Translacja z Pythona – dokumentacja końcowa

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.

Podzbiór języka Python

- Obsługiwanymi typy: int, float, bool
- Możliwość definiowania funkcji z adnotacjami informującymi o typie argumentów i zwracanej wartości
- Zmienne są statycznie typowanie, każda zmienna musi mieć adnotację o typie zmiennej
- Obsługa instrukcji warunkowych if elif else
- Obsługa pętli while
- Obsługa wyrażeń zawierających operatory matematyczne +,-.*,/,% oraz logiczne not, and, or
- Obsługa wyjścia przy pomocy funkcji print
- Obsługa wywołań zdefiniowanych przez użytkownika funkcji

Przykłady

```
#include <iostream>

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

int main()
{
    float x;
    int y;
    x = 2.4;
    while(x > 0 && 1 == 1)
    {
        x = x - 1;
        std::cout << x << std::endl;
    }
    y = 3 + 2 * 2;
    std::cout << sum(x, y) << 2 << std::endl;
    return 0;
}
```

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

x : float = 2.4

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

Błąd leksera:

Wejście:

x : int = 2

& y = 3

Wyjście:

Lexical error: line 2

Błąd parsera:

Wejście:

x : int = x++

Wyjście:

syntax error: token PLUS, line 1

```
#include <iostream>

int factorial(int n)
{
    if(! n)
    {
        return 1;
    }
    m = n - 1;
    return factorial(m);
}

int main()
{
    int x;
    int i;
    float z;
    factorial(5);
    x = factorial(10);
    std::cout << x << factorial(7) << std::endl;
    i = 0;
    while(i < 100)
    {
        if(i % 7 == 0)
        {
            z = factorial(i);
            z = z * z;
        }
        else
        {
            z = factorial(i);
            z = z * z * z;
        }
    }
    x = 2;
    std::cout << std::endl;
    return 0;
}
```

```
def factorial(n : int) -> int:
        if not n:
                return 1
        m = n - 1
        return factorial(m)

factorial(5)
x : int = factorial(10)

print(x, factorial(7))

i = 0
while i < 100:
        i : int
        if i % 7 == 0:
                z : float = factorial(i)
                z = z * z
        else:
                z : float = factorial(i)
                z = z * z * z

x = 2

print()
```

Lista zdefiniowanych tokenów

'def' - DEF,
'if' - IF,
'elif' - ELIF,
'else' - ELSE,
'while' - WHILE,
'None' - NONE,
'int' - INT,
'float' - FLOAT,
'bool' - BOOL,
'return' - RETURN,
'print' - PRINT,
':' - COLON,
',' - COMMA,
'->' - RETURN_TYPE,
'\+' - PLUS,
'-' - MINUS,
'*' - MULTIPLY,
'/' - DIVIDE,
'\(' - LP,
'\)' - RP,
'%' - MODULO,
'==' - ISEQUAL,
'!=' - ISNOTEQUAL,
'<=' - ISEQUALLESS,
'<' - ISLESS,
'>=' - ISEQUALLESS,
'>' - ISMORE,
'=' - EQUALS,
'and' - AND,
'or' - OR,
'not' - NOT,
'\d+\.\d+' - VALUE_FLOAT,
'\d+' - VALUE_INT,
'True|False' - VALUE_BOOL,
'[a-zA-Z_][a-zA-Z0-9_]*' - IDENTIFIER
wzrost liczby tabulacji w nowej linii – INDENT
zmniejszenie liczby tabulacji w nowej linii – DEDENT
taka sama liczba tabulacji w nowej linii – NEWLINE

Gramatyka

```
program = statements
statements = statement | statement statements
statement = assignment_statement | function_statement | return_statement | whi
le_statement |
if_statement | print_statement | func_call |  NEWLINE
statement_block = INDENT statements DEDENT
assignment_statement = IDENTIFIER [COLON type] EQUALS expression
function_statement = DEF IDENTIFIER LP[IDENTIFIER COLON type] {COMMA IDENTIFIE
R’ COLON
type} RP RETURN_TYPE(type | NONE) COLON statement_block
return_statement = RETURN expression
```

```
while_statement = WHILE expression COLON statement_block
if_statement = IF expression COLON statement_block [else_statement | elif_stat
ement]
elif_statement = ELIF expression COLON statement_block [else_statement | elif_
statement]
else_statement = ELSE COLON statement_block
print_statement = PRINT LP[expression] {COMMA expression} RP
expression = func_call | operation
operation = [unary_logic_op] (value | IDENTIFIER) {(binary_op | binary_logic_o
p | comparison_op) (value | IDENTIFIER)}
func_call = IDENTIFIER LP (value | IDENTIFIER) {COMMA (value | IDENTIFIER)} RP
unary_logic_op = NOT
binary_op = PLUS | MINUS | MULTIPLY | DIVIDE | MODULO
binary_logic_op = AND | OR
comparison_op = ISEQUAL | ISNOTEQUAL | ISLESS | ISEQUALLESS | ISMORE | ISEQUAL
MORE
type = INT | FLOAT | BOOL
value = VALUE_INT | VALUE_FLOAT | VALUE_BOOL
```

Sposób uruchomienia

Wymagania: biblioteka anytree (do obsługi drzew)

Sposób uruchomienia: python -m transpiler.codegen <ścieżka do pliku wejściowego> <ścieżka do pliku wyjściowego>

Skrypt wczytuje plik wejściowy z kodem w pythonie oraz generuje plik wyjściowy z odpowiednikiem kodu w c++ albo wypisuje odpowiedni błąd na wyjściu.

Moduły

- **Lekser** – odpowiedzialny za rozbicie tekstu wczytanego z pliku na tokeny, które trafiają do parsera. Zwraca iterator do wygenerowanych tokenów. Podczas generowania w przypadku wystąpienia niezdefiniowanego tokenu, rzucany jest LexerError.
- **Parser** – odpowiedzialny za sprawdzenie czy wczytany ciąg tokenów jest zgodny z gramatyką zdefiniowanego podzbioru oraz zbudowanie ast oraz wypełnienia tablicy zmiennych, służącej do inicjowania zmiennych. Parser jest zstępujący. W przypadku niezgodnej gramatyki wczytywanego kodu, parser wywołuje ParserError.
- **Codegen** – odpowiedzialny za przejście drzewa rozbioru oraz tablicy zmiennych i wygenerowanie wyjściowego kodu w c++. Na tym etapie, nie są generowane błędy

  Dodatkowe struktury danych:

- **Token** – obiekt reprezentujący pojedynczy token, przechowujący typ, wartość oraz linię wystąpienia tokenu
- **LexerError** – wyjątek rzucany przy analizie leksykalnej, wypisujący linię wystąpienia błędu
- **ParserError** – wyjątek rzucany w trakcie parsowania , wypisujący token, przy którym wystąpił błąd

- **Tablica symboli** – zawieraja używane w kodzie wejściowym zmienne oraz informacje o ich typie i zasięgu, wypełniana przez parser
- **Tablica akceptowalnych tokenów** – predefiniowana tablica wszystkich zdefiniowanych symboli, używana przez lekser

Reguły translacji

- Stała struktura kodu w C++, pojawiająca się zawsze:

```
#include <iostream>

//definicje funkcji

int main()

{

    // inicjalizacja zmiennych o zasięgu globalnym

    // translacja pythonowych instrukcji

    return 0;

}
```

- Na początku funkcji inicjowane są wszystkie zmienne lokalne
- Zamiana INT, FLOAT, BOOL, NONE na kolejno ‘int’, ‘float’, ‘bool’, ‘void’
- Zamiana funkcji postaci ‘def func(arg1: type1, arg2:type2) -> type3:’ na ‘type3 func(type1 arg1, type2 arg2)’
- INDENT oznaczane przez ‘{‘, a DEDENT przez ‘}’
- Zamiana ‘if expression:’ na ‘if(expression)’
- Zamiana ‘elif expression:’ na ‘else if(expression)’
- Zamiana ‘else:’ na ‘else’
- Zamiana ‘while expression:’ na ‘while(expression)’
- Operatory porównania, przypisania oraz arytmetyczne pozostają bez zmian
- Zamiana operatorów logicznych ‘not’, ‘and’, ‘or’ na kolejno ‘!’, ‘&&’, ‘||’
- Zamiana ‘print(expression1, expression2)’ na ‘std::cout << expression1 << expression2 << std::endl’

Testowanie

W test_lexer.py, test_parser.py znajdują się testy jednostkowe odpowiednich etapów translacji. W test_codegen.py testowanie odbywa się poprzez wczytywanie kodu wejściowego przykładowych plików z kodem w pythonie oraz porównanie przetłumaczonego przez skrypt kodu z oczekiwanym kodem wyjściowym w c++.