**Test Case 1:**

Hello everyone. Today we're going to talk about the Interpreter Activity, and as always, we hope that you will have gone through this either by yourself or with some friends before watching this video. Your objectives are to be apple to read the source cold for a simple interpreter and to understand how everything words. We're going to talk about lifting and how we make actual Haskell functions work on our interpreter types, and then we're going to add a few features, namely, we're going to add valuable and comparison operations. [MUSIC] Now first let's talk about the types. We have two levels for this; we have a type to represent the source code that the programmer types in and that's going to be the Exp type from lines five through seven. We're going to state with two kinds of expressions; the integer expression and the integer operation expression. We're going to have one type to represent the output, that's Val, and we're going to start off with just integer values, but we'll add more later. Finally, to be able to add variables later, we're going to have an environment type. When we have a variable, we need some way to look up what the corresponding value is. The type on the slide is listed as a list of pairs with string and expression, and that's a bug because we want to have a value instead. Now there are some languages that might allow you to put expression inside of variables, that's pretty rare.

**Error:** [MUSIC], able -> apple, code -> cold, works -> words, variables -> valuable, start -> state

**Corrected Document:**

Hello everyone. Today we're going to talk about the Interpreter Activity, and as always, we hope that you will have gone through this either by yourself or with some friends before watching this video. Your objectives are to be able to read the source code for a simple interpreter and to understand how everything works. We're going to talk about lifting and how we make actual Haskell functions work on our interpreter types, and then we're going to add a few features, namely, we're going to add variables and comparison operations. Now first let's talk about the types. We have two levels for this; we have a type to represent the source code that the programmer types in and that's going to be the Exp type from lines five through seven. We're going to start with two kinds of expressions; the integer expression and the integer operation expression. We're going to have one type to represent the output, that's Val, and we're going to start off with just integer values, but we'll add more later. Finally, to be able to add variables later, we're going to have an environment type. When we have a variable, we need some way to look up what the corresponding value is. The type on the slide is listed as a list of pairs with string and expression, and that's a bug because we want to have a value instead. Now there are some languages that might allow you to put expression inside of variables, that's pretty rare.

**Test Case 2**

[SOUND] Hello everyone. &gt;&gt;  This week we're going to take a break from what we've been doing before, we're going to introduce some theory. So in this one we're going to talk about semantics. Semantics comes from a Greek word which means meaning, and this is the mathematical study of the meaning of computer programming languages and their constructs. So in order to do this, we're going to introduce three different things. One is called a judgment, which is something that alerts a property about a piece of computer code or programming language, construct or syntactic object. We have proof rules that define when judgments are valla, and these are recursive inductive objects. And we have proof trees which will look like proof rules stacked together to prove properties about something more compline. Now this treatment of semantics is borrowed heavily from Robert Harper's book, which I cited the last slide. And I highly encourage you to take a look at that book as a different way of approaching programming languages which you may find helpful. So judgment is simply an assertion about some syntactic object we're interested in. So for example, we might want to talk about whether or not numbers are even or odd. Here, three is odd could be a judgement. I am asserting that three is an odd number. Here's another judgement. This is something called big steps semantics, and here what we're saying is that two plus three, and then we have this down arrow, five. And what this is asserting is that two plus three is equal to five. We have this thing, which comes from typing semantics. We'll talk more about what the funny T on its side simple is later. But this is just saying that if you have the expression, 2.4 greater than 3.5, then this is a boolean type of expression. When we have judgements, we want to be able to say when a judgement is valid or when it's not valid. I mean you can just write down any judgement that you want but how do you know it's true or not? How do you know it's accurate? I could have written down, for example, that three is even and that would be false. So we have these definitions, these are called rules, which tell us when a judgment is true or when a judgment is not true. So the way it works, you have this thing that looks kind of like a fraction, it's not a fraction. But you have this horizontal bar, and on top of it you have a bunch of judgements. Now these judgements are called assumptions or premises, and on the bottom you have a final judgement which is called the conclusion. And the idea is if all of the J's on top, J1 through Jn, are true then J on the bottle is also true. And often on the side you will also see a label which gives a name to this rule, and this way we can refer to it easily later. It's possible for a rule to have no assumptions whatsoever, and this is called an axiom. And in that case, what will happen is you'll just have a rule with a line and nothing on top.[music] You'll just have the conclusion and perhaps its label. Another thing that you can see is something called a side condition.

**Error:** [SOUND], &gt;&gt; asserts -> alerts, valla -> valid, complex -> compline, judgment -> judgement, symbol -> simple, bottom -> bottle, .[music]

**Corrected Document:**

Hello everyone. This week we're going to take a break from what we've been doing before, we're going to introduce some theory. So in this one we're going to talk about semantics. Semantics comes from a Greek word which means meaning, and this is the mathematical study of the meaning of computer programming languages and their constructs. So in order to do this, we're going to introduce three different things. One is called a judgment, which is something that asserts a property about a piece of computer code or programming language, construct or syntactic object. We have proof rules that define when judgments are valid, and these are recursive inductive objects. And we have proof trees which will look like proof rules stacked together to prove properties about something more complex. Now this treatment of semantics is borrowed heavily from Robert Harper's book, which I cited the last slide. And I highly encourage you to take a look at that book as a different way of approaching programming languages which you may find helpful. So judgment is simply an assertion about some syntactic object we're interested in. So for example, we might want to talk about whether or not numbers are even or odd. Here, three is odd could be a judgement. I am asserting that three is an odd number. Here's another judgement. This is something called big steps semantics, and here what we're saying is that two plus three, and then we have this down arrow, five. And what this is asserting is that two plus three is equal to five. We have this thing, which comes from typing semantics. We'll talk more about what the funny T on its side symbol is later. But this is just saying that if you have the expression, 2.4 greater than 3.5, then this is a boolean type of expression. When we have judgements, we want to be able to say when a judgement is valid or when it's not valid. I mean you can just write down any judgement that you want but how do you know it's true or not? How do you know it's accurate? I could have written down, for example, that three is even and that would be false. So we have these definitions, these are called rules, which tell us when a judgment is true or when a judgment is not true. So the way it works, you have this thing that looks kind of like a fraction, it's not a fraction. But you have this horizontal bar, and on top of it you have a bunch of judgements. Now these judgements are called assumptions or premises, and on the bottom you have a final judgement which is called the conclusion. And the idea is if all of the J's on top, J1 through Jn, are true then J on the bottom is also true. And often on the side you will also see a label which gives a name to this rule, and this way we can refer to it easily later. It's possible for a rule to have no assumptions whatsoever, and this is called an axiom. And in that case, what will happen is you'll just have a rule with a line and nothing on top. You'll just have the conclusion and perhaps its label. Another thing that you can see is something called a side condition.

**Test Document 3**

Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it doesn't matter in waht oredr the letters in a word are, the only iprmoetnt thing is that the frist and last letter be at the rghit place. The rest can be a toatl mess and you can still raed it without problem. This is bcuseae the huamn mind does not read every letter by itself, but the word as a wlohe.

**Corrected document:**

According to a research at Cmabrigde Uinervtisy, it doesn't matter in what order the letters in a word are, the only important thing is that the first and last letter be at the right place. The rest can be a total mess and you can still read it without problem. This is because the human mind does not read every letter by itself, but the word as a whole.

**Error:** according -> Aoccdrnig, research -> rscheearch, what -> waht, order -> oredr, important -> iprmoetnt, first -> frist, right -> rghit, total -> total, read -> raed, because -> bcuseae, human -> huamn, whole -> wlohe