Translacja z Pythona – dokumentacja wstępna

Autor: Jakub Ficek

Temat projektu

Celem projektu jest stworzenie translatora skrośnego dla wybranego podzbioru języka Python do języka C++. Program będzie napisany w języku Python3.

Założenia

- Obsługiwanymi typami są int, float, bool
- Wszystkie zmienne są statycznie typowane (zmienna nie może być najpierw typu int a potem float)
- Obsługa własnych funkcji z adnotowanymi typami argumentów i zwracanej wartości
- Obsługa instrukcji warunkowych if elif else
- Obsługa pętli while
- Obsługa wyrażeń matematycznych i logicznych
- Obsługa wyjścia przy pomocy funkcji print

Przykłady

```
x = 5
while x > 0:
        if x % 3 == 0:
                print(0)
        elif x % 3 == 1:
                print(1)
        else:
                print(2)
        x = x - 1
```

```
#include <iostream>


using namespace std;

int main()
{
        int x = 5;
        while(x > 0)
        {
                if(x % 3 == 0)
                {
                        cout << 0 << endl;
                }
                else if(x % 3 == 1)
                {
                        cout << 1 << endl;
                else
                {
                        cout << 0 << endl;
                }
                x = x - 1;
        }
        return 0;
}
```

```
def sum(a: int, b: int) -> int:
        return a + b

x = 2

while x > 0 and 1 == 1:
        x = x - 1
        print(x)
y = 3 + 2 * 2
print(sum(x, y))
```

```
#include <iostream>

using namespace std;

int sum(int a, int b)
{
        return a + b;
}

int main()
{
        int x = 2;
        while(x > 0 && 1 == 1)
        {
                x = x - 1;
                cout << x << endl;
        }
        int y = 3 + 2 * 2;
        cout << sum(x, y) << endl;
        return 0;
}
```

Lista zdefiniowanych tokenów

‘def’, ‘:’, ‘if’, ‘elif’, ‘else’, ‘while’, ‘(‘, ‘)’, ‘int’, ‘float’, ‘bool’, ‘None’, ‘return’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’, ‘=’, ‘->’, ‘and’, ‘or’, ‘not’, ‘+’, ‘-‘, ‘*’, ‘/’, ‘%’, ‘print’

Gramatyka

program = statements

statements = statement | statement statements

statement_block = indent statements dedent

statement = assignment_statement | function_statement | return_statement | while_statement | if_statement | print_statement | expression_statement | EOL

assignment_statement = identifier ‘=’ expression_statement

function_statement = ‘def’ identifier ‘(‘ [identifier ‘:’ type] { ‘,’ identifier ‘:’ type } ‘)’ ‘->’ (type | ‘None’) ‘:’ EOL statement_block

return_statement = ‘return’ expression_statement

while_statement = ‘while’ expression_statement ‘:’ EOL statement_block

if_statement = ‘if’ expression_statement ‘:’ EOL statement_block [el_statement]

el_statement = else_statement | elif_statement

elif_statement = ‘elif’ expression_statement ‘:’ EOL statement_block [el_statement]

else_statement = ‘else’ ‘:’ EOL statement_block

print_statement = ‘print’ ‘(‘ expression_statement ‘)’

expression_statement = func_call | operation

operation = { unary_op | unary_logic_op } expression_statement { binary_op | binary_logic_op |comparison_op expression_statement }

func_call = identifier ‘(‘ [identifier | value] { ‘,’ (identifier | value) } ‘)’

unary _op = ‘+‘ | ‘-’

unary_logic_op = ‘not’

binary_op = ‘+’ | ‘-‘ | ‘*’ | ‘/’ | ‘%’

binary_logic_op = ‘and’ | ‘or’

comparison_op = ‘==’ | ‘!=’ | ‘<’ | ‘<=’ | ‘>=’ | ‘>’

type = ‘int’ |’float’ | ‘bool’

value = int |float |bool

identifier = [a-zA-Z_][a-zA-Z0-9_]*

intend – wzrost liczba początkowych tabulacji o 1 względem poprzedniej linii

dedend – spadek liczby początkowych tabulacji o 1 względem poprzedniej linii

Sposób uruchomienia

Program na wejściu dostaje ścieżkę do pliku z kodem napisanym w języku Python, jako wyjście program generuje plik z analogicznym kodem napisanym w C++ albo wypisuje komunikat o odpowiednim błędzie (błąd może wystąpić na etapie analizy leksykalnej, składniowej lub syntaktycznej).

Moduły

- **Moduł obsługi plików** – odpowiedzialny za wczytywanie tekstu z pliku i tworzenie pliku wyjściowego
- **Lekser** – odpowiedzialny za rozbicie tekstu wczytanego z pliku na tokeny, które trafiają do analizatora składniowego.
- **Parser** – odpowiedzialny za sprawdzenie czy wczytany ciąg tokenów jest zgodny z gramatyką zdefiniowanego podzbioru oraz zbudowanie drzewa rozbioru.
- **Analizator semantyczny** – odpowiedzialny za sprawdzenie poprawności znaczenia utworzonego przez parser drzewa oraz dodając informacje o typach do drzewa
- **Generator kodu** – odpowiedzialny za konstrukcję kolejnych instrukcji w C++ na podstawie informacji uzyskanych z poprzednich analiz
- **Moduł obsługi błędów** – prezentuje błędy w czytelny sposób

  Na etapie analizy leksykalnej, składniowej i semantycznej mogą wystąpić błędy, gdy wczytywany kod jest niezgodny z założeniami. W takim przypadku odpowiedni analizator generuje komunikat o błędzie, czytelny dla użytkownika.

Testowanie

Testowanie będzie odbywać się przy użyciu krótkich przykładów testujących, porównując uzyskany przez program kod z oczekiwanym.