TKOM Projekt

Protipy

- Składnię piszemy w EBNF (stosujemy czcionkę o stałej szerokości) (warto skorzystać z narzędzi do EBNFa)

Ewentualne rozszerzenia

mapowanie identyfikatorów dostępu
def func() -> public func
def _func() -> protected func
def __func() -> private func

Pytania

Czy translator ma sprawdzać takie rzeczy czy np. stałe nie została zmodyfikowana po deklaracji?
Co powinno być przesyłane do analizy semantycznej?
Czy zwalidowane drzewa wyprowadzeń możemy już translować?
Czy powinienem tokenizować symbol **:** ?

Damian Kolaska

TKOM Dokumentacja Etap 1

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

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

```
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

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
    }
}
```

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

- aplikacja powinna tłumaczyć kod Pythona na możliwy do uruchomienia kod C#
- aplikacja powinna móc wczytać kod źródłowy Pythona z pliku
- program powinien móc wyświetlić zbudowane struktury składniowe

Wymagania niefunkcjonalne

- program nie powinien niepotrzebnie zużywać zasobów. Wątki powinny blokować się w oczekiwaniu na dane.
- limit długości łańcuchów znakowych: 20000 znaków.
- program powinien osiągać możliwie maksymalną współbieżność
- program powinien dostarczać czytelne logi wykonania i zapisywać je do pliku

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 kontynuuować 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,
    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,
}
```