TB-Lang

Joshua Jeffries Hook
Jack Parsons

Language Features

Syntax

Code snippets are included in the appendix, along with brief descriptions. These screenshots showcase the **syntax of our language**, and demonstrate our **syntax highlighting** support package for Visual Studio Code (this package is included in the submission zip file - called 'tb-lang'). A BNF for the grammar is also provided in the appendix.

Variables and Pointers

Users can define, redefine, and access variables in the current scope. These variables follow similar naming conventions to C (e.g. they can't start with digits - this avoids issues such as "123 = True"), and are dynamically typed - allowing simple code which is quick to write and easy to understand. To provide type safety guarantees, variables are type checked by our preprocessor at the start of runtime (see type section). This ensures variables have the correct type when used by functions/operations before executing the code, meaning runtime type exceptions don't occur.

Variables can be referenced using the '**&**(x)' operation. This will get the address of expression x in the store, adding x to the store if it is not a variable. Other variables can then be bound to this reference, thus creating a pointer variable. The '*(y)' operation can be used to retrieve the value stored at address y, and operations/functions can be used to alter it. Examples of these operations can be seen in Figure 1 and Figure 2.

Operations

We have included common operations, with the typical functionality of most languages. Examples of these can be seen in the appendix.

- Maths operations '**+**', '**-**', '*', '/', '**%**', '**^**'.
- Boolean operations '**&&**', '**||**', '**!**'.
- List operations '**:**', '**+**'.
- Comparison operations '**<**', '**>**', '**==**', '**!=**'.
- Assignment operations '**+=**', '**-=**', '*=', '/=', '**&=**', '**|=**', '**^=**', '**++**', '**--**' (these are purely syntactic sugar, included for programmer convenience).

Functions

Users are able to define their own functions, which take in arguments and return a value. The type of a function must be declared before a function can be defined - this aids in static type checking of function calls and definitions (see type section). Additionally, we allow **higher order functions** (e.g. passing a function as a parameter to another function as shown in Figure 13 - this also shows a typical **'map' function** implementation in our language).

The user is then able to define the function definitions themselves, where we use a Haskell-style **pattern-matching** approach to match function call arguments with function definitions. Definitions can be defined as one-liners, or as function blocks - this maintains tidy functions while also allowing for added complexity. Some function definitions are shown in Figure 6. When returning from a function, you can use 'return(x)' (where x is what is being returned). Without defining a return, the return value is implied as None.

Functions are **lexically scoped**, however for programmer convenience we've added global variables. These can only be accessed and assigned from within a function using the 'global' keyword, as shown in Figure 7. Furthermore, pointers can be passed into functions, allowing for pass-by-reference (by default, arguments are passed-by-value) - shown in Figure 8.

We have also included various in-built functions in our language, these can be seen below (examples of these can be seen in the Appendix):

- Functions which operate on lists - **tail** : ([a]) -> [a], **head** : ([a]) -> a, **drop** : (Int, [a]) -> [a], **take** : (Int, [a]) -> [a], **length** : ([a]) -> Int, **get** : (Int, [a]) -> a - these have functionality similar to their counterparts in Haskell (with the exception of "get", which is equivalent to "!!" in Haskell).
- I/O functions - **out** : (a) -> NoneType, **in** : (Int) -> *Stream, **setIn** : ([Int]) -> NoneType (see input section).
- Functions on iterables - **pop** : (*(Itr a)) -> a, **popN** : (Int, *(Itr a)) -> [a], **peek** : (*(Itr a)) -> a, **peekN** : (Int, (*Itr a)) -> [a], **isEmpty** : (*(Itr a)) -> Bool, **hasElems** : (Int, *(Itr a)) -> Bool. These have their intuitive functionality.
- Exception functions - **throw** : (Exception) -> NoneType (see exceptions section).

Control Flow

For control flow, we have provided both ‘for’ and ‘while’ loops (syntactically very similar to C), as well as ‘if’ statements which include optional ‘elif’ and ‘else’ parts - syntactic sugar allowing for more readable and concise code. See Figures 3, 4, 5.

Streams and input

Using the inbuilt ‘in(Int)’ function, users can create pointer variables to input streams. By using pointers, it means we only need to maintain one copy of each of the streams, allowing programmers to access the same stream through different means (letting multiple variables point to the same stream). Additionally, this allows streams to be passed by reference into functions.

Input is retrieved lazily, so operations using a stream will read input into the stream's buffer only if required (i.e. when the operation requires more values than currently in the stream). To improve efficiency, we read up to 100 lines from the input into the stream at a time, as well as include an inbuilt ‘setIn([Int])’ function. This allows the user to declare which streams should be read in, avoiding storing unused streams.

Exceptions

When errors occur during runtime, exceptions are thrown - for example ‘head([])’ would throw an EmptyListException. Programmers can throw exceptions themselves using the inbuilt ‘throw(Exception)’ function (which takes the exception to be thrown). We have included a try-catch statement to allow users to handle thrown exceptions in an appropriate way (see Figure 9). A stack trace is printed to stderr for uncaught exceptions; this message includes function calls on the stack frame, and their line numbers in the program - Figure 10 shows an example of this.

We have provided the following exceptions in our language: EmptyListException, IndexOutOfBoundException, StreamOutOfInputException, InvalidParameterException, NonExhaustivePatternException, InvalidInputException. These exceptions are thrown at intuitive times e.g. InvalidInputException is thrown when reading in a line which doesn’t consist of integers (e.g. it contains a letter).

Type System and Static Type Checking

Our preprocessor checks the type safety of expressions and variables, also ensuring that variables are not accessed before they have been defined. This provides the programmer with confidence that if the program runs, there will not be any type errors or “variable not defined” access errors at runtime. When the preprocessor finds a type conflict, it prints out a helpful error message to stderr, and stops the program - shown in Figure 11.

Due to our variables being dynamic and if statements being unscoped, situations may arise where a variables’ type is ambiguous (e.g. a variable could exit an if statement being of type Int, or being of type Bool, depending on the conditions). To prevent operation/function type errors, in the preprocessor variable types are maintained as a list, and when an operation or function needs to constrain the type of the variable, then we check if the list is a singleton and throw a type error if not - shown in Figure 12.

Users must provide static type definitions for functions when they define them, allowing us to check that functions are well typed. We evaluate the return types of function definitions to ensure that they return the correct type, as well as evaluate the types of arguments in function calls to ensure they follow the functions declared input types.

To provide more functionality, we have implemented a form of **generics**, making functions more adaptable and making their types less constrained by adding more generalization. When a function has a generic in its return type, we don’t know what the actual return type will be (i.e. it may be [a]). When calling a function like this, we evaluate the types of the arguments and match them against the types from the function definition, to determine the actual type of the returned generic. Figure 13 shows an example ‘map’ function which makes use of generics. Generic type classes have also been provided, to add constraints to generics - shown in Figure 14:

- **Ord a -** constrains generic ‘a’ to be an **Int**.
- **Itr a -** constrains generic ‘a’ to be a **Stream**, or a **List**.
- **Eq a -** constrains generic ‘a’ to be an **Int**, a **Bool**, or a **List**.

The implementation of generics provides our language with a form of **parametric polymorphism**, while the addition of class constraints provides it with **ad-hoc polymorphism**.

Syntax Errors
The “posn” wrapper in Alex is used to store the line and column numbers of tokens. Subsequently, any lexing or parsing errors will be printed as syntax errors (to stderr) with the line and column numbers of the invalid tokens. See an example of this in Figure 15.

Execution Model
We generate an abstract syntax tree (AST) using our lexer (Alex) and parser (Happy). This is fed into the preprocessor which runs prior to execution. If no access or type errors are found, then evaluation begins.

Preprocessor
We implemented the preprocessor using **big-step semantics** for evaluation, using a recursive evaluation style. This is because it is a faster and more efficient way to prove type safety, due to the need for fewer rules (compared to small-step semantics) since it skips intermediate steps. The main drawback of big-step semantics concerns programs without a final configuration (e.g. one with an infinite loop). This is not an issue in this instance, as we don’t actually perform iteration or function calls within the preprocessor, we only evaluate the types of expressions (so could never enter an infinite loop). In summary, using small-step would create unnecessary work and add complexity.

Our Haskell ‘process :: Expr -> ProcessState -> IO ([Type], ProcessState)’ function handles all of the preprocessing. It takes in a parsed expression, as well as the current ProcessState - this is a tuple of (local, global), which are both typing environments (they map variables to a list of possible types).

The process function recursively evaluates the inputted expression, and on reaching the base case, it returns an updated ProcessState and a list of types which the expression could be (allowing us to take into account situations where an expression could have different types, e.g. in an if statement). During evaluation, the ProcessState is used (variable types are looked up in the local/global typing environments) and maintained (variable types are modified or added to the local/global typing environments).

Evaluator
Conversely, for evaluating the program we used **small-step semantics**, based on a CESK machine (which models a finite state machine). When evaluating, some programs may not have a final configuration, making big-step inappropriate here. Furthermore, small-step semantics allow us to have control over the exact order of operations, which is necessary for more advanced control features such as runtime exception handling.

Our ‘step :: State -> IO State’ function takes in a State - this is a 6-tuple containing the following:
- Expr - the expression to be evaluated (the Control).
- Environment - a map of variable names to integer addresses.
- Store - a map of Address to evaluated expressions.
- Address - the next available address in the Store, used for optimisation.
- CallStack - a function call stack, used for exception messages.
- Kon - a list of Frames - data types which inform the step function of unfinished computations (the Kontinuation).

The step function takes the parsed program, along with an Environment and Store that contain the inbuilt functions and variables. The function performs a single transformation step to a State, each time evaluating a segment of an expression. The Kontinuation is then used to store any remaining, unevaluated segments of the expression in the order in which they need to be evaluated (modelling a stack frame).

We use the Environment and Store to hold variables, making variable lookup a two-stage process. This means that if a variable isn’t in the Environment, then it can’t be accessed; thus allowing us to maintain scope by storing an older Environment on the Kontinuation, and putting it back into the state once we have left that scope. The Store is passed between all states. This allows for global variable modification, and for different named variables to access the same Store value (this is used for pass-by-reference in functions). We avoid the need for garbage collection by storing the next available address in a similar fashion, meaning that out-of-scope variables in the store will later be overwritten as new variables are introduced.

The IO monad is used by the step function so that I/O can be performed during the evaluation of the program.

Appendix

BNF Grammar

$\textbf{Int} \;=\; [0-9]+$

$\textbf{Bool} \;=\; \mathit{True} \mid \mathit{False}$

$\textbf{Var} = ([a-zA-Z]|\_)([a-zA-Z]|[0-9]|\_)*$

$\textbf{E} \;=\; \mathit{while}\ (\ \textbf{E}\ )\ \{\ \textbf{E}\ \} \mid \mathit{for}\ (\ \textbf{E}\ ;\ \textbf{E}\ ;\ \textbf{E}\ )\ \{\ \textbf{E}\ \}$
$\mid \mathit{if}\ (\ \textbf{E}\ )\ \{\ \textbf{E}\ \}\ \textbf{EL} \mid \mathit{if}\ (\ \textbf{E}\ )\ \{\ \textbf{E}\ \}$
$\mid \mathit{try}\ \{\ \textbf{E}\ \}\ \mathit{catch}\ (\ \textbf{P}\ )\ \{\ \textbf{E}\ \}$
$\mid \mathit{type}\ \textbf{Var}\ \textbf{FT} \mid \mathit{func}\ \textbf{Var}\ (\ )\ =\ \textbf{E} \mid \mathit{func}\ \textbf{Var}\ (\ \textbf{P}\ )\ =\ \textbf{E}$
$\mid \mathit{return}\ (\ \textbf{E}\ ) \mid \mathit{return}\ (\ )$
$\mid \textbf{Var}\ (\ \textbf{P}\ ) \mid \textbf{Var}\ (\ ) \mid \textbf{Var}\ ++ \mid \textbf{Var}\ -- \mid ++\ \textbf{Var} \mid --\ \textbf{Var}$
$\mid \&\ \textbf{E} \mid *\ \textbf{Var}\ =\ \textbf{E} \mid *\ \textbf{Var} \mid *\ (\ \textbf{E}\ ) \mid !\ \textbf{E}$
$\mid \textbf{V} \mid \textbf{B} \mid \textbf{O} \mid \textbf{C} \mid \textbf{L}$
$\mid \textbf{E}\ ;\ \textbf{E} \mid \textbf{E}\ ;$

$\textbf{FT} \;=\; (\ )\ \rightarrow\ \textbf{TL} \mid ()\ \rightarrow\ \textbf{TL} \sim (\ \textbf{PC}\ )$
$\mid (\ \textbf{TL}\ )\ \rightarrow\ \textbf{TL} \mid (\ \textbf{TL}\ )\ \rightarrow\ \textbf{TL} \sim (\ \textbf{PC}\ )$
$\mid (\ \textbf{FTP}\ )\ \rightarrow\ \textbf{TL} \mid (\ \textbf{FTP}\ )\ \rightarrow\ \textbf{TL} \sim (\ \textbf{PC}\ )$

$\textbf{FTP} \;=\; \textbf{TL}\ ,\ \textbf{FTP} \mid \textbf{TL}\ ,\ \textbf{TL}$

$\textbf{TL} \;=\; (\ \textbf{TL}\ ) \mid \textbf{FT} \mid [\,] \mid [\ \textbf{TL}\ ] \mid *\ \textbf{TL} \mid \textbf{Int} \mid \textbf{Bool} \mid \mathit{None} \mid \mathit{Stream} \mid \mathit{Itr}\ \textbf{TL} \mid \textbf{Var}$

$\textbf{PC} \;=\; \textbf{TC}\ ,\ \textbf{PC} \mid \textbf{TC}$

$\textbf{TC} \;=\; \mathit{Eq}\ \textbf{Var} \mid \mathit{Itr}\ \textbf{Var} \mid \mathit{Itr}\ \textbf{Var} \mid \mathit{Ord}\ \textbf{Var} \mid \mathit{Printable}\ \textbf{Var}$

$\textbf{V} \;=\; \mathit{global}\ \textbf{GV} \mid \textbf{GV} \mid \mathit{global}\ \textbf{Var} \mid \textbf{Var}$

$\textbf{GV} \;=\; \textbf{Var} +=\ \textbf{E} \mid \textbf{Var} -=\ \textbf{E} \mid \textbf{Var} *=\ \textbf{E}$
$\mid \textbf{Var}\ /=\ \textbf{E} \mid \textbf{Var}\ \hat{}=\ \textbf{E} \mid \textbf{Var}\ \ =\ \textbf{E} \mid \textbf{Var}\ |=\ \textbf{E} \mid \textbf{Var}\ =\ \textbf{E}$

$\textbf{P} \;=\; \textbf{E}\ ,\ \textbf{P} \mid \textbf{E}$

$\textbf{EL} \;=\; \mathit{elif}\ (\ \textbf{E}\ )\ \{\ \textbf{E}\ \}\ \textbf{EL} \mid \mathit{elif}\ (\ \textbf{E}\ )\ \{\ \textbf{E}\ \} \mid \mathit{else}\ \{\ \textbf{E}\ \}$

$\textbf{B} \;=\; \{\ \textbf{E}\ \} \mid \{\ \} \mid (\ \textbf{E}\ )$

$\textbf{O} \;=\; \textbf{E}\ ==\ \textbf{E} \mid \textbf{E}\ !=\ \textbf{E} \mid \textbf{E}\ >=\ \textbf{E} \mid \textbf{E}\ <=\ \textbf{E} \mid \textbf{E}\ >\ \textbf{E} \mid \textbf{E}\ <\ \textbf{E}$
$\mid \textbf{E}\ :\ \textbf{E} \mid \textbf{E} + \textbf{E} \mid \textbf{E} - \textbf{E} \mid \textbf{E} * \textbf{E} \mid \textbf{E}\ /\ \textbf{E} \mid \textbf{E}\,\hat{}\,\textbf{E} \mid \textbf{E}\ \%\ \textbf{E} \mid \textbf{E}\ \&\&\ \textbf{E} \mid \textbf{E}\ ||\ \textbf{E}$

$\textbf{C} \;=\; \textbf{E}\ ,\ \textbf{C2} \mid \textbf{E}\ ,\ \textbf{E}$

$\textbf{L} \;=\; -\textbf{Int} \mid \textbf{Int} \mid \textbf{Bool} \mid \mathit{None}$

Program Figures

Figure	Screenshot	Description

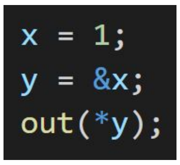
Figure 1	
	`x = 1;`
	`y = &x;`
	`out(*y);`
		This demonstrates using the **address** ‘&’ and **pointer** ‘*’ operators. Variable ‘y’ is declared to point to the address of variable ‘x’.
		Output: 1

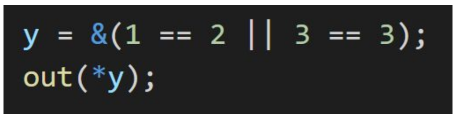
Figure 2	
	`y = &(1 == 2 || 3 == 3);`
	`out(*y);`
		This demonstrates **pointers** using the address ‘&’ operator on an expression. The addressed expression is stored, and then ‘y’ is declared to point to the expression's address.
		Output: True

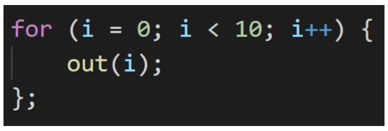
Figure 3	
	`for (i = 0; i < 10; i++) {`
	    `out(i);`
	`};`
		This demonstrates the use of a **‘for’ loop**. The variable ‘i’ is defined and incremented while the condition *‘i < 10’* is being met.
		Output: 0 1 2 3 4 5 6 7 8 9 10

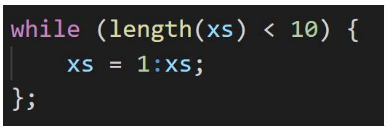
Figure 4	
	`while (length(xs) < 10) {`
	    `xs = 1:xs;`
	`};`
		This demonstrates the use of a **‘while’ loop**. It loops while the condition *‘length(xs) < 10’* is being met.

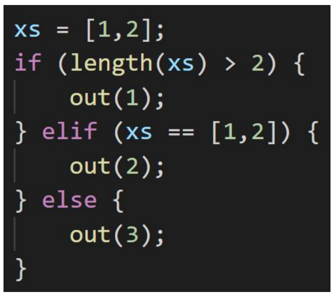
Figure 5	
	`xs = [1,2];`
	`if (length(xs) > 2) {`
	    `out(1);`
	`} elif (xs == [1,2]) {`
	    `out(2);`
	`} else {`
	    `out(3);`
	`}`
		This demonstrates use of **‘if’** statements, with the optional **‘elif’** and **‘else’** parts.
		Output: 2

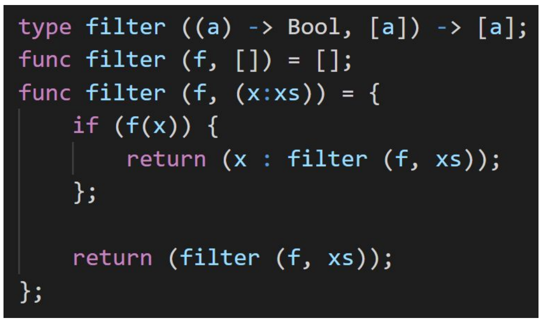
Figure 6	
	`type filter ((a) -> Bool, [a]) -> [a];`
	`func filter (f, []) = [];`
	`func filter (f, (x:xs)) = {`
	    `if (f(x)) {`
	        `return (x : filter (f, xs));`
	    `};`
	
	    `return (filter (f, xs));`
	`};`
		Demonstrate **defining functions** as a one-liner or as a function block.

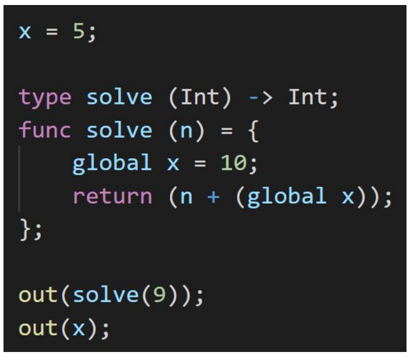
Figure 7	
	`x = 5;`
	
	`type solve (Int) -> Int;`
	`func solve (n) = {`
	`    global x = 10;`
	`    return (n + (global x));`
	`};`
	
	`out(solve(9));`
	`out(x);`
		Demonstrate assigning and accessing **global variables**.
		
		Output: 19 10

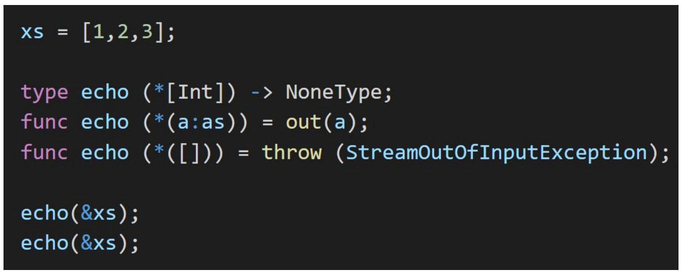
Figure 8	
	`xs = [1,2,3];`
	
	`type echo (*[Int]) -> NoneType;`
	`func echo (*(a:as)) = out(a);`
	`func echo (*([])) = throw (StreamOutOfInputException);`
	
	`echo(&xs);`
	`echo(&xs);`
		Demonstrate **pass-by-reference** into function parameters.
		
		Output: 1 2

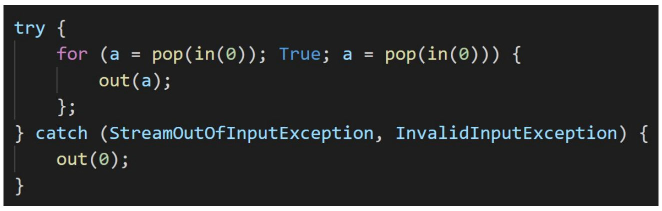
Figure 9	
	`try {`
	`    for (a = pop(in(0)); True; a = pop(in(0))) {`
	`        out(a);`
	`    };`
	`} catch (StreamOutOfInputException, InvalidInputException) {`
	`    out(0);`
	`}`
		Demonstrates **try-catch** statements.
		
		Input: a b
		Output: 0

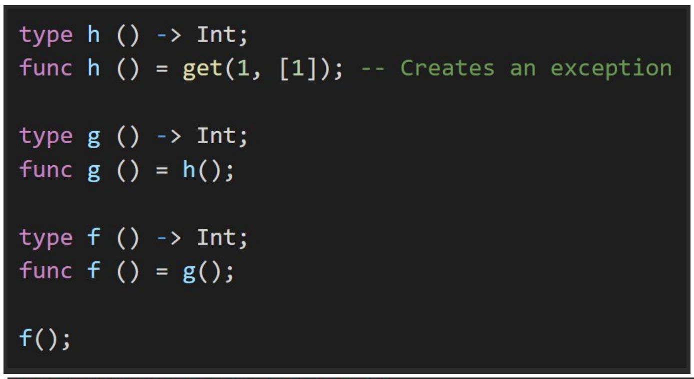
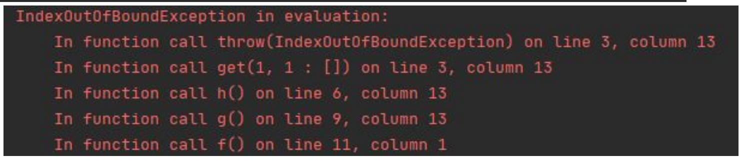
Figure 10	
	`type h () -> Int;`
	`func h () = get(1, [1]); -- Creates an exception`
	
	`type g () -> Int;`
	`func g () = h();`
	
	`type f () -> Int;`
	`func f () = g();`
	
	`f();`
	
	`IndexOutOfBoundException in evaluation:`
	`    In function call throw(IndexOutOfBoundException) on line 3, column 13`
	`    In function call get(1, 1 : []) on line 3, column 13`
	`    In function call h() on line 6, column 13`
	`    In function call g() on line 9, column 13`
	`    In function call f() on line 11, column 1`
		Demonstrates a **stack trace** created from within multiple function calls.

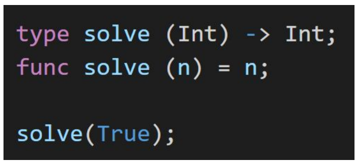
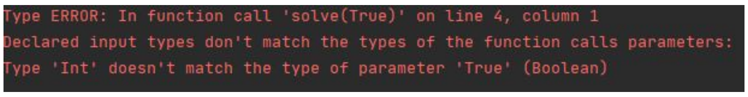
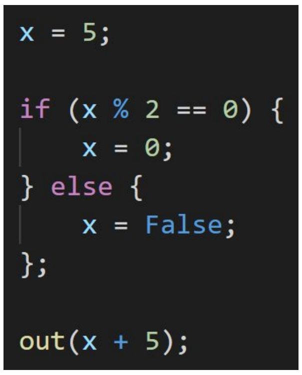
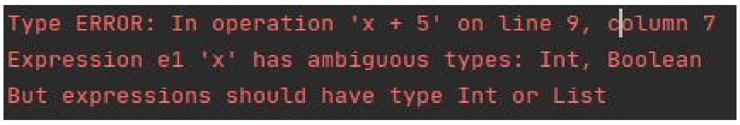
Figure 11	
	`type solve (Int) -> Int;`
	`func solve (n) = n;`
	
	`solve(True);`
	
	`Type ERROR: In function call 'solve(True)' on line 4, column 1`
	`Declared input types don't match the types of the function calls parameters:`
	`Type 'Int' doesn't match the type of parameter 'True' (Boolean)`
		Demonstrates a **type error** within a function call, caused due to the argument types not matching the function definitions types.

Figure 12



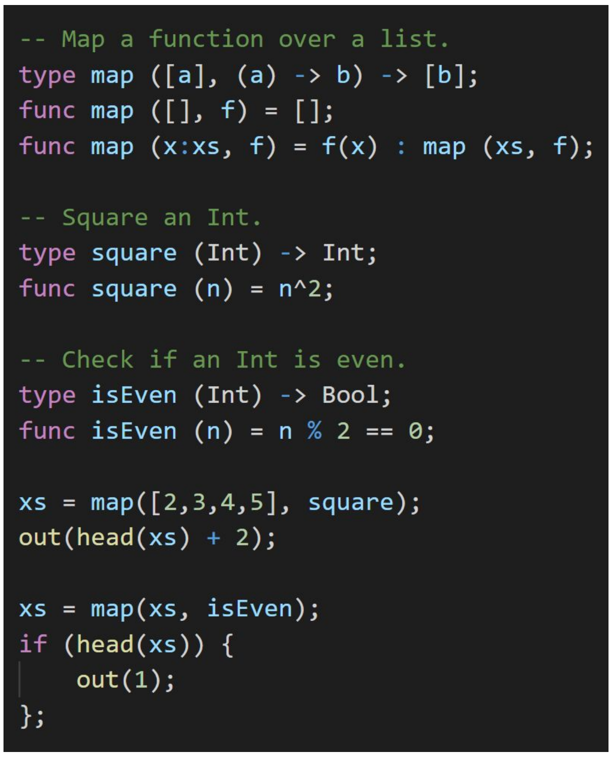
```
x = 5;

if (x % 2 == 0) {
    x = 0;
} else {
    x = False;
};

out(x + 5);
```



```
Type ERROR: In operation 'x + 5' on line 9, column 7
Expression e1 'x' has ambiguous types: Int, Boolean
But expressions should have type Int or List
```

Demonstrates an **ambiguous type error** thrown by the preprocessor, caused as following the if statement, ‘x’ could be an Int or a Bool.

Figure 13

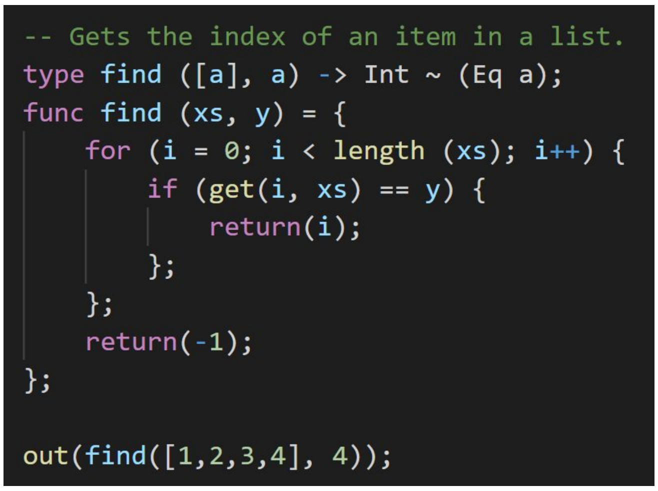


```
-- Map a function over a list.
type map ([a], (a) -> b) -> [b];
func map ([], f) = [];
func map (x:xs, f) = f(x) : map (xs, f);

-- Square an Int.
type square (Int) -> Int;
func square (n) = n^2;

-- Check if an Int is even.
type isEven (Int) -> Bool;
func isEven (n) = n % 2 == 0;

xs = map([2,3,4,5], square);
out(head(xs) + 2);

xs = map(xs, isEven);
if (head(xs)) {
    out(1);
};
```

Demonstrates using **generics**, and **higher order functions** by implementing a “map” function. This acts like a standard map function in most languages.

Output: 6 1

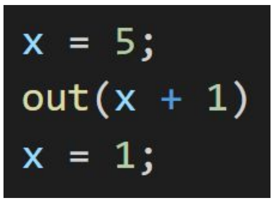
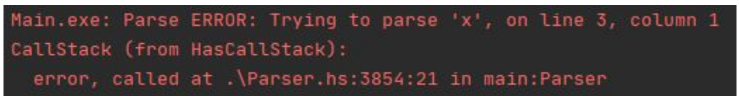
Figure 14



```
-- Gets the index of an item in a list.
type find ([a], a) -> Int ~ (Eq a);
func find (xs, y) = {
    for (i = 0; i < length (xs); i++) {
        if (get(i, xs) == y) {
            return(i);
        };
    };
    return(-1);
};

out(find([1,2,3,4], 4));
```

Demonstrates using **class constraints** to ensure that the values passed in to the list and the value can be equality tested.

Output: 3

Figure 15	
	```
	x = 5;
	out(x + 1)
	x = 1;
	```
	 
	```
	Main.exe: Parse ERROR: Trying to parse 'x', on line 3, column 1
	CallStack (from HasCallStack):
	  error, called at .\Parser.hs:3854:21 in main:Parser
	```
		Demonstrates a **syntax error** (due to a missed semicolon at the end line 2).