Dokumentacja

TKOM - interpreter języka z wbudowanym ułamkiem zwykłym

Adam Małkowski

FUNKCJONALNOSC

Projekt zakłada napisanie interpretera własnego języka w paradygmacie imperatywnym.

Język ma pozwalać na:

- deklarowanie oraz wykonywanie funkcji
- posiadać instrukcję warunkową oraz pętle
- pozwalać na deklarowanie zmiennych lokalnych i globalnych
- pozwalać na realizacje wyrażeń logicznych oraz arytmetycznych
- pozwalać na komunikacje z otoczeniem - tj. mieć możliwości obsługi wejścia/wyjścia
- mieć wbudowaną obsługę ułamka zwykłego

ANALIZA WYMAGAN FUNKCJONALNYCH I NIEFUNKCJONALNYCH - OBSLUGA BLEDÓW

WYMAGANIA FUNKCJONALNE

- możliwość wykonania kodu zgodnego z gramatyką
- informowanie użytkownika o błędach leksykalnych, strukturalnych oraz semantycznych - w miarę możliwości jasna i przejrzysta
- działanie operacji wejścia/wyjścia w kodzie realizowanych wbudowanymi funkcjami
- zapisywanie przebiegu interpretacji do pliku oraz wyświetlanie na terminalu

WYMAGANIA NIEFUNKCJONALNE

- brak ograniczenia długości kodu (z dokładnością do ograniczeń sprzętowych, np. kod nie mieści sie na dysku itp.)
- brak ograniczeń na liczbę zmiennych i funkcji (z dokładnością do pojemności RAM)
- interpretowanie jedynie poprawnego kodu
- zapewnienie pewności poprawności w interpretacji

SPOSÓB URUCHOMIENIA - WEJ/WYJ

Interpreter działa w trybie tekstowym. Aplikacja będzie wyświetlać się jako konsola, w której będziemy mogli podać ścieżkę do pliku z kodem oraz obserwować i kreować odpowiednio operacje wyjścia i wejścia.

Zapis z konsoli, zapis działania aplikacji oraz komunikaty nt. przebiegu interpretacji będą zapisywane do określonego pliku tekstowego.

GRAMATYKA

PODSTAWY

```
mala_litera = "a"|"b"|"c"|...|"y"|"z";
wielka_litera = "A"|...|"Y"|"Z";
litera = mala_litera | wielka_litera;

zero = "0";
cyfra_niezerowa =  "1"|...|"9";
cyfra = zero | cyfra_niezerowa;

znak = znak_specjalny  | "!", "@", "#" ....; (w tym białe znaki)
znak_specjalny = "//", "/"";

lancuch = """{znak | cyfra | litera}""";

znak_liczby = "+" | "-";
liczba  = liczba_calkowita ["." {cyfra}];
liczba_calkowita  = [znak_liczby] cyfra_niezerowa {cyfra} | zero;

stala_logiczna = "PRAWDA" | "FALSZ";

identyfikator = mala_litera {litera | cyfra};

operator_porowania = "<" | "<=" | "=" | "!=" | ">" | ">=";
operator_przypisania = ":=";
zakoncznik = ";";

operator_multiplikatywny = "*" | ":";

funkcja_tablica = "(" { wyrazenie_arytmetyczne } ")" | "[" wyrazenie_arytmetyczne "]"
atom_wartosc = identyfikator funkcja_tablica

atom_arytmetyczny = liczba | ulamek_zwykly | atom_wartosc;
atom_logiczny = stala_logiczna | atom_wartosc;
```

ARYTMETYKA

wyrazenie_arytmetyczne = wyrazenie_arytmetyczne2 wyrazenie_arytemtyczne_prim;
wyrazenie_arytmetyczne_prim = znak_liczby wyrazenie_arytmetyczne_2
wyrazenie_arytmetyczne_ prim | Epsilion;
wyrazenie_arytmetyczne = wyrazenie_arytmetyczne3 wyrazenie_arytemtyczne_2_prim;
wyrazenie_arytmetyczne_2_ prim = operator_multi_arytmetyka
wyrazenie_arytmetyczne_3 wyrazenie_arytmetyczne_2_ prim | Epsilion;
wyrazenie_arytmetyczne_3 =[" -"]("(" wyrazenie_arytmetyczne ")" |
atom_arytmetyczny);

LOGIKA

wyrazenie_logiczne_1 = wyrazenie_logiczne_2 wyrazenie_logiczne_1_prim;
wyrazenie_logiczne_1_prim = "I" wyrazenie_logiczne_2 wyrazenie_logiczne_1_prim |
Epsilion;
wyrazenie_logiczne_2 = wyrazenie_logiczne_3 wyrazenie_logiczne_2_prim;
wyrazenie_logiczne_2_prim = "LUB" wyrazenie_logiczne_3 wyrazenie_logiczne_2_prim |
Epsilion;

wyrazenie_logiczne_3 = ["NIE"] ("(" wyrazenie_logiczne_1 ")" | wyrazenie_logiczne);
wyrazenie_logiczne = wyrazenie_arytmetyczne operator_porowania
wyrazenie_arytmetyczne | atom_logiczny;

UŁAMEK ZWYKLY

ulamek_zwykly = liczba_calkowita "/" liczba_calkowita;

ZMIENNE

wartosc = wyrazenie_logiczne | wyrazenie_arytmetyczne | lancuch;
typ = wielka_litera {litera};

deklaracja = typ identyfikator zmiennej_funkcji
zmiennej_funkcji = ["["wyrażenie_arytmetyczne"]"] [operator_przypisania wartosc]
zakoncznik | "(" {typ identyfikator} ")" blok_kodu;

INSTRUKCJE STERUJACE

blok_kodu = "{" {instrukcja_warunkowa | petla | deklaracja_zmiennej |
odwolanie_sie_do_id | zwroc} "}";
zwroc = "ZWROC" wartosc;
instrukcja_warunkowa = "JEZELI" wyrazenie_logiczne blok_kodu
["W_PRZECIWNYM_PRZYPADKU" blok_kodu];
petla = "DOPOKI" wyrazenie_logiczne blok_kodu;

FUNKCJE

wywolanie_funkcji = "(" { wartosc } ")";
przypisanie_wartosci = ["["liczba_calkowita"]"] operator_przypisania wartosc
zakoncznik;
odwolanie_sie_do_id = identyfikator wywolanie_funkcji | przypisanie_wartosci

KOD

program = {deklaracja}

STRUKTURA PROGRAMU

Pakiety:

code - zawiera podstawowe klasy reprezentujące kod programu - podstawowe typy instrukcji oraz klasy pomocnicze (np. Function)

code.arithmetci - zawiera klasy reprezentujące elementy wyrażeń arytmetycznych

code.logic - zawiera klasy reprezentujące elementy wyrażeń logicznych

code.logic.comparasioninstructions - zawiera klasy reprezentujące porównania arytmetyczne

errors - zawiera potencjalne wyjątki

interpretator - tutaj znajdują się podstawowe funkcje interpretacji oraz main - inicjalizacja wszystkich obiektów, przygotowanie i rozpoczęcie parsowania oraz wykonywania, obsługa błędów występujących w trakcie

lexer - zawiera klasy analizatora leksykalnego

parser - zawiera klasę parsera

Program jest podzielony logicznie na 4 modułu:

Analizator leksykalny

Parser

Moduł wykonywawczy

Moduł obsługi błędów

Program, zgodnie z wyższym opisem jest napisany w paradygmacie obiektowym. Moduł obsługi błędów opiera się na mechanizmie wyjątków.

ROZROZNIANE TOKENY

- identyfikator : np. zmienna
- ALBO
- JEZELI
- DOPOKI
- I
- LUB
- ZWROC
- FALSZ
- PRAWDA
- "{", "}", "(", ")", ";", "[", "]"
- liczba : np. 2, 5.5, 5/9
- operatory porównawcze: <=, <, >, >=, =, !=,
- operatory addytywne: +, -
- operatory multiplikatywne: *, :
- operator przypisania: :=
- łańcuch znakowy : np. "lancuch kotów"

WYKORZYSTANE STRUKTURY DANYCH

Podczas parsowania tworzona jest struktura kodu składająca się z listy funkcji oraz zmiennych globalnych. Każda funkcja (obiekt Function) posiada szablon zmiennych lokalnych oraz listę instrukcji (Instructions).
Program posiada globalny stos (będący stosem, stosów, wektorów - wektor zawiera zmienne danego bloku, pośrednie stosy odpowiadają wywołaniom funkcji) na który odkładane są kopie szablonów zmiennych lokalnych danego bloku kodu.

Do przechowywania wartości użyta jest klasa SpecyficValue mogąca przyjąć wartości typów -int, double, string, boolean oraz fraction.

W programie występują typy wyliczeniowe:

- TokenType - możliwe tokeny
- Type - możliwe typy danych
- Sign - możliwe znaki arytmetyczne

Wyrażenia arytmetyczne oraz logiczne są reprezentowane jako drzewa złożone z elementarnych działań arytmetycznych/logicznych.

Wyjątki dziedziczą po klasie BaseException która jest poszerzeniem klasy Exception o pole z dodatkowymi informacjami do wyświetlenia.

ROZPISKA CZĘSCI STRUKTUR (pola)

KLASY:

```
CodeBlock{
        public List<Instruction> instructions;
        public List<Variable> localVariablesTemplate;
        private CodeBlock parent;
        private SpecyficValue returnValue;
}

Function{
        private String id;
        private Type returnType;
        public List<Variable> arguments;
        public CodeBlock bodyOfFunction;}

Assignment : Instruction{
        private Variable variable;
        private GetingValue value;
        private boolean table;
        private ArithmeticExpression index;
}

FunctionCall : Instruction {
        public List<Instruction> arguments;
        private Function function;
        private SpecyficValue returnValue;
}

ConditionalInstruction : Instruction {
        private CodeBlock codeBlockWhenTrue;
        private CodeBlock codeBlockWhenFalse;
        private Boolean conditionalValue;
        private LogicExpression conditional;
        private SpecyficValue returnValue;
}

GettingValue : Instruction {
        private ArithmeticExpression arithmetic;
        private LogicExpression logic;
        private String string;
}

Loop : Instruction {
        private CodeBlock bodyOfLoop;
        private SpecyficValue returnValue;
```

```
}


ReturnInstruction : Instruction {
        private LogicExpression logicValue;
        private ArithmeticExpression arithmeticValue;
        private String stringValue;
}

SpecyficValue{
        private Integer intValue;
        private Double doubleValue;
        private Boolean booleanValue;
        private String stringValue;
        private Integer numerator;
        private Integer denumerator;

        private Type type;
}

Variable{
        private Type type;
        private String id;
        private Boolean table;
        private ArithmeticExpression tableSize;
        private SpecyficValue value;
        public Vector<SpecyficValue> tableValue;
}

VariableUse : Instruction {
        private String id;
        private ArithmeticExpression index;
        private Boolean table;
}

ArithmeticExpression : Instruction {
        protected ArithmeticExpression left;
        protected ArithmeticExpression right;
        protected Boolean unary = true;
        protected SpecyficValue value;
        protected Instruction instructionValue;

        protected Sign sign;
}

LogicExpression : Instruction {
        protected LogicExpression left;
        protected LogicExpression right;
        protected Boolean unary = true;
```

```
        protected SpecyficValue value;
        protected Instruction instructionValue;
}
Comparation : LogicExpression {
        protected ArithmeticExpression leftSite;
        protected ArithmeticExpression rightSite;
}

Program {
        public List<Function> functions;
        public Stack<Stack<List<Variable>>> stack;
        public CodeBlock context;
}
```

OBIEKTY:

```
        private Lexer lex;
        private Parser parser;
        public static Program program;
```

W zmiennej program znajdują się lista funkcji, stos oraz context (realizujący zmienne globalne)

Rezultaty testowania:

Testy zostały przeprowadzane w formie wykonywania kodu programu wykorzystującego różne funkcjonalności. Do powyższej dokumentacji załączony jest folder w którym znajdują się pary plik z kodem - wynik działania interpretera pokazujące przykłady użycia programu, efekt jego działania oraz komentarz do przykładów w plikach wynikowych .

Program był również testowany w trakcie pisania co nie zostało udokumentowane.

Udokumentowane testy:

-użycia zmiennej globalnej
-użycia funkcji
-użycia funkcji rekurencyjnej (w dwóch wersjach)
-użycia pętli i prostych wyrażeń arytmetycznych
-użycie wyrażeń arytmetycznych
-przykrywanie zmiennych
-wykorzystanie tablicy oraz błąd pobrania niezainicjalizowanej wartości
-użycie niezadeklarowanej zmiennej
-dzielenie przez zero (błąd)
-zła struktura kodu (brak typu przy deklaracji funkcji - błąd)
-brak funkcji głównej (błąd)
-brak ciała definicji wykorzystanej funkcji (błąd)
-wielokrotna deklaracja zmiennej w tym samym bloku (błąd)