

Techniki kompilacji 103D-INIOP-ISP-TKOM		
Temat: Język wspierający wybrane operacje finansowe i walutowe	Wykonał: Ciarka Jakub 270849	
Prowadzący:	Data: 16.01.2023	Ocena:

Spis treści

1.	Projekt wstępny	3
1.1.	Cel i zakres projektu	3
1.1.1.	Temat projektu.....	3
1.1.2.	Wymagania funkcjonalne.....	3
1.2.	Specyfikacja języka	4
1.2.1.	Założenia ogólne	4
1.2.2.	Założenia wstępne dla składni języka	4
1.2.3.	Typowanie w języku	5
1.2.4.	Typy oraz wspierane przez nie operacje	5
1.2.5.	Organizacja dostępu do zmiennych, czas życia obiektów i uchwytów do obiektów	7
1.2.6.	Przekazywanie zmiennych do funkcji, zwracanie wyników z funkcji	8
1.2.7.	Pętle	8
1.2.8.	Operacje warunkowe	9
1.2.9.	Konfiguracja typów walutowych.....	9
1.2.10.	Pozostałe założenia dla języka	10
1.2.11.	Wynikowa gramatyka języka.....	10
1.3.	Architektura narzędzia	12
1.3.1.	Lekser	12
1.3.2.	Parser	13
1.3.3.	Interpreter.....	14
1.4.	Uruchomienie	14
1.5.	Testowanie	14
2.	Dokumentacja końcowa.....	15
2.1.	Podsumowanie implementacji	15
2.2.	Budowanie wersji produkcyjnej oprogramowania	15
2.3.	Uruchomienie programu.....	16

2.4.	Przykłady z działania programu (testowy plik źródłowy).....	16
2.4.1.	Deklarowanie funkcji i rekurencja i operacje arytmetyczne.....	16
2.4.2.	Pętle	17
2.4.3.	Obiektowość i operacje finansowe	18
2.4.4.	Obiektowość, kolekcje i operacje na kolekcjach.....	19
2.4.5.	Obsługa błędów	20

1. Projekt wstępny

1.1. Cel i zakres projektu

1.1.1. Temat projektu

Tematem projektu jest stworzenie języka dostarczającego mechanizmy wspierające obsługę walut oraz operacji finansowych z uwzględnieniem przewalutowań.

1.1.2. Wymagania funkcjonalne

Poniżej przedstawiono listę wymagań omówionych z prowadzącym projekt na spotkaniu konsultacyjnym z 27.10.2022. Na spotkaniu omówiony został cel, temat i zakres projektu. Ustalone zostały następujące wymagania:

- Język powinien dostarczać podstawowe możliwości *typowego* języka programowania. Ponadto powinien zostać wyposażony w operacje specyficzne dla operowania na wartościach pieniężnych w różnych walutach.
- Język powinien dostarczać możliwość konfiguracji w zakresie obsługiwanych walut oraz stawek transferów między walutami. Stawki przewalutowań mogą różnić się w zależności od kierunku przewalutowania.
- Wynikiem projektu powinno być narzędzie pozwalające na interpretację (wykonanie) kodu źródłowego dostarczonego w formie pliku (oprócz kodu źródłowego wymagany plik z konfiguracją). Na potrzeby testowania jednostkowego i integracyjnego projekt powinien zawierać moduły pozwalające na wprowadzanie konfiguracji i kodu źródłowego w postaci obiektów tekstowych zamiast plików na dysku.
- W języku powinny zostać dostarczone mechanizmy pozwalające na wygodne dla użytkownika realizowanie podstawowych operacji finansowych takich jak:
 - zwiększenie stanu konta o zadaną kwotę,
 - zwiększenie stanu konta o zadany procent,
 - zmniejszenie stanu konta o zadaną kwotę,
 - zwiększenie stanu konta o zadany procent,
 - przelew zadanej kwoty między kontami,
 - przelew między kontami odpowiadający procentowi z pierwszego konta.

Opisane operacje powinny umożliwiać automatyczne przewalutowania w zależności od zadanych danych wejściowych.

- Powinien zostać zaimplementowany mechanizm obsługi kolekcji kont. Kolekcje powinny oferować typowe dla tego typu operacje, takie jak:
 - Dodawanie nowych elementów,
 - Usuwanie elementów,
 - Dostęp indeksowy,
 - Modyfikacje obiektu pod indeksem ,
 - Przeszukiwanie ze względu na warunek dostarczany jako funkcja lambda.
- Poza mechanizmami specyficznym dla operacji walutowych język powinien dostarczać możliwość typowe dla języka imperatywnego w tym:
 - Operowanie na zmiennych logicznych, całkowitych oraz niecałkowitych,
 - Instrukcje warunkowe,
 - Instrukcje pętli,
 - Możliwość deklaracji i wywoływania funkcji.

1.2. Specyfikacja języka

1.2.1. Założenia ogólne

- Projekt będzie składał się z implementacji leksera (wspólnego dla kodu źródłowego i konfiguracji), parsera konfiguracji, parsera kodu źródłowego oraz interpretera.
- Wszystkie elementy projektu (lekser, parser, interpreter) zostaną napisane w języku C#.
- Do uruchomienia programu będzie należało dostarczyć dwa pliki: plik z konfiguracją zmiennych walutowych oraz plik z kodem źródłowym programu do interpretacji.

1.2.2. Założenia wstępne dla składni języka

- Instrukcje w projektowanym języku będą kończyły się znakiem średnika.
- Instrukcje będą organizowane w bloki ograniczane znakami nawiasów klamrowych. W ramach bloków interpreter będzie tworzył nowe zakresy zmiennych, ale będzie możliwy dostęp zakresów bloków obejmujących dany blok,
- Język będzie pozwalał definiowanie oraz rekurencyjne wywołania funkcji. Głębokość rekurencji będzie ograniczona dostępną wielkością stosu programu interpretera,
- Zakłada się ograniczenie na maksymalną ilość argumentów funkcji równą 255 elementów,
- Zakłada się ograniczenie na maksymalną długość identyfikatorów równą 255 znaków,
- W skład identyfikatorów mogą wchodzić jedynie litery małe i duże ze zbioru ASCII, cyfry (na wszystkich pozycjach oprócz pierwszej) oraz znak podkreślenia "_",
- Komentarze jednolinijkowe będą rozpoczynały się parą znaków "//",
- Stringi będą mogły zawierać znaki specjalne oraz być escapowane znakiem "\",

Ponadto zakłada się, że kod źródłowy będzie dostarczany w pojedynczym pliku. W pliku tym będzie można wprowadzać definicje funkcji. Jedną z funkcji o nazwie *main* będzie funkcja startową dla przetwarzania.

Przykład pliku źródłowego

```
int nwd(int a, int b)
{
    if(a==b)
    {
        return a;
    }
    if (a > b)
    {
        return nwd(a-b, b); # wywołania rekurencyjne
    }
    return nwd(b-a);
}

int factorial(int a)
{
    if(a == 1)
    {
        return 1;
    }

    return a * factorial(a-1);
}

void main()
```

```

{
    int x = 3;
    int y = 10;

    print("NWD 3 i 10 wynosi = ", nwd(x, y)); # NWD 3 i 10 wynosi = 1
    print("Silnia 3 wynosi = ", factorial(x, y)); # Silnia 3 = 6
}

```

1.2.3. Typowanie w języku

Typowanie w języku będzie statyczne i silne. Operacje wykonywane na zmiennych różnych typów wymagać będą jawnej konwersji (operator `to`). Podobnie operacje na walutach (typy USD, EUR, PLN...) będą wymagały jawnej konwersji (realizującej przewalutowanie).

Przykładowo poniższe operacje są poprawne:

```

decimal a = 123.1;
int b = 12;
var c = a + b to decimal; # konwersja ma wyższy priorytet
                          # niż operacje arytmetyczne i logiczne

```

a poniższa operacja jest niepoprawna ze względu na niezgodność typów:

```
var c = a + b; # dodanie int i decimal
```

Podobnie dla walut poniższe operacje wykonują jawna konwersję między walutami:

```

USD a = 123USD;      # wartość zakończona nazwą typu stanowi literał
                     # danej waluty
PLN b = 12PLN;
var c = a + b to USD; # wynik jest zmienną w typie USD

```

a poniższa operacja jest niepoprawna ze względu na niezgodność typów:

```
var c = a + b;
```

Jedynym wyjątkiem od typowania silnego będą operacje na klasie `Account`, dla którego dostarczone zostaną dedykowane operacje walutowe realizujące niejawnie przewalutowania.

```

var a = Account(USD, 123);
var b = Account(PLN, 123);
a >> 100CHF; # zmniejszenie stanu konta (niejawna konwersja CHF do USD)
b << 100CHF; # zwiększenie stanu konta (niejawna konwersja CHF do PLN)
a >> 100CHF >> b; # przelew między kontami
                  # (niejawna konwersja CHF do USD i CHF do PLN))

```

Zakłada się dostępność słowa kluczowego `var`, który stanowi dowolny typ dedukowany z wyrażenia przypisującego wartość. Typ zostanie wydedukowany w czasie parsowania (mechanizm analogiczny do `var` w C#, czy Java w wersji 10 lub wyższej). Przykładowo:

```

var a = 124.12;      # decimal
var b = 123D;        # litera d w literale wskazuje na typ decimal
var c = 123;         # int
var d;               # niedopuszczalne (nie można wywnioskować typu)

```

1.2.4. Typy oraz wspierane przez nie operacje

W języku dostępne będą następujące typy:

- **Typy proste** (*bool, int, decimal*),
Typy walutowe (*PLN, USD, CHF....*): ich ilość oraz nazwy zależne będą od zawartości pliku konfiguracyjnego. Plik konfiguracji będzie przetwarzany przed plikiem kodu źródłowego. Efekt przetwarzania konfiguracji będzie rozbudowywał zbiór typów języka o zdefiniowane typy walut. Ponadto przewiduje się możliwość używania literałów tych typów. Literał kwoty w danej walucie będzie zakończony nazwą typu waluty np. 100.15USD, 12.75PLN. Dla typów walutowych zostaną zdefiniowane podstawowe operacje arytmetyczne takie jak dodawanie i odejmowanie. Ponadto zakłada się możliwość mnożenia z typami liczbowymi *int* i *decimal*.
- **Klasa typu** (*Type*): będzie to klasa pomocnicza pozwalająca przetwarzać typ języka. Dzięki temu możliwe będzie zdefiniowanie wyrażenia zwracającego typ:

```
var currencyType1 = EUR;
EUR amount = 100.25 to currencyType1;
    # możliwy wyjątek typu runtime przy niezgodności typu

var currencyType2 = Currency("USD");
    # zakłada się możliwość dostarczenia funkcji natywnej
    # która parsowałaby string do typu Type
USD amount = 50 + 50 to currencyType2;
```

- **Klasa konta** (*Account*): będzie to typ na którym będą zdefiniowane będą operacje specyficzne dla walut. Operacje te realizować będą niejawne konwersje walut. Przykładowe operacje walutowe to:

```
var a = Account<PLN>(100000);

# Zmniejszenie stanu konta o wartość
a >> 100 CHF;
a >> 100 to CHF;

# Zmniejszenie stanu konta o procent
a %> 0.1;

# Zwiększenie stanu konta o wartość
a << 100 CHF;
a << 100 to CHF;

# Zwiększenie stanu konta o procent
a <% 0.1;

# Przelew między kontami na wartość
a >> 100 CHF >> b;
a >> 100 to CHF >> b;

# Przelew między kontami na procent pierwszego z kont
a %> 0.1 >> b;

# wszystkie opisane powyżej operacje zwracaj obiekt Account
# zawierający informację o kwocie operacji pieniężnej
print(amount.Value); # 1000
print(amount.Currency); # PLN

# Obliczenie procentowej wartości z konta
decimal prctAmount = a % 0.1;
decimal prctAmount = a % getPrct();
```

Klasa walutowa ponadto zawiera pola, które opisują jej cechy:

```
var a = Account<PLN>(100000);
print(a.Value) # 1000000 (typ decimal)
print(a.Currency) # PLN (typ Type)
```

- **Klasa kolekcji** (*Collection of type*): będzie klasą generyczną pozwalającą na przechowywanie wartości zadanego typu:

```
# Tworzenie kolekcji
var accounts = Collection<Account<PLN>>();
```

zaimplementowane zostaną typowe dla kolekcji operacje:

```
# Dodawanie
Accounts.Add(Account<PLN>(120));
Accounts.Add(Account<PLN> (10));

# Dostęp indeksowy
var account = Accounts[1];
PLN balance = Accounts[1].Balance;

# Modyfikacje na indeksie
accounts[1] = Account<USD>(0);
accounts[1].name = "TESTOWE";

# Usuwanie
Accounts.Delete(1);

# Przeszukiwanie z delegatem
Account<USD> first = Accounts.First(x => x.Ballance > 10000);
Account<USD> last = Accounts.Last(x => x.Ballance > 10000);
Collection<Account> usdAccounts = Accounts.Where(
    x => x.Ballance > 10000);
```

1.2.5. Organizacja dostępu do zmiennych, czas życia obiektów i uchwytów do obiektów
Zakłada się że zmienne będą uchwytami do obiektu (jak zmienne referencyjne w języku C#).

Obiekty są niemutowalne względem operacji arytmetycznych i logicznych. Przykładowo operacja:

```
int a = 1;
a = a + 1;
```

tworzy nowy obiekt o wartości $a + 1$ i przypisuje go do uchwytu o nazwie a .

Obiekty są mutowalne względem operacji finansowych. Przykładowo operacja:

```
myAccount >> 100 PLN;
```

Zmienia stan konta obiektu *myAccount*.

Obiekty są mutowalne względem operacji na polach obiektów oraz zmian realizowanych przez metody:

```
account.name = "Nowa nazwa"; # zmiana wartość pola obiektu account
```

Zmienne typów prostych i walutowych będą tworzone przez przypisanie wartości z literału lub wyniku ewaluacji wyrażenia. W przypadku typów obiektowych konieczne będzie jawne wywołanie konstruktora:

```
var konto = Account<PLN>(120000);
```

Istotnym zagadnieniem jest sposób niszczenia obiektów. W tym celu wykorzystanie zostanie mechanizm *Garbage Collector* ukryty w maszynie wirtualnej C# realizującej kod interpretera. Dzięki niemu po utracie wszystkich referencji do zbioru obiektów w pamięci, będą one cyklicznie usuwane ze sterty programu interpretera. Rozwiązanie to pozwoli tworzyć obiekty w funkcji i zwracać je jako wynik bez zagrożenia wycieków pamięci.

Czas życia zmiennych będzie sięgał jedynie do końca bloku, w którym zostały zdefiniowane. Każdy blok będzie posiadał dedykowany słownik przypisań (*nazwa uchwytu, obiekt*) oraz będzie wskazywał na słownik zakresu rodzica. Po wyjściu z zakresu odniesienie do słownika zostanie zapomniane, a jego zawartość obsłużona przez *Garbage Collector*.

1.2.6. Przekazywanie zmiennych do funkcji, zwracanie wyników z funkcji

Domyślnie zmienne są przekazywane do funkcji przez referencję. Zachowanie zmiennej przekazanej przez referencję będzie wzorowane na językach C#/Java. Operacje modyfikujące obiekt przekazany przez argument (wykonane w funkcji) będą widoczne po opuszczeniu funkcji. Efekt operacji przypisania nie będzie propagował poza funkcję

Przykładowo wywołanie poniższej funkcji nie spowoduje zmiany wartości obiektu *a* w zakresie wywołującym funkcję:

```
void test(int a)
{
    a = 5; # przypisanie nowego obiektu do kopii uchwytu
}

int a = 1
test(a);
print(a); # 1
```

Zakłada się definicję dla wszystkich typów bezargumentowej metody *Copy*, dzięki której będzie można wykonać głęboką kopię obiektu. Wykonując taką kopię przed przekazaniem obiektu jako argument wywołania funkcji możemy uzyskać efekt taki, jak przy przekazaniu zmiennej przez wartość. Metodę *Copy* można również wykorzystać w dowolnym innym wyrażeniu kodu źródłowego.

```
var a = Account(PLN);

doSomething(a.Copy()); # funkcja wykonuje logikę na kopii obiektu
```

1.2.7. Pętle

Zakłada się implementacji pętli *while* oraz *foreach*. Przykładowe wykorzystanie pętli *while* może mieć następującą postać:

```
# pętla while
int a = 1;

while(a < 100)
{
    a = a + 1;
```



```

}
print(a); # 100

# pętla foreach

Collection<int> col;
col.Add(1);
col.Add(10);
col.Add(5);

foreach(int b in col)
{
    print(b);
}

```

1.2.8. Operacje warunkowe

Zakłada się implementację instrukcji warunkowej *if* z opcjonalnym członem *else*.

W przypadku kaskady wywołań *if* człon *else* zawsze dotyczy najbliższego mu członu *if*. Przykładowo dla poniższego wywołania:

```

if(a == 1) print("1");
if(a == 2) print("2");
if(a == 3) print("3");
else print("1");

```

else dotyczy członu *if* z warunkiem *a == 3*.

1.2.9. Konfiguracja typów walutowych

Język będzie dostarczał możliwość konfiguracji dostępnych walut oraz kwot przewalutowań za pomocą niezależnego pliku konfiguracyjnego. Poniżej przedstawiono przykładową zawartość takiego pliku konfiguracyjnego:

	USD	CAD	EUR	GBP	HKD	CHF	JPY	AUD	INR	CNY;
USD	1	1.36	1.01	0.87	7.85	1	148.74	1.56	82.73	7.3;
CAD	0.73	1	0.74	0.64	5.77	0.74	109.27	1.15	60.78	5.36;
EUR	0.99	1.35	1	0.86	7.76	0.99	147.04	1.54	81.78	7.22;
GBP	1.15	1.56	1.16	1	9	1.15	170.57	1.79	94.87	8.37;
HKD	0.13	0.17	0.13	0.11	1	0.13	18.95	0.2	10.54	0.93;
CHF	1	1.36	1.01	0.87	7.84	1	148.5	1.56	82.6	7.29;
JPY	0.01	0.01	0.01	0.01	0.05	0.01	1	0.01	0.56	0.05;
AUD	0.64	0.87	0.65	0.56	5.03	0.64	95.34	1	53.03	4.68;
INR	0.01	0.02	0.01	0.01	0.09	0.01	1.8	0.02	1	0.09;
CNY	0.14	0.19	0.14	0.12	1.07	0.14	20.37	0.21	11.33	1;

Formatowi pliku konfiguracyjnego stawia się następujące wymagania:

- Symbole i wartości w poszczególnych wierszach rozdzielone są dowolną ilością białych znaków. Wiersze kończą się symbolem ``
- Pierwszy wiersz stanowią rozdzielone białymi znakami symbole walut. Symbolowe walut mogą składać się z dowolnych trzyliterowych zbiorów znaków literowych ASCII. Symbol waluty jest niewrażliwy na wielkość liter, co oznacza że symbole USD, usd i Usd będą sobie równoważne.
- Kolejne wiersze zaczynają się symbolem waluty, do której się odnoszą. Kolejne elementy wiersza to kwoty przewalutowań z danej waluty do waluty, której odpowiada dana kolumna.

Należy podać kwoty przewalutowań dla wszystkich zdefiniowanych walut. W kolumnie odpowiadającej tej samej walucie np. PLN -> PLN musi pojawić się kwota równa 1. W przeciwnym razie zgłaszany będzie błąd semantyczny.

- Kwoty przewalutowań nie muszą być zwrotne, np. przewalutowanie PLN -> USD -> PLN nie musi dać takiej samej kwoty jak przed przewalutowaniem.
- Kolejność deklaracji przewalutowań nie musi odpowiadać kolejności identyfikatorów walut nagłówek. Musi natomiast pojawić się dokładnie jeden wiersz dla każdej waluty (o tym której waluty dotyczy wiersz świadczy identyfikator stanowiący pierwszy token danego wiersza).
- Żaden z typów walutowych nie może mieć nazwy takiej samej jak słowa kluczowe dostępne w języku. Przykładowo niedopuszczalna nazwą dla waluty jest INT, AND etc.

Powyższym założeniom odpowiadana następująca gramatyka:

```

Configuration      = header { row } ;
Header             = { currencyType } `;` ;
Row               = currencyType { currencyConversionAmount } `;` ;
currencyType      = letter letter letter;
Letter            = `a` | `b` | ... | `z` | `A` | ... | `Z` ;
currencyConversionAmount = digitExcludingZero { digit } [`.`{ digit }];
digitExcludingZero = `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` ;
digit             = `0` | diginumbert excluding zero ;

```

Identyfikator waluty w pliku walutowym stanie się typem walutowym dostępnym w języku.

1.2.10. Pozostałe założenia dla języka

- Wiele języków wspiera składnię kaskadowego przypisania np. $x = y = z = 5$; . W projektowanym języku świadomie rezygnuje się z tej składni, ze względu, że często skutkuje ona nieczytelnymi konstrukcjami. Innym przykładem nieczytelnych konstrukcji znanych np. z języka C jest np. przepisanie wewnątrz warunku w instrukcji warunkowej. Zakłada się że operacja przypisania nie zwraca wartości.

•

1.2.11. Wynikowa gramatyka języka

Na podstawie opisan matyką:

```

program           = { funtionDecl } ;
funtionDecl       = functionReturnType indetifier `(` parameters `)` block ;

anyStmt           =
    exprStmt      |
    ifStmt        |
    returnStmt    |
    whileStmt     |
    foreachStmt   |
    declarationStmt |
    fincancialFromStmt |
    fincancialToStmt |
    block ;

exprStmt          = expression [= expression] `;`
ifStmt            = `if` `(` expression `)` anyStmt [ `else` anyStmt ] ;
returnStmt        = `return` expression `;`
whileStmt         = `while` `(` expression `)` anyStmt;
foreachStmt       = `foreach` `(` ( `var` | anyType )

```

```

                                identifier in expression `)` anyStmt ;
declarationStmt      = (anyType | `var`) identifier `=` expression `;` ;
fincancialToStmt    = expression [ ( `<<` | `<%` ) expression ] `;` ;
fincancialFromStmt  = expression [ ( `>>` | `%>` ) expression
                                [ `>>` objectExpr ] `;` ;

block                = `{` { anyStmt } `}` ;

expression           = orExpr ;
orExpr              = andExpr { `or` andExpr } ;
andExpr             = comparativeExpr { `and` comparativeExpr } ;
comparativeExpr      = additiveExpr { ( `!=` | `==` | `>` | `>=` | `<` | `<=` )
                                additiveExpr } ;

additiveExpr         = multiplicativeExpr { ( `+` | `-` ) multiplicativeExpr } ;
multiplicativeExpr   = negatonExpr { ( `*` | `/` ) negatonExpr } ;
negatonExpr          = [ `!` | `-` ] conversionExpr ;
conversionExpr        = conversionExpr [ `to` expression ] ;
prctOfExpr           = objectExpr [ % expression ] ;
objectExpr           = term {
                                `.` identifier [ `( ` arguments `)` ] |
                                [ expression ]
                                };

term                 = identifier
                    |
                    functionCallExpr
                    |
                    anyType
                    |
                    constructorCallExpr
                    |
                    anyLiteral
                    |
                    `( ` expression `)` ;

functionCallExpr     = identifier `( ` arguments `)` ;
constructorCallExpr  = anyType `( ` arguments `)` ;

functionReturnType    = anyType | `void` ;
anyType              = listType | complexType | basicType | currencType ;
listType             = `Collection` `<` anyType `>` ;
complexType          = `Account` | `Currency` | `Type` ;
basicType            = `bool` | `int` | `decimal` ;
currencyType         = LISTA DOSTARCZANA Z ZEWNĄTRZ (plik konf.) ;
arguments            = [ argument { `,` argument } ] ;
argument             = expression | lambda ;
lambda               = `lambda` [anyType] identifier `=>`(expression | block) ;
parameters           = [ parameter { `,` parameter } ] ;
parameter            = anyType identifier ;
anyLiteral           = textLiteral | booleanLiteral | numericLiteral [
                                currencyType ] ;

nullLiteral          = `null` ;
textLiteral          = `"` { DOWOLNY ZNAK UNICODE } `"` ;
booleanLiteral        = `true` | `false` ;
currencyLiteral       = digitExcludingZero { digit } [ `.` { digit } ]
                                currencType ;

nummericLiteral       = digitExcludingZero { digit } [ `.` { digit } ] [ `D` ] ;
intlLiteral          = digitExcludingZero { digit } ;
identifier            = alpha { alpha | digit } ;
alpha                = letter | `_` ;
letter               = `a` | `b` | ... | `z` | `A` | ... | `Z` ;
digitExcludingZero    = `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` ;
digit                = `0` | digitExcludingZero ;

```

1.3. Architektura narzędzia

Projektowane narzędzie będzie można w sposób logiczny podzielić na trzy części:

- Lekser – jego zadaniem będzie analiza tekstu wejściowego, wyodrębnianie z niego zdefiniowanych w specyfikacji języka różnych rodzajów tokenów. Tokeny będą udostępniane wyższym warstwom przez interfejs strumieniowy. Parser będzie również wykrywał i zgłaszał błędy składniowe (np. nierozpoznany token, zbyt długi identyfikator, przepełnienie zakresu zmiennej),
- Parser – jego zadaniem będzie zbudowanie na podstawie strumienia tokenów drzewa odpowiadającego rozbiorowi kodu zgodnemu z algorytmem rekursywnie zstępującego parsowania. Ponadto zadaniem parsera będzie analiza semantyczna kodu źródłowego, w tym m. in. sprawdzenie zgodności typów zwracanych przez wyrażenia z oczekiwanym typem w wynikowym miejscu ich ewaluacji oraz określanie typów dedukowanych odpowiadającym definicji zmiennej ze słowem kluczowym `var`.
- Interpreter – jego zadaniem będzie wykonanie kodu źródłowego z wykorzystaniem mechanizmów udostępnianych przez środowisko wykonawcze platformy .NET (do obsługi wejścia – wyjścia oraz przechowywania zmiennych).

1.3.1. Lekser

Na potrzeby projektu zostanie przygotowany wspólny lekser na potrzeby analizy leksykalnej zarówno pliku konfiguracyjnego oraz pliku z kodem źródłowym.

Projekt zakłada, że lekser będzie korzystać z dodatkowej warstwy abstrakcji nad źródłem kodu do analizy. Warstwę tą stanowić będzie interfejs udostępniający zestaw operacji pozwalających na odczytywanie kolejnych znaków oraz odczytywanie pozycji w tekście. Planuje się dostarczenie dwóch implementacji. Pierwsza z nich będzie pozwalała na odczyt danych z pliku, a więc będzie przeznaczona do docelowej pracy narzędzia. Druga pozwoli na czytanie kodu ze zmiennej tekstowej i będzie przeznaczona głównie na potrzeby testowania.

Lekser będzie definiował interfejs umożliwiający operacje charakterystyczne dla strumieni, w tym m.in.:

- Pobranie obecnego tokena,
- Sprawdzenie kolejnego tokena bez jego konsumpcji,
- Sprawdzenie n kolejnych tokenów bez ich konsumpcji. Idea działania tej metody nie narusza założenia leniwego budowania tokenów i będzie zakładać budowę minimalnej liczby tokenów w przód a następnie odkładanie ich w kolejce do czasu ich konsumpcji.
- Sprawdzenie poprzedniego tokena,
- Sprawdzenie m -tego poprzedniego tokena.

Tokeny zawierać będą następujące informacje:

- Typ tokenu,
- Wartość tokenu – niezależne zmienne wartości dla poszczególnych typów (`bool`, `int`, `decimal` i `string`),
- Wartość typu (w przypadku literałów będzie to informacja jakiego typu w języku jest wartość reprezentowana przez token, a więc który typ reprezentacji wartości ma poprawną wartość),
- Numer linii kodu źródłowego, w której znajduje się token,
- Pozycję pierwszego znaku tokenu w linii kodu źródłowego,
- Pozycję pierwszego znaku tokenu w strumieniu wejściowym,

- Pozycję w strumieniu wejściowym pierwszego znaku linii, w której znajduje się token (na potrzeby obsługi błędów – pozwoli na łatwe wyświetlenie całej linii tekstu)

Lekser może zwracać następujące błędy:

- Nieoczekiwany znak (znak nie pojawiający się w definicji języka),
- Nieprawidłowy token (wewnątrz tokenu pojawia się niewłaściwa konfiguracja znaków – np. nie zamknięta znakiem cudzysłowu stała tekstowa),
- Przekroczenie dopuszczalnej liczby znaków identyfikatora,
- Przepełnienie stałej liczbowej.

Na lekser będzie można nakładać filtry implementujące identyczny interfejs co sam lekser, które pozwolą na dodatkowe modyfikacje strumienia tokenów. Przewiduje się m.in. implementację filtra do usuwania komentarzy.

Utworzenie obiektu klasy leksera będzie umożliwiało dostarczanie opcjonalnej klasy konfiguracji (umożliwiającej zmiany względem konfiguracji domyślnej). Obiekt opcji pozwoli na dostarczenie zestawu typów walutowych pochodzących z uprzednio przetworzonego pliku konfiguracyjnego. Przy jego pominięciu typy walutowe będą ignorowane. Dzięki temu możliwe będzie wykorzystanie tego samego leksera do odczytu konfiguracji, przy czym podczas przetwarzania konfiguracji typy walutowe będą traktowane jako identyfikatory, a kwoty przewalutowań jako literały liczbowe. Ponadto obiekt opcji pozwoli na modyfikowanie ograniczeń względem maksymalnej długości identyfikatorów oraz podobnych cech leksera.

1.3.2. Parser

Zadaniem parsera będzie zbudowanie na podstawie strumienia tokenów drzewa odpowiadającego rozbirowi zgodnemu z algorytmem rekursywnie zstępującego parsowania. Zadanie to będzie realizowane przez klasę *Parser*. Klasa ta będzie zawierała zestaw prywatnych funkcji pomocniczych, gdzie każda będzie tworzyła odpowiadające jej obiekty drzewa rozbioru lub rekursywnie wywoływała kolejną metodę wg reguł precedensu definiowanych przez gramatykę. Efektem parsowania będzie zwrócenie obiektu korzenia drzewa.

Struktura drzewa rozbioru kodu źródłowego będzie wymagała dostarczenia zestawu obiektów odpowiadających możliwym węzłom tego drzewa. Obiekty te w zależności od elementu języka będą zawierały następujące informacje:

- Wskaźniki na węzły potomne,
- Wartość (dla literałów),
- Typ zwracanej wartości – do kontroli poprawności typowania oraz dedukcji typu dla słowa kluczowego `var`,
- Operator dla (operacji arytmetycznych i logicznych),
- Obiekty potomne w strukturze drzewa rozbioru,

Parser może zwracać wiele różnych rodzajów błędów w tym m.in.:

- Nieoczekiwany token,
- Niezgodność typów.

W przypadku natrafienia na błąd parser będzie synchronizowany do kolejnej instrukcji (rozpoznawanej przez token średnika).

Zakłada się możliwość kontynuacji przetwarzania do osiągnięcia z góry zadanej liczby błędów powyżej której, błędy kaskadowe będą stanowiły istotną część kolejnych problemów.

1.3.3. Interpreter

Zadaniem interpretera będzie wykonanie operacji opisywanych przez drzewo wygenerowane przez parser. Interpreter będzie powiązany z obiektami drzewa rozbioru przez wzorzec wizytatora. Poszczególne klasy węzłów drzewa będą implementowały generyczną metodę *visit*, gdzie typ generyczny będzie definiował typ wartości zwracanej. Metoda *visit* wywoływać będzie odpowiednią metodę klasy podanego jako argument wizytatora (w tym przypadku interpretera).

Podobnie do interpretera do struktury drzewiastej mogą być podpinane również inne narzędzia w np. realizujące analizę leksykalną.

Parser będzie mógł generować błędy typu *runtime*, w tym m.in.:

- Brak możliwości konwersji między typami (operator *to*),
- Odwołanie do nieistniejącej zmiennej.

Zmienne i zakresy realizowane będą obsługiwane za pomocą słowników powiązań (*nazwa uchwytu, obiekt*). Każdy blok będzie miał do użytku dedykowany słownik, który dodatkowo będzie zawierał referencje do słownika zakresu obejmującego rozważany blok (szczegóły mechanizmu opisane w sekcji 1.2.5).

Czas życia obiektów i uchwytów do obiektów został opisany w 1.2.5. Zakłada on wykorzystanie przez interpreter dedykowanego dla platformy .NET mechanizmu *Garbage collector*.

1.4. Uruchomienie

Przewiduje się obsługę programu przez interfejs wiersza poleceń. Jako argumenty wywołania będzie należało podać lokalizację pliku źródłowego oraz pliku z konfiguracją. Prawdopodobnie zaproponowane zostaną również dodatkowe flagi serujące pozwalające m.in. na określenie liczby błędów skutkującej definitywnemu przerwaniu prasowania lub inne zmiany w zachowaniu projektowanego narzędzia.

1.5. Testowanie

Każdy z modułów wyposażony zostanie w zestaw testów jednostkowych chroniących przed regresją przy ewentualnych późniejszych zmianach.

Przygotowany zostanie również zestaw testów integracyjnych sprawdzających działanie jednocześnie kilku modułów narzędzia.

Testy jednostkowe i integracyjne zostaną przygotowane z wykorzystaniem biblioteki xUnit.

Zaproponowane zostaną również przykładowe scenariusze testów manualnych stanowiące propozycję testów akceptacyjnych projektowanego narzędzia.

2. Dokumentacja końcowa

2.1. Podsumowanie implementacji

Projekt wraz z testami zawiera ok. 12000 linii kodu. Do implementacji przygotowano testy jednostkowe uwzględniające ponad 340 przypadków.

Kod źródłowy zawiera dwa projekty. Pierwszy z nich o nazwie *Application* stanowi implementację logiki biznesowej. Został on wewnętrznie podzielony na kilka katalogów:

- *Infrastructure* – w którym znajdują się klasy zawierające logikę biznesową w tym poszczególnych warstw leksera, parsera konfiguracji, parsera plików źródłowych, logiki do analizy statycznej i interpretera.
- *Models* – zawierające klasy stanowiące kontenery na dane w tym klasy reprezentujące poszczególne elementy gramatyki, klasy błędów, klasy reprezentujące wartości programowanego języka.
- *TestFiles* – katalog zawierający pliki testowe,
- *Examples* – katalog zawierający kod wykorzystywany na potrzeby testowania manualnego w trakcie implementacji.

W drugi z projektów umieszczonych w repozytorium znajdują się testy jednostkowe poszczególnych elementów logiki biznesowej. Przygotowano zestawy testów dla poszczególnych elementów:

- Warstwy wejściowej leksera,
- Logiki leksera,
- Filtru wyjściowego leksera do pomijania komentarzy,
- Parsera plików konfiguracyjnych,
- Parsera plików źródłowych,
- Logiki do analizy statycznej,
- Interpretera

2.2. Budowanie wersji produkcyjnej oprogramowania

Wersję produkcyjną projektu można zbudować za pomocą narzędzi graficznych Visual Studio lub za pomocą narzędzia *dotnet tools*, dostępnego z poziomu wiersza poleceń.

W przypadku visual studio należy kliknąć prawym przyciskiem myszy na projekt w eksploratorze plików solucji, następnie wybrać opcję *publish*. W oknie dialogowym należy wskazać sposób publikacji jako *folder* oraz wprowadzić pożądaną lokalizację. Po zatwierdzeniu we wskazanej lokalizacji wygeneruje się zestaw plików wśród których będzie jeden plik z rozszerzeniem *.exe* pozwalający na uruchomienie programu.

W przypadku *dotnet tools* należy otworzyć wiersz poleceń w katalogu projektu. Następnie należy wprowadzić komendę:

```
dotnet publish --output <OUTPUT_DIRECTORY>
```

której użycie będzie miało identyczny efekt jak w przypadku narzędzia *publish* w visual studio.

Na potrzeby testów program można oczywiście uruchomić również z poziomu visual studio lub dowolnego innego IDE pozwalającego na pracę z platformą .NET.

2.3. Uruchomienie programu

Program przyjmuje dwa argumenty wejściowe:

- Ścieżkę do pliku źródłowego (obowiązkowy),
- Ścieżkę do pliku konfiguracyjnego (opcjonalny, jeśli nie zostanie dostarczony typy walutowe nie będą wspierane).

Uruchomienie programu bez argumentów wejściowych spowoduje uruchomienie testowego pliku źródłowego, na podstawie którego można obejrzeć najważniejsze funkcjonalności wspierane przez język (przykłady zaprezentowane w pliku testowym zostaną omówione w poniższej sekcji opracowania).

2.4. Przykłady z działania programu (testowy plik źródłowy)

2.4.1. Deklarowanie funkcji i rekurencja i operacje arytmetyczne

Przygotowano implementację dwóch przykładowych funkcji realizujących algorytm do obliczania NWD oraz silni. Co istotne w obu algorytmach pojawia się więcej niż jedno miejsce powrotu oraz oba algorytmy funkcjonują w sposób rekurencyjny. Działanie programu sprawdzono na następujących przykładach:

```
int nwd(int a, int b)
{
    if(a==b)
    {
        return a;
    }

    if (a > b)
    {
        return nwd(a-b, b); # wywołania rekurencyjne
    }

    return nwd(b-a, a);
}

int factorial(int a)
{
    if(a == 1)
        return 1;

    return a * factorial(a-1);
}

void main()
{
    int x = 100;

    int y = 9;
    print("nwd (", x, ", ", y, ") = ", nwd(x, y));

    y = 10;
    print("nwd (", x, ", ", y, ") = ", nwd(x, y));

    y = 15;
    print("nwd (", x, ", ", y, ") = ", nwd(x, y));

    y = 100;
    print("nwd (", x, ", ", y, ") = ", nwd(x, y));

    y = 150;
    print("nwd (", x, ", ", y, ") = ", nwd(x, y));

    int z = 5;
    print("Silnia ", z, " wynosi = ", factorial(z));

    z = 15;
    print("Silnia ", z, " wynosi = ", factorial(z));
```



```

nwd (100, 9) = 1
nwd (100, 10) = 10
nwd (100, 15) = 5
nwd (100, 100) = 100
nwd (100, 150) = 50
Silnia 5 wynosi = 120
(LINE: 22, column: 13) Overflow at arithmetic operation
Line:      return a * factorial(a-1);
           ^ HERE !
Trace:
  on Application.Models.Grammar.MultiplicativeExpr
  on Application.Models.Grammar.BlockStmt
  on Application.Models.Grammar.FunctionDecl
  on Application.Models.Grammar.FunctionCallExpr
  on Application.Models.Grammar.MultiplicativeExpr
  on Application.Models.Grammar.BlockStmt
  on Application.Models.Grammar.FunctionDecl
  on Application.Models.Grammar.FunctionCallExpr
  on Application.Models.Grammar.MultiplicativeExpr
  on Application.Models.Grammar.BlockStmt
  on Application.Models.Grammar.FunctionDecl
  on Application.Models.Grammar.FunctionCallExpr
  on Application.Models.Grammar.ExpressionArgument
  on Application.Models.Grammar.FunctionCallExpr
  on Application.Models.Grammar.BlockStmt
  on Application.Models.Grammar.FunctionDecl

```

Działanie algorytmu NWD daje prawidłowe wyniki, co świadczy o poprawności mechanizmu wywołania funkcji zadeklarowanej w kodzie źródłowym. W zaprezentowanym przykładzie pokazano ponadto deklarację i przypisanie zmiennych oraz wyjątek, który pojawił się ze względu na przepełnienie wartości zmiennej typu *int*. Na potrzeby debugowania do prezentacji wyjątków czasu wykonania dołożono ślad wywołania interpretera, prezentujący kolejne konstrukcje gramatyczne, przez które przechodził interpreter.

2.4.2. Pętle

W ramach implementacji przygotowano obsługę pętli *while* i *foreach*. Poniżej pokazano przykłady ich wywołania.

```

int i;

while(i < 10)
{
    print("i = ", i);
    i = i + 1;
}

```

```

i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9

```

```

var accounts = Collection< Account<USD> >();

# Dodawanie
accounts.Add(test1);
accounts.Add(test2);

accounts.Add(test1);
accounts.Add(test2);
accounts.Add(test3);
accounts.Add(test1);

var foundItems = accounts.Where(lambda Account x => x.Balance > 1000D);

int j = 1;
foreach(Account<USD> acc in foundItems)
{
    print(j, ". found = ", acc.Balance, " ", acc.Currency);
    j = j + 1;
}

```

```

1. found = 5000 USD
2. found = 1011,00 USD
3. found = 5000 USD
4. found = 1012,00 USD
5. found = 1011,00 USD

```

2.4.3. Obiektość i operacje finansowe

W ramach projektu przygotowano natywną klasę *Account*. Przykłady jej zastosowania przedstawiono poniżej:

```
var test1 = Account<USD>();
var test2 = Account<USD>(500);
var test3 = test2.Copy();

print("Account test1 = ", test1.Ballance, " ", test1.Currency);
print("Account test2 = ", test2.Ballance, " ", test2.Currency);
print("Account test3 = ", test3.Ballance, " ", test3.Currency, "\n");

test1.Ballance = 2000D;
test2.Ballance = 5000D;
test3.Ballance = 22D;

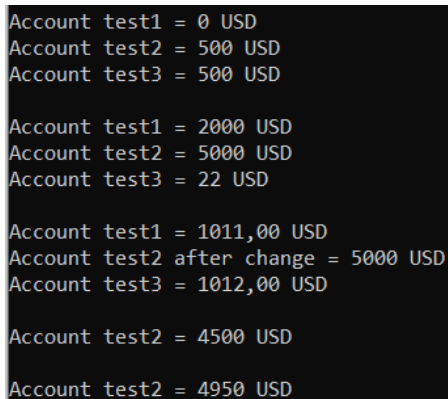
print("Account test1 = ", test1.Ballance, " ", test1.Currency);
print("Account test2 = ", test2.Ballance, " ", test2.Currency);
print("Account test3 = ", test3.Ballance, " ", test3.Currency, "\n");

test1 << 1 USD;
test1 >> 1000 EUR >> test3;

print("Account test1 = ", test1.Ballance, " ", test1.Currency);
print("Account test2 after change = ", test2.Ballance, " ", test2.Currency);
print("Account test3 = ", test3.Ballance, " ", test3.Currency, "\n");

test2 %> 10;
print("Account test2 = ", test2.Ballance, " ", test2.Currency, "\n");

test2 <% 10;
print("Account test2 = ", test2.Ballance, " ", test2.Currency, "\n");
```



```
Account test1 = 0 USD
Account test2 = 500 USD
Account test3 = 500 USD

Account test1 = 2000 USD
Account test2 = 5000 USD
Account test3 = 22 USD

Account test1 = 1011,00 USD
Account test2 after change = 5000 USD
Account test3 = 1012,00 USD

Account test2 = 4500 USD
Account test2 = 4950 USD
```

Klasa zawiera konstruktor bezargumentowy oraz konstruktor przyjmujący wartość początkową. Klasa zawiera również metodę kopiującą która generuje niezależną instancję obiektu. W ten sposób uzyskano trzy niezależne instancje o nazwach test1, test2 i test3.

Klasa zawiera właściwości *Ballance* oraz *Currency*. Pierwsza przechowuje stan konta jako wartość *decimal*, druga typ waluty jako obiekt typu. Obie z tych właściwości można wyświetlać w funkcji *print*.

W przykładzie pokazano ponadto działanie operacji finansowych, a więc wypłaty z konta (z opcjonalnym przelew na konto), wpłaty na konto oraz tych samych operacji ale w odniesieniu do wartości procentowych, a nie bezwzględnych. Warto zwrócić uwagę że jedna operacji realizowana jest w walucie EUR, gdzie konta mają walutę USD. W przypadku operacji finansowych przewalutowanie realizowane jest automatycznie.

2.4.4. Obiektość, kolekcje i operacje na kolekcjach

```
var accounts = Collection< Account<USD> >();

accounts.Add(test1);
accounts.Add(test2);

var account = accounts[0];
print("accounts[0] = ", account.Ballance, " ", account.Currency);
account = accounts[1];
print("accounts[1] = ", account.Ballance, " ", account.Currency);

accounts.Delete(0);
account = accounts[0];
print("accounts[0] = ", account.Ballance, " ", account.Currency);

accounts.Add(test1);
accounts.Add(test2);
accounts.Add(test3);
accounts.Add(test1);

var found = accounts.First(lambda Account x => {
    return x.Ballance > 1000;
});

if(found != null)
{
    print("found = ", account.Ballance, " ", account.Currency);
}
else
{
    print("found = none");
}

found = accounts.First(lambda Account x => x.Ballance > 1000000);

if(found != null)
{
    print("found = ", account.Ballance, " ", account.Currency);
}
else
{
    print("found = none");
}

var foundItems = accounts.Where(lambda Account x => x.Ballance > 10000);

int j = 1;
foreach(Account<USD> acc in foundItems)
{
    print(j, ". found = ", acc.Ballance, " ", acc.Currency);
    j = j + 1;
}
```

```
accounts[0] = 1011,00 USD
accounts[1] = 4950 USD
accounts[0] = 4950 USD
found = 4950 USD
found = none
1. found = 4950 USD
2. found = 1011,00 USD
3. found = 4950 USD
4. found = 1012,00 USD
5. found = 1011,00 USD
```

Powyżej pokazano kilka przykładów operacji na listach obiektów. Zaprezentowano operacje dodawania oraz usuwania elementów z listy. Następnie zaprezentowano metodę *First*, która zwraca pierwszy element listy spełniający dany predykat lub pustą referencję. Na bazie testu wartości null możemy stwierdzić, czy zwrócona referencja wskazuje na instancję obiektu konta, a następnie zaprezentować jego wartość oraz walutę. Test wykonujemy dwukrotnie, aby pokazać wynik w warunkach znalezienia obiektu oraz braku wyników. Ostatnim przykładem jest pokazana już wcześniej iteracja po obiektach należących do listy instancji spełniających dany predykat.

2.4.5. Obsługa błędów

Faza interpretacji drzewa obiektów została podzielona na dwie fazy. Pierwszą z nich jest analiza statyczna kodu, gdzie sprawdzana jest zgodność typów dla wszystkich operacji. Co istotne większość błędów na etapie analizy statycznej nie dyskwalifikuje dalszego przetwarzania dlatego przy jednym przejściu przez kod możliwe jest wskazanie wielu problemów. Poniżej pokazano przykłady niektórych błędów wykrywanych podczas analizy statycznej:

- Operacja arytmetyczna na niezgodnych typach:

```
# wrong types arithmetic operations
decimal wr1 = 5.1 + 10;
decimal ok1 = 5.1 + 10D;
```

```
(LINE: 167) Unexpected type of expression: "INT", expected "DECIMAL"
Line:    decimal wr1 = 5.1 + 10;
                ^ HERE !
```

- Odwołanie do nieistniejącej funkcji:

```
# not declared function call
iAmNotDeclared();
```

```
(Line 171) Function iAmNotDeclared() with specified signature does not exists.
Line:    iAmNotDeclared();
                ^ HERE !
```

- Niezgodność typów podczas przypisania:

```
# wrong types on declaration
int wr2 = 10.1;
```

```
(LINE: 174) Unexpected type of expression: "DECIMAL", expected "INT"
Line:    int wr2 = 10.1;
                ^ HERE !
```

- Brak możliwości auto dedukcji typu:

```
# not possible to deduce var type
var wr3;
```

```
(LINE: 177, column: 4) Invalid variable declaraion, can't deduce type
Line:    var wr3;
                ^ HERE !
```

- Redefinicja zmiennej w tym samym bloku:

```
decimal ok1 = 5.1 + 10D;
```

```
# variable redefinition  
decimal ok1 = 10D;
```

```
(LINE: 171) Variable ok1 redefinition attempt  
Line:    decimal ok1 = 10D;  
        ^ HERE !
```

Drugą fazą jest interpretacja, gdzie mogą pojawić się problemy, których wykrycie było niemożliwe na etapie analizy statycznej. Poniżej pokazano kilka błędów czasu wykonania:

- Dzielenie przez zero:

```
(LINE: 146, column: 12) Zero division exception  
Line:    print(10/0);  
        ^ HERE !  
Trace:  
    on Application.Models.Grammar.MultiplicativeExpr  
    on Application.Models.Grammar.ExpressionArgument  
    on Application.Models.Grammar.FunctionCallExpr  
    on Application.Models.Grammar.BlockStmt  
    on Application.Models.Grammar.FunctionDecl
```

- Przepelnienie się wartości przy operacji arytmetycznej:

```
(LINE: 149, column: 19) Overflow at arithmetic operation  
Line:    int m = 100000 * 100000 ;  
        ^ HERE !  
Trace:  
    on Application.Models.Grammar.MultiplicativeExpr  
    on Application.Models.Grammar.BlockStmt  
    on Application.Models.Grammar.FunctionDecl
```

- Operacja na referencji wskazującej na NULL

```
# Null reference error exception example  
Account<USD> nulltest;  
nulltest.Ballance;
```

```
(LINE: 153, column: 13) Null reference exception (Property Ballance)  
Line:    nulltest.Ballance;  
        ^ HERE !  
Trace:  
    on Application.Models.Grammar.ObjectPropertyExpr  
    on Application.Models.Grammar.BlockStmt  
    on Application.Models.Grammar.FunctionDecl
```

- Odwołanie poza listę

```
# Index out of range exception example  
Collection< int > ints = Collection< int >();  
ints.Add(2);  
ints.Add(3);  
print(ints[2]);
```

```
(LINE: 159, column: 15) Index reference beyond collection size (index 2)
Line:    print(ints[2]);
          ^ HERE !
Trace:
  on Application.Models.Grammar.ObjectIndexExpr
  on Application.Models.Grammar.ExpressionArgument
  on Application.Models.Grammar.FunctionCallExpr
  on Application.Models.Grammar.BlockStmt
  on Application.Models.Grammar.FunctionDecl
```