Damian Kolaska

- Damian Kolaska
- **TKOM Dokumentacja końcowa**
  - Specyfikacja podzbioru
    - **operatory**
    - **stałe**
    - **zmienne**
    - **domyślna klasa oraz domyślnie importowane moduły**
    - Wcięcia
    - Gdzie mogą znajdować się definicje funkcji
    - Brak rzutowania
    - Nawiasy
    - Pusta linia na końcu programu
  - Składnia
  - Sposób uruchomienia
  - Struktura projektu
    - Pliki
    - Komunikacja
  - Obsługa błędów
  - Struktury danych
  - Sposób testowania
    - Lekser
    - Parser
    - Analizator semantyczny
  - Typy Tokenów

TKOM Dokumentacja końcowa

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

Specyfikacja podzbioru

operatory

arytmetyczne
+, -, *, /
Tłumaczone do C# bez zmian
przypisania
=
Tłumaczone do C# bez zmian
porównania
>, >=, <, <=, ==, !=
Tłumaczone do C# bez zmian
logiczne
not, and, or
Python

```
not x
x and y
x or y
```

C#

```
!x
x && y
x || y
```

stałe

Stałe tworzymy poprzez wyrażenie przypisanie.
Jeśli dany symbol pojawił się po raz pierwszy, a po jego
prawej stronie znajduje się stałe wyrażenie to traktujemy to jako stałą.
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

Zmienne tworzymy następująco
Python

```
var0 = int() # TODO handle this
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 oraz domyślnie importowane moduły

Zmienne, metody, klasy itd. nienależące do żadnej klasy, w C# trafiają do domyślnej klasy *Program* oraz mają publiczny dostęp.
Dodatkowo domyślnie importowane są moduły System.IO oraz System
Python

```python
var = 3
def func(arg: int) -> int:
    loc_var = int(2)
    loc_var = loc_var + 2
    return loc_var * arg
```

C#

```csharp
using System.IO;
using System;

internal static class Program
{
    static int func(int arg)
    {
        int loc_var = 2;
        loc_var = loc_var + 2;
        return loc_var;
    }
    static void Main(string[] args)
    {
        const int var = 3;
    }
}
```

Wcięcia

Zakładam, że wcięcie będzie się składać albo z pojedynczego znaku tabulacji lub z czterech spacji. Wcięcie na ilość spacji niepodzielną przez 4 uznaję za nieprawidłowe.

Gdzie mogą znajdować się definicje funkcji

Założyłem, że deklaracje funkcji mogą znajdować się tylko na najwyższym poziomie zagnieżdżenia. Ponadto znajdują się one zawsze na samym szczycie wynikowego programu.

Brak rzutowania

Dla uproszczenia zakładam brak rzutowania typów. Typy po obu stronach wyrażenia muszą być dokładnie takie same.

Nawiasy

Dopuszczam wyrażenia logiczne bez nawiasów.
Jeśli znajdują się one wewnątrz wyrażenia **if** lub **while**, w C# dodawane są nawiasy.
Ponadto nie dopuszczam używania operatorów arytmetycznych wewnątrz wyrażeń logicznych.
Trzeba w takim wypadku zadeklarować dodatkową zmienną.
Python

```python
if x == 1:
    pass
while y > 2:
    pass
x = not x
```

C#

```
if (x == 1) {}
while (y > 2) {}
x = !x;
```

Pusta linia na końcu programu

Program musi posiadać przynajmniej jedną pustą linię na końcu. W przeciwnym wypadku parser może nie zachowywać się prawidłowo.

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" | "str" | "bool"
digit ::= [0-9]

logicalUnaryOperator ::= "<=" | "<" | ">=" | ">" | "==" | "!="
arithmeticUnaryOperator ::= "*" | "/" | "-" | "+"

integerConstant ::= ([1-9] digit*) | "0"
decimalConstant ::= ([1-9] digit*) | "0" "." digit* [1-9]
logicalConstant ::= True | False
string ::= '"' ([^btnfr"'] | ("\" [btnfr"']))* '"'
constantValue ::=
    integerConstant | decimalConstant | logicalConstant | string

parameter ::= constantValue | identifier

logicalExpression ::=
    "(" recursiveLogicalExpression ")" | (parameter logicalUnaryOperator
parameter) | (not? parameter)

recursiveLogicalExpression ::=
    (logicalExpression | (logicalExpression logicalUnaryOperator
logicalExpression))

arithmeticExpression ::=
    "(" recursiveArithmeticExpression ")" | parameter | (parameter
arithmeticUnaryOperator paramter)

recursiveArithmeticExpression ::=
    (arithmeticExpression | (arithmeticExpression arithmeticUnaryOperator
arithmeticExpression))

statement ::=
    funcCallOrVarDefOrAssign | ifStatement | whileLoop | forLoop | functionDef

funcCallOrVarDefOrAssign ::=
     function_call | variableDef | assignment
```

```
function_call ::=
     identifier "(" ((parameter ",")* parameter)? ")"

ifStatement ::=
    "if" logicalExpression ":" newline
    (tab statement newline)+

whileLoop ::=
    "while" logicalExpression ":" newline
    (tab statement newline)+

forLoop ::=
    "for" identifier "in" "range"
    "(" integerConstant "," integerConstant ")" ":" newline
    (tab statement newline)+

functionDef ::=
    "def" identifier
    "(" ( ((identifier ":" type) ",")* (identifier ":" type) )? ")"
    ("->" type)? ":" newline
    (tab statement newline)+

assignment ::=
    identifier "=" (constantValue | identifier | function_call |
logicalExpression | arithmeticExpression)

variableDef ::=
    identifier "=" type "(" ((parameter ",")* parameter)? ")"

program ::= (statement newline)*
```

Sposób uruchomienia

```
Translator.exe input.py output.cs
Translator.exe input.py
Translator.exe
```

Struktura projektu

Pliki

- CharacterSource
    - FileCharacterSource.cs - źródło znaków wczytujące znaki z pliku
    - ICharacterSource.cs - interfejs dla źródeł znaków
    - StringCharacterSource.cs - źródło znaków wczytujące znaki z łańcucha znakowego
- Lexer
    - Token
        - ITokenSource.cs - interfejs dla źródeł tokenów
        - Token.cs - definicja tokenu
        - TokenSourceMock - źródło tokenów pobierające tokeny z listy podanej w konstruktorze
        - TokenType.cs - definicja typów tokenów

        - TokenValue.cs - klasa zarządzająca wartością tokena
    - Lexer.cs - definicja leksera
- Parser
    - Parser.cs - definicja parsera
    - RValue.cs - definicja r-wartości (zmienna, stała, wywołanie funkcji, wyrażenia nawiasowe)
    - Statement.cs - definicja wyrażenia (defincja funkcji, wywołanie funkcji, wyrażenie warunkowe, pętla warunkowa, ...)
- SemanticAnalyzer.cs - definicja analizatora semantycznego
- Translator.cs - definicja translatora
- TranslationError - definicja klasy wyjątku
- ProgramObject.cs - definicja klasy programu
- Program.cs - główna klasa programu

Komunikacja

Źródłem dla Lexera jest strumień znaków.
Lexer -> Parser : strumień tokenów
Parser -> SemanticAnalyzer : obiekty klasy Statement
SemanticAnalyzer -> Translator : zwalidowane i zmodyfikowane obiekty klasy Statement
Wynikiem pracy Translatora jest plik źródłowy z rozszerzeniem *.cs*.

Obsługa błędów

Lexer
W przypadku błędu zwracany token *Unknown*. Numer linii oraz kolumny odczytywany z tokena.

Parser
W przypadku błędu parser tworzy obiekt typu BadStatement z informacją o błędzie oraz tokenem, który spowodował błąd.

SemanticAnalyzer
W przypadku błędu zgłasza wyjątek typu TranslationError

Translator
W przypadku błędu zgłasza wyjątek typu TranslationError

Struktury danych

- lista *List* używana do przechowywania tokenów oraz wyrażeń
- słownik *Dictionary* do składowania tablicy symboli

Sposób testowania

2 źródła znaków:

- Z pliku
- Z łańcucha znakowego

Lekser

Testy jednostkowe. Proste testy korzystają z źródła znaków opartego na łańcuchu znakowym. Bardziej złożone pobierają znaki z plików.

Parser

- Testy jednostkowe - na wejściu podajemy strumień tokenów. Na wyjściu oczekujemy zbudowania określonej struktury składniowej lub zgłoszenia błędu.
- Test pseudo-jednostowe - podajemy nazwę pliku oraz oczekiwane struktury składniowe.

Analizator semantyczny

- Testy pseudo-jednostkowe - podajemy nazwę pliku i to czy analiza powinna zakończyć się sukcesem

Typy Tokenów

```
public enum TokenType
{
    TabToken,
    NewlineToken,
    End,
    Identifier,

    IntToken,
    StrToken,
    BoolToken,
    FloatToken,

    AssignmentSymbol,
    Colon,
    Comma,
    LeftParenthesis,
    RightParenthesis,
    Return,
    Arrow,

    Plus,
    Minus,
    Star,
    Slash,

    LessThan,
    GreaterThan,
    EqualSymbol,
    NotEqualSymbol,
    LessEqualThan,
    GreaterEqualThan,

    NotToken,
    AndToken,
    OrToken,

    ForToken,
    WhileToken,
```

```
    IfToken,
    DefToken,
    InToken,
    RangeToken,

    LogicalConstant,
    DecimalConstant,
    IntegerConstant,
    StringLiteral,

    UnknownToken
}
```