Testing

# Video demonstrating testing

I have made a video demonstrating the application’s capabilities:  
[bit.ly/paskellRTE](http://bit.ly/paskellRTE) (*Paskell*)

# UI Testing

The simplest component of the project was the UI component of the IDE. Its primary functionalities that require testing include all file access functionality and the custom control component, in the form of the EditorTextBox class; the other functionalities, for example the correct buttons being appropriately enabled, don’t have enough failure cases to be worth considering. Of the potential failure points in the UI, for the file access functionality every possibility can be exhausted by regular use; I demonstrate the majority of these possibilities in the video, but every aspect of file use is addressed by the app. This includes:

* Creating a new file or opening one, including appropriately titling the editor instance “Untitled” or the file name
* Saving files, which also includes specifying a save location for new files (still titled “Untitled”)
* Specifying the save location failing, so not saving and not updating the saved status
* Saving files as, which involves specifying a save location regardless of if the file is new
* Saving all files, which just repeats the saving files process for all files until trying to save a new file fails when specifying a location fails
* Closing files, which if not saved asks if it should be saved
* Saving much like the regular save function
* Cancels if specifying the save location fails
* Cancelling the closing if the user chooses to cancel
* Closing the whole IDE, repeating the closing files process for all files until closing a file cancels or saving that file if not saved fails

For the custom control component, the custom functionality tested includes:

* Custom scrolling
* Properly processing arrow keys and page up/down keys for scrolling
* Updating line numbers to account for text

Testing by exhaustion for all of these test conditions requires only using the IDE and they function correctly or they don’t; there don’t exist exceptional circumstances for them not to be applicable.

# Compiler and Runtime environment

Testing for the backend is more complicated, as there exist countless possibilities for code which cannot be exhausted; instead the system must be tested by identifying testable cases, which can be all tested separately and within given conditions to prove that it does what it is supposed to generally, and so should work for all possibilities.

## Compiler

The compiler is the first level of interfacing with the user by directly processing user input in the editor. The problems that arise are errors in code, and these need to be made useful to the user by providing useful error output which is displayed in the output window. Most of the possible test cases were addressed in the video, but every aspect includes:

* Missing tokens
  + Tested in video
  + This includes an equality and consecutive words
* Invalid tokens
  + Tested in video
  + Very simply tokens are either valid or not, and match the right token type or not
  + Testing for invalid tokens is done by attempting to run code, and invalid code is underlined
  + Matching the correct token types is implicit in the remainder of compilation be able to work properly, so testable in that regard
* Invalid type signatures
  + Tested in video
  + Sequential function maps and invalid bracket nesting throw errors
  + Generally no other errors can be thrown from this context as consecutive tokens that aren’t function maps and ignoring brackets are used to indicate their end, so the errors occur within the expression signature
* Invalid expression signature
  + Tested in video
  + Invalid token types (should only be words)
  + Invalid amount of parameters, the correct amount as indicated by type signature
* Invalid expression definition
  + All the error cases for this are addressed in the design section
  + This includes all the cases of subexpressions not having the right type signature, either for the parameter of a base subexpression or for the base subexpression itself once totally considered

## Runtime environment

Given a working compiler, the runtime environment encounters no failures besides incorrect use of base expressions with generic type signatures. Throughout development, testing of the compiler in one way is possible by observing undefined behaviour in the runtime environment in debugging, but primarily is done simply by examining expression outputs; conversely, once the compiler is set to be functioning properly, the runtime environment can be tested and debugged effectively directly from the IDE itself, using the code editor and having the compiler provide properly constructed expressions.

At one point in development, such testing enabled the consideration of more deeply implemented selection processes. This was a developmental example of the testing being effective while building the runtime environment.