TKOM / ZPR2025L - dokumentacja wstępna

Krzysztof Gólcz 325159

1. Temat projektu

Celem projektu jest utworzenie interpretera niestandardowego języka (własnego) z konkretnymi założeniami opisanymi poniżej.

2. Założenia funkcjonalne

- Język wspierał typy: całkowitoliczbowe, zmiennoprzecinkowe, ciągi znaków, logiczne oraz funkcje
- Język będzie umożliwiał dokonywanie operacji arytmetycznych dzięki operatorom dodawania (+), odejmowania (-), mnożenia (*) oraz dzielenia (/)
- Język będzie udostępniał operatory logiczne takie jak operator równości (==), operator różnicy (!=), operator “większy niż” (>), operator “mniejszy niż “(<), operat
- Język będzie udostępniał operator przypisania (=)
- Język będzie miał typowanie silne, dynamiczne
- Język będzie miał zmienne domyślnie mutowalne
- Język będzie wspierał przekazywanie argumentów przez kopię
- Język będzie wspierał instrukcje warunkowe if
- Język będzie wspierał instrukcję pętli while
- Język będzie wspierał operator konkatenacji (+)
- Język będzie wspierał przypisywanie funkcji do zmiennej oraz wywoływanie jej później z odpowiednimi argumentami
- Język będzie wspierał operator dekoracji funkcji (@@), który umożliwi zmianę sposobu działania funkcji dekorowanej
- Język będzie udostępniał operator kompozycji funkcji (|), który umożliwi tworzenie funkcji która, przekazuje argumenty wyjściowe jednej funkcji na wejście kolejnej
- Język będzie umożliwiał przykrywanie zmiennych globalnych przez zmienne lokalne oraz zmienne lokalne, przez inne znajdujące się w aktualnym zakresie (scope)
- Język będzie miał ograniczoną długość identyfikatorów (40 znaków)
- Język będzie ograniczał maksymalną wielkość zmiennych (od -2^32 do 2^32-1 dla int, 200 znaków dla string)
- Interpreter będzie wspierał wykrywanie oraz zatrzymywanie nieskończonych rekurencji
- Interpreter będzie wskazywał błąd w przypadku niezgodności typów
- Interpreter będzie umożliwiał sygnalizowanie błędów z komunikatem
- Interpreter będzie udostępniał wbudowaną funkcję print

3. Przykłady pokazujące funkcjonalności

- **Dozwolone typy zmiennych:**

```
var intiger = 2;
var floating_point = 3.5;
var string_var = “abcd”;
var my_bool = True;
var func_var = fun(var x) [ return x; ]
```

- **Operacje arytmetyczne**

```
var added  = 1 + 3;
var sub = 5 - 2 - 1 // Wynik to 2
var mult = 3 * 2;
var division = 6 / 2;
```

- **Operatory logiczne**

```
if(a > b || b == a && c != a && d < c && d >= q && q <= p)
```

- **Typowanie dynamiczne, silne**

```
var my_var = “abcd”;
my_var = 1; // dynamiczne
var my_float = 1.5;
my_float = my_float + (my_var as float); // typowanie silne
```

- **Domyślna mutowalność**

```
var my_var = 1;
const my_constant = 2; // jawnie zadeklarowana jawnie
```

- **Przekazywanie argumentów przez kopię**

```
fun example(var a)
[
    a = a + 1;
]

fun main()
[
    var a = 1;
    var my_fun = example;
    my_fun(a);
    // TEST(a == 1);
    return 0;
]
```

- **Obsługa instrukcji warunkowej if**

```
var a = 1;
var b = 2;
if (a < b)
[
    a = b +1;
```

```
]
```

- **Obsługa pętli while**

```
var a = 1;
while(a < 5)
[
        a = a + 1;
]
```

- **Obsługa kokatenacji**

```
var my_str = "Ania";
var my_str2 = "i Basia";
my_str = my_str+my_str2;
```

- **Wywołanie funkcji wewnątrz zmiennej**

```
fun example()
[
      return fun(var a, var b) [return a+b;];
]

fun main()
[
      var my_fun = example();
      my_fun(1, 2);
      return 0;
]
```

- **Dekorator funkcji**

```
var ident = fun(var x) [ return x; ]
var decorated = ident @@ fun(var f,var arg) [ return f(arg + 1); ]
// TEST(decorated(1) == 2)
```

- **Operator kompozycji funkcji**

```
var square = fun(var x) [ return x * x; ];
var increment = fun(var x) [ return x + 1; ];
var square_then_increment = double | increment;
// TEST(square_then_increment (2) == 5)
```

- **Przysłanianie zmiennych globalnych**

```
var glob = 1;
fun my_function()
[
        var glob = 1.4;
        // TEST(glob == 1.4)
]
```

- ***Ograniczona długość identyfikatorów***

```
var my_very_very_long_varrrrrrrrrrrrrrrrrrrrrrrrrrr = 10; // niepoprawne, ponieważ
                                  // długość nazwy zmiennej jest większa niż 40
```

- **Ograniczenie wielkości zmiennych**

var a = -2147483649 // Błąd
var a = 2147483649 // Błąd

- **Wykrywanie nieskończonych rekurencji**

fun recursion(var a)
[
return recursion(a); // spowoduje runtime error z poziomu
] // implementowanego interpretera

- **Wbudowana funkcja print**

string a = "abcd";
print(a);

4. Składnia języka

- Część składniowa:

```
Program            = { FunctionDeclaration | Declaration};
FunctionDeclaration = "fun", id, "(", [ Parameters ], ")",
                      StatementBlock ;
Parameters         = Parameter, {",", Parameter }
Parameter          = ("const var" | "var"), id ;
StatementBlock     = "[", Statement, { Statement }, "]" ;
Statement          = FunctionCall, ";"
                   | IfStatement
                   | Declaration, ";"
                   | ReturnStatement, ";"
                   | Assign, ";"
                   | WhileStatement
                   | FunctionCall ;
FunctionCall       =  Factor ;
CallArguments      = "(", [ ArgumentList ], ")" ;
ArgumentList       = Expression, { ",", Expression } ;
Expression         = SimpleExpression
                   | TypeCastExpression
                   | FunctionLiteral;
SimpleExpression   = Term {( "+" | "-" | "|" | "@@" ), Term} ;
Term               = Factor, { ( "*" | "/" ), Factor } ;
Factor             = BaseFactor, { CallArguments } ;
BaseFactor         = Number
                   | LiteralString
                   | id
                   | "(", Expression, ")"
                   | FunctionLiteral;
TypeCastExpression = Expression, "as", Type ;
FunctionLiteral    = "fun", "(", [ Parameters ], ")",
                      StatementBlock ;
```

```
IfStatement         = "if", "(", LogicalExpr, ")", StatementBlock,
                      ["else", StatementBlock] ;
LogicalExpr         = RelExpression, {LogicOperator, RelExpression};
RelExpression       = Expression, { RelOperator, Expression } ;
Declaration         = ("var" | "const var"), id, ["=", Expression] ;
ReturnStatement     = "return", ( Expression | FunctionLiteral );
Assign              = id, "=", Expression ;
WhileStatement      = "while", "(", LogicalExpr, ")",
                      StatementBlock ;
id                  = [Letter, "_"], { Letter | Digit | "_" } ;
Number              = Integer | Decimal ;
```

- Część leksykalna:

```
Integer             = (NonZeroDigit, { Digit } ) | "0";
Digit               = "0" | NonZeroDigit ;
NonZeroDigit        = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
                    | "9" ;
Decimal             = Integer, ".", Digit, { Digit } ;
Type                = "float", "int", "string" ;
Letter              = "A" | "B" | … | "Z" | "a" | "b" | … | "z" ;
LogicOperator       = "&&" | "||";
RelOperator         = "==" | "!=" | ">" | "<" | ">=" | "<=" ;
LiteralString       = '"', { StringCharacter }, '"' ;
StringCharacter     = EscapeSequence | AnyCharNoSpecial;
EscapeSequence      = "\\", ( "\\" | "\"" | "\'" | "n" | "t") ;
AnyCharNoSpecial    = ? Wszystkie znaki oprócz " i \ ? ;
```

5. Dozwolone operacje

Konwersje

int -> float (np. 12 -> 12.0)
int -> string
float -> string
float -> int (utrata precyzji np. 12.15 -> 12)

Operator +

int + int - operacja dodawania
float + float - operacja dodawania
string + string - konkatenacja dwóch ciągów znakowych (drugi “przyklejany do pierwszego)
int + float - niedozwolone (wymaga konwersji int na float lub na odwrót)
string + int - niedozwolone
string + float - niedozwolone

Operator -

int - int - operacja dodawania
float - float - operacja dodawania
int - float - niedozwolone (wymaga konwersji int na float lub na odwrót)
string - string - niedozwolone

Operator *

int * int - operacja mnożenia
float * float - operacja mnożenia
int * float - niedozwolone (wymaga konwersji int na float lub na odwrót)
Operator /
int / int - operacja mnożenia
float / float - operacja mnożenia
int / float - niedozwolone (wymaga konwersji int na float lub na odwrót)
Operator ==
int == int - operacja porównania liczb
float == float - operacja porównania liczb
string == string - porównanie wartości ciągów znakowych
Analogicznie dla operatora “!=”
Operator >
int == int - operacja porównania liczb
float == float - operacja porównania liczb
string == string - niedozwolone
Analogicznie dla “<”, “<=”, “>=”

6. Obsługa błędów

Błędy będą obsługiwane przez oddzielny moduł, który będzie informował programistę o typie błędu oraz gdzie się błąd znajduje. Przykładowy komunikat błędu:
“Missing “(“ character at line 7“
Moduł ten, będzie działał następująco:
Klasa ErrorHandler będzie posiadała strukturę:
string message;
int line;
int column;
W przypadku gdy któryś z modułów napotka błąd, to zostanie on zgłoszony do metody klasy ErrorHandler z odpowiednią instancją struktury (wiadomość, wiersz, kolumna). Metoda ta zgłosi błąd przy pomocy *throw* z odpowiednią wiadomością, przez co program zostanie zatrzymany.

7. Sposób uruchomienia

Program będzie budowany przy pomocy *make*, np.:
mkdir build
cd build
cmake ..
make

Program będzie uruchamiany następująco:
./bibl <nazwa_pliku>
Spowoduje do wczytywanie pliku znak po znaku do programu oraz analizowanie wczytanych znaków. Wynik programu będzie zwracany na wyjście standardowe konsoli po czym będzie można przekierować wynik do pliku przy pomocy przekierowania strumienia do pliku (znak

“>” na systemie Linux). Do testów natomiast program będzie używał łańcuchów znakowych string, ponieważ nie ma to wpływu na realizację leksera, parsera ani interpretera.

8. Opis realizacji

Program będzie zrealizowany z modułów:

- Analizator leksykalny
- Analizator składni
- Analizator semantyczny
- Interpreter drzewa
- Moduł obsługi błędów

Analizator leksykalny - pobiera kolejne pojedyncze znaki (nie bajty) z wejścia po czym grupuje je w tokeny oraz zgłasza błąd do modułu obsługi błędów.
Analizator składniowy (parser) - grupuje tokeny (utworzone przez lekser) w strukturę drzewa oraz sprawdza poprawność analizowanych tokenów. W przypadku napotkania błędów, zgłasza ten fakt do modułu obsługi błędów.
Analizator semantyczny - sprawdza poprawność drzewa (zbudowanego przez parser) oraz zgłasza błędy do modułu obsługi błędów.
Interpreter drzewa - Wykonuje operacje na podstawie drzewa stworzonego przez analizator składniowy oraz przeanalizowanego przez analizator semantyczny.
Moduł obsługi błędów - zostaje powiadomiony w przypadku wystąpienia błędu i wypisuje go na wyjście programu kończąc jego działanie

9. Opis testowania

Testowanie będzie się składało z:

- Testów jednostkowych - sprawdzenie modułów w odosobnieniu od innych.
  lekser - odpowiednio rozpoznaje i buduje tokeny, czy poprawnie zgłasza błędy
- Testów integracyjnych - sprawdzenie modułów w połączeniu z innymi modułami.
  parser - na podstawie weryfikuje otrzymane od leksera tokeny, z których buduje drzewo. W przypadku napotkania błędu, np. niespodziewanego tokenu, informuje moduł błędów
  analizator semantyczny - sprawdza, czy drzewo stworzone przez parser jest poprawne i w przypadku błędu informuje moduł obsługi błędów
- Testy end to end - interpreter drzewa, będzie testowany przy użyciu testów end to end, co oznacza, że zostanie uruchomiony na konkretnych plikach, a jego wyjście zostanie porównane do znanych wyników, które powinny wyjść przy wykonaniu tych plików

10. Bardziej skomplikowane przykłady testowe

```
const var glob = 1.3;
fun test_scope(var a, var b)
[
        var glob = 1;
        // TEST(glob = “1”)
```

```
        if (glob == 1)
        [
                var glob = “abc”;
                // TEST(glob == “abc”)
        ]
        // TEST(glob = 1)
        return glob;
]

fun generate()
[
        return fun inc(var x) [ return x + 1; ];
]

fun addition(var a, var b)
[
        return a + b;
]

fun  square(var a)
[
        return  a*a;
]

fun test_return(var a)
[
        if(a == 1)
        [
                print(a);
                return a;
        ]
        a = 999; // instrukcje po return się nie wykonują
        return a;
]

fun main()
[
        var outcome_gen = generate()(1);
        //TEST(outcome_gen == 2)
        var concatenatied = addition | square;
        var concatenatied_val = concatenatied(1, 3);
        //TEST (concatenatied_val = 16)
        var casted_add = 3 + (glob as int);
        //TEST (casted_add == 4)
```

```
        var decorated_fun =square @@ fun(f, a) [ return f(a)*2; ];
        var decorated_val = decorated_fun(glob);
        //TEST( decorated_val == 3.38)
]
```

Więcej przykładowych plików testowych znajduje się w katalogu test_cases.