



**L-Università ta' Malta**  
Faculty of Information &  
Communication Technology

# **Compiler Theory & Practice**

## **Part 2**

Karl Attard 203501(L)  
B.Sc. (Hons) Artificial Intelligence

---

Study-unit: **Compiler Theory & Practice**  
Unit Code: **CPS2000**  
Lecturer: **Dr Sandro Spina**

## **FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY**

### **Declaration**

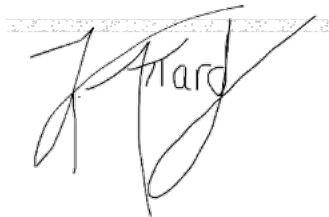
Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines” (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

We, the undersigned, declare that the assignment submitted is our work, except where acknowledged and referenced.

We understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected and will be given zero marks.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).



**Karl Attard**

\_\_\_\_\_  
Student Name

\_\_\_\_\_  
Signature

**CPS2000**

**Compiler Theory & Practice – Part 2**

\_\_\_\_\_  
Course Code

\_\_\_\_\_  
Title of work submitted

**07/07/2021**

\_\_\_\_\_  
Date

## Table of Contents

<b>INTRODUCTION.....</b>	<b>4</b>
<b>TASK 1: CHAR TYPE AND ARRAYS .....</b>	<b>5</b>
CHAR TYPE .....	5
<i>Testing</i> .....	6
ARRAYS.....	7
<i>Testing</i> .....	8
<b>TASK 2: AUTO TYPE .....</b>	<b>10</b>
TESTING .....	10
<b>TASK 3: STRUCTS .....</b>	<b>12</b>
<b>TASK 4: FUNCTION OVERLOADING .....</b>	<b>13</b>
TESTING .....	13
<b>CONCLUSION.....</b>	<b>15</b>

## Introduction

For this part of the project, it was required to develop a new Tea2Lang compiler, which is basically addition features and functionalities to the previous TeaLang compiler. In this documentation, I will be discussing how these tasks were implemented and tested.

## Task 1: Char Type and Arrays

### Char Type

To begin with, char type is essentially similar to String, but only one printable character is allowed rather than multiple. Prior to implementing this primitive data type, it was imperative to update the EBNF grammar rules to cater for such a type.

The following were the additions and changes made to the EBNF:

- `<Literal> ::= <BooleanLiteral>, <IntegerLiteral>, <FloatLiteral>, <StringLiteral>, <CharLiteral>`
- `<CharLiteral> ::= ‘’ <Printable> ‘’`
- `<Type> ::= ‘float’ | ‘int’ | ‘bool’ | ‘string’ | ‘char’`

Therefore, with respect to lexical analysis, a new keyword called ‘char’ had to be added to the list of keywords, together with 2 new tokens, one to handle the ‘char’ keyword and the other to handle the single quotes. Moreover, the DFSA was slightly modified too, just by adding the single quote as a transition between the starting state and state 12.

Then, the parser was slightly modified. Firstly, a new node which extends the ExpressionNode called ‘CharLiteralNode’ was created. The property of this class is just a variable of type ‘char’ which will hold the value of the expression. Now, the ‘Parser.java’ class was modified to handle the syntax of such a variable type.

The following is an example of a variable declaration of type char:

- `let c : char = ‘y’;`

Therefore, the ‘parseType()’ function was modified in such a way to deal with this new keyword. Moreover, the ‘parseLiteral()’ method a new case statement to handle the char type parsing. This was done by entering a particular case statement when single quotes are encountered within this function. This case statement works the same way as the string literal, but this time outputting an error if there is 1 or more printable characters and if the printable character is not enclosed within two single quotes. This function then returns a new AST CharLiteralNode with the only printable character.

Moving on to XML, a new visit method had to be added to cater for such a new type. Moreover, this new method had to be overridden in XMLVisitor, SemanticAnalysisVisitor class as well as InterpreterExecVisitor class. The implementation of each is the same as the other literals, but this time it obviously deals with the ‘char’ type,

## Testing

To test the char type, sound programs and bad programs were written.

The first use case is a sound program:

```
let ch : char = 'c';  
print ch;
```

In the above snippet, char variable declaration is being declared and then outputted. The following is the outputted after running this program:

```
----- XML Output -----  
<Program>  
  <VarDeclaration>  
    <Var Type="char">  
      <Identifier>ch</Identifier>  
    </Var>  
    <CharLiteral>c</CharLiteral>  
  </VarDeclaration>  
  <Print>  
    <Identifier>ch</Identifier>  
  </Print>  
</Program>  
  
----- Interpreter Output -----  
c  
  
TeaLang program successfully executed!
```

As can be seen, both XML generation and output are correct.

The following snippet breaks the rules of the program:

```
let ch : char = 'chhh';  
print ch;
```

The following is the output for this snippet:

```
Syntax Error!  
Parser expected one printable character but got 'chhh '  
HINT: Every char literal must only have one printable character. Ex: x:char = 'c'
```

Therefore, the above outputs show that this char type has been implemented successfully. Hence, the above tests passed,

## Arrays

Arrays were only implemented up to XML generation. In this section, I will be explaining the implementation for arrays starting from lexical analysis up to XML generation.

Prior to implementing arrays, it was imperative to define the new EBNF rules. The following rules were added and/or modified in the EBNF:

- `<ArrayDecl> ::= 'let' <identifier> '[' <Digit> ']' ':' <Type> [ <ArrayAssignment> ]`
  - Ex: `let arr1[2] : int = {1,2}`
- `<ArrayAssignment> ::= '{' <Literal> { '.' <Literal> } '}'`
- `<Factor> ::= <Literal> | <Identifier> | <FunctionCall> | <SubExpression> | <Unary> | <Array>`
- `<Array> ::= <identifier> '[' <digit> ']'`

Therefore, to implement arrays, some changes had to be made to the lexer. To begin with, new tokens had to be created for the open/close square brackets and added to the transition of the DFSA between starting state and state 12. Hence, the open/close square brackets were added in the switch statement of the 'getColumnType()' and also in the function 'setTokenType()' to set the respective tokens for these symbols.

When it comes to parsing, it was first required to create new AST nodes that will cater for arrays. The following are the classes created:

- **ArrayDeclarationNode** – This class includes 4 properties, IdentifierNode, IntegerLiteralNode, TypeNode, and ArrayList of assignments. Generic typing was implemented for the array list since assignments in arrays can be of any type (float, int, string, etc). Two constructors are implemented to cater for different types of array declaration (with or without array assignments values). Getters and Setters instance functions were also implemented together with an 'accept()' method. This node is created when an array is declared.
- **ArrayStatNode** – This class has 2 properties, the IdentifierNode and IntegerLiteralNode. This class is created for when arrays are called in expressions.

After having created such classes, it is now time to implement the parse methods within the Parser class. The respective function were implemented, called 'parseArrayDeclaration()' and 'parseArray()' respectively. The former method is parsing the `<ArrayDecl>` EBNF rule by ensuring that the syntax is correct. This method returns a new ArrayDeclarationNode. The latter method parses the `<Array>` EBNF rule by ensuring that all syntax is not broken and subsequently, returns a new ArrayStatNode. As previously, the parser returns the AST.

Then, the XML is using the returned AST to generate XML. XML was generated by implementing a visitor method for each of the above new classes in the Visitor class. Hence, in the XMLVisitor, these methods must be overridden and implemented. The XML is printed to the console through a series of indents, new lines and prints to generate the appropriate XML.

For instance, the below snippet of code shows the implementation to generate the XML for array declarations.

```
@Override
public void visit(ArrayDeclarationNode arrayDeclarationNode)
{
    System.out.println("<ArrayDeclarationNode>");
    incrementIndent();
    System.out.println("<Array Type=\"" + arrayDeclarationNode.getType().getType() + " Size=\"" + arrayDeclarationNode.getInteger().getValue() + "\">");
    incrementIndent();
    arrayDeclarationNode.getIdentifierNode().accept( v: this);
    if(arrayDeclarationNode.getAssignments() != null)
    {
        incrementIndent();
        System.out.println("<ArrayValues>");
        for(int i = 0; i < arrayDeclarationNode.getAssignments().size(); i++)
        {
            //System.out.println(arrayDeclarationNode.getAssignments().get(0).getValue());
            System.out.println(arrayDeclarationNode.getAssignments().get(i));
        }
        System.out.println("</ArrayValues>");
    }
    decrementIndent();
    System.out.println("</Array>");
    printIndents();
    //arrayDeclarationNode.get().accept(this);
    decrementIndent();
    System.out.println("</ArrayDeclarationNode>");
}
```

In a nutshell, the above code is generating the XML for array declaration. This XML generation outputs the type of the array declaration, its size and its values if assignment also took place.

## Testing

The following is a sound code snippet highlighting array declaration with and without assignments:

```
let arr1[2] : int = {1,6};
let arr2[3] : float;
```

The following is the output of the XML (after lexical analysis and parsing):



```

<Program>
  <ArrayDeclarationNode>
    <Array Type="int Size="2">
      <Identifier>arr1</Identifier>
      <ArrayValues>
com.compiler.Parser_Task2.AST.IntegerLiteralNode@7106e68e
com.compiler.Parser_Task2.AST.IntegerLiteralNode@7eda2dbb
</ArrayValues>
    </Array>
      </ArrayDeclarationNode>
    <ArrayDeclarationNode>
      <Array Type="float Size="3">
        <Identifier>arr2</Identifier>
      </Array>
        </ArrayDeclarationNode>
  </Program>

```

As can be seen, the XML is generated correctly. The only issue would be that the class is being printed rather than the classe's attributes in the assignment array values part.

Another use case is a code snippet which breaks the syntax rules:

```
let arr1[2} : int = {1,6};
```

As can be seen above, a curly bracket replaced a closing bracket, leading to syntax errors. The following syntax error was outputted:

```

Syntax Error!
Parser expected ']' but got '}'
HINT: Array declarations must be in the form: let x[2]:int;

```

Therefore, in both the above use cases, the tests were successful since the expected output was also the actual output.

## Task 2: Auto Type

The auto type is a type which determines the type of variable automatically based on the value it has been declared. This type can only be used in variable declarations and must not be used in formal parameters.

With regards to the lexer, a new keyword called 'auto' was added to the list of keywords. Then, a new token was created to handle this keyword during lexical analysis.

In the parsing stage, in the 'getType()' function another case statement was added in order to return the 'auto' type. Moreover, with regards to syntax, within 'parserFormalParams()' it was important to check the type of the node. If the type is 'auto', then a syntax error is outputted since these are not allowed.

When it comes to XML generation, nothing needs to be done since the type of node will be outputted when the XMLVisitor class is passed to TypeNode.

In semantic analysis, the main reasoning behind such implementation is to assign the type of the variables based on the expression they are assigned to. This type deduction is done in 2 instances, in variable declaration and function returns. In the former, deduction is carried out once the expression has been analysed, and based on this expression, the correct type is assigned. In the latter, the return type is set according to the type of the expression in the return statement.

Finally, when it comes to interpreter output, nothing needs to be done. Hence, auto type deduction only concerns lexical analysis, parser and semantic analysis.

### Testing

The following is the first use case showing a sound program:

```
let var : auto = 7;  
print var;
```

The following is the output for this program:

```
----- XML Output -----  
<Program>  
  <VarDeclaration>  
    <Var Type="auto">  
      <Identifier>var</Identifier>  
    </Var>  
    <IntegerLiteral>7</IntegerLiteral>  
  </VarDeclaration>  
  <Print>  
    <Identifier>var</Identifier>  
  </Print>  
</Program>  
  
----- Interpreter Output -----  
7  
  
TeaLang program successfully executed!
```

It can be seen that the output is correct for the above program.

Now a program breaking the rules will be tested:

```
int add(x:auto y:int){  
  return x+y;  
}
```

In this snippet, an auto type is within the parameter which breaks the rules of the program. The output for this program is below:

```
Syntax Error!  
The 'auto' type cannot be used within function's parameters
```

The above tests performed passed since the expected output is the actual output.

### Task 3: Structs

Unfortunately, this task was not implemented in this project.

## Task 4: Function Overloading

As mentioned in the other documentation of part 1, function overloading was implemented there too. To implement such a task, only the semantic analysis and interpreter had to be updated.

Function overloading involves more than one method having the same name but different parameters. This was done by checking if a function with a particular name and with the defined parameter types (number of parameters is crucial too) already exists in the current scope or not. If there already exists a function with the same name and same parameter types, then a semantic analysis error is generated to the user. It is important to note that the Scope class includes a hash map where its key is a tuple (identifier and list of parameter types) while its value is the return type. Hence, this hash map is storing the identifier, the list of parameter types and the return type of each function in the respective scope. Hence, all what is required is to search in this hash map before adding it to it.

On the other hand, in the interpreter execution phase, it involves adding the function together with its identifier, list of parameter types and return type into the hash map of the current scope when declaring functions. Then, with regards to function calls, it just retrieves the function called by passing the identifier and parameter types (since there can exist more than one function having the same name, hence, we also pass parameter types to get the needed function).

### Testing

A sound program will be tested in this section to test for function overloading.

The following is the snippet of code:

```
int add(x:int, y:int){  
    return x+y;  
}  
  
int add(x:int, y:int, z:int){  
    return x+y+z;  
}  
  
let x : auto = 2;  
print x;
```

The expected output is that the compiler does not output any errors and prints the value of x.

The following is the output:

```
----- XML Output -----
<Program>
  <FunctionDeclaration>
    <Var Type="int">
      <Identifier>add</Identifier>
    <FormalParameters>
      <FormalParameter Type="int">
        <Identifier>x</Identifier>
      </FormalParameter>
      <FormalParameter Type="int">
        <Identifier>y</Identifier>
      </FormalParameter>
    </FormalParameters>
    <Block>
      <Return>
        <ExpressionV2 Operator="+">
          <Identifier>x</Identifier>
          <Identifier>y</Identifier>
        </ExpressionV2>
      </Return>
    </Block>
  </FunctionDeclaration>
  <FunctionDeclaration>
    <Var Type="int">
      <Identifier>add</Identifier>
    <FormalParameters>
      <FormalParameter Type="int">
        <Identifier>x</Identifier>
      </FormalParameter>
      <FormalParameter Type="int">
        <Identifier>y</Identifier>
      </FormalParameter>
      <FormalParameter Type="int">
        <Identifier>z</Identifier>
      </FormalParameter>
    </FormalParameters>
    <Block>
      <Return>
        <ExpressionV2 Operator="+">
          <Identifier>x</Identifier>
          <ExpressionV2 Operator="+">
            <Identifier>y</Identifier>
            <Identifier>z</Identifier>
          </ExpressionV2>
        </ExpressionV2>
      </Return>
    </Block>
  </FunctionDeclaration>
  <VarDeclaration>
    <Var Type="auto">
      <Identifier>x</Identifier>
    </Var>
    <IntegerLiteral>2</IntegerLiteral>
  </VarDeclaration>
  <Print>
    <Identifier>x</Identifier>
  </Print>
</Program>

----- Interpreter Output -----
2

TeaLang program successfully executed!
```

The above output is correct and therefore, the testing for function overloading has passed.

## Conclusion

In conclusion, the implemented tasks seem to be working fine with no known issues. This project is an extension to the previous project and was implemented using the same IDE. Hence, to build such a project, the same process as highlighted in Part 1 needs to be followed.