Projekt wstępny

1. Opis funkcjonalny

Język w założeniu jest oparty na LOGO, języku do nauki programowania, w którym sterujemy żółwiem zostawiającym graficzny ślad swojej ścieżki. Mój język składniowo będzie oparty na aktualnie popularnych językach programowania Python i JavaScript. Oprócz przyjemniejszej składni, usprawnieniem względem LOGO będzie możliwość tworzenia wielu żółwi w jednym programie, o dowolnym kolorze oraz położeniu początkowym. Docelowo język będzie zintegrowany z graficznym interfejsem.

Język będzie obsługiwał: * instrukcję warunkową if - {elif} - [else] * pętlę for (w wariancie foreach) * operatory matematyczne: +, -, *, /, %, () * operatory logiczne: and, or, not, in, ==, !=, >, >=, <, <=, () * tworzenie własnych funkcji i wywoływanie (w tym rekurencyjne) * typy danych: turtle, color, position, orientation, int, string, boolean, null * wypisywanie danych na standardowe wyjście * operacje na żółwiach: idź do przodu, idź do tyłu, obróć się w prawo/lewo o podaną liczbę stopni (domyślnie 90), zmiana koloru i położenia * tworzenie list, dodawanie/usuwanie/odczytywanie elementów listy

2. Przykłady użycia

Kod	Działanie
`name_1 = turtle()`	Utworzenie żółwia z podstawowymi parametrami
`name_2 = turtle(color='red')`	Utworzenie czerwonego żółwia
`name_3 = turtle(color=color(0, 0, 255))`	Utworzenie niebieskiego żółwia (RGB)

`name_4 = turtle(position=(15, 20))`	Utworzenie żółwia na zadanej pozycji |
	`name_1.color = black`
	`name_1.color = color(100, 200, 50)`
	`name_1.position.x = 20`
	`name_1.position.y = -10`
	`name_1.orientation = right`
	`name_1.orientation = 90`
		Zmiana parametrów żółwia

`name_1.forward(10)`	Żółw 'name_1' idzie o 10 kroków naprzód
`name_1.backward(20)`	Żółw 'name_1' idzie o 20 kroków do tyłu
`name_1.turn_right()` `name_1.turn_left(15)`	Żółw 'name_1' skręca w prawo/lewo o 90^o (względem aktualnej orientacji)

`a = null`
`b = 10`
`c = "c"`
`d = "text"`
`f = true`
`g = false`
	Tworzenie zmiennych
	Domyślna wartość: null
	Dopuszczalne typy danych: null, integer, real number, character, string, boolean

`if(name_1.position.x > 50)`
`  name_1.turn_right()`
`elif(name_1.position.y < -15)`
`  name_1.turn_left()`
`elif(name_2.position.y >= 10)`
`  name_1.turn_right(20)`
`else`
`  print("else")`
	Złożony warunek

`if(name_1.color == red and for not g) {`
`  name_1.forward(2)`
`  name_2.backward(4)`
`}`
	Warunek z blokiem w klamrach

Kod	Działanie

`exampleList = [1, 2, 3, 'a', "text"]`
`exampleList.add(2)`
`exampleList.add("b")`
`exampleList.remove(0)`
`print(exampleList[2])`
	Utworzenie listy, dodanie elementu do listy (na koniec), usunięcie elementu spod podanego indeksu, odczytanie elementu spod podanego indeksu

`for(element in list) {`
`  print(element)`
`}`
	Wypisz każdy element z listy

`foo(param1, param2) => {`
`  i = 10`
`  return i + param1 * param2`
`}`
`bar() => {`
`  print(foo(50, 20))`
`}`
	Definicja i wywołanie funkcji 'foo'

`fun() => print("debug")`	Jednoliniowa funkcja nie wymaga klamer

3. Opis gramatyki

3.1. Konwecje leksykalne

```
literal      = string | number | "true" | "false" | "null";
string       = """, {char | escaping}, """ | "'", {char | escaping}, "'";
number       = digit | nonZeroDigit, {digit};
digit        = "0" | nonZeroDigit;
nonZeroDigit = "1" | ... | "9";
identifier   = letter, {letter | digit | "_"} | "_", {letter | digit | "_"};
comment      = "#", {char};
escaping     = "\", specialChar;
```

3.2. Gramatyka

```
program             = {functionDef};

functionDef         = identifier, "(", paramList, ")", "=>", block;
paramList           = [param, {",", param}];
param               = normalParam | defaultParam;
normalParam         = identifier;
defaultParam        = identifier, "=", literal;

block               = "{", {statement}, "}";
statement           = simpleStatement, ";"
                    | compoundStatement;
simpleStatement     = assignment
                    | functionCall
                    | returnStatement
                    | "pass";
compoundStatement   = ifStatement
                    | forStatement;

assignment          = identifier, assignmentOperator, expression;
assignmentOperator  = "="
                    | "+="
                    | "-="
                    | "*="
                    | "/="
                    | "%=";
```

```
functionCall        = identifier, {".", identifier}, "(", argList, ")";
argList             = [arg, {",", arg}];
arg                 = normalArg
                    | keywordArg;
normalArg           = expression;
keywordArg          = identifier, "=", expression;

returnStatement     = "return"
                    | "return", expression, {",", expression};

ifStatement         = "if", "(", expression, ")", (statement | block),
                      {elifBlock}, [elseBlock];
elifBlock           = "elif", "(", expression, ")", (statement | block);
elseBlock           = "else", (statement | block);

forStatement        = "for", "(", identifier, "in", expression, ")",
                      (statement | block);

expression          = expressionPart, {operator, expressionPart};
expressionPart      = literal
                    | identifier
                    | functionCall
                    | listDef
                    | listAccess
                    | "(", expression, ")";
listDef             = "[", [expression, {",", expression}], "]";
listAccess          = identifier, "[", number, "]"
operator            = mathOperator
                    | logicOperator;
mathOperator        = "+"
                    | "-"
                    | "*"
                    | "/"
                    | "//"
                    | "%";
logicOperator       = "and"
                    | "or"
                    | "in"
                    | "<"
                    | "<="
                    | ">"
                    | ">="
                    | "=="
                    | "!=";
```

4. Opis techniczny

4.1. Wymagania funkcjonalne

- Interpreter będzie się składał z analizatora leksykalnego i parsera.
- Lekser będzie odczytywał dane wejściowe (kod użytkownika) ze standardowego strumienia wejściowego, a parser będzie pobierał kolejne tokeny poprzez wywołanie odpowiedniej metody leksera (np. get_next_token()).
- Zadaniem leksera będzie wydobywanie z kodu kolejnych tekenów.
- Zadaniem parsera będzie pobieranie od leksera tokenów i składanie z nich kodu według gramatyki.
- Pobrane identyfikatory funkcji i zmiennych będą przechowywane w słowniku. Dzięki temu będzie można kontrolować niepowtarzalność.
-
- Program będzie uruchamiany przez interpreter Pythona w konsoli, czyli np. `python3 logo.py`. Konsola pozostanie otwarta jedynie do wyświetlania wartości podanej do funkcji `print`.
- Obiekty i zmienne będą przekazywane do funkcji jako referencja.
- Niedopuszczalne będzie tworzenie identyfikatorów o takiej samej nazwie jak słowo kluczowe języka lub istniejącej już identyfikator, przy czym najwyższy priorytet mają słowa kluczowe, a następnie nazwy funkcji. To znaczy, że zmienna nie może mieć takiej nazwy jak istniejąca funkcja. Duplikacja identyfikatorów będzie traktowana jako błąd już na etapie interpretacji.
- Po uruchomieniu programu, zostanie wyświetlony interfejs graficzny z podziałem na część do pisania kodu (prosty, ale przyjemny edytor) oraz na część wyświetlającą wykonanie skryptu.
- Będzie możliwość ukrycia jednej z tych części i wyświetlenie drugiej na pełnym oknie.
- Użytkownik będzie mógł utworzyć nowy skrypt, otworzyć skrypt z pliku na dysku i zapisać do pliku.
- Interfejs będzie miał listę rozwijaną do wyboru funkcji, od której rozpocznie się wykonanie programu.
- Obsługa błędów:
    - Błędy leksykalne/gramatyczne: program nie zostanie wykonany, na standardowe wyjści zostanie wypisany pierwszy napotkany błąd wraz z

stacktracem.
    - Błędy wykonania: program przerwie wykonanie w momencie napotkania błędu i wypisze typ błędu oraz stacktrace na standardowe wyjście.

4.2. Wymagania niefunkcjonalne

- Interpreter będzie napisany w języku Python w wersji 3.10.
- Do interfejsu graficznego wykorzystam bibliotekę PyQt5 w aktualnie najnowszej wersji.
- Docelowo aplikacja będzie mogła być uruchomiona na każdym systemie wspierającym Pythona w wybranej wersji i PyQt5. Minimum to Windows 10 i Ubuntu 20.04.

4.3. Rozpoznawane tokeny

```
 "true", "false", "null", """, "'", "#", ";",
"+", "-", "*", "/", "//", "%",
"and", "or", "in", "not", "<", "<=", ">", ">=", "==", "!=",
"(", ")", "{", "}", "[", "]", ",", ".", "=>",
"=", "+=", "-=", "*=", "/=", "%=",
"if", "elif", "else", "for", "return", "pass"
```

5. Opis testowania

5.1. Testowanie leksera

Testy jednostkowe sprawdzające wykrycie białych znaków, tokenów, odczytywanie różnych źródeł kodu (np. plik, tekst wpisany bezpośrednio do kodu), funkcję zwracającą kolejne tokeny

5.2. Testowanie parsera

Testy jednostkowe sprawdzające parsowanie przykładów wszystkich struktur, włączając w to przypadki rzadkie.

5.3. Testowanie finalnej aplikacji z GUI

Testy manualne polegające na sprawdzeniu wyniku przygotowanych skryptów.