Wojciech Sitek

Dokumentacja projektowa TKOM

1. Temat

Napisać prosty interpreter języka programowania wraz z możliwością dołączania bibliotek. Napisać kilka podstawowych bibliotek.

2. 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.

3. Wymagania niefunkcjonalne

1. Język programowania nazywa się Fish.
2. Język jest napisany w języku C++.
3. Język jest zoptymalizowany do szybkiego działania.

4. Sposób uruchomienia, wejście/wyjście

Interpreter jest programem napisanym w języku C++. Przed uruchomieniem powinien zostać skompilowany i zlinkowany z włączoną optymalizacją kompilatora do postaci pliku binarnego, który powinien zostać dodane do katalogu /bin, aby można je było uruchomić z każdego katalogu w powłoce. Program wykonywalny nazywa się „fish"". Interpretery można uruchomić na 2 sposoby:

a) fish –file skrypt.fi – wtedy wykonuje się skrypt *skrypt.fi*
b) *fish* **–** wtedy otwiera się wykonywanie w linii komend.

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.

5. Obsługa błędów

Program obsługuje ostrzeżenia (warning) oraz błędy (error). W przypadku gdy wystąpi błąd lub ostrzeżenie, są one wypisywane na standardowe wyjście. Gdy wykonywany jest skrypt a wystąpi błąd, jego wykonanie zostaje wtedy przerwane. Gdy wykonywane jest polecenie w powłoce, jego wykonanie zostaje przerwane.

6. Opis funkcjonalności i przykłady

Przykładowe konstrukcje językowe oraz ich semantyka jest przedstawiona poniżej:

1. **Import bibliotek zewnętrznych**

```
import virtualbox as vb
import server as s
import schedule as sch
import windowsProcess
```

2. Tworzenie zmiennych i aliasów

```
new vb.virtualbox
alias OperatingSystems List[vb.OperatingSystem]
new OperatingSystems windowsList.create(virtualbox.WindowsSeven, number=10)
new Server visServer
```

3. Pętla uproszczona

```
fori(10)
        virtualbox.prepare(windowsList[i], server=visServer, ram=3GB, size=30GB)
```

4. Funkcja

```
def windowsRun(schedule, OperatingSystems windowsList, server, vb.virtualbox){
```

5. Instrukcja warunkowa

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

6. 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){
```

7. **Pętla zakresowa**

```
        foreach(w in windowsList){
                w.restart()
        }
}
```

8. 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 napisane w interpreterze, w którym wykonywany jest program, można załadować za pomocą instrukcji:

a) „import”, wtedy ładujemy całą bibliotekę pod swoją nazwą.
b) Z dodatkiem „as”, wtedy dodajemy alias dla przedrostka używanego w skrypcie

Została zaimplementowana tylko jedna biblioteka implementująca funkcję „print” wypisującą informacje na standardowe wyjście.

2. **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:

a) new TYP ZMIENNA
b) new ZMIENNA – jeżeli typ nazywa się tak samo jak zmienna
c) new BIBLIOTEKA.ZMIENNA – jeżeli typ nazywa się tak samo jak zmienna
d) new TYP – utworzenie obiektu w locie (bez przypisania do konkretnej zmiennej)

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.

Słowa kluczowe mut oraz static niestety nie zostały zaimplementowane w interpreterze.

3. **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.

4. **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:

a) TYP ZMIENNA
b) 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.

5. 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.

6. 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”.

7. 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”.

W parserze została zaimplementowana obsługa typów, jednak nie ma statycznej obsługi typów, zgodnie z ustaleniami z Prowadzącym.

8. Listy

Jest możliwe tworzenie kontenerów w języku Fish. Kontenerem wbudowanym jest lista dwukierunkowa oparta na std::list.

9. 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*.

Klasy zostały w pełni zaimplementowane w programie, jednak nie zostały przetestowane z powodu braku czasu.

7. Formalna specyfikacja

Każdy symbol będzie zrealizowany jako oddzielna struktura danych, dziedzicząca po strukturze Symbol. Każdy token terminalny będzie zrealizowany jako stała w enumeracji wraz z wartością tekstową jako struktura Token z polami TokenType type oraz std::string value.

Klasa Symbol i Token dziedziczą po klasie Obj, jest częścią interpretera i jest używana do implementacji środowiska (klasa Env). Referencje na obiekty Obj są wartościami tablicy mieszającej każdego środowiska.

Rozróżnialne są następujące tokeny i symbole:

```
8.  IDENTIFIER = Letter, {Letter | Digit | "_"};
    CONSTANT = "_", UpperLetter, {UpperLetter};

    File = {FilePart};
    FilePart = FunctionDefinition | ClassDefinition | Statement;
    Statement = (CompoundStatement | SimpleStatement), [";"];

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

    ExpressionStatement = Expression;
    DeleteStatement = "delete", IDENTIFIER;
    ImportStatement = "import", IDENTIFIER, ["as", IDENTIFIER];
```

```
NewStatement = NewExpression;
AssignStatement = AssignExpression;
ControlStatement = "break" | "continue" | "return", [ConditionalExpression];
AliasStatement = "alias", Type, Type;

Type = IDENTIFIER;
Bool = "true" | "false"

Expression = ConditionalExpression | NewExpression | AssignExpression;

NewExpression = "new", ["mut"], ["static"], [Type], (IDENTIFIER | CONSTANT), ['(', ArgumentList,
')'] [AssignOperator, ConditionalExpresion];
AssignExpression = IDENTIFIER, AssignOperator, ConditionalExpression;

ConditionalExpression = AndExpression, {"or", AndExpression};
AndExpression = OrExpression, {"and", OrExpression};
OrExpression = [UnaryNot], RelativeExpression, {RelativeOperator, [UnaryNot],
RelativeExpression};
RelativeExpression = '(' ConditionalExpression ')' | ArithmeticExpression;

ArithmeticExpression = AddExpression, {AddOperator, AddExpression};
AddExpression = MultiplyExpression, {MultiplyOperator, MultiplyExpression};
MultiplyExpression = [UnarySign], Term;
Term = Int | Dbl | Str | Bool | '(', ArithmeticExpression, ')'
    | (IDENTIFIER | CONSTANT | FunctionCall), {ArraySubscript}, {'.', (IDENTIFIER | CONSTANT |
FunctionCall), {ArraySubscript}}


FunctionCall = IDENTIFIER, '(', ArgumentList, ')';
ArgumentList = [Argument, {',', Argument}];
Argument =  ConditionalExpression;


AssignOperator = '=' | "+=" | "-=" | "*=" | "/=" | "%=";
RelativeOperator = "==" | "!=" | "<" | ">" | "<=" | ">=";
AddOperator = "+" | "-";
MultiplyOperator = "*" | "/" | "%";
UnarySign = "+" | "-";
UnaryNot = "not" | "!";

ArraySubscript = "[", UnsignedIntTerm, "]";
UnsignedIntTerm = Term;


CompoundStatement = IfStatement | WhileStatement | ForStatement | ForiStatement |
ForeachStatement;
BlockInstruction = ("{", {Statement}, "}") | Statement;

IfStatement = "if", '(', ConditionalExpression, ')', BlockInstruction, ["else", BlockInstruction];
WhileStatement = "while", '(', ConditionalExpression, ')', BlockInstruction;
ForStatement = "for", '(', [Expression], ';', ConditionalExpression, ';' [Expression], ')',
BlockInstruction;
ForiStatement = "fori", '(', UnsignedIntTerm, ')', BlockInstruction;
ForeachStatement = "foreach", '(', IDENTIFIER, "in", Term, ')', BlockInstruction;
FunctionDefinition = "def", [Type], IDENTIFIER, '(', ParameterList, ')', BlockInstruction;
ClassDefinition = "class", Type, '{', ClassBody, '}';

ParameterList = [Parameter, {',', Parameter}];
Parameter = [Type], IDENTIFIER, ['=', Default];
```

```
Default = Term;


ClassBody = {ClassBodyStatement};
ClassBodyStatement = MemberDefinition | FunctionDefinition;
MemberDefinition = ["public" | "private" | "secret"], ["mut"], ["static"], [Type], IDENTIFIER, ['(',
ArgumentList, ')'];
```

Symbolem podstawowym jest File. Analiza leksykalna oraz parserowa została przedstawiona i zaakceptowana przez Prowadzącego.

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

9. Testowanie

Testowanie będzie przeprowadzone za pomocą testów jednostkowych Boost.test oraz przez wykonywanie przykładowych skryptów w napisanym języku.

10. Informacje dodatkowe

Projekt został napisany w języku C++. Jest to bardzo prosty język programowania pomiędzy językiem Python a C++.

11. Struktura plikowa

Większość klas znajduje się w folderze Analizator. Podzielony jest on na podstawowe klasy Lexer, Parser, Symbol oraz podfoldery Symbols, Tokens, Interpreter.

W folderze Symbols znajdują się wszystkie symbole nieterminalne, każdy w oddzielnej klasie. Każdy symbol dziedziczy po klasie Symbol i implementuje wirtualną metodę execute(Env &env) oraz konstruktor.

Obiekty oparte są w głównej mierze na klasie std::unique_ptr. Wszystkie klasy mają alias KlasaUP, oznaczająca inteligentny wskaźnik na klasę.

12. Realizacja leksera

Lekser jest zrealizowany jako klasa, która zwraca kolejne tokeny (główna metoda getNextToken()). Korzysta ona z opakowania źródła znakowego Source (źródło plikowe lub CLI), które sprawia, że lekser nie jest „świadomy”, skąd pobiera znaki.

Główna metoda getNextToken() składa się z kolejnych prób utworzenia tokenu – słowa kluczowego, napisu, liczby całkowitej, zmiennoprzecinkowej, itp. Omija białe znaki. Metoda zwraca wskaźnik inteligentny na nowy token.

Metoda jest wywoływana przez parser (konkretnie przez metodę statyczną klasy Symbol), który traktuje tokeny jako wartości podstawowe i buduje z nich syntaktykę programu.

13. Realizacja parsera

Konstruktor każdego z symboli realizuje zadanie parsera. Klasa Parser wykonuje po prostu konstruktor symbolu głównego File (lub iteracyjnie symboli składowych FilePart), a każdy z symboli rekursywnie podejmuje próbę zbudowania się. Gdy nie udaje się zbudować symbolu, metoda klasy Symbol o nazwie buildToken o kilku wariantach, usuwa zaalokowane tokeny, tak aby móc je jeszcze raz wykorzystać.

Gdy udaje się zbudować wszystkie podsymbole danego symbolu, flaga constructed obiektu bazowego Symbol jest podnoszona, co sygnalizuje, że możemy korzystać z obiektu.

Konkretne obiekty stworzone w parserze według gramatyki przedstawionej powyżej będą później służyły budowie interpretera.

14. Interpreter

Interpreter nie ma swojej klasy, ale jest oparty o kilka klas funkcyjnych takich jak między innymi:

- Env – środowisko, lokalny zasięg, w którym przechowywane są zmienne. Obiekt pamięta swojego „rodzica" i może poszukiwać identyfikatorów w górę drzewa środowisk. Może być globalny, czyli najwyższy w drzewie. Tworzymy środowisko poprzez zwykły konstruktor ze wskazaniem rodzica lub nie.
- Obj – obiekt podstawowy, służący do przechowywania obiektów w tablicy haszującej środowiska. Dziedziczą po nim wszystkie symbole i tokeny.
- Wyjątki, takie jak BreakException czy ContinueException

Cała logika interpretera znajduje się w metodach execute() poszczególnych symboli. Wywołują one symbole podrzędne, wykonują logikę np. pętlę for, while czy if, liczą wartości czy zwracają wartości różnych typów.

15. Testowanie

Do kodu źródłowego dołączonych jest kilka przykładów w pliku examples.

16. Wnioski

Na pewno nie udało się zrealizować pełni zakładanej funkcjonalności. Wydaje mi się, że było tak z powodu, iż stworzenie nowego języka podobnego do Pythona czy C++, używanych powszechnie na świecie to nie lada wyzwanie, więc na stworzenie dodatkowych bibliotek czy przetestowanie wszystkich rozwiązań po prostu zabrakło czasu. Jest to mimo wszystko projekt semestralny, a mógłby także stanowić temat dalszych, pogłębionych badań i inżynierii.