Dokumentacja

Temat: Interpreter obiektowego Logo

Konrad Opaliński

1. Opis projektu

Celem projektu jest stworzenie interpretera obiektowego logo. Ideę działania można porównać do znanej logomocji, lecz składnia języka będzie bardziej zbliżona do obecnych języków obiektowych. Dzięki temu możliwe będzie np. stworzenie więcej niż jednego żółwia, czy przypisanie mu atrybutów (np. kolor/grubość linii oraz tego, czy ma aktualnie rysować).

2. Opis języka za pomocą EBNF

digit := ‘0’ | digit_ex_zero ;

digit_ex_zero := ‘1’ .. ‘9’ ;

natural := ‘0’ | digit_ex_zero, {digit} ;

natural_ex_zero := digit_ex_zero, {digit} ;

num_val := ‘0’ | natural_ex_zero, [‘.’, natural] ;

all_visible_chars := ? all visible characters ? ;

char := ‘A’ .. ‘Z’ | ‘a’ ... ‘z’ ;

str_val := ‘\”’, {all_visible_chars - ‘\”}, ‘\”’ ;

var_value := num_val | str_val ;

type := ‘num’ | ‘str’ | ‘Turtle’ ;

identifier := char, {char | digit} ;

var_declaration := type, identifier ;

equal_oper := ‘==’ | ‘!=’ ;

relational_oper := ‘<’ | ‘>’ | ‘<=’ | ‘>=’ ;

add_oper := ‘+’ | ‘-‘ ;

mult_oper := ‘*’ | ‘/’ ;

expression := mult_expr, {add_op, mult_expr } ;

mult_expr := param_expr, {mult_op, param_expr } ;

param_expr:= [unary], (‘(‘, expression, ‘)’ | var_value | identifier, [‘(‘, args, ‘)’]) ;

unary := ‘-’ ;

parameters := [var_declaration , {‘,’, var_declaration}] ;

arg := [expression] ;

args := [expression {‘,’, expression}] ;

func_definition := ‘func’, identifier, ‘(‘, parameters, ‘)’, block_statement ;

assign_or_func_method_call := identifier, (‘(‘, args, ‘)’ | ‘.’, identifier, ‘(‘, arg, ‘)’ | ‘=’, expression ;

program := {statement | func_definition} ;

block_statement := ‘{‘, {statement}, ‘}’ ;

statement := ‘if’, ‘(‘, expression, ‘)’, block_statement, ‘else’, block_statement

| ‘repeat’, ‘(‘, expression, ‘)’, block_statement

| var_declaration

| assign_or_func_method_call

| ‘return’, expression ;

3. Dodatkowy opis składni języka oraz typu wbudowanego

Język posiada podstawowe formy reprezentowania danych tekstowych jak i liczbowych. Służą do tego odpowiednio zmienne typu str oraz num.

Poza zmiennymi typu wartościowego, istnieje również wbudowany typ referencyjny żółwia (Turtle). Jest to klasa, której instancje można tworzyć oraz nimi sterować za pomocą wbudowanych metod. Należą do nich:

- Forward(num)
  Przesuwa żółwia do przodu o wartość podaną jako parametr wywołania.

- Backward(num)
  Przesuwa żółwia do tyłu o wartość podaną jako parametr wywołania.

- Right(num)
  Obraca żółwia zgodnie ze wskazówkami zegara o kąt podany jako parametr wywołania.

- Left(num)
  Obraca żółwia przeciwnie do ruchu wskazówek zegara o kąt podany jako parametr wywołania.

- PenUp()
  “Podnosi” żółwia. Od teraz wszystkie inne wbudowane metody będą wykonywane w ten sposób, że nie będzie widoczny “ślad” ruchu żółwia (linia, którą rysuje).

- PenDown()
  Przywraca podstawowy stan żółwia po wywołaniu PenUp().

- LineColor(str)
  Zmienia kolor linii, którą żółw będzie rysował podczas przemieszczania się po płótnie. Kolor zostaje zmieniony na ten podany jako parametr wywołania, możliwe opcje to: “Black”, “Red”, “Green”, “Blue”.

- LineThickness(num)
  Zmienia grubość linii, którą żółw będzie rysował podczas przemieszczania się po płótnie. Grubość zostaje zmieniona na tą podaną jako parametr wywołania, możliwe opcje należą do zakresu od 1 do 5.

Warto tutaj wspomnieć, że ruchy, które wykonał żółw nie znikną z płótna pomimo tego, że straciło się do niego referencję. Przykładem tego może być utworzenie obiektu żółwia w ciele funkcji, wszystkie ruchy jakie wykona podczas swojego “żywotu” zostaną naniesione na płótno, natomiast po zakończeniu funkcji nie będzie już do niego dostępu.

4. Sposób uruchomienia, we/wy

Do obsługi interpretera zostanie stworzona prosta aplikacja (IDE) zawierająca miejsce do pisania kodu (CE - Code Editor) oraz płótno, na którym będą wyświetlane efekty napisanego programu.

Uruchomienie polega na naciśnięciu odpowiedniego przycisku rozpoczynającego działanie interpretera.

Wejściem dla interpretera jest kod napisany przez użytkownika w CE.

Wyjściem interpretera jest rysunek na płótnie na podstawie wejścia (kodu).
* Obsługa błędów -> patrz poniżej.

5. Moduły

Projekt jest podzielony na 3 główne moduły: Lexer, Parser, Executor. Moduły te są odpowiedzialne za:

1. **Lexer**
    - Odczytywanie tekstu ze strumienia danych wejściowych, w tym przypadku z CE
    - Tworzenie odpowiednich tokenów

    Odczytuje znak po znaku ze źródła i próbuje dopasować sekwencję znaków do istniejących tokenów w języku. Moduł ustawia atrybut Token na aktualnie znaleziony przez niego token przy każdym wywołaniu odpowiedniej metody. Dzięki temu nie musi przetwarzać całego tekstu na raz. W przypadku wystąpienie nieznanego mu znaku, zwróci odpowiednią informację użytkownikowi.

2. **Parser**
    - Grupuje tokeny otrzymane od Lexera w struktury składniowe (drzewa AST) zgodne z gramatyką języka

    Prosi Lexer o kolejne tokeny ze źródła oraz próbuje je dopasować do możliwych konstrukcji językowych. W przypadku powodzenia, dodaje do programu odpowiednie przez siebie stworzone drzewo składniowe. W przypadku niepowodzenia (konstrukcja sprzeczna/nieistniejąca w języku) poinformuje o tym użytkownika.

3. **Executor**
    - Zajmuje się wykonaniem struktur składniowych powstałych w wyniku działania Parsera

    Zajmuje się wykonywaniem drzew składniowych dostarczonych mu przez Parser w ramach programu. Wykonuje odpowiednie obliczenia oraz rysuje na płótnie. W przypadku wystąpienia błędu wykonania, np. dzielenia przez 0, poinformuje o tym użytkownika odpowiednim błędem.

6. Obsługa błędów

W przypadku, gdy program będzie zawierał błędy, interpretacja zakończy się, a informacja o błędzie zostanie zgłoszona użytkownikowi.

W zależności, na którym etapie interpretacji zostanie wykryty błąd, wówczas użytkownik dostanie odpowiednią informację o nim.

Przykładowe błędy związane z poszczególnymi modułami opisanymi powyżej:

1. Lexer

Znak, który nie występuje w składni języka:

```
num #
```

Zostanie zwrócony błąd:

Exception (Lexer): Didn't find appropriate token for text in position: col = 5, ln = 1, ch = 5

2. Parser

Brak warunku wykonania instrukcji warunkowej "if":

```
if () {}
```

Zostanie zwrócony błąd:

Exception (Parser): Expected condition in position: col = 5, ln = 1, ch = 5

3. Executor

Wywołanie funkcji, której definicja nigdzie w kodzie się nie znajduje:

```
foo()
```

Zostanie zwrócony błąd:

Exception (Executor): Did not find function called foo

7. Testowanie

Moduł Lexer posiada napisane testy jednostkowe sprawdzające rozpoznawane przez niego tokeny w tekście.

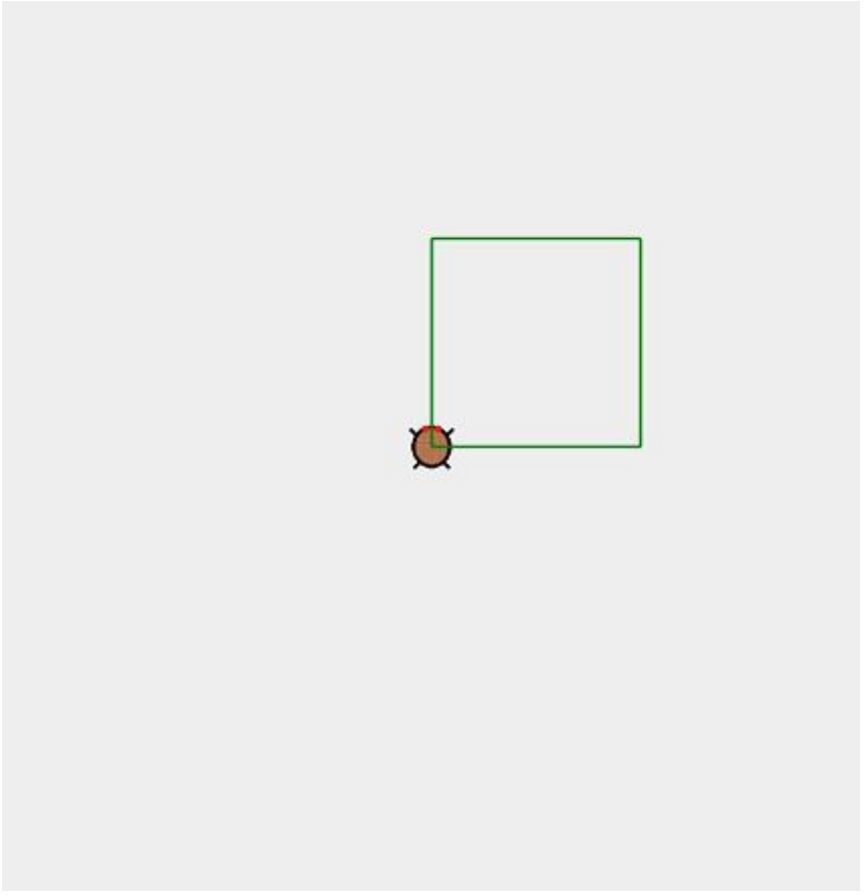
Moduł Parsera posiada napisane testy sprawdzające poprawność utworzonego drzewa składniowego. Testy polegają na odtworzeniu kodu ze stworzonego przez Parser drzewa składniowego.

Po pomyślnych wynikach testów, działanie interpretera zostało sprawdzone na dużej liczbie przykładowych kodów, w których oczekiwany wynik był porównywany z tym otrzymanym na płótnie.

8. Przykładowe konstrukcje

Poniżej przedstawiono dwie poprawne konstrukcje językowe oraz rezultaty ich wykonania.

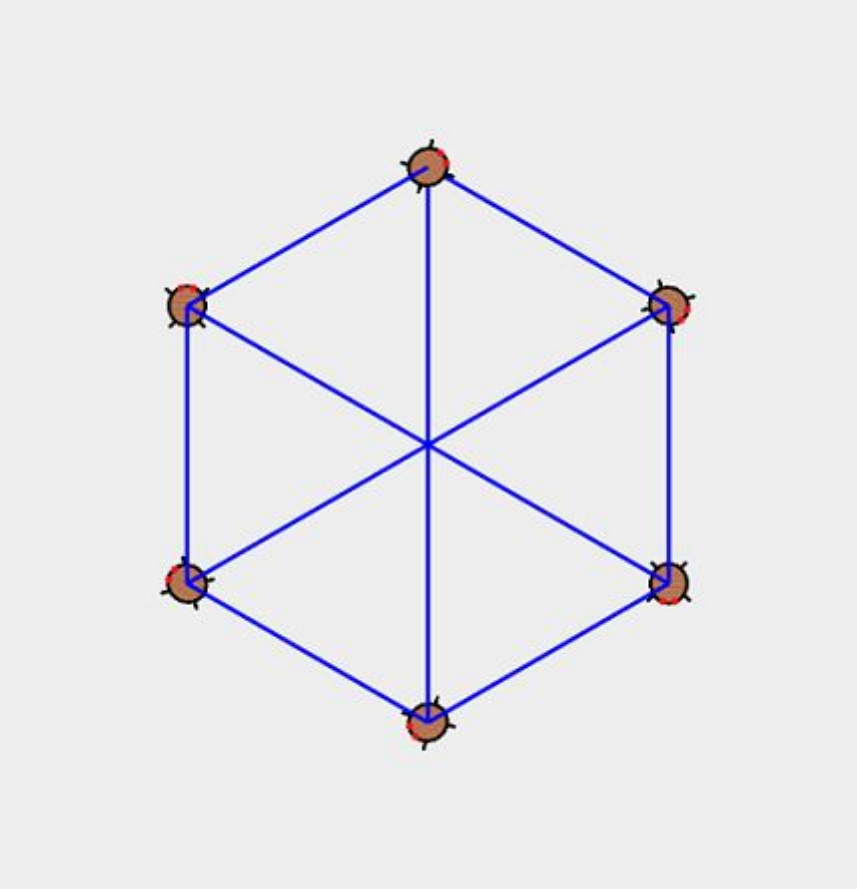
- Kwadrat



```
func square(Turtle obj, num len, str color, num thick)
{
	obj.LineColor(color)
	obj.LineThickness(thick)

	repeat(4)
	{
		obj.Forward(len)
```

```
                obj.Right(90)
        }
}

Turtle zenek


str color
color = "Green"

num thickness
thickness = 2

square(zenek, 150, color, thickness)
```

- Sześciokąt



```
func partialHex(num angle, num length, str color)
{
        Turtle turtle
        turtle.LineThickness(3)
        turtle.LineColor(color)
        turtle.Right(angle)
        turtle.Forward(length)
        turtle.Right(120)
```

```
	turtle.Forward(length)
}

func hexagon(num length, str color)
{
	num angle
	angle = 0

	repeat(6)
	{
		partialHex(angle, length, color)
		angle = angle + 60
	}
}

hexagon(200, "Blue")
```