Wojciech Sitek

Projekt wstępny TKOM

# Temat

Napisać prosty interpreter języka programowania wraz z możliwością dołączania bibliotek. Celem działania języka jest **zarządzanie infrastrukturą komputerową i serwerową**. Napisać kilka podstawowych bibliotek.

# Wymagania funkcjonalne

1. Język jest interpretowany.
2. Użytkownik może wywoływać program w powłoce lub podając plik ze skryptem.
3. Język umożliwia tworzenie klas, zmiennych globalnych i lokalnych, funkcji.
4. Język umożliwia kontrolę typów, dostarcza typy wbudowane.
5. Język umożliwia ograniczone zarządzanie pamięcią.
6. Język umożliwia tworzenie pętli, instrukcji warunkowych.
7. Język umożliwia łatwe tworzenie i importowanie bibliotek zewnętrznych.
8. Język umożliwia obsługę zmiennych środowiskowych
9. Język umożliwia łatwe zagnieżdżanie kodów innych języków.
10. Język umożliwia tworzenie komentarzy.
11. Język obsługuje ostrzeżenia i błędy.

# Wymagania niefunkcjonalne

1. Język programowania nazywa się Fish.
2. Język jest napisany w języku C++.
3. Udostępniony jest program „lake” do ładowania bibliotek zewnętrznych.
4. Język jest zoptymalizowany do szybkiego działania.

# Sposób uruchomienia, wejście/wyjście

Interpreter jest programem napisanym w języku C++. Przed uruchomieniem powinien zostać skompilowany i zlinkowany z włączoną optymalizacją kompilatora do postaci dwóch plików binarnych, które powinny zostać dodane do katalogu /bin, aby można je było uruchomić z każdego katalogu w powłoce. Program wykonywalny nazywa się „fish” oraz „lake”. Interpretery można uruchomić na kilka sposobów:

1. fish [-v] [-e ZMIENNE\_ŚRODOWISKOWE] – wtedy uruchamia się interpreter w powłoce
2. fish skrypt.fi [-v] [-e ZMIENNE\_ŚRODOWISKOWE] – wtedy wykonuje się skrypt *skrypt.fi*
3. lake install NAZWA\_BIBLIOTEKI – wtedy instaluje się do interpretera biblioteka o podanej nazwie

Wejściem do interpretera może być więc skrypt, kod wpisany ręcznie lub przekierowany do programu strumień tekstowy. Wyjściem z programu mogą być komunikaty błędów i ostrzeżeń, a także celowe wypisywanie informacji np. przez funkcję *print* języka.

# Obsługa błędów

W każdym uruchomieniu programu w języku Fish jest tworzona zmienna Bool \_\_verbose, która domyślnie jest ustawiona na false. Można ją przestawić, dodając opcję [-v] do uruchomienia interpretera fish. Steruje ona, czy ostrzeżenia są wypisywane także na standardowe wyjście. Zmiennej nie można zmienić podczas działania programu.

W przypadku gdy wystąpi ostrzeżenie, wypisywane jest na wyjście błędów (gdy jest ustawione \_\_verbose, także na wyjście standardowe).

W przypadku gdy wystąpi błąd wykonania, jest on wypisywany na wyjście standardowe i błędów. Gdy wykonywany jest skrypt, jego wykonanie zostaje wtedy przerwane. Gdy wykonywane jest polecenie w powłoce, jego wykonanie zostaje przerwane.

# Opis funkcjonalności i przykłady

Przykładowe konstrukcje językowe oraz ich semantyka jest przedstawiona poniżej:

1. **Import bibliotek zewnętrznych**

*import virtualbox as vb #komentarz*

*from server import all*

*from schedule import all*

*from windowsProcess import Process, Status*

1. **Zmienne środowiskowe**

*print(“Nazwa skryptu to: {arg[0]}. Zmiennych środowiskowych jest {arg.len}”)*

1. **Tworzenie zmiennych i aliasów**

*new vb.virtualbox*

*alias OperatingSystems List[vb.OperatingSystem]*

*new OperatingSystems windowsList.create(vb.WindowsSeven, number=10)*

*new Server visServer*

1. **Pętla uproszczona**

*for(10):*

*virtualbox.prepare(windowsList[i], server=visServer, ram=3GB, size=30GB)*

1. **Funkcja**

*def windowsRun(schedule, OperatingSystems windowsList, server, vb.virtualbox):*

1. **Instrukcja warunkowa**

*if(server.freeRAM > 3GB\*windowsList.len and server.size > 30GB\*windowsList.len):*

*virtualbox.runAll()*

*schedule.stop()*

*else: print(‘Nie udalo sie uruchomic, bo tylko ‘+server.freeRAM+” RAMu oraz {server.freeSize} wolnego dysku”)*

*new schedule(‘m’, 1, function=windowsRun, fg=true).run()*

*new Schedule stopSchedule(‘d’, 1)*

1. **Funkcja jako argument**

*def stopSchedFunc(OperatingSystems windowsList):*

*for(10): if(windowsList[i].isRunning()):*

*delete windowsList*

*stopSchedule.setFunction(stopSchedFunc).run()*

*new process = windowsList[4].getProcessWithPID(12)*

*if(virtualbox.getPreparingCount() == 0):*

1. **Pętla zakresowa**

*for(w in windowsList):*

*w.restart()*

1. **Pętla zwykła i typy wbudowane**

*for(Dbl k=3,Str s=””, i=1, finish=false; s.len<100 and i<40 and !finish; i++, k+=5):*

*if(process.getStatus() != WINDOWS\_PROC\_ACTIVE):*

*finish = true*

*elif(s.len < 10):*

*s += process.getName()*

1. **Import bibliotek zewnętrznych**

Biblioteki wcześniej załadowane do interpretera lub te lokalnie dostępne w folderze, w którym wykonywany jest program, można załadować za pomocą instrukcji:

1. „import”, wtedy ładujemy całą bibliotekę.
2. Z dodatkiem „as”, wtedy dodajemy alias dla przedrostka używanego w skrypcie
3. „from {nazwa-biblioteki} import (komponenty | all)”, wtedy dodajemy określone komponenty dostępne bez przedrostka lub dostępne bez przedrostka wszystkie komponenty, gdy podamy słowo kluczowe „all”

Gdyby istniały różnoznaczności w nazwach, wtedy wypisujemy błąd lub ostrzeżenie.

1. **Zmienne środowiskowe**

W języku są zaimportowane zmienne środowiskowe z wywołania skryptu w postaci zmiennej globalnej o nazwie „arg” o typie List, której pierwszym elementem jest nazwa programu. Istnieje możliwość zarządzania pamięciowego tą zmienną (usunięcia jej).

1. **Tworzenie zmiennych i aliasów**

Ze składnią „alias NEW\_NAME OLD\_NAME” można zastępować typy lub zmienne. Ważne, że jeżeli zastępujemy typ, musi on rozpoczynać się wielką literą, a gdy zastępujemy zmienną, musi się ona rozpoczynać z małej litery, gdyż takie są wymagania języka. Jeżeli chodzi o tworzenie zmiennych, jest kilka sposobów na utworzenie zmiennych i obiektów:

1. new TYP ZMIENNA
2. new ZMIENNA – jeżeli typ nazywa się tak samo jak zmienna
3. new BIBLIOTEKA.ZMIENNA – jeżeli typ nazywa się tak samo jak zmienna
4. new TYP – utworzenie obiektu w locie (bez przypisania do konkretnej zmiennej)
5. TYP ZMIENNA – utworzenie identyfikatora bez nadania mu obiektu (logiczne wydaje się od razu przypisanie mu istniejącego lub nowego obiektu). Gdyby w danej instrukcji nie został mu nadany obiekt, zostaje on stworzony konstruktorem domyślnym. Gdy nie ma konstruktora domyślnego, zostaje wypisany błąd.
6. ZMIENNA = OBIEKT – utworzenie zmiennej i przypisanie do niej obiektu. Typem zmiennej staje się typ obiektu

Gdy nie dodamy nic do takiej deklaracji, wywoła się konstruktor domyślny danego typu. Gdy po nazwie zmiennej dopiszemy w nawiasie okrągłym argumenty wykonania, wywoła się konstruktor dopasowany do argumentów. Gdy po zmiennej dopiszemy operator ‘=’ i obiekt po prawej stronie, nastąpi wskazanie nowo utworzonej zmiennej na dany obiekt. Typy muszą być zgodne (tzn. po prawej stronie musi znajdować się typ taki sam lub pochodny). Typem podstawowym jest Type. Po nim dziedziczą wszystkie typy.

Wszystkie zmienne i typy są domyślnie tworzone jako „const”. To znaczy, że nie można przypisać tej nazwie innego obiektu przez czas życia tego obiektu (tzn. do wykonania instrukcji delete). Aby zmienna mogła zmienić wskazanie na inny obiekt, musi być oznaczona słowem kluczowym „mut”, np.

*new mut student*

*student = new Student*

Istnieje także słowo kluczowe „static”, które dodajemy po słowie „mut”, jeżeli występuje, lub po słowie „new”, lub na początku deklaracji

1. **Pętla uproszczona**

Instrukcja for(INT) odwzorowuje konstrukcję for(Int i=0; i<INT; ++i), znaną z języka C. Zmienna i jest dostępna w środku pętli i jedną instrukcję po zakończeniu pętli.

1. **Funkcja i jej wywołanie**

Język obsługuje funkcje i metody. Składają się one na początku ze słowa kluczowego „def”, po którym następuje typ zwracany lub jego brak, gdy nie ma wartości zwracanej. Następnie występuje unikalna (brak powtórzeń z występującymi w tym zasięgu zmiennymi i funkcjami) nazwa funkcji, rozpoczynająca się z małej litery, po której, w nawiasie wypisane są argumenty, rozdzielone przecinkami, w dwóch możliwych formatach:

1. TYP ZMIENNA
2. ZMIENNA – gdy jej nazwa jest taka sama jak typ.

Możliwe jest dodanie do któregokolwiek argumentu wartości domyślnej w postaci „=DEFAULT”. Obiekt DEFAULT jest kopiowany wraz ze wszystkimi jego powiązaniami do trzeciego poziomu i przechowywany w pamięci przez cały czas, w którym jest możliwe wywołanie funkcji. Gdyby powiązania przekraczały trzeci poziom (tzn. jeden obiekt wskazuje na drugi, drugi na trzeci, trzeci na czwarty), wypisywany jest błąd wykonania.

Funkcja zostaje wywołana przez podanie jej nazwy i w nawiasie argumentów wywołania funkcji. Możliwe jest podawanie po prostu zmiennych (np. *createStudent(imie, nazwisko, srednia)*) lub podawanie nazwy argumentu i nazwy zmiennej rozdzielonej znakiem ‘=’ (np. *createStudent(imie, nazwisko, srednia=srednia, aktywny=false)*.

Funkcja o jednym identyfikatorze może mieć jedną postać. Nie jest możliwe definiowanie dwóch funkcji o tej samej nazwie w tym samym zasięgu z różnymi argumentami.

1. **Instrukcja warunkowa**

Instrukcja warunkowa rozpoczyna się słowem kluczowym „if”, następnie w nawiasie występuje wyrażenie o typie wbudowanym Bool (w przeciwnym razie wypisywany jest błąd). Występują operatory „or”, „and” oraz „not”. Po zamknięciu nawiasu z dwukropkiem rozpoczyna się blok instrukcji warunkowej. Tak jak w języku Python, po zakończeniu bloku „if” możliwa jest instrukcja „elif(WARUNEK)” lub „else”.

Używane jest wcięcie (tab) do zaznaczenia bloku podrzędnego. W przypadku gdy blok podrzędny ma jedną linię, jest możliwe zapisanie go w tej samej linii co instrukcja „if” lub „for”.

1. **Funkcja jako argument**

Jest możliwe używanie funkcji, która została wcześniej zdefiniowana jako argumentu lub zmiennej, gdyż jest to także obiekt typu Def[RETURN\_TYPE]. Gdy funkcja nie zwraca nic, nawiasy kwadratowe pozostają puste.

1. **Pętla zakresowa**

Drugim rodzajem pętli „for” jest pętla zakresowa. Zachowuje się ona dokładnie jak pętla zakresowa w języku Python.

1. **Pętla zwykła i typy wbudowane**

Trzecim rodzajem pętli jest pętla zwykła (znana z języka C). Pierwszy argument pętli to deklaracje zmiennych dostępnych podczas wykonania pętli i jedną instrukcję po zakończeniu pętli. Drugi argument to wyrażenie typu Bool określające, czy pętla wykonuje kolejny przebieg. Trzeci argument to instrukcje, wykonywane po zakończeniu każdego przebiegu pętli.

Typy wbudowane są określone w bibliotece standardowej i są to m.in.: Int, Uns, Dbl, Def[], Str, Bool, Char, List[], Vec[], Union[]. Typ Uns to znany z innych języków typ „unsigned int”. Dla typu Int, Uns oraz Dbl dostępny jest operator postinkrementacji ++ oraz operator += oraz analogicznie --, -=, \*=, /=, %=, podobnie jak w języku C. Zdefiniowane są słowa kluczowe „true” i „false” jako możliwe wartości typu Bool.

1. **Listy**

Jest możliwe tworzenie kontenerów w języku Fish. Wszystkie kontenery dziedziczą po (pustym) typie Container. Są dwa typy wbudowane, także dziedziczące po typie Container:

1. List – zaimplementowana jako lista dwukierunkowa
2. Vec – zaimplementowany jako wektor z C++
3. **Inne języki**

W język jest wbudowany mechanizm *lang*, który ułatwia wbudowanie w język kodów innych języków, takich jak C, C++, Java, Python, język powłoki (Sh), możliwe jest także dodanie innych języków. Jego użycie wygląda następująco:

*lang Python(Int x):*

*import pandas as pd*

*c = pd.read\_csv(“moje\_csv”)*

*i = 0*

*while(i<x):*

*print(c)*

*i += 1*

Gdy język jest interpretowany, wtedy wykonywany jest po kolei każda linia w czasie wykonania. Gdy zaś język jest kompilowany, do tymczasowego pliku zapisywany jest cały kod w tym języku, a następnie kompilowany i wykonywany zgodnie z zasadami tego języka.

1. **Klasy**

W tym języku dostępne jest także tworzenie klas. Klasa tworzy nowy typ, który dziedziczy po typie Type lub typie zdefiniowanym przez programistę. Konstruktor to metoda o nazwie takiej samej jak klasa (i małej pierwszej literze). Wszystkie pola klasy są domyślnie prywatne, można je uczynić publicznymi poprzedzając je słowem kluczowym „public”. W każdej metodzie dostępne są obiekty „me” (to samo co self w języku Python) oraz „super” (dostęp do metod klasy nadrzędnej). Słowo kluczowe „pass” funkcjonuje tak samo jak w języku Python. Oto przykładowa klasa z dziedziczeniem:

*class Student(Osoba):*

*public wydzial*

*public Osoba nauczyciel*

*Dbl srednia*

*def student(Str imie, Str nazwisko, Dbl srednia, Osoba nauczyciel, wydzial, Bool aktywny=true):*

*pass*

*def Str toStr():*

*return „Jestem studentem o sredniej {me.srednia}”*

*def setSrednia():*

*pass*

Gdy w konstruktorze pojawiają się argumenty o nazwie takiej samej jak pola klasy, są automatycznie przypisywane do pól klasy. Dla pól prywatnych domyślnie są tworzone metody getZMIENNA() oraz setZMIENNA(). Aby dezaktywować te metody, należy je nadpisać w sposób pokazany powyżej. Metoda „toStr” jest używana do wypisywania informacji o obiekcie np. za pomocą funkcji wbudowanej *print*. Metody „def TYP toTYP()” pozwalają na konwersję obiektu na określony typ. Szczególnie przydaje się to przy tekstach i wartościach logicznych. Metody są automatycznie wykorzystywane do konwersji na dany typ, gdy jest on wymagany. Są obsługiwane metody operatorowe z C++ (np. operator+, operator++).

# Formalna specyfikacja

Każdy token nieterminalny będzie zrealizowany jako oddzielna struktura danych, dziedzicząca po strukturze Token. Każdy token terminalny będzie zrealizowany jako stała w enumeracji TerminalTokens. Lista tokenów będzie zrealizowana jako lista dwukierunkowa „tokens”. Analizator leksykalny i składniowy i wszystkie jego struktury danych będą należały do przestrzeni nazw „analizator”.

Rozróżnialne są następujące tokeny:

## Analiza leksykalna

### Tokeny pośrednie:

**<letter>** := <upper\_letter> | <lower\_letter>

**<upper\_letter>** := [A-Z]

**<lower\_letter>** := [a-z]

**<zero>** := ‘+0’ | ‘-0’ | ‘0’

### Produkty:

**<signed\_int>** := ‘-‘ [1-9] [0-9]\*

**<unsigned\_int>** := ‘+’? [1-9] [0-9]\* | <zero>

**<dbl>** := <int> (‘.’ [0-9]+)?

**<lower\_var>** := <lower\_letter>{1,63}

**<var>** := <lower\_letter> <letter>{0,62}

**<type>** := <upper\_letter> <letter>{0,62}

**<constant>** := ‘\_\_’ <upper\_letter>{1,62}

**<lang\_line>** := ‘!\t’ [^\n]{0, 1023}

**Tokeny terminalne**

Zadaniem analizatora leksykalnego jest przetworzenie linii kodu w listę tokenów. Spacje oraz komentarze będą pomijane – służą one jedynie jako separatory dla niektórych tokenów. Sąsiadujące tokeny składające się z liter muszą być oddzielone co najmniej jedną spacją. Tabulacja pełni rolę oznaczenia bloku wewnętrznego (analogicznie do języka Python).

## Analiza składniowa

**<int>** := <signed\_int> | <unsigned\_int>

**<library>** := <lower\_var>

**<name>** := <var> | <lower\_var>

**<variable>** := (<library> ‘.’)? <name>

**<list\_variable>** := <list\_object> | <variable>

**<list\_object>** := <list\_variable> ‘[‘ (<unsigned\_int> | <int\_object> ‘]’

**<library\_shortcut>** := <library>

**<simple\_object>** := <type> | <function> | <variable> | <list\_object>

**<function>** := <name>

**<statement>** := <simple\_stmt> NEWLINE | <compound\_stmt> NEWLINE

**<simple\_stmt>** := <expr\_stmt> | <delete\_stmt> | <pass\_stmt> | <import\_stmt> | <new\_stmt> | <assign\_stmt> | <augassign\_stmt> | <loop\_stmt>

**<new\_stmt>** := NEW <type> | NEW? MUT? STATIC? (<type> <variable> | <variable>) | <new\_stmt> ‘(‘ <arguments> ‘)’

**<assign\_stmt>** := <variable> ‘=’ <object>

**<import\_stmt>** := ‘import’ <library> (‘as’ <library\_shortcut>)? | ‘from’ <library> import (‘all’ | <object>+)

**<pass\_stmt>** := ‘pass’

**<object>** := <simple\_object> | <new\_stmt> | <assign\_stmt>

**<expression>** := <object> | <expression> <operator> <expression>

**<operator>** := ‘+’ | ‘-‘ | ‘\*’ | ‘%’

**<expr\_stmt>** := <object> | <object> <run\_method>\* | <expression>

**<run\_method>** := ‘.’ <function>

**<augassign\_stmt>** := <object> ((<augassign> <object>)|<postaugassign>)

**<augassign>** := ‘+=’ | ‘-=’ | ‘\*=’ | ‘/=’ | ‘%=’

**<postaugassign>** := ‘++’ | ‘—'

**<delete\_stmt>** := ‘delete’ <object>

**<compound\_stmt>** := <if\_stmt> | <for\_stmt> | <while\_stmt> | <try\_stmt> | <funcdef> | <classdef> | <lang\_stmt>

**<if\_stmt>** := ‘if’ ‘(‘ <bool\_stmt> ‘)’ ‘:’ NEWLINE <body> (‘elif’ ‘(‘ <bool\_stmt> ‘)’ ‘:’NEWLINE <body>)\* (‘else’ ‘:’ NEWLINE <body>)?

**<bool\_stmt>** := ‘true’ | ‘false’ | <expression> | <bool\_stmt> (‘and’ | ‘or’) <bool\_stmt> | not <bool\_stmt>

**<body>** := (TAB <statement>)+

**<for\_stmt>** := ‘for’ ‘(‘ (<unsigned\_int>

| <unsigned\_int\_object>

| <variable> ‘in' <container\_object>

| (((<assign\_stmt> | <new\_stmt>) ‘,’)\* (<assign\_stmt>|<new\_stmt>))? ‘;’ <bool\_stmt> ‘;’ ((<augassign\_stmt> ‘,’)\* <augassign\_stmt>)?

) ‘)’ ‘:’ <for\_body> <statement>?

**<for\_body>** := <body>

**<flow\_stmt>** := TAB (‘break’ | ‘continue’ | ‘throw’ <exc\_object>) NEWLINE

**<exc\_object>** := <object>

**<int\_object>** := <object>

**<str\_object>** := <object> | ‘\’’ [^\’]{0,1023} ‘\’’ | ‘\”’ [^{\”]{0,1023} (‘{‘ <variable> ‘}’)? ‘\”’

**<container\_object>** := <object>

**<while\_stmt>** := ‘while’ ‘(‘ <bool\_stmt> ‘)’ ‘:’ <while\_body>

**<while\_body>** := <body>

**<try\_stmt>** := ‘try’ ‘:’ <try\_body> ‘catch’ ‘(‘ <argument> (‘,’ <argument>)\* ‘)’ ‘:’ <catch\_body>

**<catch\_body>** := <body>

**<funcdef>** := ‘def’ <type>? <function> ‘(‘ <arguments> ‘)’ ‘:’ <func\_body>

**<func\_body>** := <body>

**<arguments>** := ( (<argument> ‘,’)\* <argument> )?

**<argument>** := <type>? <variable> (‘=’ <object>)?

**<classdef>** := ‘class’ (‘(‘ <type> ‘)’)? ‘:’ NEWLINE <class\_body>

**<class\_body>** := (TAB ( <new\_stmt> | <funcdef> ) NEWLINE)+

**<lang\_stmt>** := ‘lang’ <function> (‘(‘ <arguments> ‘)’)? ‘:’ NEWLINE <lang\_body>

**<lang\_body>** := (<lang\_line> NEWLINE)+

Dla wyjaśnienia, tokeny terminalne zostały napisane wielkimi literami lub określone tekstowo. Tokenem podstawowym, który podlega wykonaniu, jest token <statement>. Analizator składniowy będzie miał za zadanie zapamiętanie obecnego stanu interpretera i analizę kolejnej podanej linii kodu programu. Krok analizy będzie odbywał się każdorazowo po jednym tokenie. Analiza składniowa zajmuje się zamianą ciągu tokenów na drzewo wykonania. Na samym początku korzeniem drzewa jest token <statement>.

Drzewo wykonania jest wykonywane przez przeszukiwanie go wgłąb.

## Analizator typów

Następnie, mając w pamięci obecny stan interpretera, program analizuje zgodność typów. Gdy typy są niezgodne, wyrzuca błąd wykonania.

# Sposób realizacji

Każdy stworzony w programie obiekt będzie miał odzwierciedlenie w pamięci interpretera. Będzie on utworzony jako obiekt C++ opakowany typem języka Fish. Będzie liczona liczba wskazań na dany obiekt. Jeżeli po wykonaniu jednej instrukcji liczba odwołań będzie wynosiła 0, obiekt ten zostanie usunięty automatycznie. Można także usuwać obiekty ręcznie przez instrukcję delete.

Nie ma potrzeby nadawania innych nazw obiektom niż ich podstawowe zmienne, więc istnieją one w pamięci po prostu jako unikalne identyfikatory.

# Testowanie

Testowanie będzie przeprowadzone za pomocą testów jednostkowych na cztery sposoby:

1. Testowanie analizy leksykalnej
2. Testowanie analizy składniowej
3. Testowanie analizy typów
4. Testowanie wyników wykonania programu.

Testy zostaną wykonane w bibliotece Boost.test.