Damian Kolaska

TKOM Dokumentacja Etap 2

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
```

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
    }
}
```

Nawiasy

Dla uproszczenia składni zakładam, że nie dopuszczam notacji bez nawiasów

```
if x == 1:
    pass
```

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]

logicalUnaryOperator ::= "<=" | "<" | ">=" | ">" | "==" | "!="

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

parameter ::= value | identifier

logicalFormula ::=
    (parameter logicalUnaryOperator parameter) | (not? parameter)

logicalExpression ::=
    "(" recursiveLogicalExpression ")" | logicalFormula

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 "=" (value | identifier | function_call | logicalExpression |
arithmeticExpression)

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

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

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*.

Wstępne założenia implementacyjne

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

Obsługa błędów

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

- nieznany token

Parser
Zgłoszenie wyjątku.

- brak wcięcia/złe wcięcie
- brakujące nawiasy

```
if ((var == 1 and (var2 == 2)): # 3 nawiasy otwierające, a tylko 2 zamykające
if ((var == 1 and (var2 == 2))): # dokładamy brakujące nawiasy na koniec i kontynuujemy
```

- inna nieznana struktura składniowa

SymbolManager
Zgłoszenie wyjątku.

- wielokrotnie zadeklarowany symbol -> zakładamy za poprawną tylko pierwszą deklarację i kontynuujemy
- symbol nieznany -> przerywamy
- przypisanie zmiennej złego typu
- potraktowanie zmiennej jak funkcji

Translator
Zgłoszenie wyjątku.

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

Struktury danych

przechowywanie tokenów - tablica o dynamicznym rozmiarze, np. *ArrayList*
budowanie struktur składniowych - drzewo reprezentowane za pomocą grafu, np. *Graph*
tablica symboli - kontener asocjacyjny, np. *Dictionary*

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.

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
}
```