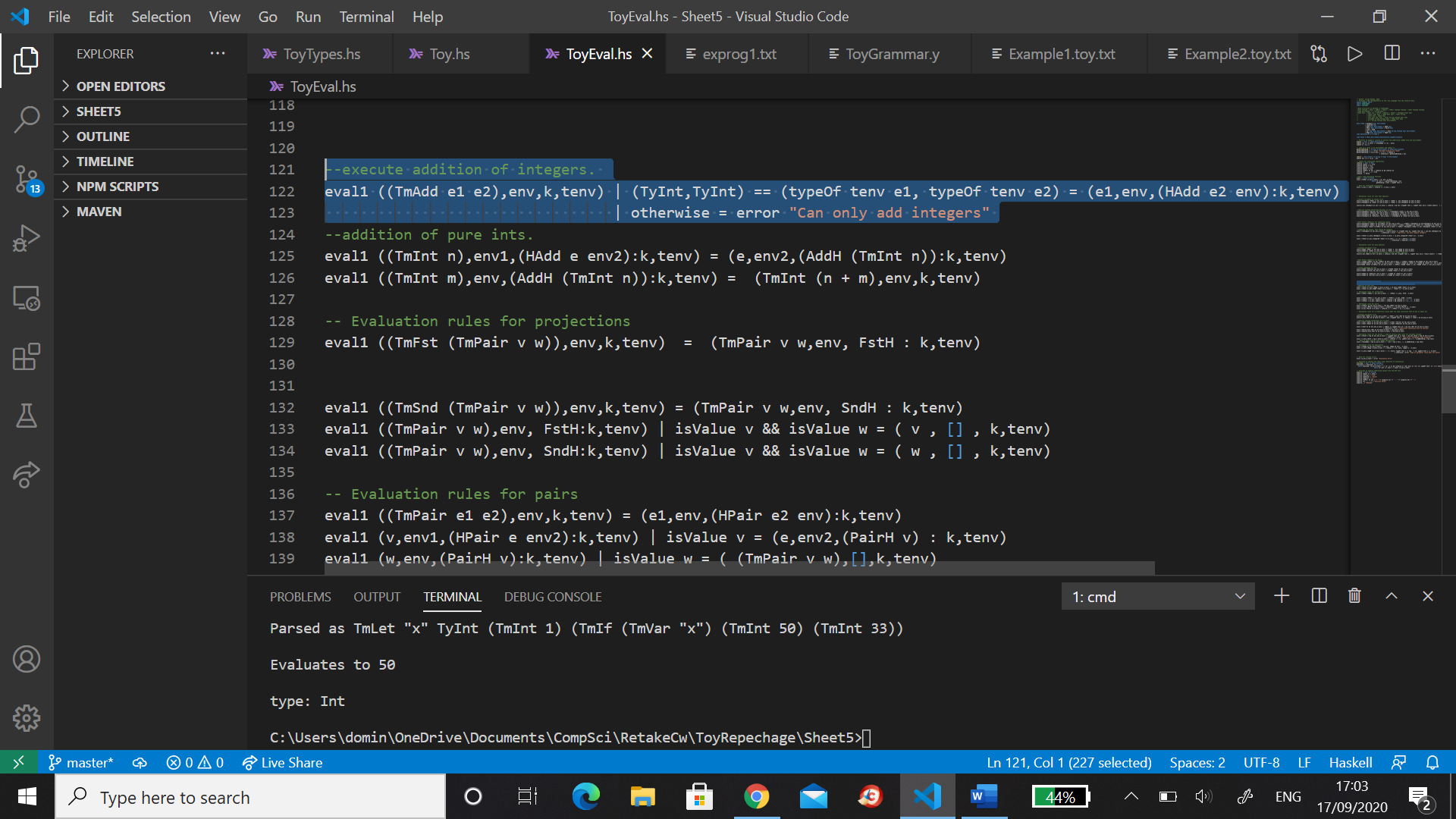
**Toy Language**

**Dynamic Typing**

Ensuring type safety at runtime meant that type checking had to be incorporated into the runtime semantics of the language. Before each runtime ‘state’ was reduced and executed, a type check was required where necessary. This occurred in cases such as addition, comparison, let expressions and anywhere a type error could occur. For example, say we were executing an addition, there would be a type check ensuring that the operands are integers during runtime, prior to execution of addition. The other type rules were enforced similarly. Figure 1 shows a code snippet of how this was implemented.



In order to execute a type check at runtime, a type environment structure had to be added to the runtime states in addition to the environment containing variables. This was used in cases where variables were implemented. For example, in the expression:

Let (x:Int) = 4 in (x + 4)

We are storing the integer x in an environment which the expression ‘(x+4)’ has access to. In addition to this, we also need to store the type of x, so that (x+4) can be type checked. But we are not storing the value, just the type of the variable, so an additional data structure is needed of the form <varname, type>.

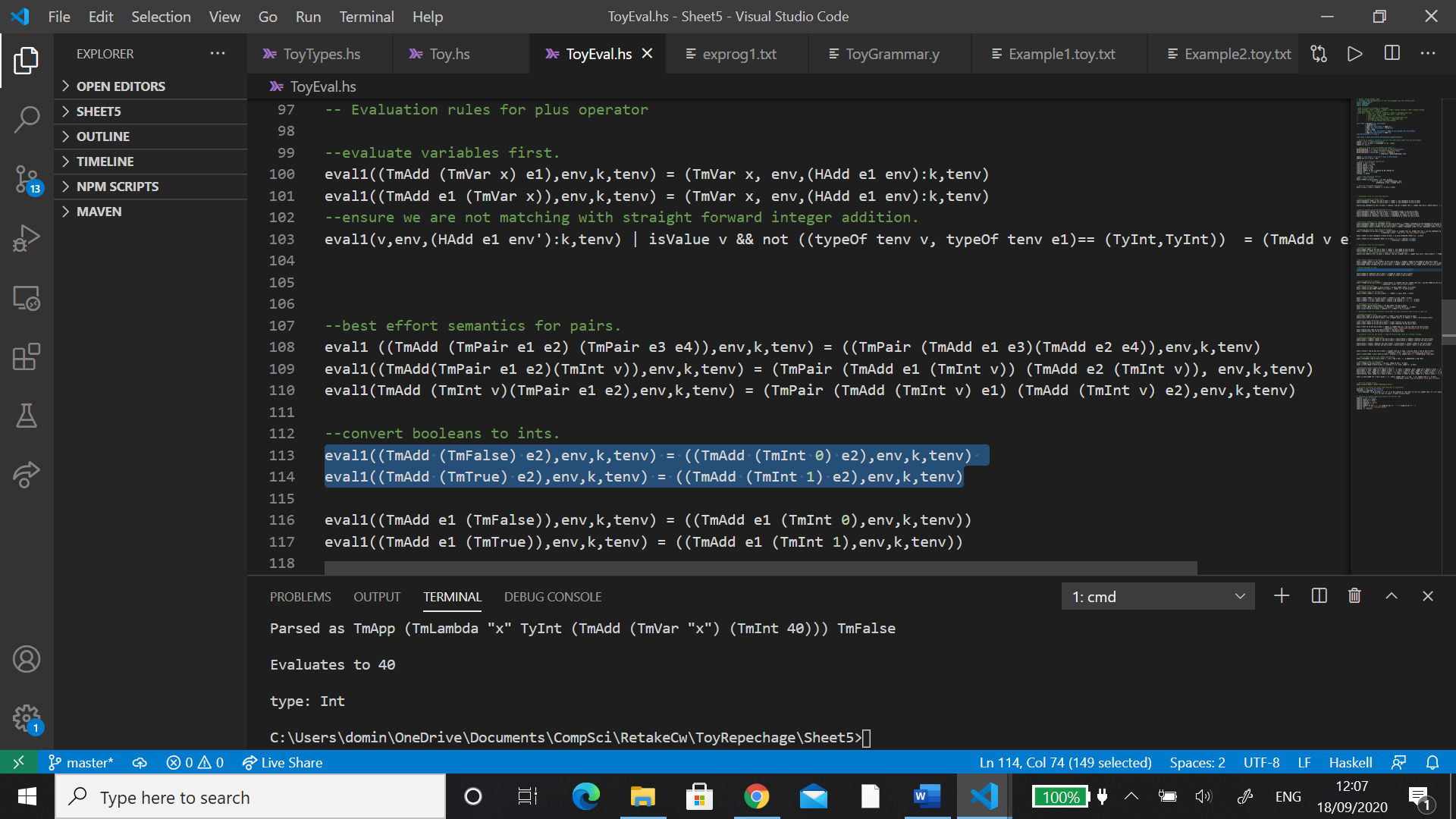
The operation of dynamic typing is essentially checking types of expressions as you go during execution, where needed, instead of checking for absolute consistency on the entire syntax tree. All the type rules are still being enforced, but dynamic typing allows some programs, such as

Let (x:Bool) = true in (if x then 50 else false)

to be executed. To be specific, in the Toy specification, the dynamic type checker does not require that the types of 50 and false are the same, because of the flexible nature of dynamic typing and also to some degree, this program will never evaluate to false, given the value of x, so a Bool could never be returned in principle. The only relevant aspect is what the program evaluates to whilst running, instead of what it could possibly evaluate to. If x were false instead of true, the overall type structure would not change, but now the program evaluates to false, which is not an integer value. The compile time checker would not allow this, but the dynamic checker only considers what is running. The only condition is that the type of x is a bool, which it is. The dynamic checker is only concerned with consistency of each part of execution, as a pose to absolute consistency with all possible run time configurations. It allows for more flexible programming.

**Implicit Casting and Best Effort Semantics**

Implicit casting allowed for the conversion of integer types to Booleans and vice versa, for convenience. For example, the interpreter allows statements like ‘3 + true’, ‘3 < true’ , ‘if 1 then true else false’ and ‘let (x:Int) = 1 in (if x then 4 else 5) to be executed. The interpreter will also implicitly cast assignments such as (x:Bool) = 1 to (x:Bool) = true ; this occurs in let expressions and function application. For instance, (\(x:Int) (x + 40)) false evaluates to 40, because false is taken to be 0. In the language, Ints and Booleans are converted wherever possible for convenience. This was simply done by adding a semantic step of converting these types as necessary, so ‘(1 + true)’ becomes (1 + 1) which is executed without a type error (see below). This allows for more flexible and error free programming.



Adjustments were made for allowing evaluation of pair types. For example, (1,2) + (3,4) evaluates to (3,6) and (1,2) < (3,4) evaluates to (true,true). Furthermore, 2 + (45,23) evaluates to (47,25) and 30 < (20,50) evaluates to (false,true). These features enhance the programmer’s expression.

Also, the conversion of bools to ints is preserved amongst pair types and throughout the language, so there is no confusion in the programmer’s mind.