

Dokumentacja wstępna

Interpreter prostego języka z wbudowanym
typem do obsługi grafiki wektorowej

Łukasz Wolanin

21 marca 2020

1 Opis funkcjonalny

Celem projektu jest stworzenie interpretera prostego języka przypominającego C, który będzie wyposażony w podstawowe instrukcje sterujące (instrukcje if, else i while). Język ten będzie umożliwiał definiowanie funkcji przez użytkownika oraz będzie posiadał wbudowany typ do obsługi grafiki wektorowej 2D. Język będzie zachowywał priorytety operatorów arytmetycznych, a zmienne będą miały zasięgi. Język będzie posiadał wbudowane funkcje do obsługi typu graficznego takie jak rysowanie linii, koła, itd. skalowanie ich, rotację itp.

1.1 Wymagania funkcjonalne

1. Możliwość deklarowania zmiennych typów int i string
2. Możliwość deklarowania zmiennych typu specjalnego: graphic
3. Możliwość definiowania swoich kolorów za pomocą typu: color
4. Możliwość deklarowania własnych funkcji przyjmujących argumenty poprzez wartość, oraz zwracających wartość o zadanym typie
5. Poprawne obliczanie wyrażeń arytmetycznych z zachowaniem kolejności operatorów
6. Język będzie umożliwiał wykorzystanie instrukcji warunkowej if - else
7. Język będzie udostępniał pętlę while
8. Język będzie obliczał wartość wyrażeń logicznych zgodnie z zasadami logiki
9. W wyrażeniach logicznych można używać operatorów porównania
10. Operowanie typem specjalnym za pomocą wbudowanych funkcji
 - funkcje to rysowania linii, okręgu, wielokątów
 - funkcje do wypełniania kolorem
 - funkcje do przemieszczania, skalowania, oraz obracania
11. Możliwość umieszczania komentarzy w kodzie
12. Język powinien umożliwiać wyświetlanie lub zapisywanie wygenerowanego obrazu
13. Język nie będzie umożliwiał tworzenia zmiennych alokowanych dynamicznie
14. Każdy program musi zawierać funkcję main
15. Język powinien ułatwiać operację na obiektach graficznych przechowywanych w postaci wektorowej

16. W wyrażeniach logicznych typ `int` jest konwertowany na wartość logiczną jak w języku C
17. Typy inne niż `int` nie mają swojej wartości logicznej
18. Położenie centrum każdego obiektu jest podawane względem środka obiektu nadrzędnego
19. Uruchomienie programu z niepoprawnymi parametrami powinno informować użytkownika o możliwych parametrach startowych

1.2 Wymagania нефunkcjonalne

1. Każdy obiekt typu graficznego będzie posiadać parametry jak:
 - wysokość i szerokość
 - pozycja, rotacja i skala (transform)
2. Wielkość pliku źródłowego nie będzie ograniczona przez interpreter
3. Komunikaty o błędach analizy plików powinny być przejrzyste i ułatwiać wyszukanie błędu
4. Program powinien działać w środowisku Windows, ale też łatwo dać się przenieść do środowiska Linuxowego

1.3 Przykłady kodu

Przykład całego programu

```
int main()
{
    graphic root = blank(800, 600);
    color red = rgb(255, 0, 0);
    //funkcja blank(int x, int y) zwraca czysty obiekt o zadanych rozmiarach

    graphic circle = circle(50);
    fill(circle, red);
    graphic triangle = triangle(80, 100, 60);
    //utworzenie kola o promieniu 50 i trojkata prostokatnego o bokach 80, 100, 60

    string in = "TAK";

    if(in == "TAK")
    {
        circle = move(circle, 200, 0); //funkcja move(x, y) zmienia polozenie obiektu
        triangle = move(triangle, -200, 0);
    }
}
```

```

else
{
circle = move(circle, -200, 0);
triangle = move(triangle, 200, 0);
}

root = add(root, circle); // dodanie obiektow do roota i stworzenie zaleznosci
root = add(root, triangle);

save(root, "sample.bmp") //zapisanie rusunku w formacie bmp

return 0;
}

```

Przykład definicji funkcji

```

graphic drawArrow()
{
graphic arrow = triangle(60,80,100);
graphic arrow2 = arrow;
arrow2 = rotate(arrow2, 180);
arrow = add(arrow, arrow2);
arrow = scale(arrow, 0.5, 1.0);
graphic arrow3 = triangle(60, 60, 60);
arrow3 = move(arrow3, 50);
arrow = add(arrow, arrow3);
return arrow;
}

```

Przykład wykorzystania pętli while i zdefiniowanej funkcji

```

graphic root = blank(800, 600);
int i = 0;
while(i < 5)
{
graphic arrow = drawArrow();
arrow = move(arrow, -300 + 80*i, 0);
root = add(root, arrow);
i = i + 1;
}

```

2 Formalny opis gramatyki

```

program = {function_def};
function_def = data_type, identifier, call_def, body;
call_def = "(", [data_type, identifier, { ",", data_type, identifier }], ")";
call_operator = "(", [(identifier|literal), { ",", (identifier|literal) }], ")";
body = "{", {instruction}, "}";
instruction = if_statement | while_loop | assign_statement |
    init_statement | function_exec | return_statement | "brake;";
if_statement = "if", "(", condition, ")", (body | instruction), [ "else",
    (body | instruction)];
while_loop = "while", "(", condition, ")", (body | instruction);
assign_statement = identifier, "=", right_val, ";";
init_statement = data_type, identifier, { ",", identifier },
    ["=", right_val, { ",", right_val }], ";";
function_exec = function_call, ";";
return_statement = "return", right_val, ";";
right_val = literal | identifier | function_call | expression;
expression = multiplicative_expr, { ("+" | "-"), multiplicative_expr };
multiplicative_expr = primary_expr, { ("*" | "/"), primary_expr };
primary_expr = ( literal | identifier | bracket_expr );
bracket_expr = "(" expression ")";
condition = and_cond, { "||", and_cond };
and_cond = equal_cond, {"&&", equal_cond };
equal_cond = relation_cond, { ("==" | "!=" ), relation_cond };
relation_cond = primary_cond, { relation_oper, primary_cond };
primary_cond = ["!"], ( braces_cond | right_val );
braces_cond = "(", condition, ")";
function_call = identifier, call_operator;
data_type = "int" | "string" | "graphic" | "color";
identifier = (letter|"_"){letter|digit|"_"};
literal = string_literal | integer_literal;
string_literal = "'", {?visible character? | ?white space? }, "'";
integer_literal = "0" | ["-"], non_zero_digit, {digit};
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
    "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" | "B" |
    "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
    "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";
non_zero_digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
digit = "0" | non_zero_digit;
relation_oper = "<" | ">" | "<=" | ">=";

```

3 Opis techniczny realizacji

Projekt zostanie zrealizowany w języku C++ w środowisku MS Visual Studio 2019. Zamierzam wykorzystać bibliotekę boost między innymi do testowania. Główną platformą docelową będzie Windows, ale planuję też uruchomić napisany interpreter na Linuxie (kompilacja GCC i wykorzystanie narzędzia Cmake do konfiguracji).

3.1 Definicja produktu końcowego

Interpreter, który będzie wynikiem projektu, będzie aplikacją konsolową przyjmującą jako argumenty ścieżkę do pliku z kodem w wyżej zdefiniowanym języku, który będzie interpretowany, a informacje o wyniku poszczególnych etapów analizy oraz wyjście programu będą wyświetlane na konsoli. Interpreter będzie także powiadamiał użytkownika o błędach wykrytych w przekazanym do interpretacji skrypcie.

3.2 Budowa interpretera

Na interpreter będą składać się moduły odpowiedzialne za poszczególne etapy procesu interpretacji

Analiza leksykalna Scanner będzie odpowiedzialny za generację tokenów z ciągu znaków jaki otrzyma na wejściu. Osobny moduł będzie odpowiedzialny za stworzenie abstrakcji strumienia danych przekazywanych do scannera tak aby nie uzależniać go od konkretnej implementacji źródła kodu. Scanner będzie połączony z modułem obsługi błędów, oraz innymi pomocnymi strukturami jak tablica tokenów. Wygenerowane tokeny będzie na bieżąco przekazywał do parsera.

Analiza składniowa Parser będzie na wejściu otrzymywał tokeny przekazane przez scanner. Następnie będzie budował z tych tokenów drzewo składniowe jednocześnie sprawdzając czy wszystkie tokeny są ułożone zgodnie ze zdefiniowaną gramatyką języka. Moduł parsera również będzie połączony z modułem obsługi błędów.

Analiza semantyczna Analizator semantyczny sprawdza to czego nie sprawdzono w poprzednich etapach. Będzie sprawdzał poprawność używanych identyfikatorów, zgodność kontekstów operacji logicznych, zgodność kontekstów operacji arytmetycznych, zgodność ilości parametrów wywołań funkcyjnych, brak nadpisywania identyfikatorów funkcji. Moduł będzie miał połączenie z modułem obsługi błędów.

Wykonanie kodu To co zwróci analizator semantyczny zostanie przekazane do modułu wykonawczego. Moduł wykonawczy ma za zadanie sekwencyjne wykonanie instrukcji zawartych w drzewie wytworzonym na poprzednich etapach.

3.3 Testowanie

Zostanie napisany zestaw testów jednostkowych sprawdzających poprawność działania poszczególnych modułów programu. Do tego celu zostanie użyta biblioteka boost.