Skryptomania - analiza funkcjonalna

Karol Kubicki

March 2025

Spis treści

1 Szczegóły implementacyjne

- Kompilator: MSVC, Clang;
- Standard: C++23;
- Kontrola wersji: Git;
- Środowisko: Visual Studio 22, CMake.

2 Analiza funkcjonalna

Na potrzeby utworzenia języka skryptowego, powstaną 4 "systemy": analizator leksykalny, parser, interpreter, środowisko użytkownika.

Analizator leksykalny jest odpowiedzialny za rozpoznanie fragmentów tekstu kodu źródłowego i przetworzenie go na ciąg symboli.

Parser, na podstawie opisanej gramatyki języka, połączy tokeny w wyrażenia gramatyczne, tworząc z nich hierarchię nazywaną abstrakcyjnym drzewem syntaktycznym (AST).

Interpreter przechodzi przez AST, faktycznie wykonując zapisane instrukcje wyrażeniami wewnątrz drzewa. Utrzymuje też pamięć alokowaną przez program.

Środowisko użytkownika najprawdopodobniej wystarczy, aby było w konsoli, ponieważ jest to jedynie język skryptowy. Jeśli czas pozwoli, i Prowadzący wyrazi taki wymóg, jestem skłonny dodać interfejs w ImGUI i OpenGL 4.6, natomiast uważam, że nie wniesie to nic szczególnego do projektu. Oprócz wykonywania, i potencjalnego "debuggowania"programu, można również wyświetlić wydruk AST. Środowisko pozwala na wykonanie kodu zawartego w zewnętrznym pliku - np.: lang -i code.txt. Do wiadomości użytkownika wyświetlone zostaną ewentualne błędy. Mógłbym spróbować, jeśli czas pozowli, dodać wykonywanie kodu linijka po linijce na – uproszczony – wzór programu gdb.

Przykładowy kod:

```
func f1(a, b, c) { a = b = c; return a + b * ++c; }
func f3() { return 1.0; }
func f2(a) { return a, a*a, f3(); }
a = 1;
b = 2.0;
c = "tekst";
a = f1(a, f2(b)[1], a + b / a);

if(a > 15) { print("a>15"); }
else {
    x = 0;
    for(i = 0; i < 3; ++i) {
        x = ++x * x;
        switch(i) {
            case 2: { print("i: ", i); }
        }
    }
    a = x;
}
print("a: ", a);
```

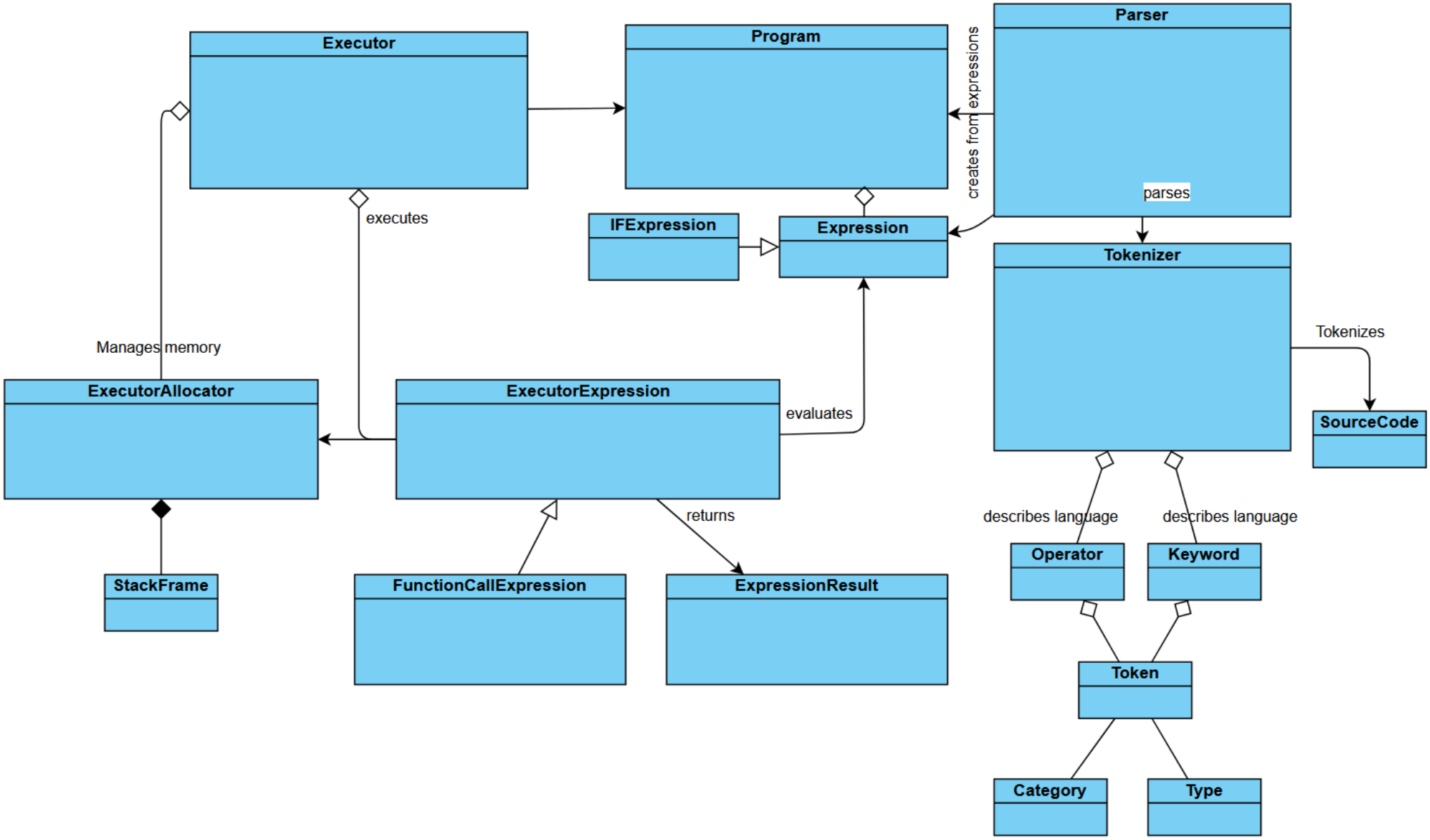
Etap II

Karol Kubicki, 319060
Projekt: Skryptomania

1. Opis klas

- **SourceCode** zawiera pojedyncze znaki kodu źródłowego programu z przeczytanego pliku.
- **Tokenizer** przetwarza SourceCode znak po znaku, tworząc w tokeny typu „Identifier”, „Operator”, Keyword”.
- **Token** reprezentuje rudymentarny składnik języka.
- **Category** i **Type** opisują rodzaj tokenu.
- **Parser** składa jeden lub więcej tokenów w wyrażenia Expression.
- **Expression** jest wyrażeniem o określonej semantyce – np. wyrażenie warunkowe „if”.
- **Program** jest ciągiem wyrażeń.
- **Executor** jest interpreterem, który potrafi wykonywać program wyrażenie po wyrażeniu – to z nim użytkownik wchodzi w interakcję i to jest środowisko wykonawczę, posiadające takie metody jak np. „ExecuteNext”, „GetAllFunctions”, „GetAllVariables”.
- **ExecutorExpression** jest klasą abstrakcyjną z metodą (w C++ funkcją członkowską) wirtualną „eval”, którą wyrażenia specjalizujące przeładowują, aby zapewnić interfejs do wykonywania faktycznych wyrażeń jak np. wyrażenie wywołania funkcji.
- **ExpressionResult** jest obiektem zawierającym rezultat wykonania wyrażenia ExecutorExpression;
- **ExecutorAllocator** jest stanem programu – to w nim przechowywane są wszystkie utworzone zaalokowane zmienne.
- **StackFrame** jest ramką stosu; tworzona jest przy wejściu i niszczona po wyjściu z obszarów o własnym zakresie, czyli takich, poza którymi zmienne wewnątrz nich są niedostępne. Przechowuje również stan aktualnego procesu wykonującego, czyli czy warunek w wyrażeniu warunkowych jest prawdziwy lub czy zostało napotkane wyrażenie powrotu (return) i nie należy dalej wykonywać kodu.

2. Diagram klas



Etap III
Projekt: Skryptomania

Karol Kubicki, 319060

Link do repozytorium: https://github.com/KKarol01/lang

1 Raport

Program kompiluje się z użyciem programu Cmake, który może zostać zintegrowany z Visual Studio. Visual Studio uruchamiamy w folderze głównym z kodem źródłowym, po czym wystarczy nacisnąć F5, aby wszystko się skompilowało i uruchomiło. Kompilacja w trybie DEBUG dodaje dodatkowo wypis abstrakcyjnego drzewa syntaktycznego.

Język skryptowy wspiera konstrukty takie jak: PrimaryExpression, PostfixExpression, UnaryExpression, AddExpression, SubExpression, MulExpression, DivExpression, AssignExpression, FuncDeclExpression, FuncCallExpression, ExprListExpression, ReturnStmntExpression, BreakStmntExpression, IfStmntExpression, ForStmntExpression, PrintStmnt, LogicalOpExpression, LogicalCompExpression.

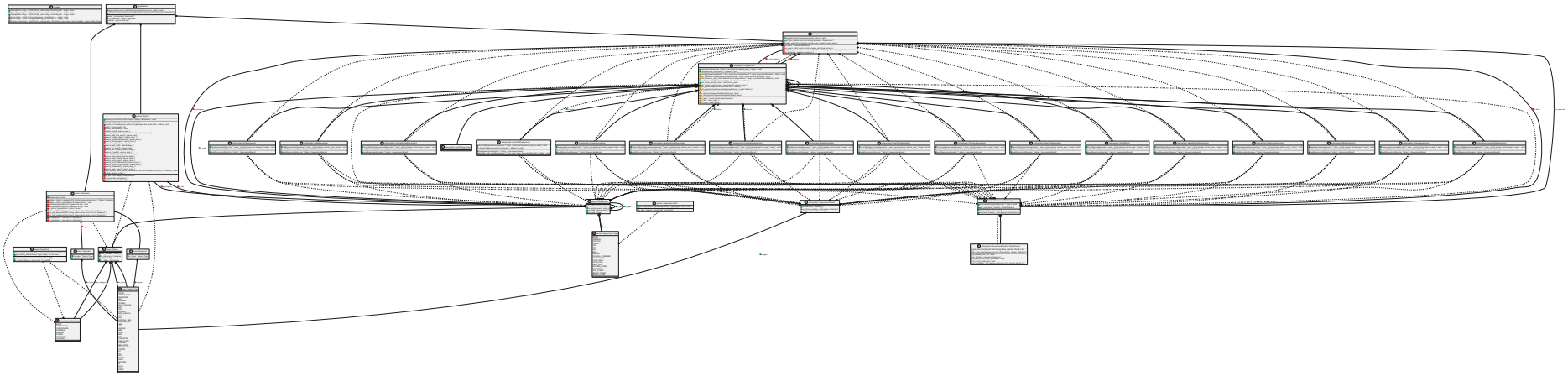
Program uruchamia się z konsoli poleceń w następujący sposób: ./scriptlang.exe sciezka/do/pliku. Przykładowy kod znajduję się w folderze „assets" i jest kopiowany automatycznie do folderu z wynikowym plikiem wykonywalnym. Przykładowy kod źródłowy zawiera celowy błąd mnożenia napisu przez liczbę dla zaprezentowania możliwości zgłaszania błędów.

Kod programu podzielony jest na 4 oddzielne moduły: analizator leksykalny, parser, interpreter i środowiko użytkownika (w kodzie środowisko odpowiada klasie App).

Program jest w całości programem konsolowym.

Język posiada funkcje bez, jedno i wieloargumentowe. Funkcje mogą zwracać wiele wartości. Są instrukcję if oraz for, przy czym dla fora jest instrukcja break, a dla funkcji return. W forze można deklarować wiele zmiennych (bez przecinka). Jest to język dynamiczny i dedukcja typu zmiennej następuje podczas przypisywania do niej wartości. W czasie działania programu, zmienna może zmieniać swój typ wielokrotnie. Zmienne można wypisywać na ekran za pomocą wieloargumentowej funkcji print(). Kompilator potrafi pokazywać błędy składniowe oraz logiczne (podczas działania programu) z zaznaczoną linią i dopiskiem, co się dzieje i czego brakuje. Brakuje w nim tablic, ale do zadań obliczeniowych bez nich jest odpowiedni – potrafi obliczyć liczbę Fibonacciego. Możliwym byłoby również obrót trójwymiarowego punktu za pomocą macierzy lub osi i kątu obrotu. Język również posiada ramki stosu do rekursywnych lub zagnieżdżonych wywołań funkcji bądź instrukcji for.

2 Zaktualizowany diagram klas



Rysunek 1: Diagram klas