Raport techniczny z wykonania projektu

Interpreter języka Pseudo

Maciej Jamroży, Robert Jacak, Kacper Gałek

czerwiec 2025

Spis treści

1 Cel i założenia projektu **3**

2 Diagram klas **4**

3 Przebiegowość interpretera PseudoInterpreter **4**
3.1 Zalety zastosowanego rozwiązania interpretera 5

4 Kroki przetwarzania **6**

5 Moduły i ich odpowiedzialność **7**
5.1 Pseudo.g4 (ANTLR4) 7
5.2 Functions.py 12
5.3 Variables.py 12
5.4 StackFrame.py & Stack.py 13

6 Zarządzanie zakresem (scope) i stosem wywołań **14**
6.1 Klasa `StackFrame` 15
6.2 Klasa `Stack` 15
6.3 Mechanizm dostępu do zmiennych 15
6.4 Przepływ wywołań (call stack) 16
6.5 Korzyści i dodatkowe uwagi 17

7 Ewaluacja wyrażeń i porządek wykonywania **18**

8 Obsługa błędów **20**

9 Głębokość rekurencji i ochrona stosu **20**

10 Napotkane problemy i ich rozwiązania **21**

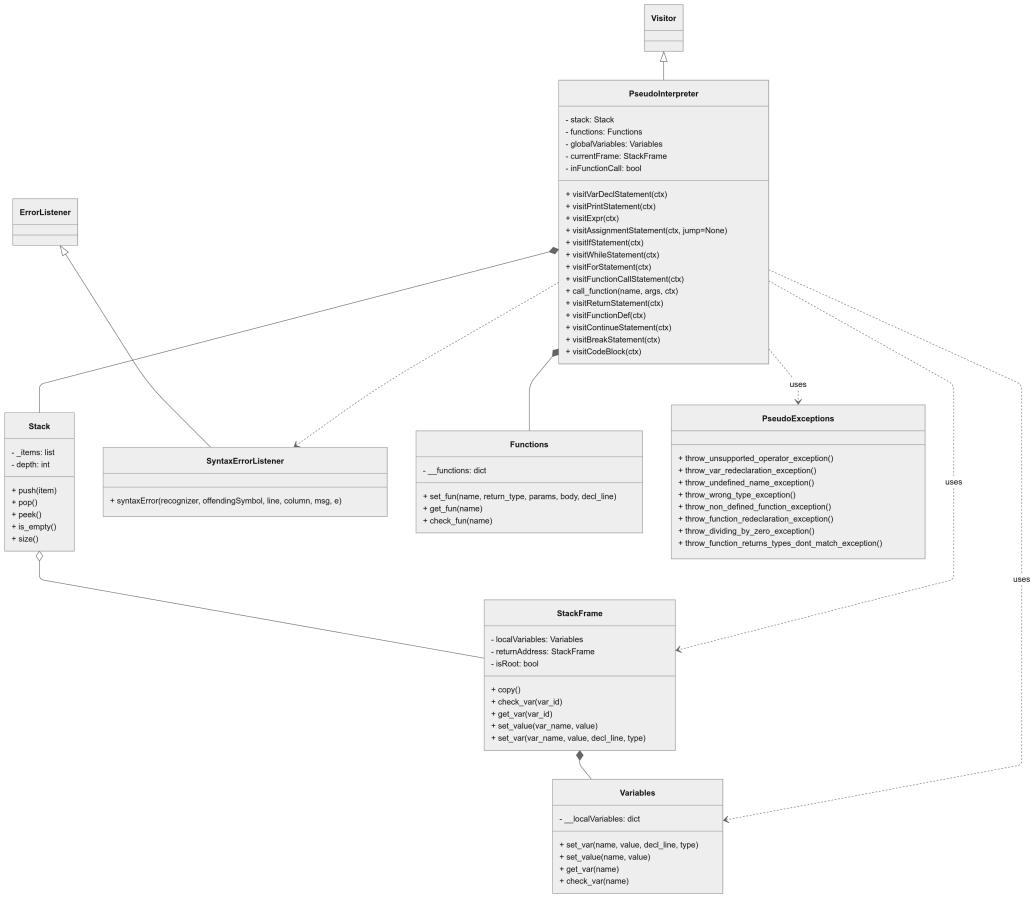
11 Podsumowanie i dalsze prace **21**

1 Cel i założenia projektu

Celem projektu było stworzenie prostego, a zarazem elastycznego języka programowania „pseudokodowego” – *Pseudo* – wraz z interpreterem pozwalającym na:

- Swobodę składniową (liczne aliasy, różne style bloków).
- Statyczne, silne typowanie i natychmiastowe raportowanie błędów.
- Praktyczne poznanie procesu budowy interpretera przy użyciu ANTLR4.
- Obsługę standardowych konstrukcji: instrukcje warunkowe, pętle, funkcje, wejście/wyjście.
- Możliwość późniejszego rozszerzenia (tablice, moduły, generacja kodu pośredniego).

2 Diagram klas



Rysunek 1: Diagram klas Pseudo

3 Przebiegowość interpretera PseudoInterpreter

Interpreter `PseudoInterpreter` działa w trybie **jednoprzebiegowym (single-pass)**. Oznacza to, że kod źródłowy analizowany i wykonywany jest podczas jednego przejścia przez drzewo składniowe wygenerowane przez ANTLR.

Rejestrowanie funkcji

Podczas odwiedzania węzłów drzewa:

- Gdy interpreter napotka definicję funkcji (`visitFunctionDef`), zapisuje ją w słowniku funkcji (`Functions`) bez wykonywania jej ciała.
- Dzięki temu każda funkcja jest zarejestrowana i dostępna do wywołania nawet przed jej rzeczywistym miejscem w kodzie.

Wywoływanie funkcji

Podczas wywołania funkcji (`visitFunctionCallStatement`):

- Tworzona jest nowa ramka stosu (`StackFrame`).
- Ciało funkcji jest odwiedzane z przekazanymi argumentami.
- Lokalne zmienne funkcji są przechowywane niezależnie od globalnych.

Obsługa rekurencji pośredniej

Dzięki temu mechanizmowi możliwa jest także **rekurencja pośrednia**, ponieważ wszystkie definicje funkcji są dostępne już przed wykonaniem jakiegokolwiek ich ciała.

3.1 Zalety zastosowanego rozwiązania interpretera

Zaprojektowany interpreter oparty na wzorcu odwiedzającego (`Visitor`) oraz jednoprzebiegowej analizie drzewa składniowego przynosi szereg korzyści:

Prostota implementacji

- Brak konieczności przechowywania pośrednich reprezentacji (np. AST lub kodu pośredniego).
- Mniej złożony kod — konstrukcje są przetwarzane bezpośrednio w momencie ich napotkania.
- Wykorzystanie mechanizmu `Visitor` ANTLR zapewnia czytelne rozdzielenie logiki dla poszczególnych węzłów gramatyki.

Obsługa funkcji i rekurencji

- Definicje funkcji są rejestrowane w momencie ich napotkania podczas jednego przebiegu.
- Funkcje muszą być zdefiniowane przed ich wywołaniem — nie jest możliwe wywołanie funkcji przed definicją.
- Rekurencja pośrednia działa poprawnie, ponieważ wywołanie funkcji powoduje odwiedzenie jej ciała dopiero podczas wywołania.
- Każde wywołanie funkcji tworzy nową ramkę stosu, co dobrze odwzorowuje lokalność zmiennych.

Efektywność czasowa i pamięciowa

- Kod jest analizowany i wykonywany w jednym przebiegu, co skraca czas działania interpretera.
- Nie są tworzone duże pośrednie struktury danych, co ogranicza użycie pamięci.

4 Kroki przetwarzania

Plik źródłowy Program zapisany w pliku z rozszerzeniem `.pseudo` jest wczytywany przy użyciu `FileStream`.

Lexing `PseudoLexer` przekształca ciąg znaków na strumień tokenów (obiekt `CommonTokenStream`).

Parsing `PseudoParser` tworzy drzewo składniowe (parse tree) na podstawie tokenów.

Visiting `PseudoInterpreter.visit(tree)` odwiedza węzły drzewa i wykonuje instrukcje:

- Deklaracje zmiennych i przypisania
- Instrukcje warunkowe (`if`), pętle (`while`, `for`)
- Wywołania funkcji (w tym rekurencyjne)

- Instrukcje wyjścia (`print`

5 Moduły i ich odpowiedzialność

5.1 Pseudo.g4 (ANTLR4)

Gramatyka definiuje tokeny i reguły:

- `program: ((functionDef | statement) ';')* EOF`
- `statement`: print, przypisanie, if/elseif/else, while, for, varDecl, codeBlock
- `functionDef`: wiele wariantów (function, fun, def, składnia „->")
- `expr`: obsługa operatorów aryt., por., logicznych i aliasów
- `codeBlock`: `{ body }`, `begin body end`, `block body end`

Listing 1: Cała Gramatyka Pseudo

```
grammar Pseudo;

program: (((functionDef | statement | expr) ';') | (
   codeBlock))* EOF;

statement:
        printStatement
        | assignmentStatement
        | ifStatement
        | whileStatement
        | forStatement
        | functionCallStatement
        | returnStatement
        | breakStatement
        | continueStatement
        | varDeclStatement
        ;
```

```
printStatement: ('print' | 'shout') '(' expr ')';

assignmentStatement
        : id = ID op = ('='| 'is'| '<<'| '<-') expr
        | id = ID op = ('++' | '--')
        | parent = PARENT assignmentStatement
        ;

ifStatement:
        'if' ('(' expr ')' | expr) (':' | 'then') body (
                'elseif' ('(' expr ')' | expr) (':' | '
                   then') body
        )* ('else' ':' body)? 'end' ('if')?;

whileStatement: 'while' '(' expr ')' ':' body 'end' ('
   loop')?;

forStatement:
        'for' '(' (entryStmt = initStatement)? ';' expr?
            ';' assignmentStatement? ')' ':' body 'end'
           (
                'loop'
        )?;

initStatement: varDeclStatement | assignmentStatement;

breakStatement: 'break' ('loop')? | 'exit' ('loop')?;

continueStatement: 'continue' ('loop')? | 'next' ('loop
   ')?;

functionDef
        : 'function' type = TYPE name = ID '(' params =
           paramList? ')' ':' block = body 'end' ('
           function')?
        | 'fun' type = TYPE name = ID '(' params =
           paramList? ')' ':' block = body 'end' ('fun')
           ?
        | 'def' type = TYPE name = ID '(' params =
           paramList? ')' ':' block = body 'end' ('def')
```

```
            ?

        | 'function' name = ID '(' params = paramList?
            ')' '->' type = TYPE ':' block = body 'end'
            ('function')?
        | 'fun' name = ID '(' params = paramList? ')'
            '->' type = TYPE ':' block = body 'end' ('fun
            ')?
        | 'def' name = ID '(' params = paramList? ')'
            '->' type = TYPE ':' block = body 'end' ('def
            ')?
        ;

returnStatement: 'return' val = expr;

paramList: param (',' param)*;

param: type = TYPE name = ID;

functionCallStatement: name = ID '(' args = argumentList
   ? ')';

argumentList: expr (',' expr)*;

codeBlock
        : '{' body '}'
        | 'begin' body 'end'
        | 'block' body 'end'
        ;

body: (((functionDef | statement | expr) ';') | (
   codeBlock))*;

varDeclStatement:
        (global = 'global')? TYPE ID (
                op = ('=' | 'is' | '<<' | '<-') expr
        )?;

expr: ('input' | 'scan' | 'listen') '(' (STRING)? ')'
        | functionCallStatement
```

```
        | expr op = (MULT | DIV) expr
        | expr op = (PLUS | MINUS) expr
        | expr op = (
                GREATER
                | SMALLER
                | EQUAL
                | DIFFERENT
                | GREATEREQUAL
                | SMALLEREQUAL
        ) expr
        | expr op = (
                GREATER
                | SMALLER
                | EQUAL
                | DIFFERENT
                | GREATEREQUAL
                | SMALLEREQUAL
        ) expr
        | expr op = INTDIV expr
        | op = MINUS expr
        | expr op = AND expr
        | expr op = OR expr
        | op = NOT expr
        | op = PARENT expr
        | op = TYPE '(' expr ')'
        | '(' expr ')'
        | STRING
        | NUMBER
        | DOUBLE
        | BOOL
        | ID;

STRING: '"' (ESC | ~["\\])* '"' | '\'' (ESC | ~['\\])*
   '\'';
fragment ESC: '\\' ["'\\rbnt];

NUMBER: [0-9]+;
DOUBLE: [0-9]+ '.' [0-9]+;
BOOL: 'True' | 'False';
```

```
WS: [ \t\n\r]+ -> skip;

SINGLE_LINE_COMMENT: '//' ~[\r\n]* -> skip;
MULTI_LINE_COMMENT: '/*' .*? '*/' -> skip;

PLUS: '+';
MINUS: '-';
MULT: '*';
DIV: '/';
INTDIV: '/#';
INCREMENT: '++';
DECREMENT: '--';
GREATER: '>' | 'greater than';
SMALLER: '<' | 'smaller than';
GREATEREQUAL: '>=' | 'greater or equal than';
SMALLEREQUAL: '<=' | 'smaller or equal than';
EQUAL: '==' | 'equals';
DIFFERENT: '!=' | 'differs';
AND: '&&' | 'and';
OR: '||' | 'or';
NOT: '!' | 'not';
PARENT: 'parent::';

TYPE:
        TYPE_INT
        | TYPE_FLOAT
        | TYPE_STRING
        | TYPE_BOOL
        | TYPE_VOID;

TYPE_INT: 'int';
TYPE_FLOAT: 'float';
TYPE_STRING: 'string';
TYPE_BOOL: 'boolean';
TYPE_VOID: 'void';

ID: [a-zA-Z_][a-zA-Z0-9_]*;
```

5.2 Functions.py

- Przechowuje dane funkcji w prostym słowniku `__functions` dla optymalizacji (słowniki są haszowane).
- `set_fun(name, return_type, params, body, decl_line)`:
  - Dodaje wpis:

    `functions[name]` = {"`return_type` : ..., "`params` : ..., "`body` : ..., "`decl_line` : ...}

    * `return_type` — string kodujący typ zwracany.
    * `params` — słownik, każda para to nazwa parametru i jego typ.
    * `body` — obiekt kontekstu ANTLR wskazujący na węzeł drzewa ciała funkcji.
    * `decl_line` — numer linii definicji, przydatny w błędach redeklaracji.
  - Umożliwia późniejszą weryfikację sygnatur i typu zwracanego.
- `get_fun(name)`:
  - Zwraca mapę danych dla podanej funkcji.
  - Rzuca `NameError`, jeśli funkcja nie została zdefiniowana.
- `check_fun(name)`: zwraca `True` jeśli funkcja istnieje, `False` w przeciwnym wypadku.
- **Zastosowanie:** w `visitFunctionCallStatement` interpreter sprawdza `check_fun` przed wywołaniem i pobiera metadane przez `get_fun`.

5.3 Variables.py

`Variables` to klasa enkapsulująca pamięć jednego scope'u:

- `__localVariables`: prywatny słownik

  { `name` → {"`value` : . . . , "`type` : . . . , "`decl_line` : . . .} }.

  - `value` — może być int, float, str lub bool

  - type — stringowy kod typu, wykorzystywany do weryfikacji w przypisaniach.
  - `decl_line` — numer linii w kodzie Pseudo użyty w komunikatach o błędach.

- `set_var(name, value, decl_line, type)`:
  - Dodaje nową zmienną, jeżeli `name` nie istnieje.
  - Przechowuje wartość, typ i numer linii deklaracji.
  - Rzuca wyjątek przy próbie redeklaracji.
- `set_value(name, value)`:
  - Aktualizuje `value` istniejącego wpisu w `__localVariables`.
  - Jeśli nie ma zmiennej lokalnie, deleguje do nadrzędnego scope'u (poprzez `returnAddress` w `StackFrame`).
- `get_var(name)`:
  - Zwraca całą strukturę {`"value","type","decl_line"`}.
  - Rzuca `NameError` jeśli zmienna nie istnieje.
- `check_var(name)`: zwraca `True` gdy `name` jest kluczem w `__localVariables`.

Współpraca z interpreterem:

- `visitVarDeclStatement` używa `set_var` do deklaracji.
- `visitAssignmentStatement` pobiera `value=visit(expr)` i wywołuje `set_value`.
- `visitExpr.ID()` odczytuje `get_var` by uzyskać wartość zmiennej.

5.4 StackFrame.py & Stack.py

StackFrame Reprezentuje jeden *scope* wykonania:

- `localVariables`: instancja `Variables` dla zmiennych lokalnych.
- `returnAddress`: wskaźnik do ramki nadrzędnej (scope wywołującego), używany do rekurencyjnego lookup'u zmiennych i przypisań.

Stack Prosty kontener LIFO dla `StackFrame`:

- `push(frame)` – dodaje ramkę na wierzch, zwiększa `depth`.
- `pop()` – usuwa i zwraca wierzchnią ramkę, zmniejsza `depth`.
- `peek()` – podgląd wierzchniej ramki bez usuwania.
- `is_empty()`, `size()` – pomocnicze metody.
- **Limit głębokości: 30** – jeżeli `depth > 30`, rzucany jest wyjątek `"Error: Stack overflow!"`, co chroni przed niekontrolowaną rekurencją i wyczerpaniem pamięci.

Schemat działania

1. **Start:** Tworzymy `initialFrame`, `push(initialFrame)`.
2. **Wejście w funkcję/block:**
   - Funkcja → `geniusCopy()` + `push()`.
   - Block lub pętla → `normalCopy()` + `push()`.
3. **Wyjście:** `pop()` – przywrócenie poprzedniego scope'u.

Dzięki temu każdy fragment kodu (funkcja, blok, iteracja) pracuje na oddzielnej ramce — zapewniając bezpieczne, izolowane przetwarzanie zmiennych i pełną kontrolę nad hierarchią scope'ów.

6 Zarządzanie zakresem (scope) i stosem wywołań

Idea Każdy kontekst wykonania programu—czy to główny moduł, funkcja, czy blok kodu—otrzymuje swoją *ramkę* (ang. `StackFrame`). Ramki są przechowywane na *stosie* (`Stack`), co pozwala:

- Izolować zmienne lokalne w danym scope.
- Dziedziczyć zmienne globalne i dostęp do zakresu nadrzędnego.
- Obsługiwać rekurencję i zagnieżdżone wywołania funkcji w naturalnym porządku LIFO.

6.1 Klasa `StackFrame`

- **localVariables**: instancja `Variables`, przechowująca wyłącznie zmienne zadeklarowane w tym scope.
- **globalVariables**: referencja do wspólnej, globalnej instancji `Variables`, gdzie trzymamy wszystkie zmienne globalne.
- **returnAddress**: wskaźnik do ramki nadrzędnej (scope wywołującego), wykorzystywany do:
  - Odczytu niezadeklarowanych lokalnie zmiennych (lookup rekurencyjny).
  - Przypisywania do zmiennych nadrzędnych (jeśli nie ma ich lokalnie).

6.2 Klasa `Stack`

- Implementacja LIFO dla `StackFrame`.
- `push(frame)`: wstawia nową ramkę na szczyt, inkrementuje licznik głębokości (`depth`).
- `pop()`: usuwa ramkę ze szczytu, dekrementuje `depth`.
- `peek()`: zwraca bieżącą (górną) ramkę bez usuwania.
- `is_empty()`, `size()`: pomocnicze metody diagnostyczne.
- *Ochrona przed przepełnieniem*: jeśli `depth > 30`, rzucany jest wyjątek `"Error: Stack overflow!"`. Zapobiega to niekontrolowanej rekurencji i wyciekom pamięci.

6.3 Mechanizm dostępu do zmiennych

Deklaracja zmiennej W `visitVarDeclStatement` interpreter wybiera, w zależności od flagi `global`, czy:

- `currentFrame.localVariables.set_var(...)` — tworzy zmienną w bieżącym scope,
- `currentFrame.globalVariables.set_var(...)` — tworzy zmienną globalną.

Odczyt i przypisanie W momencie odczytu (`get_var`) lub zmiany wartości (`set_value`) interpreter:

1. Sprawdza `localVariables.check_var(name)`.
2. Jeśli nie ma, przekazuje zapytanie na `returnAddress` (rekurencyjnie) aż do ramki głównej.
3. Jeśli zmienna nie istnieje w globalVariables, wołana jest funkcja, która rzuca wyjątek ze stosownymi informacjami `throw_undefined_name_exception`.

```
global int x = 1;

function void foo():
    // nowa ramka
    int x = 5
    {
    //nowa ramka
    int x = 2;        // shadowing: lokalne x
    print(x);         // wypisze 2
    x = x + 1;        // localVariables["x"] = 3
    print(x);        // wypisze 3
    print(parent::x); // wypisze 5
    print(parent::parent::x) wypisze 1
    }
end function;

foo();
print(x);             // wypisze 1
```

6.4 Przepływ wywołań (call stack)

1. **Start programu:** Tworzymy obiekt `StackFrame` z ustawionym `isRoot=True`. Następnie wrzucamy go na stos przy pomocy `stack.push()` i ustawiamy jako `currentFrame`.
2. **Wywołanie funkcji:**

(a) Tworzymy kopię bieżącej ramki przy pomocy `StackFrame.copy()`.

(b) Ustawiamy `returnAddress` tej nowej ramki na `currentFrame`.

(c) Wrzucamy nową ramkę na stos przy pomocy `stack.push()`.

(d) Aktualizujemy `currentFrame` na nową ramkę.

(e) Bindowanie parametrów do lokalnego scope'u funkcji — za pomocą `localVariables.set_var()`.

(f) Wywołujemy `visit(body)`, czyli wykonanie ciała funkcji poprzez odwiedzanie drzewa składniowego.

(g) Po zakończeniu działania funkcji wykonujemy `stack.pop()` i przywracamy poprzednią ramkę jako `currentFrame`.

3. *Obsługa pętli i rekurencji:*

   - Przed każdą iteracją lub wywołaniem rekurencyjnym wykonujemy `normalCopy()` bieżącej ramki, co pozwala na:
     - Zachowanie stanu zmiennych sprzed iteracji.
     - Niezależne działanie zmiennych lokalnych w kolejnych iteracjach.
   - Po zakończeniu iteracji stara ramka jest przywracana (pop).

6.5 Korzyści i dodatkowe uwagi

- **Izolacja lokalnych danych** – funkcje i bloki nie mogą przypadkowo zmieniać zmiennych spoza swojego scope.
- **Shadowing** – dopuszczalne, lokalna zmienna o tej samej nazwie zasłania globalną.
- **Bezpieczeństwo** – przepełnienie stosu (reku- rencja nieskończona) wykrywane i hamowane limitem 30 ramek.
- **Elastyczność** – różne style składni blokowej (`{...}`, `begin/end`, `block/end`) korzystają z tego samego mechanizmu stack/scope.

- **Łatwość debugowania** – każdy wyjątek semantyczny (np. niezadeklarowana zmienna) zawiera numer linii, co ułatwia lokalizację błędu.

7 Ewaluacja wyrażeń i porządek wykonywania

Interpreter Pseudo wykorzystuje metodę `visitExpr` w klasie `PseudoInterpreter`, która realizuje rekursywną ewaluację drzewa wyrażeń w sposób naturalny, z zachowaniem kolejności lewo–prawo oraz z uwzględnieniem priorytetów operatorów i optymalizacji short–circuit. Poniżej szczegóły:

1. **Rekurencyjna ewaluacja lewo–prawo** `visitExpr(ctx)` dla węzła binarnego najpierw wywołuje `visitExpr(ctx.expr(0))`, następnie `visitExpr(ctx.expr(1))`, a dopiero potem wykonuje operację na otrzymanych wynikach. Dzięki temu:
   - Zachowany jest porządek, w jakim pojawiają się operandy w kodzie.
   - Możliwe jest bezproblemowe łączenie dowolnej liczby zagnieżdżonych podwyrażeń.
2. **Priorytety operatorów** Aby zapewnić poprawne grupowanie i kolejność działań, interpreter rozróżnia sześć poziomów priorytetu (od najwyższego do najniższego):

   $$() > \textit{unarne} > * / > + - > \textit{porównania} > \texttt{AND} > \texttt{OR}$$

   - *Nawiasy* (`(expr)`) zawsze nadrzędnie nad innymi operatorami.
   - *Operatory unarne* (`-expr`, `!expr`, rzutowania typu) mają wyższy priorytet niż mnożenie/dzielenie.
   - *Mnożenie i dzielenie* (`*`, `/`, `/`) rozliczane przed dodawaniem i odejmowaniem.
   - *Dodawanie i odejmowanie* (`+`, `-`) przed porównaniami.

- *Operatory porównania* (`>`, `<`, `==`, `!=`, `>=`, `<=`) łączą się przed logicznymi.
- `AND` (, and) wykonuje się przed `OR` (||, or).

3. **Short–circuit (przerywane ewaluacje dla logicznych)** Interpreter stosuje optymalizację, która zapobiega niepotrzebnej ewaluacji prawego argumentu w wyrażeniach logicznych:

   - `A AND B` – jeżeli `A` ewaluowało się na `false`, `visitExpr` natychmiast zwraca `false` i *nie* wywołuje `visitExpr(ctx.expr(1))`.
   - `A OR B` – jeżeli `A` ewaluowało się na `true`, `visitExpr` natychmiast zwraca `true` i pomija wywołanie dla `B`.

   Dzięki temu zyskujemy na wydajności, bo nie przetwarzamy drugiego podwyrażenia, gdy nie jest to konieczne.

4. **Rzutowania typów (casts)** W gramatyce dopuszczone są syntaktyczne wyrażenia rzutujące:

   `int(expr)`, `float(expr)`, `string(expr)`, `boolean(expr)`

   - W `visitExpr` rozpoznajemy `ctx.op.type == PseudoParser.TYPE`,
   - Najpierw odwiedzamy wewnętrzne `expr`,
   - Następnie próbujemy skonwertować wynik do zadeklarowanego typu,
   - W razie niepowodzenia wywołujemy `throw_conversion_exception` z numerem linii/kolumny i informacją o niezgodnej wartości.

   To pozwala bezpiecznie wymuszać typ określonego podwyrażenia i zarządzać błędami konwersji.

5. **Obsługa wyjątków semantycznych** Podczas ewaluacji `visitExpr` mogą wystąpić sytuacje niezgodne z typami:

   - Próba dodania napisu do liczby lub odwrotnie (`"foo" + 3`).
   - Dzielenie przez zero (`x / 0`).
   - Niezgodny typ w operacji logicznej (np. `5  true`).

W każdej z tych sytuacji interpreter wywołuje odpowiednie `throw_..._exception(line, column, ...)`, co kończy wykonanie z czytelnym komunikatem i pozycją błędu.

8 Obsługa błędów

SyntaxErrorListener – natychmiastowe przerwanie parsowania przy błędzie składni.

PseudoExceptions – semantyczne:

- `throw_undefined_name_exception` – niezadeklarowana zmienna.
- `throw_wrong_type_exception` – niezgodność typów.
- `throw_unsupported_operator_exception` – operator nie działa na danych typach.
- `throw_conversion_exception` – nieudana konwersja typów.
- `throw_var_redeclaration_exception` – podwójna deklaracja zmiennej.
- `throw_function_redeclaration_exception` – wielokrotna definicja funkcji.

9 Głębokość rekurencji i ochrona stosu

- Maksymalna liczba ramek na stosie: `30`.
- Próba utworzenia 31–szej ramki → wyjątek `"Error: Stack overflow!"`.
- Chroni przed nieskończoną rekurencją i przepełnieniem pamięci.

10 Napotkane problemy i ich rozwiązania

- **Aliasowanie operatorów** (`==/equals`, `!=/differs`) → ujednolicenie tokenów w analizatorze leksykalnym.
- **Różne style bloków** (`{...}`, `begin/end`, `block/end`) → jedna reguła `codeBlock`, jednak każdy blok generuje nową ramkę.
- **Zakresy zmiennych** → stos ramek (`StackFrame`) z dziedziczeniem globalVariables i returnAddress.

11 Podsumowanie i dalsze prace

Projekt Pseudo stanowi kompletny, lecz elastyczny fundament do nauki kompilatorów i interpreterów. Potencjalne rozszerzenia:

- Interpreter Pseudo napisany w pythonie, skompilowany do czystego C, w celu poprawy wydajnosci.
- Typy złożone: tablice, struktury danych użytkownika.
- Moduły i przestrzenie nazw.
- Obsługa wyjątków w samym Pseudo: `try/catch`.