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 tworzenie komentarzy.
10. 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 linkuje 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 przez program.

# Obsługa błędów

Program obsługuje ostrzeżenia (warning) oraz błędy (error). W przypadku gdy wystąpi błąd lub ostrzeżenie, są one wypisywane na wyjście błędów. Gdy wykonywany jest skrypt a wystąpi błąd, 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*

*import server as s*

*import schedule as sch*

*import windowsProcess*

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(virtualbox.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){*

*fori(10){*

*if(windowsList[i].isRunning()){*

*delete windowsList*

*}*

*}*

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

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

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

1. **Pętla zakresowa**

*foreach(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*

*}*

*else{*

*if(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 pod swoją nazwą.
2. Z dodatkiem „as”, wtedy dodajemy alias dla przedrostka używanego w skrypcie

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 fori(INT) odwzorowuje konstrukcję for(Int i=0; i<INT; ++i), znaną z języka C. Zmienna i jest dostępna w środku 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 zmienną bez zewnętrznych powiązań i jest przechowywany w pamięci przez cały czas, w którym jest możliwe wywołanie funkcji.

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 nawiasem klamrowym rozpoczyna się blok instrukcji warunkowej. Tak jak w języku Python, po zakończeniu bloku „if” możliwa jest instrukcja „else”. Jawnie nie jest możliwa konstrukcja „else if”.

Używane są klamry (tak jak w języku C) do zaznaczenia bloku podrzędnego. Klamry są obowiązkowe nawet, gdy blok podrzędny ma jedną linię.

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. Rozpoczyna się ona od słowa kluczowego „foreach”.

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, Unsigned, Dbl, Def[], Str, Bool, Char, List[]. Typ Unsigned to znany z innych języków typ „unsigned int”. Dla typu Int, Unsigned 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.

Wartość 0 oraz pusty napis jest konwertowana na „false”, a pozostałe wartości są konwertowane na „true”.

1. **Listy**

Jest możliwe tworzenie kontenerów w języku Fish. Wszystkie kontenery dziedziczą po (pustym) typie Container. Jest jeden typ wbudowany, dziedziczący po typie Container: List – zaimplementowana jako lista dwukierunkowa

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*

*secret pesel*

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

*def Str toStr(){*

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

*}*

*def setSrednia(){}*

*}*

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 użyć słowa kluczowego „secret” przed zmienną. 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++, 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

Letter = UpperLetter | LowerLetter

UpperLetter = "A" |...| "Z"

LowerLetter = "a" |...| "z"

Zero = ["+" | "-"], "0";

NonZero = "1" |...| "9";

Digit = NonZero | Zero;

Sign = ? ASCII 32-126 ?

Str = "\"" {(Sign - "{") | ("{", Expression, "}")} "\"" | "'" {Sign} "'"

Bool = "true" | "false"

ConstantValue = Int | Dbl | Str | Bool

Int = SignedInt | UnsignedInt;

SignedInt = ["-"], NonZero, {Digit};

UnsignedInt = ["+"], NonZero, {Digit} | Zero;

Dbl = Int, [".", {Digit}];

Variable = LowerLetter, {Letter};

LowerVar = LowerLetter, {LowerLetter};

Type = UpperLetter, {Letter};

Constant = "\_\_", UpperLetter, {UpperLetter};

## Analiza składniowa

Body = {Statement};

Statement = CompoundStatement | SimpleStatement;

SimpleStatement = ExpressionStatement | DeleteStatement | ImportStatement | NewStatement | AssignStatement | ControlStatement;

ExpressionStatement = Expression, Newline;

DeleteStatement = "delete", Variable, Newline;

ImportStatement = "import", Library, ["as", Library], Newline;

NewStatement = NewExpression, Newline;

AssignStatement = AssignExpression, Newline;

ControlStatement = ("break" | "continue"), Newline;

Newline = '\n' | ';';

Expression = [(ArithmeticExpression | NewExpression), {',', (ArithmeticExpression | NewExpression)}];

NewExpression = "new", ["mut"], ["static"], [Type], Variable, ['(', ArgumentList, ')'];

AssignExpression = NewExpression, AssignOperator, Term;

ConditionalExpression = AndExpression, {"or", AndExpression};

AndExpression = OrExpression, {"and", OrExpression};

OrExpression = ["not"], RelativeExpression, {RelativeOperator, ["not"], RelativeExpression};

RelativeExpression = '(' ConditionalExpression ')' | ArithmeticExpression;

ArithmeticExpression = AddExpression, {AddOperator, AddExpression};

AddExpression = MultiplyExpression, {MultiplyOperator, MultiplyExpression};

MultiplyExpression = [UnarySign | PreIncrement], Term;

Term = ConstantValue | '(', ArithmeticExpression, ')'

| ((Variable, [PostIncrement] | Constant | FunctionCall), {ArraySubscript}), {'.', (Variable, [PostIncrement] | Constant | FunctionCall), {ArraySubscript}}

FunctionCall = Variable, '(', ArgumentList, ')';

ArgumentList = [Argument, {',', Argument}];

Argument = Term | ArithmeticExpression;

Library = LowerVar;

AssignOperator = '=' | "+=" | "-=" | "\*=" | "/=" | "%=";

RelativeOperator = "==" | "!=" | "<" | ">" | "<=" | ">=";

AddOperator = "+" | "-";

MultiplyOperator = "\*" | "/" | "%";

UnarySign = "+" | "-";

PreIncrement = "++" | "--";

PostIncrement = "++" | "--";

ArraySubscript = "[", (UnsignedIntTerm | ([UnsignedIntTerm], ':', [UnsignedIntTerm]), "]";

UnsignedIntTerm = Term;

CompoundStatement = IfStatement | WhileStatement | ForStatement | ForiStatement | ForeachStatement | FunctionDefinition | ClassDefinition;

BlockInstruction = "{", {Statement}, "}";

IfStatement = "if", '(', ConditionalExpression, ')', BlockInstruction, ["else", BlockInstruction];

WhileStatement = "while", '(', ConditionalExpression, ')', BlockInstruction;

ForStatement = "for", '(', Expression, ';', ConditionalExpression, ';' Expression, ')', BlockInstruction;

ForiStatement = "fori", '(', UnsignedIntTerm, ')', BlockInstruction;

ForeachStatement = "foreach", '(', Variable, "in", ListVariable, ')', BlockInstruction;

FunctionDefinition = "def", [Type], Variable, '(', ParameterList, ')', BlockInstruction;

ClassDefinition = "class", Type, '(', Type, ')', '{', ClassBody, '}';

ParameterList = [Parameter, {',', Parameter}];

Parameter = [Type], Variable, ['=', Default];

Default = Term;

ListVariable = Variable, [ArraySubscript];

ClassBody = {ClassBodyStatement};

ClassBodyStatement = (MemberDefinition | FunctionDefinition), Newline;

MemberDefinition = ["public" | "private" | "secret"], ["mut"], ["static"], [Type], Variable, ['(', ArgumentList, ')'];

Tokenem podstawowym jest Body.

## 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.