Kontekstowy język programowania ogólnego przeznaczenia

Dokumentacja projektowa

Wersja wstępna

Przedmiot TKOM

Semestr letni 24

Małgorzata Kozłowska

Prowadzący: Piotr Gawkowski

Spis treści

1 Wprowadzenie

Celem projektu jest stworzenie języka ogólnego przeznaczenia wzbogaconego o aspektowość. W tym celu zostaną stworzone niezbędne struktury wbudowane języka, które będą przechowywały dane parametrów wejściowych. Oprócz tego funkcje wzbogacone zostaną o parametry - związane z wartością zwracaną oraz parametrami wejściowymi. Zostanie zdefiniowany typ aspektu, który będzie wchodził w interakcje z funkcjami. Aspekty będzie można deklaratywnie włączać i wyłączać wewnątrz ciała funkcji.

2 Zakładana funkcjonalność - przykłady obrazujące dopuszczalne konstrukcje językowe

2.1 Podstawowe typy danych

Język będzie składał się z dobrze znanych podstawowych typów danych - `int`, `float`, `str`, `bool`. Oprócz nich na potrzeby języka zdefiniowane zostaną nowe wbudowane typy danych: `aspect`, wraz z jego parametrem (`enabled`) oraz kontenerowe typy danych wbudowanych `param, intab` ze zdefiniowanymi strukturami oraz własnymi parametrami. Warto zaznaczyć, że `param`, intab są wewnętrznymi strukturami, do których mamy dostęp w sposób nie bezpośredni, lecz przez:

- parametr `func.intab` - zwraca strukturę typu `intab`, która jest tablicą przechowującą obiekty typu `param` dla danej funkcji. Obiekty typu `param` stanowią reprezentację parametrów wejściowych, jakie zostały podane przy wywołaniu funkcji.
- iterację po elementach struktury `intab` (np. w pętli `for`), bądź indeksowanie - będzie wówczas dostępny wewnętrzny element struktury `intab` - obiekt typu `param`

W sposób szczegółowy nowe typy danych (`intab`, `param`) zostaną opisane poniżej, w sekcji 2.4.

2.2 Konwersja typów

Język będzie statycznie, słabo typowany. Poniżej zamieszczone są zarówno dopuszczalne konwersje typów zmiennych, jak i te, które przy wywołaniu programu wygenerują błąd.

Przykłady konwersji typów

Konwersja	Użycie (typy)	Użycie (przykład)	Czy dopuszczalne ?
`int → float`	`int + float = float`	`1 + 2.5 = 3.5`	TAK

float → int	float - float = int	3.5 - 1.5 = 2	TAK
int → str	string + int + string	"Ala ma aż " + 5 + "kotów." >>> Ala ma aż 5 kotów.	TAK
float → str	string + float	"Dzisiejszy kurs EURO to: " + 4.29 >>> Dzisiejszy kurs EURO to: 4.29	TAK

str → bool	if (str) {...}	str myStringFlag = "true";
		if (myStringFlag) {...}
		⇔
		if (true) {...}
			TAK

str → bool	if (str) {...}	str myStringFlag = "false";
		if (myStringFlag) {...}
		⇔
		if (false) {...}
			TAK

int → bool	if (int) {...}	int myIntFlag = 1;
		if (myIntFlag) {...}
		⇔
		if (true) {...}
			TAK

int → bool	if (int) {...}	int myIntFlag = 0;
		if (myIntFlag) {...}
		⇔
		if (false) {...}
			TAK

Niewłaściwe są zaś:

Konwersja	Użycie (typy)	Użycie (przykład)	Czy dopuszczalne ?

str → bool	if (str) {...}	int myIntFlag = 2;
		if (myIntFlag) {...}
		⇔
		if (true) {...} ?
			NIE

int → bool	if (int) {...}	int myIntFlag = -1;
		if (myIntFlag) {...}
		⇔
		if (true) {...} ?
			NIE

str →	str + float = float	3.5 - 1.5 = 2;	NIE

`float`			
`str → itn`	`str + int = int`	`3.5 - 1.5 = 2;`	NIE

Oprócz tego, język posiada możliwość jawnej konwersji typu zmiennej, aby móc wymusić określony typ zmiennej:

```
int myInt = 6;
float myFloat = myInt as float;
print(myFloat)
>>> 6.0

bool trueBool = true;
str trueString;
trueString = trueBool as str;
```

Dostępne są konwersje: `as int`, `as float`, `as str`, `as bool`.

2.3 Typy wbudowane (standardowe):

string - `str`
integer - `int`
float - `float`
boolean - `bool`

2.4 Oraz typy stworzone pod kątem przeznaczenia języka

inputs' table - `intab`

- jest to tablica, składająca się ze zbioru argumentów wywołania funkcji typu `param`, podobnie jak Struct w C, lecz każda zmienna - wewnętrzny element kolekcji musi być typu `param`

parameter - `param`

- jest to obiekt, zawierający trójkę zmiennych - typ, nazwa oraz wartość danego argumentu wywołania funkcji - podobny do Struct w C, lecz z określonym typem kolejnych elementów wewnętrznych

```
# Niestandardowe typy danych
param myParam;
intab myInputTable;

# SZABLON budowy tablicy typu intab
myInputTable = [
    param: myParam,   # podstawowe typy danych
           ...   # nazwa zmiennej
               ];
```

```
# SZABLON budowy tablicy typu param
myParam = [
    type: myType,   # typ danych
    name: myName,   # nazwa zmiennej
    value: myValue, # wartość zmiennej
          ];

# przykład
int funcParam = 1; # deklaracja zmiennej typu int

exampleFunc(funcParam);  # zmienna ta później użyta jest w wywołaniu funkcji jako
                         # parametr wejściowy
 # tworzy się wewnętrzna struktura typu intab : [funcParam]

# zawierająca jeden element - jedną zmienną param, która posiada taką trójkę
# informacji: [int, funcParam, 1]

/*

UWAGA - nie można explicite stworzyć (zadeklarować oraz zainicjować) zmiennych typu
intab oraz param - powyższy przykład ilustruje, jakie obiekty tworzą się podczas
przekazania zmiennych wywołania funkcji oraz w jaki sposób zorganizowana jest ich
struktura.

*/
```

Zarówno typ `intab` jak i typ `param` posiadają wbudowane atrybuty, do których możemy się odwołać, przypisać nowe wartości.

Dla `intab`:

- atrybut `count` - zwracający wartość typu int, mówiącą o liczbie obiektów typu param w tablicy intab (o liczbie parametrów wejściowych wywoływanej funkcji)

Przykład:

```
# zdefiniowanie zmiennych, które później przekażemy funkcji jako parametry wejściowe
int example1 = 1;
str example2 = "example";
bool example3 = true;

exampleFunc(example1, example2, example3);  # wywołanie funkcji z parametrami
                                            # wejściowymi (która została gdzieś
                                            # wcześniej zdefiniowana)

/* wewnątrz funkcji stworzy się jedna struktura intab exampleInputTable,
przechowująca parametry: param exampleParam1 - parametr, który zawiera informacje o
```

```
zmiennej example1 (o jej typie (int), nazwie (example1) i wartości (1)), param
exampleParam2, param exampleParam3 (analogicznie jak w exampleParam1). Warto
zaznaczyć, że są to struktury wewnętrzne, których jawnie nie możemy zainicjować w
kodzie. Tak, jak poniżej zostało to pokazane, do struktury intab można dostać się
tylko przez atrybut funkcji o tejże samej nazwie. */
str param_number = exampleFunc.intab.count as str;
print(param_number);

>>>3

# bo intab exampleInputTable = [exampleParam1, exampleParam2, exampleParam3]
```

Dla `param`:

- atrybut `type` - zwracający typ zmiennej przekazanej jako parametr wejściowy wywołania funkcji
- atrybut `name` - zwracający nazwę tegoż że parametru wejściowego wywołania funkcji
- atrybut `value` - zwracający wartość parametru wejściowego

Przykład (kontynuacja wcześniejszego kodu):

```
str exampleParamType =  exampleFunc.intab[0].type;
str exampleParamName =  exampleFunc.intab[0].name;
str exampleParamValue =  exampleFunc.intab[0].value as str; # dochodzi do
                                                            # konwersji typu na str
print(exampleParamType);
>>> int
print(exampleParamName);
>>> example1
print(exampleParamValue);
>>> 1
```

2.5 Zmienne

2.5.1 Deklaracja zmiennych oraz przypisywanie wartości

```
# Deklaracja zmiennych

int myInt;                              # deklaracja liczby całkowitej
float myFloat;                          # deklaracja liczby zmiennoprzecinkowej
str myString;                           # deklaracja stringu
bool myBool;                            # deklaracja boola

# Przypisywanie wartości zmiennych

myInt = 1;
```

```
myFloat = 2.1;
myString = "mój pierwszy string";
myBool = true;

int mySecondInt = 5;                          # deklaracja i przypisanie liczby
                                              # całkowitej
float mySecondFloat = 3.4;                    # deklaracja i przypisanie liczby
                                              # zmiennoprzecinkowej

str mySecondString = "mój drugi string";      # deklaracja i przypisanie stringu

bool mySecondBool = false;                    # deklaracja i przypisanie boola

# BŁĘDNE przypisanie wartości do zmiennej

myWrongInt = 5;  # wygeneruje błąd, że zmienna nie została wcześniej zainicjowana
```

Zmienne będą widoczne w obrębie bloku kodowego {...}, bądź w całym programie, jeśli zostały globalnie zadeklarowane. Są one mutowalne - ich wartość może być wielokrotnie zmieniana.

2.5.2 Obsługa typów znakowych

```
# Obsługa typu znakowego

# Tworzenie zmiennych

str pierwszyString;
pierwszyString = "Ala ma";

str drugiString = " kota.";

# \n - znak końca linii
# \t - znak tabulacji
# \ - znak escape


str newLineString = "Ala\n ma kota.";
>>> Ala
>>>  ma kota.


str tabString = "Ala\t ma kota.";
>>> Ala     ma kota.
```

```
str escapedString = "Ala\\n ma kota.";
>>> Ala\n ma kota.


str specjalnyString

specjalnyString = "Ala powiedziała: \" Mam kota!\"";
>>> Ala powiedziała: "Mam kota!"

#Operacje na stringach

str pierwszyString;
pierwszyString = "Ala ma";

str drugiString = " kota.";

str calyString;

calyString = pierwszyString + drugiString  # calyString = "Ala ma kota.";
>>> Ala ma kota.
```

2.5.3 Obsługa komentarzy

```
# komentarz jednoliniowy
/* komentarz
wielolinijkowy */
```

2.5.4 Deklaracja funkcji i aspektów

Funkcjom jako parametry będą przekazywane wartości zmiennych.

Deklaracja funkcji:

```
# SZABLON funkcji
func exampleFunction(type: variableName): returnType  # jeśli funkcja nic nie
zwraca
        {                                             # to powinno być to
            # function body;                          # jawnie napisane - null
            return  # return value;
        }
```

Funkcja posiada dwa wbudowane atrybuty: `retval` oraz `intab`. `retval` odpowiada wartości zwracanej przez funkcję, zaś `intab` - tablicy parametrów wejściowych wywołania funkcji.

Deklaracja aspektu:

```
# SZABLON aspektu

aspect exampleAspect: on func start/end/call like "...":  # zamiast start możliwe są
                                                          # "end" oraz "call"
        {
            # aspect body;
            # standardowo będą to funkcje print()
        }
```

zamiast "..." podajemy "`wzorzec - wyrażenie regularne, jakie ma być znalezione w nazwie funkcji`"

Przykład aspektu oraz jak on oddziałuje z funkcją:

```
# przykład komunikacji aspektu z funkcją

aspect logResult: on func end like "write"
{
    str firstPrompt = "Function of a name: ";
    str funcName = func.name;
    str secondPrompt = ", has provided an output named: ";
    str funcRetName = func.retval.name;
    str thirdPrompt = ", of a type: ";
    str funcRetType = func.retval.type;
    str forthPrompt = ", with a value: ";
    str funcRetValue = func.retval.value;
    str finalPrompt = firstPrompt + funcName +  secondPrompt + funcRetName +
                      thirdPrompt + funcRetType + forthPrompt + funcRetValue;
    print(finalPrompt);

}

aspect logParams: on func start like "write"
{
    str firstPrompt = "Function of a name: ";
    str funcName = func.name;
    str secondPrompt = ", has provided the input parameters:\n";
    str inputParamsPrompt = "";
    int count = func.intab.count;
    for param in func.intab;
    {
        count = count - 1;
        str namePrompt = "name: " + param.name;
        inputParamsPrompt = inputParamsPrompt + namePrompt;
        str typePrompt = ", type: " + param.type;
        inputParamsPrompt = inputParamsPrompt + typePrompt;
```

```
        valuePrompt = ", value: " + param.value;
        inputParamsPrompt = inputParamsPrompt + valuePrompt;
        if (count != 0)
        {
            str newlinePrompt = "\n";
            inputParamsPrompt = inputParamsPrompt + newlinePrompt;
        }
        else
        {
            str endPrompt = ".";
            inputParamsPrompt = inputParamsPrompt + endPrompt;
        }

    }

    str finalPrompt = firstPrompt + func.name + secondPrompt + inputParamsPrompt;
    print(finalPrompt);

}


func writeCirclePerimeter(int: radius) : float
{
    logResult.enabled = true;  # deklaratywnie włączamy aspekt wewnątrz funkcji
    float area = 2 * 4.13 * radius;
    return perimeter;
}

# gdy parametr enable jest ustawiony na false aspekt nie wpływa na funkcję

func writeCircleArea(int: radius) : float
{
    logParams.enabled = true;
    logResult.enabled = false;  # deklaratywnie wyłączamy aspekt wewnątrz funkcji
    float area = 4.13 * radius * radius;
    return area;
}

int myRadius = 6;
writeCirclePerimeter(myRadius);
writeCircleArea(myRadius);

>>> Function of a name: writeCirclePerimeter, has provided an output named:
perimeter, of a type: float, with a value: 49.56
/* log o obliczonym obwodzie koła. Ze względu na to, że w ciele funkcji
writeCirclePerimeter aspekt logResult został jawnie włączony (poprzez ustawienie
```

```
pola <aspectName>.enabled = true został on wykonany */

>>> Function of a name: writeCircleArea, has provided the input parameters:
>>> name: myRadius, type: int, value: 6.
/* log o obliczonej powierzchni koła. Aspekt logResult jest wyłączony dla funkcji
writeCircleArea (poprzez logResult.enabled = false). Stąd też nie ma logu o
obliczonej powierzchni koła. */
```

Przykład definicji funkcji z rekursją, instrukcją warunkową if - else oraz pętlami while:

```
int myInt = 5;
float myFloat = 5.5;

# przykład instrukcji warunkowej if else
func equalFive(float numToEvaluate) : bool
{
      if (numToEvaluate == 5)
      {
            bool returnVerdict = true;
            return returnVerdict;
      }
      else
      {
            bool returnVerdict = false;
            return returnVerdict;
      }
}

print(equalFive(myInt))
>>> true

print(equalFive(myFloat))
>>> false

# przykład rekursywnego wywołania funkcji oraz pętli while
func decrementToZero(int numToDecrement) : null
{
      while (numToDecrement > 0)
      {
            decrementToZero(numToDecrement - 0);
      }
      str outputStr = "Number got decremented to zero";
      print(outputStr);
      return null;
}
```

Język będzie posiadał funkcję wbudowaną `print()`, która wyświetlać będzie argument wywołania na standardowym wyjściu. Jej parametrem wywołania może być tylko zmienna typu `str` - zmienne typu liczbowego (`float, int`) oraz typu bool muszą uprzednio zostać przekonwertowane.

3 Formalna specyfikacja i składnia

3.1 Priorytety oraz asocjacyjność operatorów

Priorytety oraz asocjacyjność operatorów

(Operator Precedence and Associativity)

Operator	Opis	Priorytet (1 - najwyższy)	Associativity
()	Nawiasowanie	1	Lewostronna
[]	Subskrypcja	2	Lewostronna
.	Dot operator	3	Lewostronna
-x	unarna negacja	4	Prawostronna
*, /	Mnożenie, dzielenie	5	Lewostronna
+, -	Dodawanie, odejmowanie	6	Lewostronna
<, <=, >, >=, !=, ==	Znaki porównania, nierówność, równość	7	Lewostronna
&&	Logical AND	8	Lewostronna
||	Logical OR	9	Lewostronna

Przykład:

```
print((3 * (6 + 2) - 4) / 2)
>>> 10

print((3 * (6 + 2) - 4) / 2) > 12)
>>> false
```

3.2 Notacja EBNF

```
program ::= { declaration_statement
            | function_declaration
            | aspect_declaration};
```

```
declaration ::= type, identifier;

function_declaration ::= "func", identifier, "(", [ parameters ], ")", ":",
return_type, block;

return_type ::= type | "null";


block ::= "{", { statement }, "}";

statement ::= selection_statement
            | declaration_statement
            | assignment_or_call_statement
            | iteration_statement
            | return_statement;

declaration_statement ::= declaration, [ "=", expression ];

selection_statement ::= "if", "(", condition, ")", block, ["else", block];

condition ::= expression;

iteration_statement ::= "for", identifier, "in", expression, block
                      | "while", "(", condition, ")", block;

assignment_or_call_statement ::= object_access, ["=", expression ] ";";

return_statement ::= "return", [expression], ";";


aspect_declaration ::= "aspect", identifier, ":", aspect_trigger, block;

aspect_trigger ::= "on", aspect_target, aspect_event, "like", regular_expression;

regular_expression ::= string;


aspect_event ::= "start"
               | "end"
               | "call";

aspect_target ::= "func";

parameters ::= [parameter, { ",", parameter }];
```

```
parameter ::= declaration;


type ::= "int"
       | "float"
       | "string"
       | "bool";

expression ::= and_term, {"||", and_term};

and_term ::= relation_term, { "&&", relation_term};

relation_term ::= additive_term, [relation_operator, additive_term];

relation_operator ::= ">="
                    | ">"
                    | "<="
                    | "<"
                    | "=="
                    | "!=";

additive_term ::= multiplicative_term, { ("+" | "-"), multiplicative_term};

multiplicative_term ::= unary_term, { ("*" | "/" ), unary_term};

unary_term ::= ["-"], casted_term;

casted_term ::= term, {"as", type};

term ::= literal
           | object_access
           | "(", expression, ")";

object_access ::= item, {".", item};

item ::= identifier_or_call, {"[", int, "]"};

identifier_or_call ::= identifier, [ "(", parameters, ")" ];

literal ::= int | float | bool | string;

int ::= "0" | digit_positive, {digit};

float ::= int, ".", digit, { digit };

string ::= '"' , { character }, '"';
```

```
bool ::= "true" | "false";

newline ::= "\n"
          | "\r\n"
          | "\n\r";

tab ::= "\t";

escape_char ::= "\";

digit ::= digit_positive | "0";

digit_positive ::= "1"
                 | "2"
                 | "3"
                 | "4"
                 | "5"
                 | "6"
                 | "7"
                 | "8"
                 | "9";


identifier ::= letter, {alphanumeric};

alphanumeric ::= letter | 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";
```

4 Obsługa błędów i typy komunikatów o błędzie

Format błędu:

```
ERR! [<row_number>:<column_number>]: <error_message>
```

Przykłady kilku typów komunikatów o błędzie:

1. Brak wcześniej deklaracji zmiennej

```
notDeclaredVariable = 6;
```

```
ERR! [1:1]: unexpected token 'notDeclaredVariable';
```

2. Operacja nie obsługiwana dla danego typu

```
true + false;
```

```
ERR! [1:5]: operator '+' not applicable to types: ‘bool’,’ bool’;
```

3. Wartość zwracana niezgodna z tą zadeklarowaną

```
func wrongReturn(int a): int
{
      return true;
}
```

```
ERR! [3: 10]: expected 'int' got 'bool' in return
```

Napotkanie błędu w programie powoduje natychmiastową terminację jego wywołania.

5 Opis sposobu testowania

Cały projekt - jego poprawność, spójność itd., będzie testowana głównie za pomocą testów jednostkowych (gdzie w izolowany sposób będzie sprawdzana poprawność implementacji komponentów projektów, takich jak np. analizator leksykalny, składniowy, semantyczny) oraz testów integracyjnych (gdzie sprawdzana będzie poprawność komunikacji pomiędzy dwoma komponentami na ich “miejscach styku”). Wszystko to ma służyć zapewnieniu sprawnego przepływu pomiędzy odpowiednimi fazami kompilacji.