<file> -> <*><def> EOF

/* With how the code starts off, it begins from a file and splits off into an abstraction of def* and contains the EOF, signaling the end of the function/variable definition. The code can also contain multiple function definitions or variable definitions*/

<def> -> <funcDef> | <varDef>

/* Each abstraction of def*s can deviate into either being a variable definition or a function definition. Variable definitions can exist inside and outside functions. */

<funcDef> -> 'let' ID '('<paramList>')' '->' <type> <blockExpr>

/* An abstraction of a function definition can begin with the 'let' keyword, followed by an ID, followed by a pair of parentheses and inside these parenthesis would be a list of parameters the function can take in and then is followed by '- >' and then the type signifying a value of what type is going to return whenever the function terminates followed by a blockExpr, which can derive into more syntax that can be statements or expressions. */

<varDef> -> 'let' ID':' <type> ':=' <expr>

/* The abstraction of a variable definition begins with the 'let' keyword, followed by an id and what datatype the id is, whether it be primitive types like ints or bools, a unit type, a condition type, or array types such as integer arrays or boolean arrays, followed by a ":=" being assigned to an <expr>, which can be derived futher.*/

<paramList> -> *(<param> ',') [<param>]

/* The abstraction of paramList can derive into "(param ',), where a sequence of parameters for function definitions are defined, and each param is in the format of ID ":" type. Functions can have multiple parameters and the order of these parameters are important whenever these functions are called. Abstraction of param can also be optional.*/

<param> -> ID ':' <type>

/* Each parameter can be defined as follows" ID ':' type, where ID is a variable that is explicitly useful in function definitions or using built in constructs that require the usage of parameters. The type of each ID is what data type that particular variable is, and mainly for function calls, the correct actual parameter ID or literal value would have to be passed.*/

<type> -> ID

|'unit'

|<type> '[' ']'

| <type> ‘when’ <expr>

/* Abstractions of type can be defined as follows: “type = ID | ‘unit’ | type ‘[‘ ‘]’ | type ‘when’ expr”. Type can be one of the following: ID, ‘unit’ type ‘[‘ ‘]’, or type ‘when’ expr. The first alternate definition gives variables aparticular defined data type, as to promote a static type system. The ‘unit’ implies that IDs can ither return unit, a void type in the language, or using unit for a main() function that should not return anything. The type ‘[‘ ‘]’ allows for arrays of data types to be created, since integer array variables are feasible as well as working with function definitions. The type ‘when’ expr allows particular functions to implement a type condition enforcement rule of sorts, as to prevent functions from being misused.*/

<expr> -> <factor>

|('+' <expr> | '-' <expr> | 'not' <expr>)

| (<left> '*' <right> | <left> '/' <right>)

| (<left> '+' <right> | <left> '-' <right>)

| (<left> [’not’] ‘=’ <right> | <left> ['not'] '<' <right> | <left> ['not'] '>' <right> |

<left> ['not'] '<=' <right>| <left> ['not'] '>=' <right>)

| <left> ’and’ <right>

| <left> ’or’ <right>

| (<left> ':=‘ <right> | <left> ':+=' <right> | <left> ':-=' <right> | <left> ':*=' <right>| <left> ':/=' <right>)

/* For the abstraction of expr, it can either be one of the following. A factor is a simple factor expression with no rightExpr and no leftExpr. Each of these options have a left which equates to an expr, a right which equates to an expr, and an operator that varies depending on what type of expression it is. For unary expressions, operator can be: ‘+’, ‘-‘, or ‘not’. For multiplication expressions, operator can be: ‘*’ or ‘/’. For addition expressions, operator can be: ‘+’ or ‘-‘. For comparison expressions, a not keyword can be used and the operators can be derived into: ‘=’, ‘<’, ‘>’, ‘<=’, or ‘>=’. For and expressions, the operator is: ‘and’ allowing for logical AND conditions. For or expressions, the operator is ‘or’ allowing for logical OR conditions. For assignment expressions, the operator can be: ‘:=’, ‘:+=’, ‘:-=’, ‘:*-‘, or ‘:/=’; In most of these expressions, the expr is repeated onto the RHS allowing for more recursive breakdowns of the syntax to occur. Expressions that contain a left and a right are considered to be Binary expressions.*/

<right> -> <expr>

<left> -> <expr>

/* For the abstractions of right and left, both of them will divulge into <expr> as they can be represented with either more expressions, factors, or gets to be used whenever dealing with conditional expressions. Both the right and left abstraction can be useful for binary expressions that have a left side and a right side and either: perform a mathematic operation on the right and left operands, apply conditions to them, or reassign values to preex*/

<factor> -> <labeledBloxExpr>

| '(' <expr> ')'

| '|' <expr> '|'

| ID

| <funcCallExpr>

| <controlFlowExpr>

| <factor indexOp>

| <arrayLiteral>

| l<literal>

| 'it'

/* In the abstraction of factor, it can derive into one of the above definitions. A <labeledBlockExpr> will be derived into a label followed by blockExpr. The blockExpr will be a collection of statements and terminate with an expr if applicable. For (<expr>), this will mainly deal with Order of Operations or making particular calls. For |<expr>|, this is useful for returning the length of an array of elements. <funcCallExpr> deals with syntax regarding method calls, such as the actual parameters that are passed in whenever the call is done. <controlFlowExpr> deals with control flow statements and constructs such as: repeat, while, if, and jump. <factor indexOp> deal with indexing through the array, such as what subscripts for array indexing are allowed and if the index is within the bounds of the array. <arrayLiteral> allows array objects to be created with a set of values. The derivation of <literal> allow values that variables can be easily set to, such as words for strings or numerical values for integers. The

'it' will be useful for imposing type conditions for method calls, as the 'it' keyword relates to the parameter in a method expression. */

<labeledBlockExpr> -> [<label>] <blockExpr>

/* In the abstraction of labeledBlockExpr, it can derive into these definitions. Labels are optional for this expression, and after the label follows a blockExpr, which can be derived further. */

<label> -> ID ':'

/* In the abstraction of label, it derives into an ID followed by a ':' token. Labels given to block expressions will allow certain block expressions to be named, as if they act as a lamdba expression. */

<blockExpr> -> '{' *<statement> [<expr>] '}'

/* In the abstraction of blockExpr, it deviates into a set of statements to be executed, followed by an <expr> abstraction that can be optional. Block expressions are important when creating a method and defining what statements a method does. */

<statement> -> <varDef> | [<expr>] ';'

/* In abstractions of statement that will be inside block expressions, it can deviate into variable definitions or optional expresssions that behaves when a line of codes gets to execute, such as let x: int := 5;

prints("Hello World"); A single line of code that needs to be executed will terminate with a ';' as to not cause any syntax errors and denote what occured.*/

<funcCallExpr> -> ID '(' (<expr> ',')* [<expr>] ')'

/* In the abstraction of funcCallExpr, it will denote to these tokens, where ID is the name of a function that has been defined elsewhere. Since funcCall Expr deals with syntax that deals with method calls, the name of the method occurs first followed by any actual parameters passed into the function necessary for it to do something. At least 0 actual parameters can be useful for function calls if the function's method header explicitly allows for it. */

<controlFlowExpr> -> <ifExpr> |<loopExpr> | <whileExpr> | <jumpExpr>

/* The abstraction of controlFlowExpr can derive into these control expressions. For these constructs, most of them will impose a condition that determines whether the control flow should execute or keep doing something. For while expressions, it can loop through a minimal set of statements until the condition that keeps the loop going becomes false. Jump statements are useful when trying to break out of a control flow structure or start the next iteration of a

loop. For if expressions, it will exactly behave like an if statement allowing something to occur if the statement is true.*/

<ifExpr> -> 'if'' <cond> 'then' <thenBody> ['else' <elseBody>]

/* If expressions will be if statements that check for a condition, and if the condition is true then it will go into the thenBody, otherwise it will go to the elseBody. Body expressions for else can be optional. Each abstraction of cond, thenBody, and elseBody are defined further.*/

<cond> -> <expr>

/* In abstractions of cond, it can deviate into an expr, which would be considered to be a boolean condition that needs to be evaluated. If the condition within the if expression resolved to be true, the then body for that if statement would get to execute. Otherwise it will check the other conditional statements and a generic case as a fallback. */

<thenBody> -> <expr>

/* In abstractions of thenBody, it can deviate into an expr, which would be a set of deviations expr can be and whatever expression it would need to be, such as a block expression with a set of statements inside or more nested if statements that need to be detailed. The thenBody expressions correspond to if statements whose condition has resolved to true.*/

<elseBody> -> <expr>

/* In abstractions of elseBody, it can deviate into an expr, which would be a set of deviations expr can be and become whatever it would need to be, such as a block expression with a set of statements that make up the elseBody or more nested statments if necessary. Like the thenBody, the else body will contain its own set of statements if the previous conditional statements that needed to be evaluated were false.*/

<loopExpr> -> [<label>] 'loop' <body>

/* In abstractions of loopExpr, it will deal with allowing repetition structures to work as intended, depending on the conditional imposed for the control flow structure. Labels for loop expressions are optional, and the body deviates into an <expr>. A set of statements that are inside the loop's body will only execute for a number of times. */

<body> -> <expr>

/* Similar to other nonterminal abstractions, the abstraction of body gets to deviate into the abstraction of <expr>, where <expr> further breaks down into whatever is necessary, mainly a block expression. */

<whileExpr> -> [<label>] 'while' <cond> 'loop' <body>

/* Abstractions of whileExpr will be derive into the set of nonterminal and terminal symbols, where the condition of the while loop determines whether the loop gets to run and when it gets to stop. Abstraction of body is the set of expressions/statements the while loop executes whenever it gets to run.*/

<jumpExpr> -> <jump> [<atLabel>] [<expr>]

/* Abstractions of jumpExpressions will be used to exit or contine the next iteration of a control flow structure, such as using a return statement to return a value out of a method or to terminate the method early. A break statement will stop the iteration of a loop and break out of it. This type of expression can have an @ label and an expr body, though both of those are optional. */

<jump> -> 'return' |'break' |'continue

/* A jump statement can either be: 'return', 'break', or 'contine' depending on what the jump expression is intended to do. Returns are useful whenever a method needs to return a particular value or terminate the method prematurely. A break statement is useful to exit a while loop or a normal loop and continue with the remainder of the code. A continue statement allows the next iteration of a while loop or a loop to occur, skipping over everything below it. */

<atLabel> -> '@' ID

/* The atLabel abstraction is only useful for jump expressions, as it is used in control flow to determine where the code's continuation happesn at. These label types have to include an ID to jump toward.*/

<indexOp> -> '[' <expr> ']' | '[' <from> '...' <to> ']'

/* Abstractions of indexOp allows for these derivations. The first derivation involves being able to index through an array, and get the element from that index, which is from derivations of <expr>. The second form deals with array slicing, where the elements of an array to be returned/viewed are segmented with the <from> and the <to>, where these nonterminals specify the index to begin at and the index to end from.*/

<from> -> [<expr]

<to> -> [<expr>]

/* Abstractions of from and to deviate into <expr>, where <expr> can be any possible expression. For array slicing operations, it would have to break down into an index value where the slicing should start from if specified. If no <from> index is specified, then it starts at the beginning of the array. If no <to> index is specified, then it goes to the end of the array.*/

<arrayLiteral> -> '[' *(<arrayLiteralItem> ‘,’) [<arrayLiteralItem>] ']

/* If an array needs to be hardcoded into the program, then it can be created as an array literal. Each element of the array is of the same data type, and elements are separated by using commas. Hardcoded Arrays can either have elemtents or nothing at all.*/

<arrayLiteralItem> -> <spread> <expr>

/* For abstractions of this type, array literal item can be values from an arrayLiteral, where it is possible to use slices on this array where expressions can either be values that are hardcoded in or using slices of a different array. */

<spread> -> '...'

/* Abstractions of spread deal with slicing or segmenting an array based on index values. */

<literal> -> STRING_LITERAL | INT_LITERAL | ('true' | 'false') | 'unit'

/* When assigning certain types of values to variables, it can either be one of four types, integers, strings, booleans, or unit. INT_LITERAL allows for integers, STRING_LITERAL allows strings, booleans allow for 'true' or 'false' and unit acts as a return type that allows for functions or variables to not have a type at all.*/

ID -> letter | _ <*>(letter | _ | number)

/* Variable identifiers would be defined with a data type or act as a label. The naming scheme of identifiers requires that it starts with a letter or an '_' character. The remaining characters for the Identifier can contain any sequence of alphanumeric characters or underscores and the number of characters an identifier takes up does not matter. For example: 1Apple is not a valid identifier, but Apple1 or _Apple are valid, since they do not start off with a number. */

INT_LITERAL -> <*>any numerical value

/* INT_LITERAL values will be whole number integer values that can either be stored as a positive whole number or a negative whole number. Integer variables created will be assigned an INT_LITERAL value or these values will be useful for other expressions. The limit of integer values that can be stored have to be range from -2^31 to (2^31 -1). Unary operators such as '+', '-', '/', or '*' can work with integers properly, since it is possible to perform math operations onto these literal values. */

STRING_LITERAL -> ' " ' ~["]* ' " '

/* String literals that must be created must be wrapped using double quotes, so that it does nto conflict with any reserved words the language implements. String literals can be stored into string variables, be used as output, or append itself to an existing string variable.*/

COMMMENT -> ignored

/*Comments in the source code will be ignored by the device running the code, as it is not necessary to look at. This type of comment only works for block comments. Anything inside the block comment tokes will be ignored, and it must have a corresponding closing '*/' token. Similar to C-style cmments, they are ignored.*/

LINE_COMMENT -> ignored

/* When the code is reading a line and finds these comment tokens, it will ignore the rest of the line that comment token is on. Comments are useful, if only to explain the logic behind what the code should be doing at a particular point. Escape characters part of the line comment will be ignored. Similar to C-style line comments.*/

WS -> ignored

/* Whitespace that makes up source code will be ignored, as these whitespaces are optional or not necessary since the language does not implement strictness with indentations, unlike Python. For escape characters part of this abstraction, it will only be possible for these sequences to be used in assigning string variables values or outputting a string to the console. Similar to C-style whitespaces.*/