CMP409 Compilers and Languages

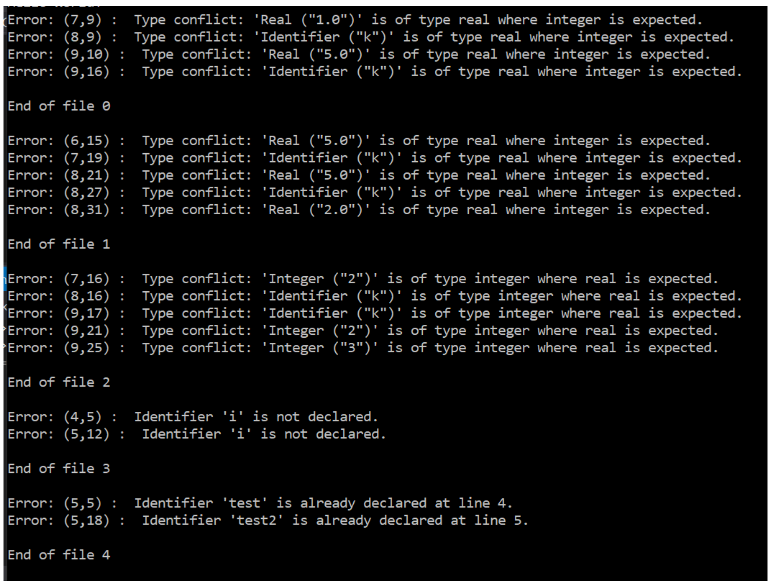
Dorota Marczak, 1604779

The evaluation of the implemented PAL compiler in terms of syntax and semantic analysis.

The PAL Compiler was evaluated with two sets of semantic and syntactic tests, read from representative folders in the form of an array of files. The tests were designed to verify the completeness of the implemented compiler in detecting bugs and/or potential vulnerabilities of the system. For the purposes of the semantic analysis, the *Semantics* class was facilitated with three helper methods:

- *CheckID()*, determining whether a given identifier token has been declared in current scope
- *DeclareID(),* appending a token to the symbol table (created based on the current scope) and reporting a semantic error when the token with the same name has already been declared
- *CheckTypesSame(),* determining whether given expression is type-consistent.

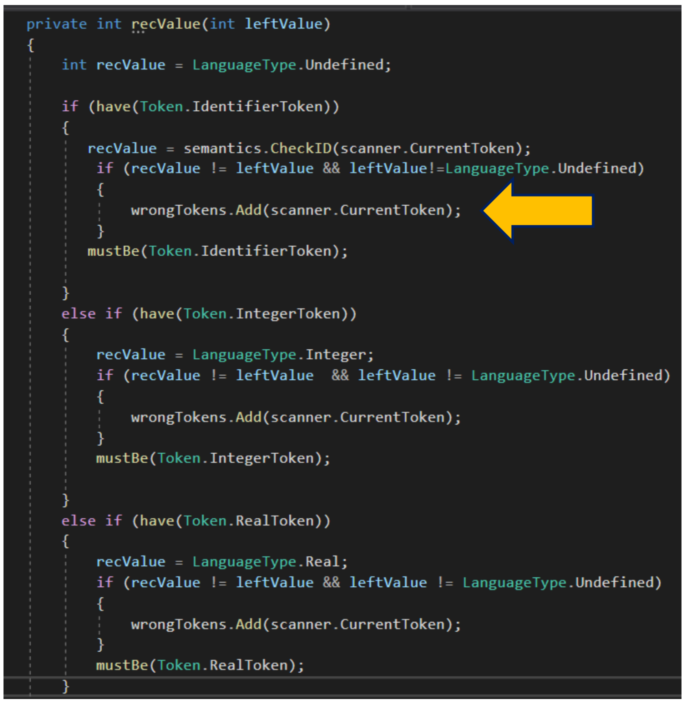
The results of the semantic analysis (Fig.1) shown that the compiler correctly recognises errors related to undeclared/already declared variables, as well as those caused by incorrect type of tokens in a given expression; including variable assignment, mathematical calculations and logical statements. The implementation of a Type checker required initialisation of a new IToken list, that would store tokens representing type different than expected in current expression. In example, a statement *i = (5.0 * k + 2)* (Line 9 of file 0, Fig.1), where *i* is of type Integer and *k* is a Real, results in two semantic errors related to type differences between *i* (Integer) and *k, 5.0 (Real).* The comparison between types was implemented through passing the expected type as a function parameter, which enabled to verify type consistence in *recValue()* – recogniser method determining the type of given token and, if the token represents a variable, verifying whether it has been previously declared in the scope. This way, the error list returned by the compiler was more accurate in terms of the error themselves and their location in text, returning the name of each of incorrect tokens instead of a general information about expression type-inconsistency.



```
Error: (7,9) :  Type conflict: 'Real ("1.0")' is of type real where integer is expected.
Error: (8,9) :  Type conflict: 'Identifier ("k")' is of type real where integer is expected.
Error: (9,10) :  Type conflict: 'Real ("5.0")' is of type real where integer is expected.
Error: (9,16) :  Type conflict: 'Identifier ("k")' is of type real where integer is expected.

End of file 0

Error: (6,15) :  Type conflict: 'Real ("5.0")' is of type real where integer is expected.
Error: (7,19) :  Type conflict: 'Identifier ("k")' is of type real where integer is expected.
Error: (8,21) :  Type conflict: 'Real ("5.0")' is of type real where integer is expected.
Error: (8,27) :  Type conflict: 'Identifier ("k")' is of type real where integer is expected.
Error: (8,31) :  Type conflict: 'Real ("2.0")' is of type real where integer is expected.

End of file 1

Error: (7,16) :  Type conflict: 'Integer ("2")' is of type integer where real is expected.
Error: (8,16) :  Type conflict: 'Identifier ("k")' is of type integer where real is expected.
Error: (9,17) :  Type conflict: 'Identifier ("k")' is of type integer where real is expected.
Error: (9,21) :  Type conflict: 'Integer ("2")' is of type integer where real is expected.
Error: (9,25) :  Type conflict: 'Integer ("3")' is of type integer where real is expected.

End of file 2

Error: (4,5) :  Identifier 'i' is not declared.
Error: (5,12) :  Identifier 'i' is not declared.

End of file 3

Error: (5,5) :  Identifier 'test' is already declared at line 4.
Error: (5,18) :  Identifier 'test2' is already declared at line 5.

End of file 4
```

Figure 1 - The result of semantic analysis

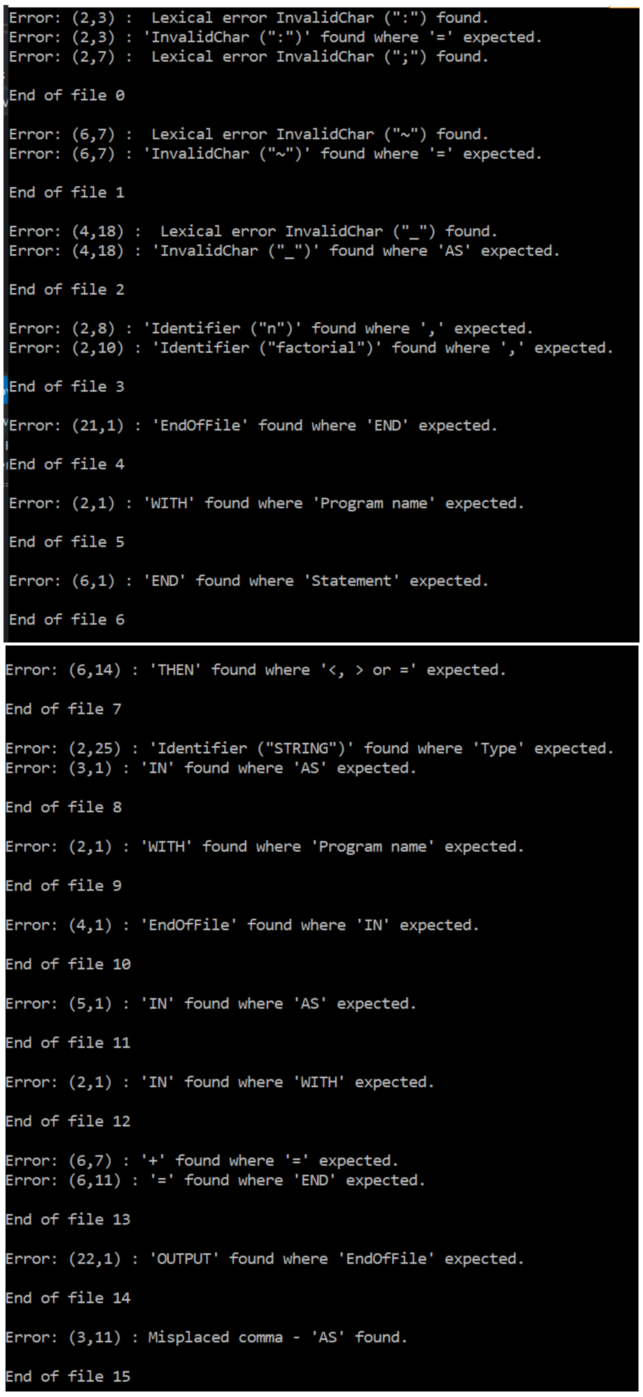
Any token of type different than expected (excluding *LanguageType.Undefined,* typical to first token of boolean or output statements) was appended by the aforementioned recogniser method to the *wrongToken* list (as shown in Fig.2). The list of errors was then displayed by corresponding function - *recTerm(), recAssignment()* or *recBoolean(),* covering current stage of compilation based on EBNF specification. Once the list of error token has been displayed, the list is cleared to avoid duplication of errors.



```
private int recValue(int leftValue)
{
    int recValue = LanguageType.Undefined;

    if (have(Token.IdentifierToken))
    {
       recValue = semantics.CheckID(scanner.CurrentToken);
        if (recValue != leftValue && leftValue!=LanguageType.Undefined)
        {
            wrongTokens.Add(scanner.CurrentToken);
        }
       mustBe(Token.IdentifierToken);

    }
    else if (have(Token.IntegerToken))
    {
        recValue = LanguageType.Integer;
        if (recValue != leftValue  && leftValue != LanguageType.Undefined)
        {
            wrongTokens.Add(scanner.CurrentToken);
        }
        mustBe(Token.IntegerToken);

    }
    else if (have(Token.RealToken))
    {
        recValue = LanguageType.Real;
        if (recValue != leftValue && leftValue != LanguageType.Undefined)
        {
            wrongTokens.Add(scanner.CurrentToken);
        }
        mustBe(Token.RealToken);
    }
```

Figure 2 - RecValue()

Syntactical analysis was performed on a set of 16 test programs; each missing different tokens/components required by EBNF specification. The errors reported by parser presented high efficiency of the compiler in detecting unexpected tokens, along with their location in text and expected terminal and/or non-terminal symbols (Fig.3)



```
Error: (2,3) :  Lexical error InvalidChar (":") found.
Error: (2,3) : 'InvalidChar (":")' found where '=' expected.
Error: (2,7) :  Lexical error InvalidChar (";") found.

End of file 0

Error: (6,7) :  Lexical error InvalidChar ("~") found.
Error: (6,7) : 'InvalidChar ("~")' found where '=' expected.

End of file 1

Error: (4,18) :  Lexical error InvalidChar ("_") found.
Error: (4,18) : 'InvalidChar ("_")' found where 'AS' expected.

End of file 2

Error: (2,8) : 'Identifier ("n")' found where ',' expected.
Error: (2,10) : 'Identifier ("factorial")' found where ',' expected.

End of file 3

Error: (21,1) : 'EndOfFile' found where 'END' expected.

End of file 4

Error: (2,1) : 'WITH' found where 'Program name' expected.

End of file 5

Error: (6,1) : 'END' found where 'Statement' expected.

End of file 6
```

```
Error: (6,14) : 'THEN' found where '<, > or =' expected.

End of file 7

Error: (2,25) : 'Identifier ("STRING")' found where 'Type' expected.
Error: (3,1) : 'IN' found where 'AS' expected.

End of file 8

Error: (2,1) : 'WITH' found where 'Program name' expected.

End of file 9

Error: (4,1) : 'EndOfFile' found where 'IN' expected.

End of file 10

Error: (5,1) : 'IN' found where 'AS' expected.

End of file 11

Error: (2,1) : 'IN' found where 'WITH' expected.

End of file 12

Error: (6,7) : '+' found where '=' expected.
Error: (6,11) : '=' found where 'END' expected.

End of file 13

Error: (22,1) : 'OUTPUT' found where 'EndOfFile' expected.

End of file 14

Error: (3,11) : Misplaced comma - 'AS' found.

End of file 15
```

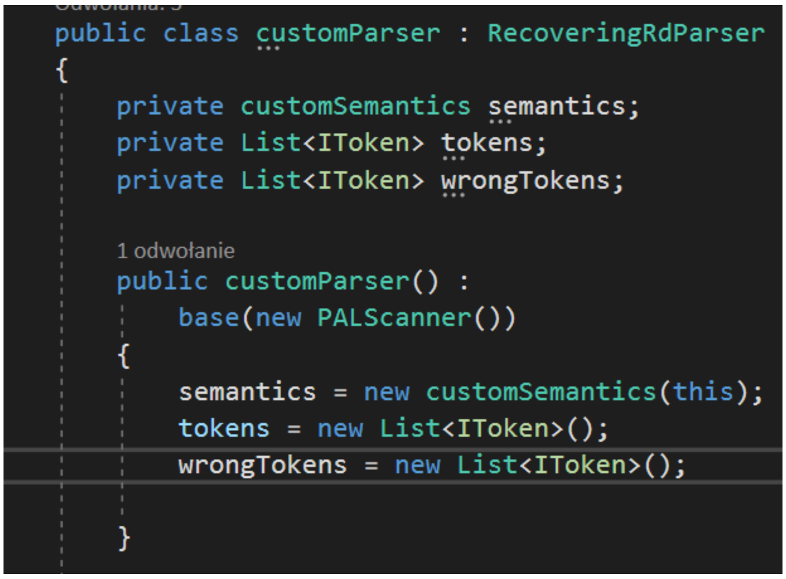
Figure 3 - The results of syntax and lexical analysis

The methodology behind VarDecls implementation

The recogniser method responsible for declaring variables has been based on the following fragment of the EBNF specification.

```
<VarDecls> ::= (<IdentList> AS <Type>)* ;
```

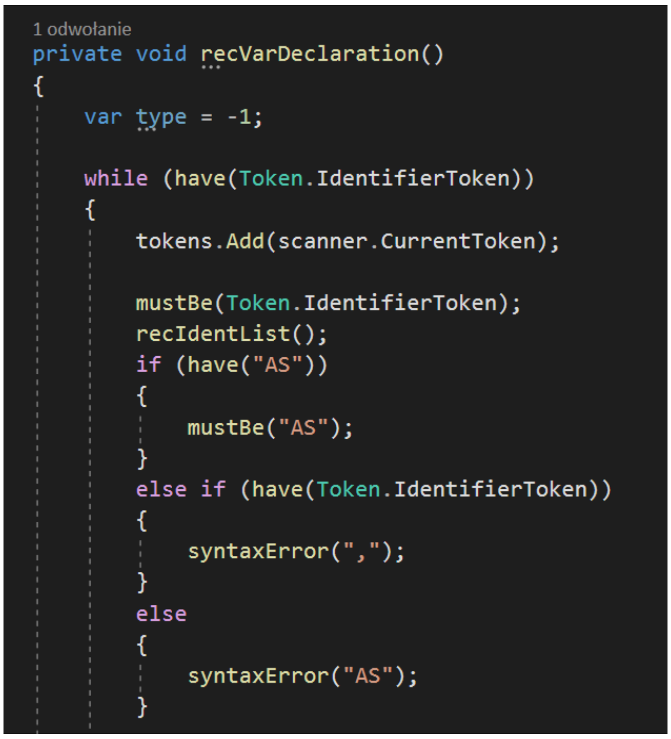
To account for the declarations consisting of multiple variables, a private IToken list *tokens* was declared and initialised in the constructor of the *customParser* class (Fig.4), inheriting its methods from *RecoveringRdParser.* The implementation of the list allowed to temporarily store identifiers being the components of a single declaration and associated metadata - such as their location in the text – until they were assigned a type.



```
public class customParser : RecoveringRdParser
{
    private customSemantics semantics;
    private List<IToken> tokens;
    private List<IToken> wrongTokens;

    1 odwołanie
    public customParser() :
        base(new PALScanner())
    {
        semantics = new customSemantics(this);
        tokens = new List<IToken>();
        wrongTokens = new List<IToken>();

    }
```

Figure 4 – CustomParser class

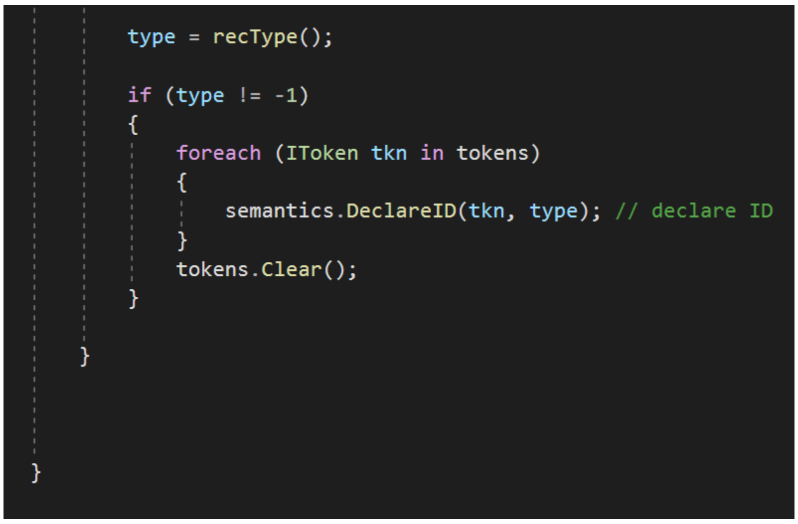
The *recVarDeclaration()* method (Fig.5,6) commences with *while* loop, which aims at recognising new variable declarations in the program code. Once the detected Identifier token is appended to the token list, the program calls *recIdentList()* function (Fig.7) that retrieves other variables included in a given declaration by recursion. Syntax analysis related with the aforementioned methods focuses on validating the grammar of given code based on provided EBNF specification, reporting an error when a declaration is incomplete (i.e. when a literal or a punctuation mark is missing).



```
1 odwołanie
private void recVarDeclaration()
{
    var type = -1;

    while (have(Token.IdentifierToken))
    {
        tokens.Add(scanner.CurrentToken);

        mustBe(Token.IdentifierToken);
        recIdentList();
        if (have("AS"))
        {
            mustBe("AS");
        }
        else if (have(Token.IdentifierToken))
        {
            syntaxError(",");
        }
        else
        {
            syntaxError("AS");
        }
```

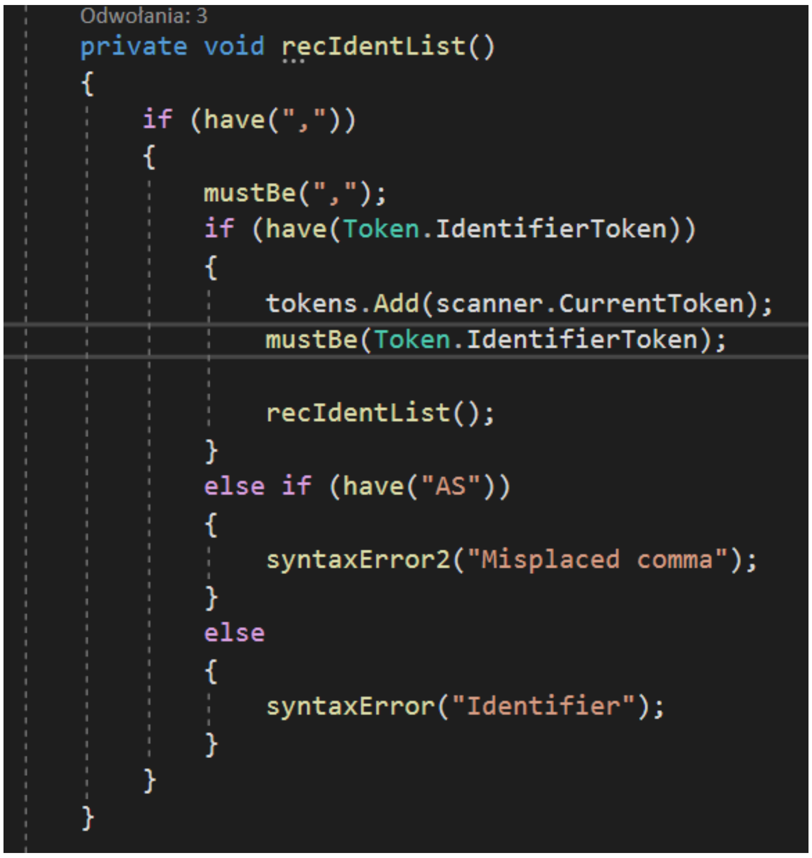
Figure 5 – Recogniser method responsible for variable declaration (part I)

When the program returns to *recVarDeclaration()* with a list of identifier tokens, *recType()* is called to determine the type of the variables being declared. Having a complete list of identifier tokens, the program begins their declaration process; any previously declared tokens will be recorded and displayed as semantic errors by helper method of the *Semantics* class object (Fig.8). Once all valid Identifier token has been added to the symbol table, the process is repeated for the rest of the declarations and finished when a statement is being encountered.



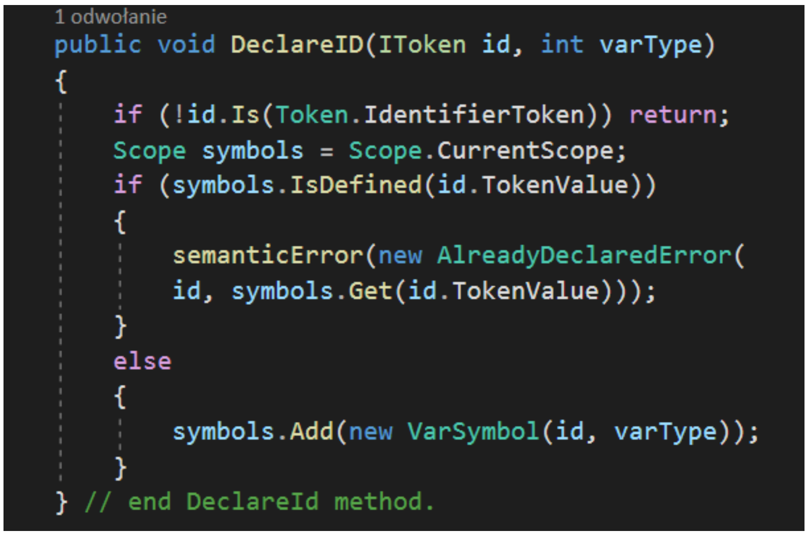
```
        type = recType();

        if (type != -1)
        {
            foreach (IToken tkn in tokens)
            {
                semantics.DeclareID(tkn, type); // declare ID
            }
            tokens.Clear();
        }

    }



}
```

Figure 6 - Recogniser method responsible for a variable declaration (part II)



```
Odwołania: 3
private void recIdentList()
{
    if (have(","))
    {
        mustBe(",");
        if (have(Token.IdentifierToken))
        {
            tokens.Add(scanner.CurrentToken);
            mustBe(Token.IdentifierToken);

            recIdentList();
        }
        else if (have("AS"))
        {
            syntaxError2("Misplaced comma");
        }
        else
        {
            syntaxError("Identifier");
        }
    }
}
```

Figure 7 - Recogniser method creating a list of Identifier tokens



```
1 odwołanie
public void DeclareID(IToken id, int varType)
{
    if (!id.Is(Token.IdentifierToken)) return;
    Scope symbols = Scope.CurrentScope;
    if (symbols.IsDefined(id.TokenValue))
    {
        semanticError(new AlreadyDeclaredError(
        id, symbols.Get(id.TokenValue)));
    }
    else
    {
        symbols.Add(new VarSymbol(id, varType));
    }
} // end DeclareId method.
```

Figure 8 - Helper method provided by Semantics class to verify identifiers and to add them to the symbol table