Contents

Introduction

In the second part of the assignment we were tasked with implementing additional features to extend the capabilities and functionality of the TeaLang programming language. The focus in this part rather than building working components to compile an input program, was instead shifted to using the existing implemented structures and components and make them capable of handling additional features and include new keywords, new handling of syntax, extended type checking on new additions and finally the interpreting and evaluation if results on these new functionalities. Some components required changes in all of the 4 main components of the our Compiler, whilst other could make use of pre-existing functionalities and pick off from there. We will go through each task, and describe in detail what additions were made to the existing components and their intended effect.

Task 1 – Chars and Arrays

Chars

Character data types were the first part of task 1, the focus here was to implement a new data type. This data type simply represents a single character. This character can hold any printable character, and must specifically be of a single character length.

EBNF

Before implementing anything within our compiler, we must first figure out the rules that account for char data types within our language. The new rules are simply an extension of pre-existing rules where data types are concerned.

First we define what a char type object is like so.

<Char_Value> :== " ' " <Printable> " '"

We also add the char keyword to the existing list of type names and that char values are accepted as a literal

<Type> :== "char" | …
<Literal> :== <Char_Value> |

With these new extensions of the EBNF all other uses of char types and values are accounted for since they make use of the literal non-terminal symbol and the type non-terminal symbol. All other cases like checking that the types are matching do not occur in the EBNF but rather are handled by Semantic Analysis.

Lexer

During the lexical analysis phase, we first must define a new transition state process for defining char literals. The process must only allow a single quote to indicate the start of a char literal, then a printable character, followed immediately by a transition through another single quote to a final state. Functionality for escape characters using the backslash was not implemented due to time constraints however this would only require an intermediary transition using the backslash between the first and second state, and then resuming transitions as normal.

When evaluating the final lexeme and move on to creating a new token, to additional token types have been created. The first type is for Char_Value literals,which are created when a character sequence reaches final state 24, the second type of token is for the reserved Char keyword to indicate the type, this is accessed from final state 10, with the condition that the supplied sequence of characters is equivalent to “char”.

Parser

When Parsing char values, no additional changes have been made to the syntax from the EBNF, all that needs to be implemented is the addition of the char keyword and value token types as part of the FIRST set within the parse table for all the cases where other types and literal values are accessed. The char keyword and char value token were added as terminals
This was done by checking in what instance types were listed in the parse table and assigning the char keyword to the same production rule case, and similarly for char value literals, checking where literal value tokens produce a valid production rule case and assigning the same production rules to char value tokens.
Illustrate by example.
Adding the AST Nodes for char values still work as intended as no new production cases and we are adding char type and value nodes as we would ordinarily add any other types or values.

XML Generation

As indicated in the documentation for the previous part, no additional changes have to be made in the XML Visitor for new Node types. This is because we are using a single Node Class, being identified by their type attribute and as long as the AST Generated by the Parser is valid, the XML will traverse the constructed tree and show us the representation based on the construction from the Parser.

Semantic Analysis

For Semantic Analysis we must now include some additional functionalities of defining what char type corresponds to and implement functionality for printing char values. At this stage we would also define the returns types of operations in expression on those types, however these were omitted since there is little use for , adding two chars, taking the comparison of chars, otherwise had these been needed to be implemented we would either need to typecast to integer vale and use ASCII Values when adding or subtracting two chars, or define boolean returns for equality and inequality operations such as == or != as has been previously done for other types.
When the getType() function receives as input either a char_value node, or a variable that when looked up within the symbol table corresponds to the char type, we return the type as char. This operations is then accessed through all other type check such as when assigning and declaring variables, checking function returns and parameters and evaluating function call parameters. Finally, we define the action of print statements on char values.

Interpreter

For the interpreter we perform the same process as outlined before, but this time in relation to the value itself. The type of char has no functionality other than setting new variables, print char values, and comparing char values to see if they are equal, and having the expression evaluate to true or false, returning the appropriate boolean value.

Testing

The folowing program was inputted, it features both chars and the next feature to be implemented, arrays.

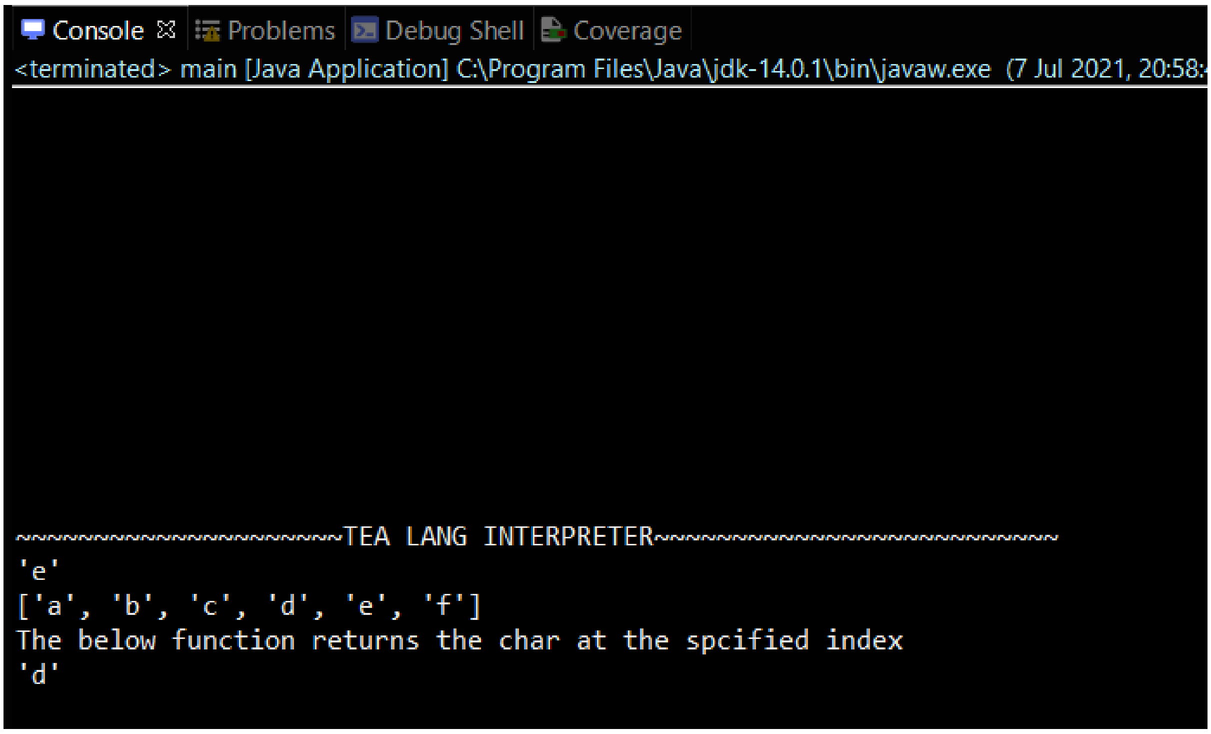
```
let x:char = 'e';

let y:char = x;


char checkCharacter( array[] : char, length : int){

return array[length];


}


print y;

let charArray[6]:char = { 'a','b','c','d','e','f'};

print charArray;

print "The below function returns the char at the spcified index";
print checkCharacter(charArray, 3);
```

XML Generation

```
<ProgramNode>
	<VariableDecl>
		<Variable_Identifier>x</Variable_Identifier>
		<Type>char</Type>
		<Char_Value>'e'</Char_Value>
	</VariableDecl>
	<VariableDecl>
		<Variable_Identifier>y</Variable_Identifier>
		<Type>char</Type>
		<Variable_Identifier>x</Variable_Identifier>
	</VariableDecl>
	<FunctionDecl>
		<Type>char</Type>
		<Variable_Identifier>checkCharacter</Variable_Identifier>
		<FormalParams>
			<Variable_Identifier = array>
				<[]>null</[]>
			</Variable_Identifier>
			<Type>char</Type>
			<Variable_Identifier>length</Variable_Identifier>
			<Type>int</Type>
		</FormalParams>
		<Block>
			<ReturnStatement>
				<Variable_Identifier = array>
					<Array_Indexing>null</Array_Indexing>
					<Variable_Identifier>length</Variable_Identifier>
				</Variable_Identifier>
			</ReturnStatement>
		</Block>
	</FunctionDecl>
	<PrintStatements>
		<Variable_Identifier>y</Variable_Identifier>
	</PrintStatements>
	<VariableDecl>
```

```
		<Variable_Identifier = charArray>
			<[]>null</[]>
		</Variable_Identifier>
		<Integer_Value>6</Integer_Value>
		<Type>char</Type>
		<Elements>
			<Char_Value>'a'</Char_Value>
			<Char_Value>'b'</Char_Value>
			<Char_Value>'c'</Char_Value>
			<Char_Value>'d'</Char_Value>
			<Char_Value>'e'</Char_Value>
			<Char_Value>'f'</Char_Value>
		</Elements>
	</VariableDecl>
	<PrintStatements>
		<Variable_Identifier>charArray</Variable_Identifier>
	</PrintStatements>
	<PrintStatements>
		<String_Value>The below function returns the char at the spcified
index</String_Value>
	</PrintStatements>
	<PrintStatements>
		<FunctionCall = checkCharacter>
			<Variable_Identifier>charArray</Variable_Identifier>
			<Integer_Value>3</Integer_Value>
		</FunctionCall>
	</PrintStatements>
</ProgramNode>
```

Result



```
<terminated> main [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe  (7 Jul 2021, 20:58:
```

```
~~~~~~~~~~~~~~~~~~~~~~~~TEA LANG INTERPRETER~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'e'
['a', 'b', 'c', 'd', 'e', 'f']
The below function returns the char at the spcified index
'd'
```

Arrays

Arrays represent a contiguous block of memory that holds a pre-determined number of elements of the same type. The size of an array is determined during declaration of an array variable and is immutable. Users have to option to initialize the array and populate its contents with values during declaration or leave it empty for now. Later on users can access specific elements of an array through the use of an index enclosed in square brackets after the array variable identifier. The whole array, or a specific element from that same array can be passed as a parameter to a function and so function parameters must also be capable of accepting entire arrays and make use of these arrays within the function scope. Arrays however cannot be returned, but elements of an array can be returned. This decision was take because in the examples shown return types of function declarations are specified as a type before the function name, whist declaring arrays is done by attaching square brackets to the variable identifier not the type as in

Let x[5]:int

Function returns would not support this naming conventions as returns are specified by a type.

EBNF

We update the EBNF to allow for accessing elements within array variables, and for declaring arrays with optional initialization. Locating every time a variable is mentioned, excluding the case for function names, we add an optional component | <identifier>"[" <Expression> "]" , which will indicate an array and the expression will evaluate to the index used in setting the maximum size during initialization and when accessing elements.

Another optional clause that needs to be included is the initialization during variable declaration, where we may allow the users to choose wether or not they wish to initialise the array, this can be added as well as an optional components along with the previous addition. Elements of an array when initialise, are identical to expression found in function calls, so we can save ourselves some time later on in the future when conducting the parser, by setting the initialised array's elements as a sequence of <Actual Params>

<VariableDecl> :== "let" <Identifier> "[" <Expression> "]" ["{" {<ActualParams>} "}"] ";"

The above statement, defines an alternative to variable declaration, which will beincluded along with the original rule, as an optional alternative.

Lexer

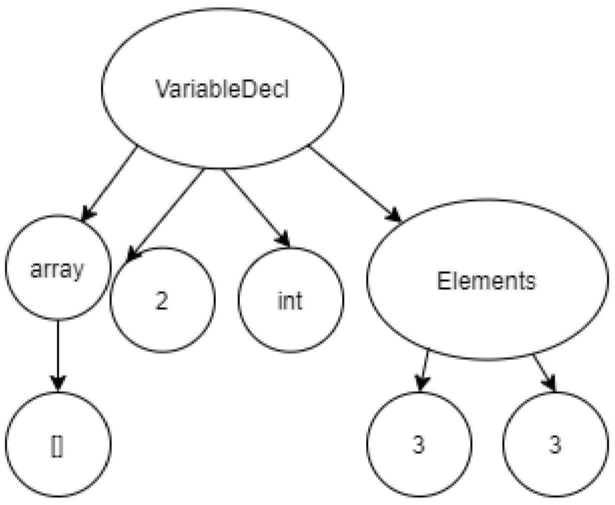
During the lexer phase, we add the square brackets as new token types. Since we already have a category of opening and closing brackets, grouped together because they are all tokens of one charcter length, we can easily add the square brackets to this final state, state 21.

Parser

For the parser, we need to identify wether the current variable is followed by an opening square brackets, this would indicate that the current varaible is an array and so the AST would need to account for this aswell by generating the index as a child of the variable identifier. Ordinarily after encountering a variable identifier within an AST, it would always be a leaf node, so letting this space be taken up by the index to indicate that the identifier referse to an array will not ruin our current implementation on other functionalities. For this case we must extend the lookahead size to include the 2 next tokens and not just one token ahead. This allows us to better predict what type of array is being created. Obviously the square brackets for opening and closing were added as new terminals, and the relevant FIRST and FOLLOW sets derived from thes non-terminals were derived namely for expression escaping in the case of indices.

In the parser 5 new cases are added, one for evaluating the index as an expression, and also for escaping the expresison tree after encountering the closing square bracket. Another which predicts wether or not the array is currently being initialised, and if it is another case fires which adds the elements, by predicting that a Function Call Parameter is next, and then utilising the previously defined cases for function call parameters, since they are identical, and a final node for exitting the array initialisation and terminating the array declaration.

A general AST for an array that is being initialised looks like this.



AST were generated this way, because during the entire process I kept in my mind the current way Semantic Analysis was being done and subsequently the Interpreter and I wanted my Visitors to be able to recognize wether or not a varaible is an array from the first node, and not allow ambiguity between different Statement archetypes.

Semantic Analsysis

In semantic analysis, an array table scope is added which holds the current variables within scopes that are known to be declared as arrays, this was done so that when array indexing occurs, we can check if the variable that is being indexed is an array. This also used in conjunction with parameters, when functions are expecting an array of a certain type, we can cross reference the function call parameter with the array table to see if the variable we are passing on to thte function is in fact an entire array. Similarly for parameters which accept normal types, and are being passed array-elements, we lookup the type value of the array identifier within the symbol table, as we still add the idnetifier here so that we have a record of what value types the array holds.
Similarly for assignment the same checks are performed.
For decleration we check that every element is of the same type, and at any indexing we must check that the type expression evaluates to an int.
In expression trees, we usually append the variable identifier as a leaf node, but know variables that represent array elements that have been indexed now have a child node containing the index. A case is added when evaluating the expression tree, that if the nodeChilds list contains the node with type Array_Indexing, then the current node is a varaible of an array, and the index is referring the specific element, so we retrieve the type of the varaiable from the array and retunr it so the method can process as intended.

Interpreter

In the interpreter stage we include another map, which maps an identifier string, to an array list of strings, this will hold the actual values of the array.
At this stage we must perform another type of check, that is the checking of indices and that they do not exceed the initialised length of the array, as well as declerations do not give us more elements that the declared length.
At any moment an index is referenced we can now evaluate it, to get its value. This bounds-checking could not be done previously because semantic analysis does not perform any acutal calculations to reach the final value, so only at this stage can we evaluate the index expression, and check that it is not larger than the declared length.
When a variable identifier belongs to an array, we pass on a copy of the value found in the array_values map belonging to the that identifier. Similarly when an element index is being referred to , we retrieve the element by locating the array list housing the values, and retriving the specified index from there.

Testing

You ay also refer to the previous section of chars, to see arrays in action.
The following function gets the average of an array of floats print the results, as well as the contents of the entire array.

```
let arr2[4]:float = {2.4, 2.8, 10.4, 12.1};

arr2[1] = 12.7;


float Average(toAverage[] : float, count : int){

let total:float = 0.0;

for(let i:int =0; i < count; i = i + 1;)
{
total = total +toAverage[i];



}



return total;
}

int f( x:int){

return x;

}


let arr4[1+5]:float = {02.4, 34.3, 23.4, 99.3,99.8};

arr4[4] = 12.2;

print "YOOHOO BIG SUMMER BLOWOUT";
print arr4[4];
print arr4;
```

```
let arrw[2]:int;
```

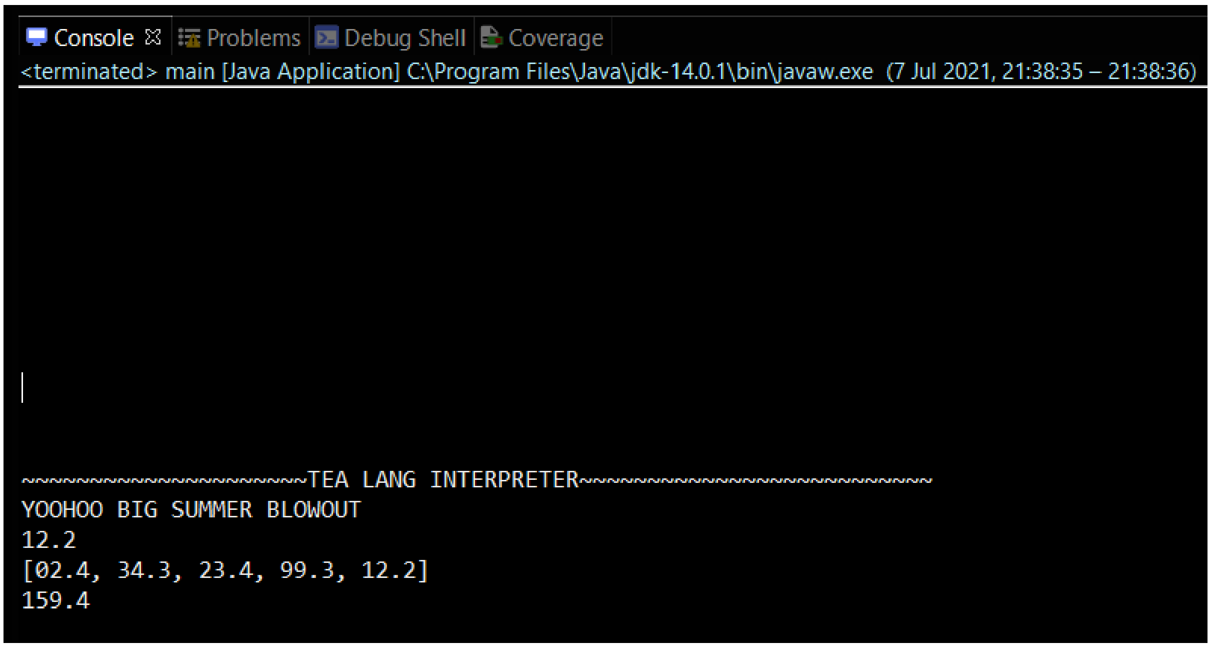
Generated XML

```
<ProgramNode>
	<VariableDecl>
		<Variable_Identifier = arr2>
			<[]>null</[]>
		</Variable_Identifier>
		<Integer_Value>4</Integer_Value>
		<Type>float</Type>
		<Elements>
			<Float_Value>2.4</Float_Value>
			<Float_Value>2.8</Float_Value>
			<Float_Value>10.4</Float_Value>
			<Float_Value>12.1</Float_Value>
		</Elements>
	</VariableDecl>
	<VariableAssignment>
		<Variable_Identifier = arr2>
			<Array_Indexing>null</Array_Indexing>
			<Integer_Value>1</Integer_Value>
		</Variable_Identifier>
		<Float_Value>12.7</Float_Value>
	</VariableAssignment>
	<FunctionDecl>
		<Type>float</Type>
		<Variable_Identifier>Average</Variable_Identifier>
		<FormalParams>
			<Variable_Identifier = toAverage>
				<[]>null</[]>
			</Variable_Identifier>
			<Type>float</Type>
			<Variable_Identifier>count</Variable_Identifier>
			<Type>int</Type>
		</FormalParams>
		<Block>
			<VariableDecl>
				<Variable_Identifier>total</Variable_Identifier>
				<Type>float</Type>
				<Float_Value>0.0</Float_Value>
			</VariableDecl>
			<ForLoop>
				<VariableDecl>
					<Variable_Identifier>i</Variable_Identifier>
					<Type>int</Type>
					<Integer_Value>0</Integer_Value>
				</VariableDecl>
				<Operator = <>
					<Variable_Identifier>i</Variable_Identifier>
					<Variable_Identifier>count</Variable_Identifier>
				</Operator>
				<VariableAssignment>
					<Variable_Identifier>i</Variable_Identifier>
					<Operator = +>

	<Variable_Identifier>i</Variable_Identifier>
						<Integer_Value>1</Integer_Value>
					</Operator>
```

```
                     </VariableAssignment>
                     <Block>
                            <VariableAssignment>
                                   <Variable_ >total</Variable_ >
                                   <Operator = +>
                                          <Variable_ >total</Variable_ >
                                          <Variable_Identifier = toAverage>
                                                 < _Indexing>null</ _Indexing>
                                                 <Variable_ >i</Variable_ >
                                          </Variable_ >
                                   </Operator>
                            </VariableAssignment>
                     </Block>
              </ForLoop>
              <ReturnStatement>
                     <Variable_Identifier>total</Variable_Identifier>
              </ReturnStatement>
       </Block>
</FunctionDecl>
<FunctionDecl>
       <Type>int</Type>
       <Variable_Identifier>f</Variable_Identifier>
       <FormalParams>
              <Variable_Identifier>x</Variable_Identifier>
              <Type>int</Type>
       </FormalParams>
       <Block>
              <ReturnStatement>
                     <Variable_Identifier>x</Variable_Identifier>
              </ReturnStatement>
       </Block>
</FunctionDecl>
<VariableDecl>
       <Variable_Identifier = arr4>
              <[]>null</[]>
       </Variable_Identifier>
       <Operator = +>
              <Integer_Value>1</Integer_Value>
              <Integer_Value>5</Integer_Value>
       </Operator>
       <Type>float</Type>
       <Elements>
              <Float_Value>02.4</Float_Value>
              <Float_Value>34.3</Float_Value>
              <Float_Value>23.4</Float_Value>
              <Float_Value>99.3</Float_Value>
              <Float_Value>99.8</Float_Value>
       </Elements>
</VariableDecl>
<VariableAssignment>
       <Variable_Identifier = arr4>
              <Array_Indexing>null</Array_Indexing>
              <Integer_Value>4</Integer_Value>
       </Variable_Identifier>
       <Float_Value>12.2</Float_Value>
</VariableAssignment>
<PrintStatements>
       <String_Value>YOOHOO BIG SUMMER BLOWOUT</String_Value>
</PrintStatements>
```

```
        <PrintStatements>
                <Variable_Identifier = arr4>
                        <Array_Indexing>null</Array_Indexing>
                        <Integer_Value>4</Integer_Value>
                </Variable_Identifier>
        </PrintStatements>
        <PrintStatements>
                <Variable_Identifier>arr4</Variable_Identifier>
        </PrintStatements>
        <VariableDecl>
                <Variable_Identifier = arrw>
                        <[]>null</[]>
                </Variable_Identifier>
                <Integer_Value>2</Integer_Value>
                <Type>int</Type>
        </VariableDecl>
        <PrintStatements>
                <FunctionCall = Average>
                        <Variable_Identifier>arr4</Variable_Identifier>
                        <Integer_Value>4</Integer_Value>
                </FunctionCall>
        </PrintStatements>
</ProgramNode>
```

Result



```
<terminated> main [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe  (7 Jul 2021, 21:38:35 – 21:38:36)

~~~~~~~~~~~~~~~~~~~~~~~TEA LANG INTERPRETER~~~~~~~~~~~~~~~~~~~~~~~~~~
YOOHOO BIG SUMMER BLOWOUT
12.2
[02.4, 34.3, 23.4, 99.3, 12.2]
159.4
```

Task 2 – Auto

The auto type needs to be able to automatically (hence the name) detect the type of a variable, from the supplied expression either during variable decleration by evaluating the RHS, or by evlauating the expression of a return statement, so that functions may be set to auto during declerations

EBNF

In the EBNF, no changes were made other than including the keyword auto as a viable type, for futre EBNF Rules.

Lexer

As indicated in the previous step, no changes were made to the makeup of the language and how tokens are built. The auto keyword is added as a reserved keyword when processing letters, so that if the supplied token is identical to the word auto, then it is recognised as the auto token.

Parser

In the Parser and AST Generation, again no changes were made, apart from adding the keyword auto as a new terminal, since an added type does not change the syntax and no new additional syntax changes have been made.

Semantic

In the semantic analysis, before adding a function or variable to the symbol table based on its type we first check if the type is auto. If it is not auto we carry on as usual, otherwise we perform the following.
First perform a type check on the expression that we are "autoing". Previously we would need to pass the expected type as a parameter, so now the typeCheck() method has been re-arranged as to work without having an expected type and instead return the resulting type, and check if the types match later on after the method call. This applies to also the normal cases and so we evaluate the expression first, and compare the evaluated and expected type later on instead of processing it all together within the function, just so we can use the same method to determine the type only.
After determining the type by either evaluating the expression during the variable decl, or during the variable initialisation, when we get the evaluated type we instantly alter the value of the AST where the type of the variable/function is held, and change the value from auto to the predicted type.
What this allows is us, is that now when we come to the interpreter stage, the AST now no longer holds auto, but holds the detected type and the interpreter can now work as if the auto was never there but the correect variable type was there all together.
For function declerations the function return type is predicted, when processing return statements and setting the first return statement type encountered, in place of auto. This would mean that later on if another return statement is encountered within the same function, it must be compliant with the type of the original return statement.

Interpreter

Explained above that no changes were needed in this component.

Testing

Overloaded Functions with auto returns were implemented and tested. The functions have a recursive auto definition, and this proved to be of no issue to the semantic analysis to recursively retrieve the value for the variable, through the function of auto return, that returns an overlaoded function that returns an int. All of them were then subsequently set to int, otherwise the interpreter would not have been able to produce results since no operations were defined for auto values.

Input Program

```
let x:auto = 5;

int double(num : int){
```

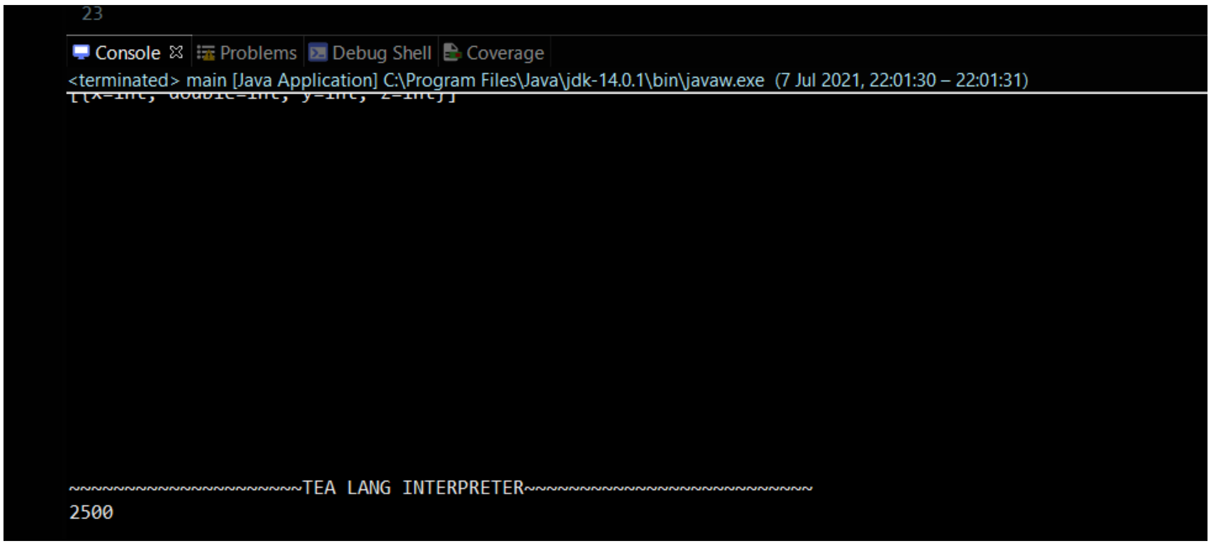
```
return num*num;

}



let y:auto = double( 12);

auto double(word : string){

return double(50);

}

let z:auto = double("nice");

print z;
```

Generated XML

```
<ProgramNode>
    <VariableDecl>
        <Variable_Identifier>x</Variable_Identifier>
        <Type>auto</Type>
        <Integer_Value>5</Integer_Value>
    </VariableDecl>
    <FunctionDecl>
        <Type>int</Type>
        <Variable_Identifier>double</Variable_Identifier>
        <FormalParams>
            <Variable_Identifier>num</Variable_Identifier>
            <Type>int</Type>
        </FormalParams>
        <Block>
            <ReturnStatement>
                <Operator = *>
                    <Variable_Identifier>num</Variable_Identifier>
                    <Variable_Identifier>num</Variable_Identifier>
                </Operator>
            </ReturnStatement>
        </Block>
    </FunctionDecl>
    <VariableDecl>
        <Variable_Identifier>y</Variable_Identifier>
        <Type>auto</Type>
        <FunctionCall = double>
            <Integer_Value>12</Integer_Value>
        </FunctionCall>
    </VariableDecl>
    <FunctionDecl>
        <Type>auto</Type>
        <Variable_Identifier>double</Variable_Identifier>
        <FormalParams>
            <Variable_Identifier>word</Variable_Identifier>
            <Type>string</Type>
        </FormalParams>
        <Block>
            <ReturnStatement>
```

```
                        <FunctionCall = double>
                              <Integer_Value>50</Integer_Value>
                        </FunctionCall>
                  </ReturnStatement>
            </Block>
      </FunctionDecl>
      <VariableDecl>
            <Variable_Identifier>z</Variable_Identifier>
            <Type>auto</Type>
            <FunctionCall = double>
                  <String_Value>nice</String_Value>
            </FunctionCall>
      </VariableDecl>
      <PrintStatements>
            <Variable_Identifier>z</Variable_Identifier>
      </PrintStatements>
</ProgramNode>
```

Result



```
23
Console ⌧  Problems  Debug Shell  Coverage
<terminated> main [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe  (7 Jul 2021, 22:01:30 – 22:01:31)
[{x=int, double=int, y=int, z=int}]

~~~~~~~~~~~~~~~~~~~~~~~~~TEA LANG INTERPRETER~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2500
```

Task 3 – Structs

Structs are new data types, that can hod nested variables and functions and created new variables of these new data types would allow these variables to access the nested varaibles and functions. Structs will be declared using the tlstruct keyword and then are a block is followed that includes variable declerations and function declarations, which instances can reference later on.

EBNF

A struct is created by the keyword tlstruct, a varaible identifier and then a block is followed which only consists of variable and function declerations. With this definition in mind the EBNF for struct decleration is as follow

<Struct_Declaration>:== “tlstruct” <Variable_Identifier> “{“ { <VaraibleDecl> | <FunctionDecl> } “}”

Structs are then initialised by simply allowing a varaible decleration on a variable identifier to be followed by another variable identifier, followed by a semi-colon or colon.

<Struct_Initialisation> :== “let” <Variable_Identifier> <Variable_Identifier> “;” | “:”

For struct variables andn function calls, we add the recognition of the. within a variable id, so we can recognize that the refrencced identifier is an object referencing a strut attribute. Therefore, within the variable_identifier I allowed the addition of a single dot within the identifiers.

Lexer

Within the lexer, the tlstruct keyword was added as a new token type whene evaluating a token built up of only letters. The lexer will then recognize when function calls or variables being referenced by an identifier are part of a struct object and they are referrring to the variable within the struct. Later on the part that is before the dot will be recognized as the struct instance, and the part that comes after as the variable identifier being accessed in the struct.

For transition states, the dot character was allowed to be added only once within a sequence of letters or underscores, so basically when reading a variable identifier, the lexer is allowed to read a single dot and produce a valid transition.

Parser

In the parser we now must build the AST accounting for the new Statemtn type of Struct Decleration, followed by its block of variable declerations and function declerations. This was done by adding a new non-terminal symbol for Struct Decleration that would be used to build the parse table, and a new terminal symbol of tlstruct. The non-terminal symbol was not neccessarry as it is never encountered in the contents of the stack, I had it included so that I could better form out the new additions and help focus on the new parse table additions as adding all new statements helped me in building my AST. Furthermore, a new non-terminal was added, that of Struct Statement, whose FIRST set allowed only the FIRST set of Variable Declaration and Function Decleration, so that we can define that in struct declaration no other statements apart these two are added. Anywhere else, where a type was existed, we now extend the FIRST set which included the type to also include the variable identifier so that struct types can now be replace types in all possible scenarios. These cases were then implemented in the stack functionality to predict the next token, and the AST was built accordingly.

Semantic Analysis

When a struct is created, this is equivalent to creating a new scope accessed only by instances of that struct. So during the Struct Declaration, we take note of the variables and functions, and map the struct name to another map, which maps the variable identifiers for variables and functions to their return types. When a new struct instance is created, the variable needs to be able to call the “struct scope” variables anywhere. When an instance is created is type its set to be equal to the Struct name, and before we first check if the mentioned struct has been declared yet by keeping a list of all the currently declared structs.

So far this was the only functionality developed for Structs, what follows here is my reasoning on how I would have gone about with implementing the rest of the Visitor class to be able to handle struct types, as well as my general idea of how the Interpreter Visitor would work.

When referencing a variable we would check wethere there is contained a dot, then check to see if the type of the LHS of the dot, this would be the instance name, and from the RHS of the dot we would get the variable or function name, with these in mind we can extract the expected type by

looking up the struct symbol table, and seeing what type is mapped to the variable/function name there.
For Struct functions, these would be implemented normally, and gone thorugh as normal, accessed in the similair fashion as accessing struct variables. When it comes to returning an instance of a class, we check within the current scope of created structs wether the variable we are returining is an instance of the specified struct, and if we require a struct as a parameter we check the actual parameter in the function call, if the variable being passed is listed as a struct instance for the struct specified by the formal parameters.

Interpreter

For the interperter, a reserved array of AST nodes holding the specific struct functions would be created to hold the struct specific functions. Appropriate value tables are created within the scope of the struct as well as to keep track of the varaibles initial value when declaring a struct instance, and so that when the struct isntance is declared it can already make use of these values. A struct instance would then have its own map for variables, in such a way that accessing the instance map, and supplying the variable name would return the struct variable value for that instance.
Struct functions would be accessed normally but instead, we access the scope functions, and append the current instance's variables to the newly created scope so that the scope makes use of the current instance's variables.
Returining and passing a struct to a function would now needed to assign a pre determined memory location and when a function parameter detects that it is supposed to receive a struct, it copies the values from this location for all of its attributes. This could be implemented in conjunction with the Formal Parameters Node, so such memory allocations could pre determined before hand, when a Struct type parametere is expected.

Testing

Testing was perfomred up until some semantic analysis, the parts of the semantic analysis cnerned were the creation of structs and the struct symbol table. Due to time constraints, much of this task were out as fleshed out as I could have wished for, as is evident by the lack of detail this report is presenting.

Task 4 – Function Overloading

Function Overloading will allow us to create multiple functions with the same name, as long as they have a different order of parameter types. This allows different function to be called by the same name, and execution flow is directed depengin on the parameters supplied.
For this to occur only changes within the Semantic Visitor and Interpreter Visitor are to be implemented, since no new syntax or token types are being added.

Semantic Analysis

Previously semantic analysis worked on the asusmption that all functions will be uniquely names, now we must account for multiple functions with the same name and extend the map of function name to function return, to one that maps a function name to an arraylist of all function returns with the same name, in the order they were declared. In a similair fashion a map that maps a function name to an array list of array lists was created. The first array list represent all the different functions with the same name, and the second interior arraylists would be the list of parameters each individual function accepts and in what order.
When checking if a function exists we no longer check by name only, but rather by the order of parameters that are presented in the function declaration and in the map of nested array lists. If no

function with this order of parameter types has been enountered within the list of this function name, then we can append the list of parameters and the return type.
When evaluating a function call, we must identify which function to access by the order of supplied parameters. Type checking is performed on each parameter, and the list of types is created, this list is then compared with the lists found in the map, if there is a match, then the overloaded function exists, otherwise no function with that order of parameters has been declared yet.

Interpreter

When evaluating the Function Declaration node, the AST node representing the point of entry for the function call is added to the array list which is mapped to the function name in another map, in the same order of declaration aswell so that the order allows us to extract the information pertaining to the same function instance. All previous operaions of storing the list of parameters along with the other list of parameters for the same overloaded function is still carried out here, as the condition to access the correct function will still be the same.
This time when we find a match of parameter lists between the function call and one of the function declerations, we save the index at which the function declaration was created, and access the AST node that is found in that index among all the other AST nodes that belong to the function with the same name.
The function is then executed as a normal function and process continues as usual.

Testing

You may refer to the auto section for another relevant test on function overloading.
The following example showcases another usage of function overlaoding, correctly entering the specified function depending on the parameters identified within the function call.

```
int function(num1:float, nice:string){

print "You have entered the original function";

return 5;

}

float function(num1:float, word:string, nice:string){

print "entered the overloaded function :)";

return num1+num1;

}

float function2(num2:float){

return num2;

}


print function( 1.0, "hello", "223.3");

print function( function(1.0, "nice", "nice"), "hello", "223.3");

let x:int = function(1.0, "hello");
```

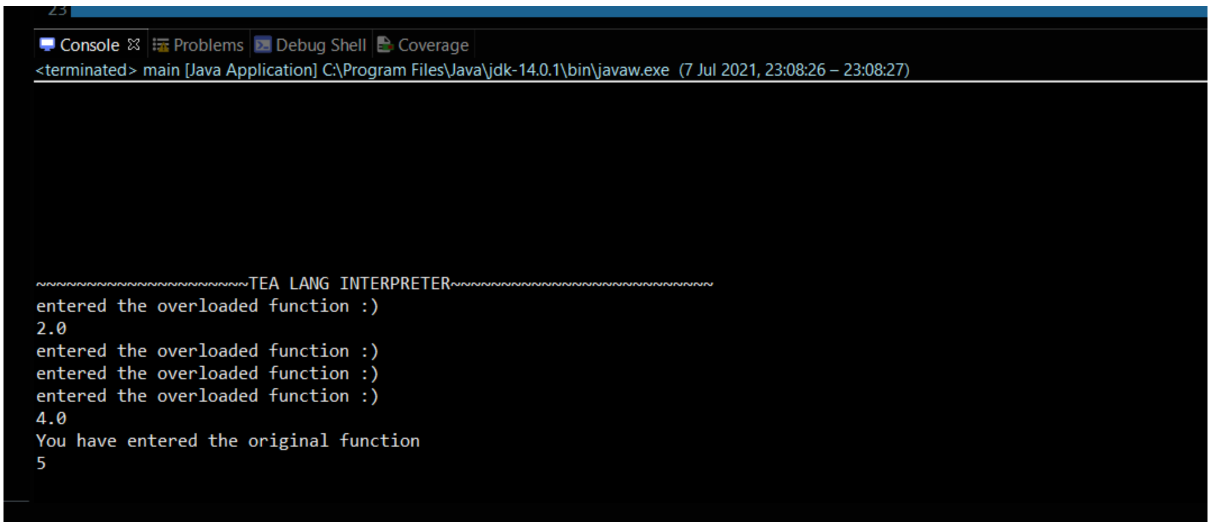
```
print x;
```

Generated XML

```
<ProgramNode>
	<FunctionDecl>
		<Type>int</Type>
		<Variable_Identifier>function</Variable_Identifier>
		<FormalParams>
			<Variable_Identifier>num1</Variable_Identifier>
			<Type>float</Type>
			<Variable_Identifier>nice</Variable_Identifier>
			<Type>string</Type>
		</FormalParams>
		<Block>
			<PrintStatements>
				<String_Value>You have entered the original 
function</String_Value>
			</PrintStatements>
			<ReturnStatement>
				<Integer_Value>5</Integer_Value>
			</ReturnStatement>
		</Block>
	</FunctionDecl>
	<FunctionDecl>
		<Type>float</Type>
		<Variable_Identifier>function</Variable_Identifier>
		<FormalParams>
			<Variable_Identifier>num1</Variable_Identifier>
			<Type>float</Type>
			<Variable_Identifier>word</Variable_Identifier>
			<Type>string</Type>
			<Variable_Identifier>nice</Variable_Identifier>
			<Type>string</Type>
		</FormalParams>
		<Block>
			<PrintStatements>
				<String_Value>entered the overloaded function 
:)</String_Value>
			</PrintStatements>
			<ReturnStatement>
				<Operator = +>
					<Variable_Identifier>num1</Variable_Identifier>
					<Variable_Identifier>num1</Variable_Identifier>
				</Operator>
			</ReturnStatement>
		</Block>
	</FunctionDecl>
	<FunctionDecl>
		<Type>float</Type>
		<Variable_Identifier>function2</Variable_Identifier>
		<FormalParams>
			<Variable_Identifier>num2</Variable_Identifier>
			<Type>float</Type>
		</FormalParams>
```

```
            <Block>
                  <ReturnStatement>
                        <Variable_Identifier>num2</Variable_Identifier>
                  </ReturnStatement>
            </Block>
      </FunctionDecl>
      <PrintStatements>
            <FunctionCall = function>
                  <Float_Value>1.0</Float_Value>
                  <String_Value>hello</String_Value>
                  <String_Value>223.3</String_Value>
            </FunctionCall>
      </PrintStatements>
      <PrintStatements>
            <FunctionCall = function>
                  <FunctionCall = function>
                        <Float_Value>1.0</Float_Value>
                        <String_Value>nice</String_Value>
                        <String_Value>nice</String_Value>
                  </FunctionCall>
                  <String_Value>hello</String_Value>
                  <String_Value>223.3</String_Value>
            </FunctionCall>
      </PrintStatements>
      <VariableDecl>
            <Variable_Identifier>x</Variable_Identifier>
            <Type>int</Type>
            <FunctionCall = function>
                  <Float_Value>1.0</Float_Value>
                  <String_Value>hello</String_Value>
            </FunctionCall>
      </VariableDecl>
      <PrintStatements>
            <Variable_Identifier>x</Variable_Identifier>
      </PrintStatements>
</ProgramNode>
```

Result



```
<terminated> main [Java Application] C:\Program Files\Java\jdk-14.0.1\bin\javaw.exe  (7 Jul 2021, 23:08:26 – 23:08:27)

~~~~~~~~~~~~~~~~~~~~~TEA LANG INTERPRETER~~~~~~~~~~~~~~~~~~~~~~~~~~
entered the overloaded function :)
2.0
entered the overloaded function :)
entered the overloaded function :)
entered the overloaded function :)
4.0
You have entered the original function
5
```

Final Remarks

Many of the functionality implemented here seems to be compliant with most of the existing functionality as well as the new ones, auto was able to detect returns from overloaded functions, and arrays were able to use the auto functionality as well as well as make use of funtioncs and parameters.
Unfortunately structs were not able to be implemented however and definite improvements can be made in all areas for optimisation, I am sure that with the next realease of TeaLang, TeaLang 3 Tokyo Drift , all these optimisations will be present and in place.