# TKOM_2021

## Zadanie "do Pythona"

Translator skrośny z C++ do Python

### Autor

Konrad Kulesza 300247

## Dokumentacja wstępna

### Założenia

- wyrażenia `#include <...>` będą ignorowane. W języku wejściowym powinno być wykorzysytwane tylko `<stdio>` ze względu na `std::cout, std::endl`
- wyrażenia `using namespace...` będą pomijane
- jeden plik na wejściu. Poprawny składniowo(możliwy do skompilowania za pomocą `g++`)
- jeden plik na wyjściu
- brak możliwości nadpisania słów kluczowych zarówno jezyka wejściowego jak i wyjściowego

### Podzbiór języka C++

- komentarze
  - jednolinijkowe
  - wielolinijkowe
- zmienne dynamiczne
- typy danych:
  - int
  - string
  - float
  - boolean
- operacje arytmetyczne(zmiennopozycyjne, stałopozycyjne)
- wypisanie na ekran
- instrukcje złożone
  - if, else
  - for
  - while
- funkcje

## We/wy

Program będzie wywoływany z argumentem wejściowym z nazwą jednego pliku do przetłumaczenia, np: `./translator nazwa_pliku.cpp`

W tym samym folderze zostanie utworzony przetłumaczony plik źródłowy w języku python `nazwa_pliku.py`

Dodatkowo, na standardowe wyjście wypisywane będą komunikaty o błędach oraz ostrzeżeniach.

# Testowanie

Do każdego modułu zostaną napisane odpowiednie testy jednostkowe(np. czy lekser odpowiednio rozpoznaje tokeny).

Zostanie przetestowana komunikacja pomiedzy modułami sąsiadującymi.

Zostaną także napisane testy funkcjonalne całego translatora.

Dodatkowo, pojawią się też testy polegające na porównaniu wyników skompilowanego kodu wejściowego i zinterpretowanego kodu wyjściowego.

### Testy funkcjonalne

Poniższe testy przedstawiają dodatkowo w jaki sposób bedą tłumaczone niektóre z konstruckji języka C++ do języka Python

```
## 1.0.variable_assignemt
int a = 5;
int b = 10;
double c = 100.00;
double result = (a+b)*c/20;
-----
a = 5
b = 10
c = 100.00
result = (a+b)*c/20
```

```
## 2.0.if_else_statment
bool cond = true;
int a = 15;
if(a<15){
    a=a+2;
}else if(cond){
    a=a-2;
}
-----
cond = true
a = 15
if a<15:
    a=a+2
elif cond:
    a=a-2
```

```
## 3.0.for_statment
for(int i=0; i<10; i=i+1){
    std::cout<<i<<std::endl;
}
-----
for i in range(0, 10):
    print(i)
```

```
## 4.0.while_statment_test
int i=0;
while( i<10 ){
    ++i;
}
-----
i=0
while i<20:
    ++i
```

```
## 5.0.function_declaration
void fun(int a){
    a = a + 2;
    std::cout<<a;
    return a;
}
-----
def fun(a: int):
    a=a+2
    print(a)
    return a
```

```
## 6.0.multiline_comment
/*
essa
komentarz
    uga buga
*/
-----
'''
essa
komentarz
    uga buga
'''
```

## Obsługa błędów

Błędy będą dzielone na dwa rodzaje:

- krytyczne - plik wynikowy nie zostaje utworzony
- niekrytyczne - ostrzeżenia; wysyłają komunikat, ale przetwarzanie pliku jest kontynuowane

Błędy będą wypisywane na standardowe wyjście.

Komunikat błędu(ostrzeżenia) będzie posiadał:

- rodzaj błędu
- miejsce wystąpienia błędu; linijka, kolumna
- opis błędu( np. na którym etapie )
- na jakim etapie przetwarzania wystąpił błąd

**Przykładowe błędy:**

```
for( i a b ); // Error! Line: 6, column: 1; Syntax error: for_statment ; parser
```

```
else if{} // Error! Line: 11, column 1; Syntax error: if_else_statment ; parser
```

```
class = 5; // Error! Line: 11, column 1; Keyword violation ; parser
```

```
0abc = 2; // Error! Line: 11, column 1; Variable name error ;
parser/analizator_semantyczny
```

```
// int a=1;
a = 3; // Error! Line: 11, column 1; Variable is undefined! ;
analizator_semantyczny
```

# Opis realizacji modułów

## Lekser

Śledzenie linii i kolumny, rozpoznawanie tokenów i przekazywanie ich do parsera. Pomijanie nadmiarowych białych znaków.

## Parser

Tworzenie drzewa składniowego. Analiza od lewej do prawej.  Rekursja lewostronnie zstępująca. Walidacja składni. Zapisywanie odpowiednich tokenów do tablicy symboli.

## Analizator semantyczny

Sprawdzanie czy struktury mają sens w kontekście języka. Sprawdzanie typów.

# Składnia

```
comment             = <single_line_comment> | <multi_line_comment>
multi_line_comment  = "/*" <string_char> "*/"
single_line_comment = "//" <string_char> <end_of_line>

print               = <print_without_nl> | <print_with_nl>
statment            = <if_statment> | <for_statment> | <while_statement>

print_with_nl       = <start_print> "std::endl"
print_without_nl    = <start_print>
```

```
start_print         = "std::cout<<" (<variable_name> | <literal> | <condition>)
while_statement     = "while" "(" <complex_condition> ")" <scope>
for_statment        = "for" "(" <variable_declaration> ";" <complex_condition>
";" <instruction> ")" <scope>
if_statment         = "if" "(" <complex_condition> ")" <scope> [ "else" <scope>]


right_value         = <literal> | <aritmetic_operation> | <variable_name>


aritmetic_operation = <arithmetic_component> <arithmetic_operand>
<arithmetic_component>
aritmetic_component = ( "("<arithmetic_operation>")" ) |<literal>|
<variable_name>| <arithmetic_operation>

function_scope      = <start_of_scope> [<return> [<right_value>] <end_of_ins>]
"}"
scope               = <start_of_scope>  "}"
start_of_scope      = "{" <instruction_block>

instruction_block   = <single_instruction>  {instruction_block}
single_instruction  = (<variable_declaration> | <variable_assignment> |
<statment> | <print>) <end_of_ins>

complex_condition   = <single_condition> | <complex_condition> [
<boolean_operator> <complex_condition>]
single_condition    = <literal> | <comparision>
comparision         = <comparison_operand> <comparison_operator>
<comparison_operand>
comparison_operand  = <variable_name> | <literal>

variable_declaration= <type> <variable_assignment>
function_declaration= <type> <variable_name> "(" [<type><variable_name>][{","
<type> <variable_name}] ")"<function_scope>
variable_assignment = <variable_name> "=" <right_value>

type                = "int" | "bool" | "float" | "string"

variable_name       = <start_of_var> , [char]
start_of_var        = <alphabet_char> | "_"

literal             = <bool_literal> | <float_literal> | <string_literal> |
<integer_literal>

string_literal      = <cudzysłow> char_string <cudzysłów>
bool_literal        = "true" | "false"
float_literal       = {digit} "." {digit}
integer_literal     = <non_zero_digit> {digit}

char_string         = (<digit> | <special_char> | <alphabet_char> | <char> |
<end_of_ins>) {char_string}

aritmetic_opeator   = "+" | "/" | "-" | "*"
boolean_operator    = "&&" | "||"
comparison_operator = "!=" | "==" | ">" | "<" | ">=" | "<="

end_of_line         = "\n" //zastanowić się
end_of_ins          = ";"
```

```
char                = <digit> | <alphabet_char>
alphabet_char       = [a-z] | [A-Z]
special_char        = "!" | "@" | "#" | "%" | "^" | "&" ...

digit               = "0"|<non_zero_digit>
non_zero_digit      = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

## Wstępny-hasłowy pomysł na implementację

- **Klasy/struktury danych**
    - Token - klasa bazowa tokenów.
    - Statment - klasa bazowa wyrażeń złożonych. Rodzaje tych wyrażeń:
        - if, else
        - while
        - for
    - Function - klasa opisująca funkcję. Atrybuty:
        - typ wartości zwracanej
        - lista argumentów
        - lista instrukcji
    - Variable - klasa opisująca zmienną. Atrubyty:
        - nazwa
        - typ danych
    - bufor tymczasowy - do przechowywania kolejnych porcji kodu wejściowego z którego wyodrębniane będą tokeny
    - tablica symboli - zawiera rekord dla każdej zmiennej.
        - globalna
        - lokalna
    - drzewo rozbioru - struktura składająca się z wierzchołków. W liściach będą przetrzymywane tokeny, natomiast w nieliściach bardziej zaawansowane struktury(np. `while_statement`)

- **Wstępny algorytm przetwarzania/generowania kodu**

1. pobieraj znaki aż do rozpoznania tokenu
    - czy token występuje w podzbiorze języka?
2. dopasuj do kontekstu
    - czy tworzy strukture?
    - jeżeli deklaracja zmiennej/funkcji - czy nie nadpisuje słowa kluczowego?
    - jeżeli przypisane - czy zmienna jest w dostępna w danym bloku?
3. sprawdź poprawność
    - czy typy zmiennych się zgadzają?
4. przetłumacz strukture języka c++ na odpowiadającą mu strukture w języku Python
    - zachowaj odpowiednią "tabulacje" bloków instrukcji(zmienna pilnująca "poziomu zagnieżdżenia" bloków)