

# TKOM Dokumentacja Etap 1

---

Damian Kolaska

## TKOM Dokumentacja Etap 1

Specyfikacja podzbioru

**operatory**

**stałe**

**zmienne**

**domyślna klasa**

Składnia

Sposób uruchomienia

Struktura projektu

Pliki

Komunikacja

Wymagania funkcjonalne

Wstępne założenia implementacyjne

Obsługa błędów

Typy Tokenów

Celem projektu jest napisanie translatora podzbioru języka Python do C#. Projekt zamierzam wykonać w języku C# (.NET Core).

Powstały kod powinien dać się skompilować i uruchomić w środowisku .NET Core.

Aplikacja powinna wczytywać kod Pythona z pliku.

## Specyfikacja podzbioru

---

### operatory

#### arytmetyczne

+, -, \*, /

#### przypisania

\=

#### porównania

>, >=, <, <=, ==, !=

#### logiczne

not, and, or

### stałe

W przypadku stałej nie specyfikujemy typu

*Python*

```
var1 = 3
var2 = 3.5
var3 = "Hello world"
```

*C#*

```
const int var1 = 3;
const double var2 = 3.5;
const String var3 = "Hello world";
```

## zmienne

Przy zmiennych typ określamy przy pomocy konstruktorów

*Python*

```
var0 = int()
var1 = int(3)
var2 = float(3.5)
var3 = str("Hello world")
var4 = MyClass("my_class_name")
```

*C#*

```
int var0;
int var1 = 3;
double var2 = 3.5;
String var3 = "Hello world";
MyClass var4 = MyClass("my_class_name");
```

## domyślna klasa

Zmienne, metody, klasy itd. nienależące do żadnej klasy, w C# trafiają do domyślnej klasy *Program* oraz

mają publiczny dostęp.

*Python*

```
var = 3
def func(arg: float) -> int:
    loc_var = int(2)
    loc_var = loc_var + 2
    return loc_var * arg
# alternatywna opcja specyfikacji return type
# return int(arg1 * arg2)
```

*C#*

```
class Program
{
    public const int var = 3;
    public int func(double arg)
    {
        int loc_var = 2;
        loc_var = loc_var + 2;
        return loc_var * arg
    }
}
```

## Składnia

---

W pliku syntax.pdf znajdują się diagramy obrazujące składnię wygenerowane przy użyciu <https://bottlecaps.de/rr/ui>

```
newline ::= "\n"
tab ::= "\t"

identifier ::= [a-zA-Z_] [a-zA-Z0-9_]*
type ::= "int" | "float" | "string" | "bool"
digit ::= [0-9]

variable ::= identifier

constant ::= ([1-9] digit*) | "0"
logical_value ::= True | False
string ::= "'" ([^btnfr"''] | ("\" [btnfr"'']))* "'"
value ::= constant | logical_value | string

function_arg ::= value | function_call | variable
function_call ::= identifier "(" ((function_arg ",")* function_arg)? ")"

comparison_operator ::= "<=" | "<" | ">=" | ">"
equality_operator ::= "==" | "!="

logical_formula ::=
    (variable equality_operator (string | constant | logical_value |
function_call)) |
    (function_call equality_operator (string | constant | logical_value |
function_call)) |
    ((constant | variable | function_call) comparison_operator (constant |
variable | function_call)) |
    ((constant | logical_value) | (not? (variable | function_call)))
logical_expression ::=
    "(" (logical_expression | (logical_expression ("and" | "or")
logical_expression)) ")"

if_statement ::= "if" logical_expression ":" newline
while_loop ::= "while" logical_expression ":" newline
for_loop ::= "for" identifier "in" "range"
    "(" (constant | variable | function_call) "," (constant | variable |
function_call) ")" ":" newline

function_def ::=
    "def" identifier "(" ( ((identifier ":" type) ",")* (identifier ":" type) )?
    ")" ":" newline
assignment ::=
    variable "=" (value | variable | function_call | logical_expression) newline
variable_def ::=
    identifier "=" type "(" ((function_arg ",")* function_arg)? ")" newline

code_block ::=
    (
        (assignment | variable_def) code_block? |
        ((if_statement | for_loop | while_loop | function_def) (tab code_block)+
    )
```

## Sposób uruchomienia

```
pythonCSharpTrs -o output.cs input.py
```

## Struktura projektu

### Pliki

- *Lexer.cs*
- *Parser.cs*
- *SemanticAnalyzer.cs*
- *Translator.cs* - translator do języka C#
- *ThreadWrapper.cs* - klasa obudowująca `System.Threading.Thread`. Zapewnia możliwość zatrzymania wątku.
- *Token.cs*
- *PyCTException.cs* - plik definiujący niestandardowe klasy wyjątków
- *SymbolManager.cs* - zarządca tablicy symboli

### Komunikacja

Źródłem dla Lexera jest strumień znaków. (`StreamReader`)

Lexer -> Parser : kolejka tokenów.

Parser -> SemanticAnalyzer : zbudowane struktury składniowe

SemanticAnalyzer -> Translator : zwalidowane struktury składniowe

Wynikiem pracy Translatora jest plik źródłowy z rozszerzeniem `.cs`.

### Wymagania funkcjonalne

- program powinien móc wyświetlić zbudowane struktury składniowe

### Wstępne założenia implementacyjne

- lexer i parser uruchamiane współbieżnie
- limit długości łańcuchów znakowych: 20000 znaków.
- zapisywanie logów wykonania do pliku
- możliwość wyświetlania drzew wyprowadzeń

### Obsługa błędów

Błędy obsługiwany przy pomocy mechanizmu wyjątków.

Aby zwiększyć przewidywalność powstanie niestandardowa klasa wyjątku oraz klasy z niej dziedziczące.

Aby zwiększyć ilość informacji zwrotnej, po części z błędów, należałoby kontynuować proces translacji.

#### Lexer

- nieznany token -> przerywamy translację
- brak wcięcia/złe wcięcie : zakładamy poprawne wcięcie

#### Parser

- brakujące nawiasy

```
if (var == 1: # dokładamy brakujący nawias i kontynuujemy  
if var == 1 and var2 == 3: # przerywamy translację
```

## SymbolManager

- wielokrotnie zadeklarowany symbol -> zakładamy za poprawną tylko pierwszą deklarację i kontynuujemy
- symbol nieznany -> przerywamy

## Translator

- stała niemieszcząca się w słowie maszynowym -> obcinamy stałą tak, aby mieściła się w zakresie i kontynuujemy

## Typy Tokenów

---

```
public enum TokenType
{
    End,
    Indent,
    Identifier,
    Type,
    Value,

    Assignment,
    Colon,
    Comma,
    LeftParenthesis,
    RightParenthesis,
    Return,

    Plus,
    Minus,
    Mult,
    Div,

    LessThan,
    GreaterThan,
    Equals,
    LessEqualThan,
    GreaterEqualThan,

    Not,
    And,
    Or,

    For,
    while,
    If,
    Def,
}
```