Damian Kolaska

TKOM Dokumentacja końcowa

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

Założenia

deklaracja stałej

Stałe tworzymy poprzez wyrażenie przypisania.
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";
```

deklaracja zmiennej

Typ zmiennej jest określany przez wyrażenie funkcyjne (int(), float(), str(), bool()).
Brak wartości domyślnej. Zmiennej można inicjować tylko pojedynczym identyfikatorem lub wartością stałą (ale nie wyrażeniem).
Python

```
# var = int(2 + 2) # błąd
# var = int(hello()) # błąd
# var = int(x + 2) # błąd
var1 = int(3)
var2 = float(3.5)
var3 = str("Hello World")
var4 = MyClass("my_class_name")
```

C#

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

domyślna klasa oraz domyślnie importowane moduły

Wyrażenia programu w Pythonie w C# trafiają do domyślnej klasy Program.
Definicje funkcji otrzymują kwalifikator *static*.
Dodatkowo domyślnie importowane są moduły System.IO oraz System
Python

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

C#

```
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 albo 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.

brak przeciążania funkcji ani domyślnych wartości argumentów

Nie dopuszczam przeciążania metod. Argumenty wywołania funkcji muszą się zgadzać dokładnie co do liczby i typu.

nawiasy

Dopuszczam wyrażenia logiczne bez nawiasów.
Jeśli znajdują się one wewnątrz wyrażenia **if** lub **while**, w C# dodawane są nawiasy.
Python

```
while y > 2:
    y = y + 1
```

C#

```
while (y > 2) {
    y = y + 1;
}
```

Ponadto nie dopuszczam używania operatorów arytmetycznych wewnątrz wyrażeń logicznych. Trzeba w takim wypadku zadeklarować dodatkową zmienną.
Python

```
# if x == y + 2: błąd
z = int(0)
z = y + 2
if (x == z)
    return 1
```

Pusta linia na końcu programu

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

Specyfikacja podzbioru

deklaracja funkcji

Python

```
def hello(x: int) -> int:
    return 1
```

C#

```
static int hello(int x)
{
    return 1;
}
```

wywołanie funkcji

Python

```
hello(2)
```

C#

```
hello(2);
```

wyrażenie if

Python

```
if (x == 1): # lub if x == 1:
    x = 2
```

C#

```
if (x == 1)
{
    x = 2;
}
```

pętla while

Python

```
while (x < 100): # lub while x < 100:
    x = x + 2
```

C#

```
while (x < 100)
{
    x = x + 2;
}
```

pętle for

Dopuszczam tylko najprostszą formę pętli for.
Python

```
for i in range (0, 5): # range(5) błąd
    x = x + 2
```

C#

```
for (int i = 0; i < 5; i++)
{
    x = x + 2;
}
```

wyrażenia przypisania

Python

```
x = 2
x = x + 2
x = (y == 2)
x = hello()
```

deklaracje zmiennych

patrz **Założenia**

deklaracje stałych

patrz **Założenia**

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
```

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*.

Implementacja

Proces translacji rozpoczyna się od leksera. Lekser odczytuje kolejne znaki ze źródła znaków (np. pliku, ale niekoniecznie). Ze znaków formuje tokeny, które są odczytywane przez Parser. Parser jest rekursywnie zstępujący, jednak wyrażenia z nawiasami są parsowane w sposób iteracyjny, sprawdzając, czy każdy kolejny token ma sens w stosunku do tokenu poprzedzającego. Parser buduje struktury Statement, przedstawiające obiekty takie jak definicja funkcji, wyrażenie warunkowe, wyrażenie przypisania itd. Struktury te są następnie przekazywane do analizatora semantycznego. Analizator zajmuje się głównie wykrywaniem błędów semantycznych, jednak może modyfikować też otrzymane struktury, np. w przypadku deklaracji stałej. Struktury zwalidowane przez analizator semantyczny trafiają do modułu translatora, gdzie są przepisywane na odpowiadające im wyrażenia w języku C#.

Obsługa błędów

W przypadku wystąpienia błędu. Błąd może być kilkukrotnie zarejestrowany.

```
[11:13:01 INF] Starting parsing...
[11:13:01 ERR] Unknown token at line:1 col:3
[11:13:01 ERR] Cannot recognize statement at line:1 col:3 error:
[11:13:01 ERR] PythonCSharpTranslator.TranslationError: Exception of type 'Pytho
nCSharpTranslator.TranslationError' was thrown.
```

Najpierw lekser wykrył nieznany token. Następnie parser widząc nieznany token, zwrócił błąd o nieznanym wyrażeniu. Na końcu analizator semantyczny zgłosił wyjątek.

Lexer

W przypadku błędu zwracany token *UnknownToken*. 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. Numer linii i kolumnę można odczytać z tokena.

SemanticAnalyzer

W przypadku błędu zgłasza wyjątek typu TranslationError. Numer linii znajduje się w treści wyjątku.

Translator

W przypadku błędu zgłasza wyjątek typu TranslationError. Numer linii znajduje się w treści wyjątku.

Struktury danych

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

Testy

Testy jednostowe

Testy jednostowe znajdują się w katalogu UnitTests.

Lekser

Dla wszystkich typów tokenów, test sprawdzające, czy lekser poprawnie je buduje.
Ponadto testy tokenizacji wybranych złożonych wyrażeń.
Proste testy korzystają z źródła znaków opartego na łańcuchu znakowym.
Bardziej złożone pobierają znaki z plików.

Parser

Testy sprawdzające, czy parser dla określonego zestawu tokenów zwraca poprawny typ wyrażenia. Dla każdego typu wyrażenia przynajmniej jeden test sprawdzający, czy wyrażenie jest poprawnie budowane.

Analizator semantyczny

Testy sprawdzające, czy dla określonego zestawu wyrażeń analizator zwróci błąd czy uzna program za poprawny semantycznie

Translator

Dla każdego typu wyrażenia, prosty test sprawdzający, czy struktura jest poprawnie translowana.

Testy funkcjonalne

Test funkcjonalne znajdują się w katalogu Translator/Resources/FunctionalTests.
Testy są podzielone na katalogi. Nazwa katalogu to nazwa testu. Plik descr.txt zawiera krótki opis testu. Plik input.py zawiera wejście dla testu. Plit expected_output.py zawiera spodziewane wyjście dla testu.
Napisałem prosty skrypt *run_all.py*, który sekwencyjnie uruchamia wszystkie testy, dla każdego testu wypisując jego opis, wejście, wyjście i plik wynikowy.

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
}
```