Raport z Implementacji Interpretera Języka ArchiCode

1. Cel projektu

Celem projektu było stworzenie interaktywnego interpretera języka ArchiCode, który pozwala na:

- definiowanie i wykonywanie funkcji (blueprintów),
- operacje arytmetyczne, logiczne, warunkowe i pętle,
- dynamiczne zarządzanie zmiennymi w różnych zakresach,
- obsługę błędów wykonania i składniowych ze śledzeniem stosu wywołań,
- prosty system typów: `int`, `float`, `bool`, `char`, `string` z inferencją i rzutowaniem.

Interpreter został zaimplementowany w języku Java przy użyciu ANTLR4 jako generatora parsera.

---

2. Filozofia i Decyzje Projektowe

Język ArchiCode został zaprojektowany z myślą o czytelności, czerpiąc z terminologii architektonicznej, gdzie kod ma przypominać techniczny plan.

- **Metafora architektoniczna:** Słowa kluczowe jak `blueprint`, `Core` czy `define` mają za zadanie budować mentalny model programu jako projektu konstrukcyjnego.
- **Model wykonania:** Kluczową decyzją było przyjęcie modelu, w którym **każde wywołanie funkcji (blueprint) tworzy własny, odizolowany kontekst wykonania** (nowy obiekt `ArchiCodeVisitorImpl` i `Memory`). Ten kontekst współdzieli jedynie globalny zakres zmiennych i definicje funkcji, co upraszcza zarządzanie stanem i zapobiega niepożądanym efektom ubocznym między wywołaniami.
- **Przeciążanie funkcji:** Język wspiera przeciążanie blueprintów. Dwie funkcje mogą mieć tę samą nazwę, o ile różnią się sygnaturą (liczbą i/lub typami parametrów).

---

3. Liczba i cel przebiegów interpretera

Interpreter ArchiCode wykorzystuje dwa główne przebiegi (etapy) przetwarzania programu:

1. **Pierwszy przebieg – analiza strukturalna (ParseTreeWalker + Listener):**
   W tej fazie drzewo parsowania przekazywane jest do `ArchiCodeListenerImpl`, co umożliwia

wczesne wykrywanie błędów (np. definicji funkcji).

2. **Drugi przebieg – wykonanie programu (Visitor):**
   Właściwe wykonanie programu odbywa się za pomocą `ArchiCodeVisitorImpl`, który odwiedza węzły AST (Abstrakcyjnego Drzewa Składni).

Model ten odpowiada klasycznemu podejściu "walk + eval", znanemu z wielu języków interpretowanych.

---

4. Implementacja tablicy symboli (Symbol Table)

Tablica symboli zarządzana jest przez klasę `Memory`, która operuje na stosie zakresów (`Stack<HashMap<...>>`). Każdy blok `{...}` oraz każda funkcja (blueprint) wprowadza nowy kontekst (scope), dzięki czemu zmienne są poprawnie izolowane.

Mechanizm:

- **Tworzenie nowego zakresu:** `memory.newScope()` dodaje nową pustą mapę na stos.
- **Usuwanie zakresu:** `memory.endScope()` usuwa bieżący zakres.
- **Tworzenie zmiennych:** `createVariable()` dodaje zmienną do aktualnego zakresu.
- **Odczyt zmiennej:** `resolveVariable()` przeszukuje stos od góry w dół, aż znajdzie zmienną.
- **Nadpisywanie wartości:** `assignValue()` nadpisuje wartość, jeśli typy są zgodne.

Jest to klasyczny, dynamiczny mechanizm zakresów oparty na stosie, znany z języków takich jak Python czy JavaScript.

---

5. Rekordy aktywacji (ramki stosu)

Każde wywołanie funkcji (blueprint) tworzy nową, izolowaną ramkę stosu wykonania, odwzorowaną przez:

- nowy obiekt `Memory`, dziedziczący globalne zmienne i funkcje,
- nowy obiekt `ArchiCodeVisitorImpl`, który operuje w tym kontekście,
- dedykowany zakres lokalny (via `newScope()`), w którym rejestrowane są parametry i zmienne lokalne.

Interpreter utrzymuje:

- **stos nazw wywołań funkcji (FunctionCallStack)** w celu raportowania błędów i diagnostyki,
- **licznik rekurencji (RecursionCounter)**, który ogranicza głębokość wywołań do 30 – zabezpieczenie przed przepełnieniem stosu.

Dzięki temu każde wywołanie funkcji działa w pełni izolowanie, a po jego zakończeniu pamięć lokalna jest zwalniana.

---

6. Gramatyka języka (ANTLR4)

Parser języka ArchiCode oparty jest na gramatyce zdefiniowanej w pliku `ArchiCode.g4`. Definiuje ona konstrukcje takie jak pętle, warunki, definicje zmiennych i funkcji.

```
// Fragment gramatyki dla kluczowych instrukcji
grammar ArchiCode;

program: statement+ EOF;

statement
    : defineStatement
    | assignStatement
    | repeatStatement
    | checkStatement
    | blueprintStatement
    | blueprintCallStatement
    | coreStatement
    ;

// ...reszta gramatyki...
```

7. Struktura i Kluczowe Komponenty

- **ArchiCodeVisitorImpl.java** – Serce interpretera. Implementuje logikę dla każdego węzła drzewa AST.
- **Memory.java** – System zarządzania pamięcią, zakresami i funkcjami.
- **Value.java (i podklasy)** – Polimorficzna reprezentacja wartości.
- **Blueprint.java** - Obiekt reprezentujący zdefiniowaną funkcję.

---

8. Główne Mechanizmy Działania

- **Zarządzanie zakresem:** Każdy blok kodu `{...}` tworzy nowy zakres przez wywołanie `memory.newScope()`.
- **System typów, rzutowanie i inferencja:** Typ zmiennej może być jawny lub wywnioskowany. Przy przypisaniach i przekazywaniu parametrów typy są sprawdzane i konwertowane.

- **Wywołanie funkcji (blueprint):** Proces ten obejmuje generowanie sygnatury, wyszukanie funkcji, tworzenie izolowanego kontekstu wykonania, wiązanie parametrów i zwrot wartości.
- **Obsługa pętli:** Pętle `repeat` automatycznie tworzą i zarządzają zmienną o nazwie `step`.

---

9. Obsługa Błędów

- **Błędy składniowe:** Obsługiwane przez `ArchiCodeSyntaxErrorListener`.
- **Błędy semantyczne i wykonania:** Wzbogacane o numer linii, kolumny, a także stos wywołań funkcji (`FunctionCallStack`). Wprowadzono limit rekurencji, aby zapobiec zawieszeniu interpretera.

---

10. Przykładowy kod źródłowy w języku ArchiCode

```
blueprint Silnia(int a) delivers float result = 1 {
    repeat 2 a + 1 {
        result = result * step
    }
    show "factorial(" a ") = " result
}

Core() delivers int coreResult {
    define float x = 0
    show "Podaj liczbę:"
    request x
    Silnia( (int)x ) // Wywołanie z jawnym rzutowaniem
}
```