac{a_1 \in A \quad a_2 \in A}{a \in}\ \text{side condition}$$

Steps example:

(a) [[*TODO: Answer Question 1(a)*
To evaluate the given term using single-step evaluation, we need to apply the appropriate reduction rule newline 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. .*]]

(b) [[*TODO: Answer Question 1(b)*
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.
Here in, 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. .]]

(c) [[*TODO: Answer Question 1(c)*
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$\longrightarrow$ 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 (15 points)

(a) [[*TODO: Answer Question 2(a)*
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. .]]

(b) [[*TODO: Answer Question 2(b)*
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 the 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.

$$\frac{t1 \to t1'}{t1 + t2 \to t1' + t2'}$$

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.

$$\frac{t_2 \xrightarrow{\textit{step}} t_2'}{n + t_2 \xrightarrow{\textit{E-Add2}} n + t_2'}$$

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 .]]

(c) *[[TODO: Answer Question 2(c)*
Now, let's see the expression 1+2+3 under the small-step rules we have defined. We'll keep using the rules until there is nothing left to do - that is,
when we will have a final value.

$$1 + 2 + 3 \xrightarrow{\text{E-PlusCompute}} 3 + 3 \xrightarrow{\text{E-PlusCompute}} 6$$

Let us go through each step:
1. We apply the E-PlusCompute rule to the leftmost addition, 1 + 2, because both 1 and 2 are values, hence we can immediately compute their sum.
2. We apply the E-PlusCompute rule to 3 + 3 because both operands are values, hence we can compute their sum. 3. The result, 6 is a value so the evaluation halts here.
Notice that we did not need to use the E-PlusLeft nor the E-PlusRight rule above, because our terms were all already values. If we had more complicated terms we would have had to use those rules to reduce sub-expressions first.
This reduction illustrates how the small-step semantics work: we do one thing at a time, from left to right, until we are done and get a value..]]