Compiler Project Report

Content	Page Number
Introduction	1
Scanner Definition	1
Parser Definition	2
Semantic Analysis Definition	3
Design	3, 10, 15
Output	6, 13, 16
Conclusion	17
References	17

Introduction

1) Scanner is the first phase to create a compiler for the Tiny language. A program in Tiny language has a simple structure as follow:

- A sequence of statements separated by semicolons.
- No procedures and no declarations
- All variables are integers which are declared by simply assigning values to them (like BASIC)
- Only two control statements which may include statement sequences:
    - If statement: has an optional else part and must be terminated by the word end.
    - Repeat statement
- Read and Write statements that perform input/output
- Multiline C comments (/* */) are used for block comments but comments cannot be nested for simplicity.
- Expressions are limited to Boolean and arithmetic expressions.
- Arithmetic expressions may involve: integer constants, variables, parentheses and any of the three integer operators -, + and * with the usual mathematical properties (precedence and associativity)
- Comparison operators are only: < and =
- The assignment operator :=
- Boolean expressions only appear as tests in control statements (no Boolean variables, assignment or I/O).

And extra works :

- Added Arithmetic operator /.

- Comparison operators >, != , >= and <= .
- Added full number parsing “e.g. 1.2E-3.5”.

Reserved Words
If
Then
Else
End
Repeat
Until
Read
Write

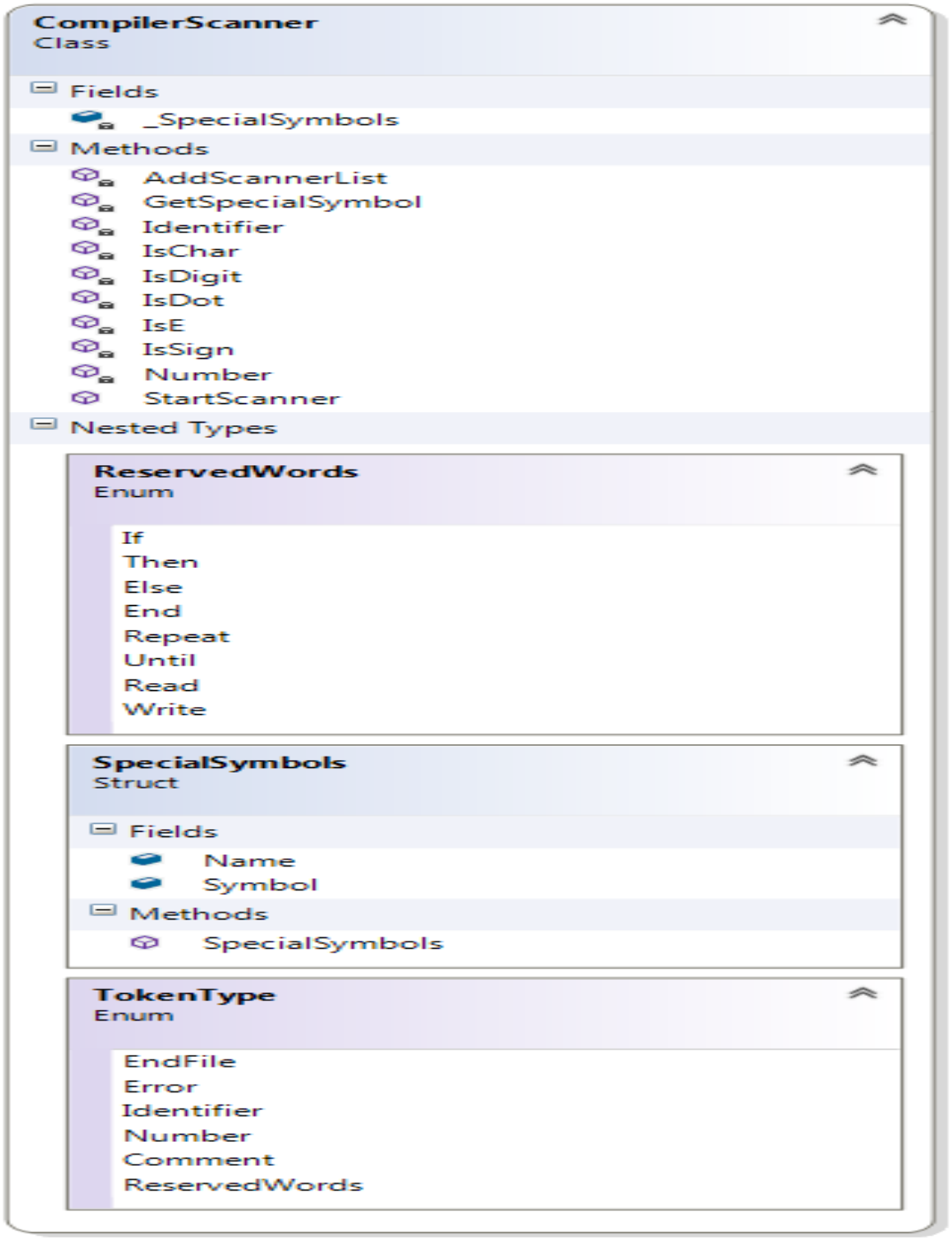
2) Parser is the second phase**. Syntax Analysis (Parsing)** is the process of **analyzing a text**, made of a sequence of tokens (for example, words), to determine its **grammatical structure** with respect to a given (more or less) formal grammar.
A program or function which performs syntax analysis is called a **syntax analyzer, or parser**.

EBNF:-
program → stmt-sequence
stmt-sequence → statement { **;** statement }
statement → if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt
if-stmt →**if** exp **then** stmt-sequence **[else** stmt- sequence] **end**
repeat-stmt → **repeat** stmt-sequence **until** exp
assign-stmt → **identifier :=** exp
read-stmt → **read identifier**
write-stmt → **write** exp
exp → simple-exp [comparison-op simple-exp]
comparison-op → **<** | =
simple-exp → term { addop term }
addop → **+** | **-**
term →factor { mulop factor }
mulop → *****
factor → **(**exp**)** | **number** | **identifier**

3) **Semantic Analysis** is the phase in which the compiler **adds semantic information** to the parse tree and builds the symbol table.

This phase performs semantic checks, such as:
Object binding (associating variable and function references with their definitions).
Definite assignment (requiring all local variables to be initialized before use).
Rejecting incorrect programs or issuing warnings.

Scanner Design:




```
/// <summary>
/// GetScannerData
```

```
/// </summary>
/// <param name="SD"></param>
/// <param name="NoError"></param>
/// <returns></returns>
public List<KeyValuePair<string, string>> GetScannerData(List<KeyValuePair<string,
string>> SD, ref bool NoError)
```

```
/// <summary>
/// Start Scanner
/// </summary>
/// <param name="FileData"></param>
/// <param name="ScannerData"></param>
public void StartScanner(string[] FileData, ref List<KeyValuePair<string, string>>
ScannerData)
```

```
/// <summary>
/// Check char or not
/// </summary>
/// <param name="ch"></param>
/// <returns>bool</returns>
private bool IsChar(char ch)
```

```
/// <summary>
/// Check Digit or not
/// </summary>
/// <param name="ch"></param>
/// <returns>bool</returns>
private bool IsDigit(char ch)
```

```
/// <summary>
/// Check Dot or not
/// </summary>
/// <param name="ch"></param>
/// <returns>bool</returns>
private bool IsDot(char ch)
```

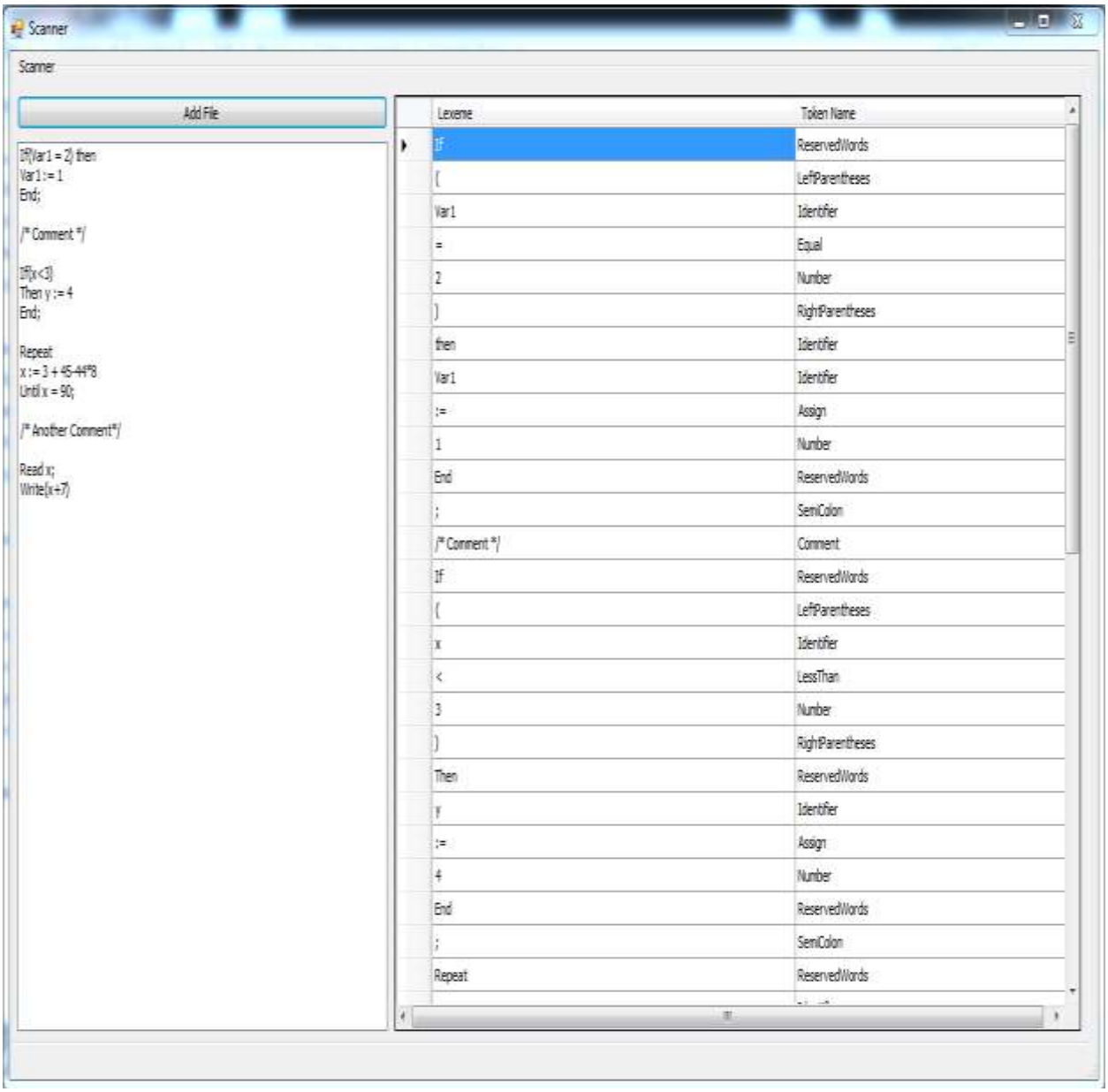
```
/// <summary>
/// Check E or not
/// </summary>
/// <param name="ch"></param>
/// <returns>bool</returns>
private bool IsE(char ch)
```

```
/// <summary>
/// Check + or - or not
/// </summary>
/// <param name="ch"></param>
/// <returns>bool</returns>
private bool IsSign(char ch)
```

```
/// <summary>
/// Get Identifier and stop index
/// </summary>
/// <param name="input"></param>
```

```
/// <param name="index"></param>
/// <returns>string</returns>
private string Identifier(string input, ref int index)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Get Number(with error or not) and stop index
/// </summary>
/// <param name="input"></param>
/// <param name="index"></param>
/// <param name="Error"></param>
/// <returns>string</returns>
private string Number(string input, ref int index, ref bool Error, string output =
"",bool E =false,bool Dot=false)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Get SpecialSymbol or empty
/// </summary>
/// <param name="current"></param>
/// <returns>string</returns>
private string GetSpecialSymbol(string current)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Add Token to ScannerData List
/// </summary>
/// <param name="ScannerData"></param>
/// <param name="output"></param>
/// <param name="result"></param>
private void AddScannerList(ref List<KeyValuePair<string, string>> ScannerData, string
output, string result)
```

Scanner Output:




```
/// <summary>
/// Create Parse Tree
/// </summary>
/// <param name="SD"></param>
/// <param name="ParserTreeView"></param>
public bool CreateParseTree(List<KeyValuePair<string, string>> SD, ref TreeNode
ParserTreeRoot)
```

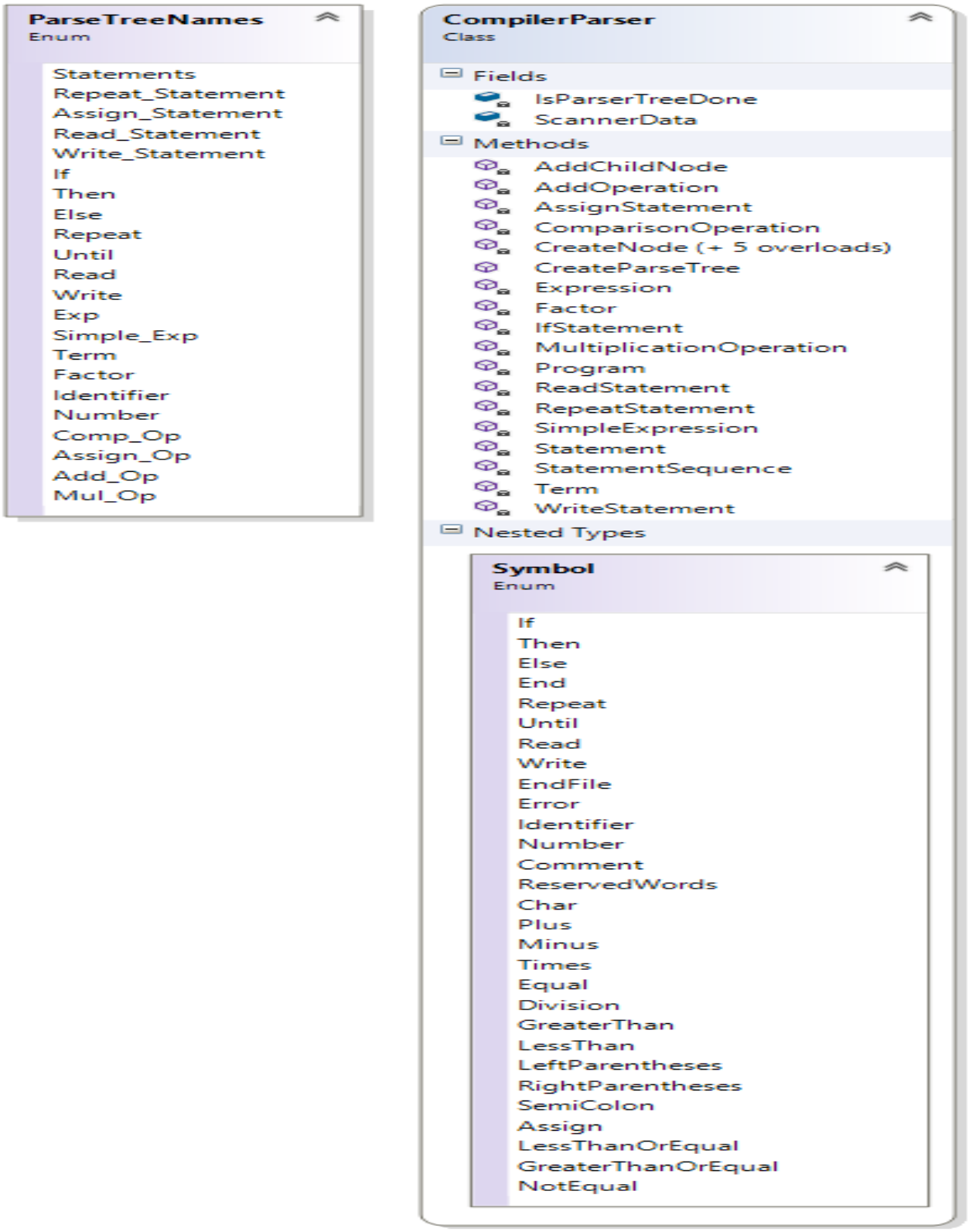
```
/// <summary>
/// Program ---> StatementSequence
/// </summary>
/// <returns></returns>
private TreeNode Program()
```

```
/// <summary>
/// StatementSequence ---> Statement { ; Statement }
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool StatementSequence(ref int CurrentIndex, ref TreeNode CurrentNode)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Statement ---> IfStatement | RepeatStatement | AssignStatement | ReadStatement |
WriteStatement
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool Statement(ref int CurrentIndex, ref TreeNode CurrentNode)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// IfStatement ---> If Expression Then StatementSequence [ Else StatementSequence ] End
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool IfStatement(ref int CurrentIndex, ref TreeNode CurrentNode)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// RepeatStatement ---> Repeat StatementSequence Until Expression
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool RepeatStatement(ref int CurrentIndex, ref TreeNode CurrentNode)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// AssignStatement ---> Identifier := Expression
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool AssignStatement(ref int CurrentIndex, ref TreeNode CurrentNode)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// ReadStatement ---> Read Identifier
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool ReadStatement(ref int CurrentIndex, ref TreeNode CurrentNode)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// WriteStatement ---> Write Expression
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
```

```
private bool WriteStatement(ref int CurrentIndex, ref TreeNode CurrentNode)
-------------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Expression ---> SimpleExpression [ ComparisonOperation SimpleExpression ]
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool Expression(ref int CurrentIndex, ref TreeNode CurrentNode)
-------------------------------------------------------------------------------------------------------------------------
/// <summary>
/// ComparisonOperation ---> < | =
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool ComparisonOperation(ref int CurrentIndex, ref TreeNode CurrentNode)
-------------------------------------------------------------------------------------------------------------------------
/// <summary>
/// SimpleExpression ---> Term  { AddOperation Term }
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool SimpleExpression(ref int CurrentIndex, ref TreeNode CurrentNode)       /// --
-------------------------------------------------------------------------------------------------------------------------
<summary>
/// AddOperation ---> + | -
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool AddOperation(ref int CurrentIndex, ref TreeNode CurrentNode)
-------------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Term ---> Factor  { MultiplicationOperation Factor }
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool Term(ref int CurrentIndex, ref TreeNode CurrentNode)
-------------------------------------------------------------------------------------------------------------------------
/// <summary>
/// MultiplicationOperation ---> *
/// </summary>
/// <param name="CurrentIndex"></param>
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool MultiplicationOperation(ref int CurrentIndex, ref TreeNode CurrentNode)
-------------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Factor ---> (Expression) | Number | Identifier
/// </summary>
/// <param name="CurrentIndex"></param>
```

```
/// <param name="CurrentNode"></param>
/// <returns></returns>
private bool Factor(ref int CurrentIndex, ref TreeNode CurrentNode)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// CreateNode
/// </summary>
/// <param name="nodeName"></param>
/// <param name="value"></param>
/// <returns></returns>
private TreeNode CreateNode(string nodeName,string value)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// AddChildNode
/// </summary>
/// <param name="Parent"></param>
/// <param name="Child"></param>
/// <returns></returns>
private TreeNode AddChildNode(TreeNode Parent,TreeNode Child)
```

Parser Design:



```
/// <summary>
/// Create Semantic Tree
/// </summary>
/// <param name="ParserTreeRoot"></param>
public void CreateSemanticTree(ref TreeNode ParserTreeRoot)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Statements
/// </summary>
/// <param name="node"></param>
private void Statements(ref TreeNode node)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// If
/// </summary>
/// <param name="node"></param>
private void If(ref TreeNode node)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Repeat
/// </summary>
/// <param name="node"></param>
private void Repeat(ref TreeNode node)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Assign_Statement
/// </summary>
/// <param name="node"></param>
private void Assign_Statement(ref TreeNode node)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Read
/// </summary>
/// <param name="node"></param>
private void Read(ref TreeNode node)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Write
/// </summary>
/// <param name="node"></param>
private void Write(ref TreeNode node)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Exp
/// </summary>
/// <param name="node"></param>
private void Exp(ref TreeNode node)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
/// Simple_Exp
/// </summary>
/// <param name="node"></param>
private void Simple_Exp(ref TreeNode node)
-----------------------------------------------------------------------------------------------------------------------
/// <summary>
```

```
/// Term
/// </summary>
/// <param name="node"></param>
private void Term(ref TreeNode node)
```

```
/// <summary>
/// Factor
/// </summary>
/// <param name="node"></param>
private void Factor(ref TreeNode node)
```

```
/// <summary>
/// Compare
/// </summary>
/// <param name="firstTerm"></param>
/// <param name="secondTerm"></param>
/// <param name="Comp_Op"></param>
/// <returns></returns>
private bool Compare(string firstTerm, string secondTerm, string Comp_Op)
```

```
/// <summary>
/// nodeOfKey
/// </summary>
/// <param name="node"></param>
/// <param name="key"></param>
/// <returns></returns>
private TreeNode nodeOfKey(TreeNode node, ParseTreeNames key)
```

```
/// <summary>
/// indexOfKey
/// </summary>
/// <param name="node"></param>
/// <param name="key"></param>
/// <returns></returns>
private int indexOfKey(TreeNode node, ParseTreeNames key)
```

```
/// <summary>
/// editNode
/// </summary>
/// <param name="parent"></param>
/// <param name="child"></param>
private void editNode(ref TreeNode parent, TreeNode child)
```

```
/// <summary>
/// editNode
/// </summary>
/// <param name="parent"></param>
/// <param name="child"></param>
/// <param name="index"></param>
private void editNode(ref TreeNode parent, TreeNode child, int index)
```

```
/// <summary>
/// format
/// </summary>
/// <param name="str"></param>
/// <returns></returns>
```

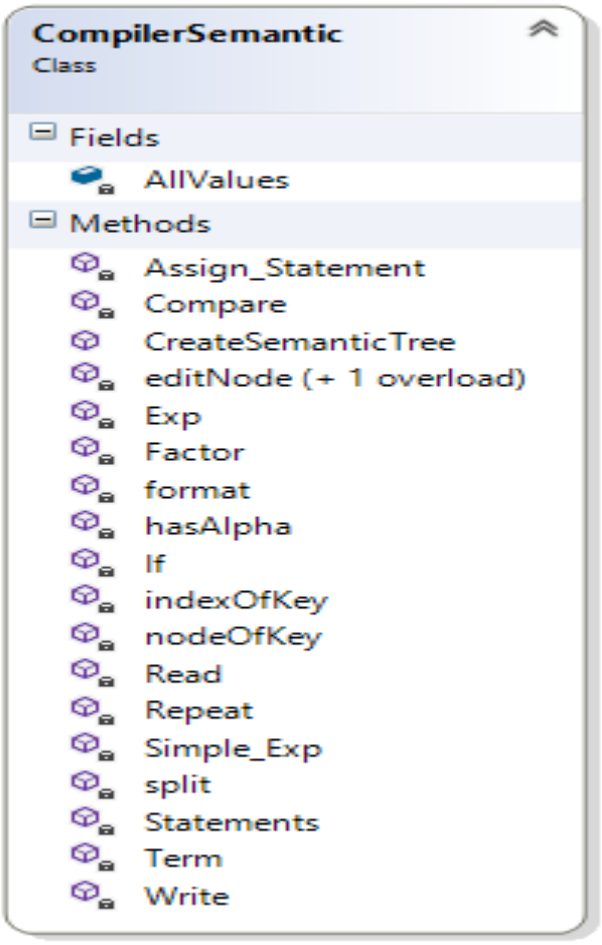
```
private string format(string str)
```

```
/// <summary>
/// split
/// </summary>
/// <param name="str"></param>
/// <returns></returns>
private string[] split(string str)
```

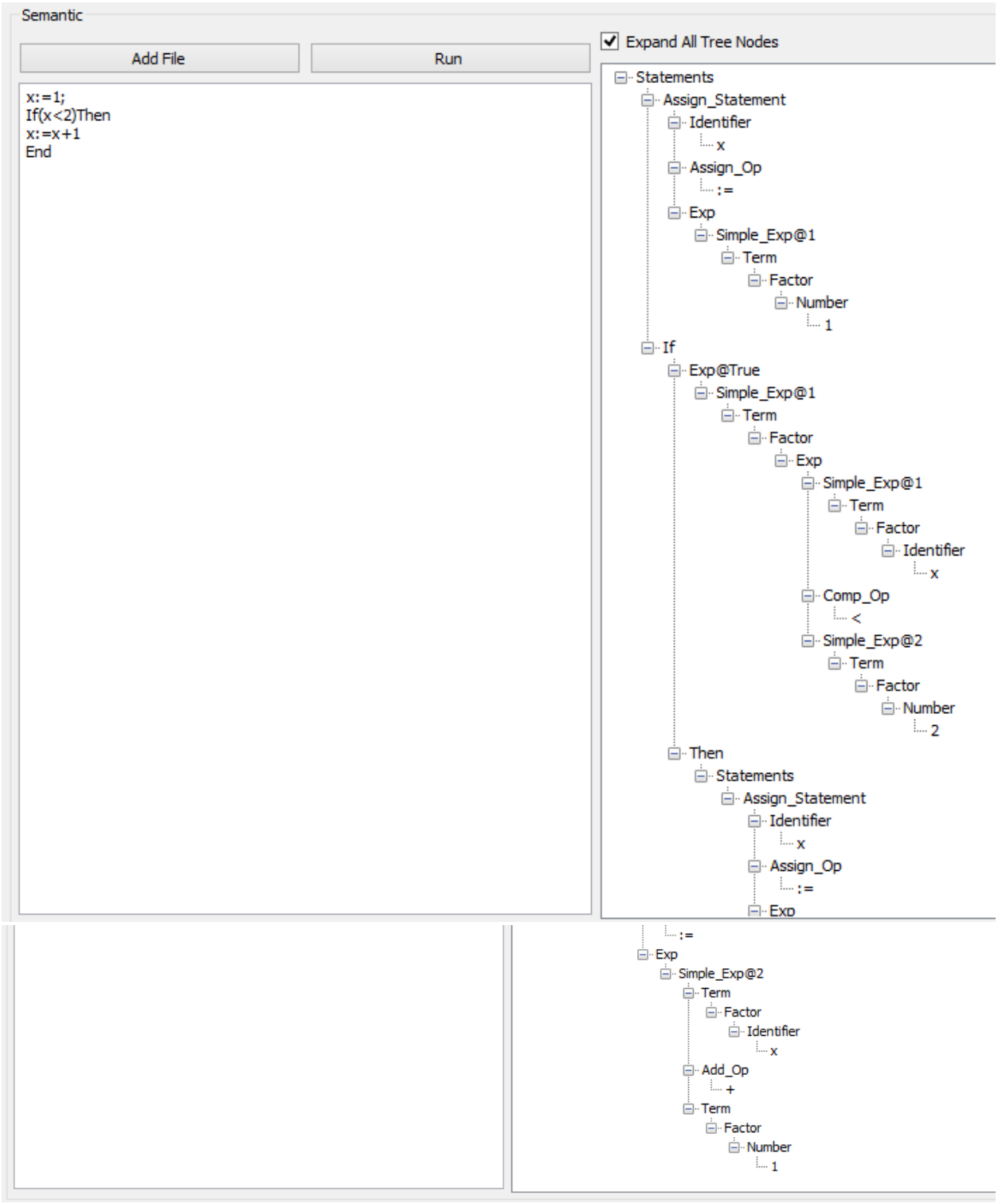
```
/// <summary>
/// hasAlpha
/// </summary>
/// <param name="str"></param>
/// <returns></returns>
private bool hasAlpha(string str)
```

Parser Output:

Semantic Design:



Semantic Output:



Conclusion:

All phase work with no bugs. Each phase dependent on previous one.

References:

Compiler’s Labs