Techniki kompilacji

Projekt

Gabriel Rębacz

Politechnika Warszawska

Wydział Elektroniki i Technik Informacyjnych

Prowadzący: dr inż. Anna Derezińska

Warszawa 2019

Treść zadania.

Napisać program przekształcający kod napisany w podzbiorze języka C#, z wyrażeniami lambda na równoważny kod bez wyrażeń lambda.

Wymagania funkcjonalne podzbioru języka

- tworzenie zmiennych

    dostępny typ: int, Action<T>, Func<T, T>

- podstawowe operacje arytmetyczne
    operator: +, -

- definiowanie funkcji

```
int nazwaFunkcji (lista parametrów)
{
    blok instrukcji
    return wynik;
}
```

- wywoływanie funkcji

```
nazwaFunkcji(lista parametrów);
```

- definiowanie klas

```
class nazwa
{
    public int a;
    private int funkcja(int b) { return b;}
}
```

- publicze, prywatne oraz statyczne metody i pola

- wykorzystanie generycznych delegatów Action i Func oraz wyrażenia lambda

```
Func<InTypes, OutType> name1;
Action<InTypes> name2;

name = (a) => {return a;};
name2 = (a) => { func();};
```

Wymagania funkcjonalne programu

- zamiana kodu z wyrażeninami lambda na kod bez wyrażeń lambda
- analiza leksykalna
- analiza składniowa
- analiza semantyczna

Wymagania niefunkcjonalne

- nazwy klas, zmiennych, interfejsów itd. Powinny zaczynać się od liter (a-z, A-Z) i mogą zawierać cyfry
- program będzie wyświetlał błędy kodu
- błędy będą dzielone na kategorie: leksykalne, składniowe, semantyczne
- program będzie podawał linijkę wystąpienia błędu

Specyfikacja języka

Wybrany język będzie podzbiorem języka C#. Język C# jest językiem czysto obiektowym, więc dodane zostaną klasy oraz ich pola i metody, które będą mogły być prywatne lub publiczne lub statyczne. Obsługiwany będzie typ int, delegaty Action i Func oraz definiowane typy użytkownika poprzez mechanizm klas. Metody również będą mogły zwracać typy int, Action, Func oraz dotatkowo void. Nie zostaną zaimplementowane operatory logiczne oraz pętle i wyrażenia logiczne. Ze względu na problematykę niezbędna będzie implementacja wyrażeń lambda oraz delegatów – czyli rodzaju wskaźników na funkcje w języku C#.

Delegat Action<T> przechowuje referencję do implementacji funkcji zwracającej void i przyjmującej parametr typu T, natomiast delegat Func<T, Y> przechwouje referencję do implementacji funkcji zwracającej typ T i przyjmującej jako parametr typ Y. Delegat Action może nie przyjmować argumentu więc można go również deklarować bez symboli <>.

Domyślnie kody programów składać się będą z pojedyńczego pliku, w którym znajdować będą się definicje klas w możliwie tylko jedna definicja metody statycznej main, w której będzie się wykonywać główna część programu złożonego z bloków wyrażeń.

Dokładniej wybrane elementy podzbioru zostaną zaprezentowane w przykładach pokazujących możliwe konstrukcje języka.

Gramatyka EBNF

```
program = ({usingStmnt}, classDefinition, { classDefinition }) |

classInitialization = "class", nazwa, "=", "new", nazwa, "(", argList ")",";"
classDefinition = "class", nazwa, "{", {["public" | "private" , ["static"] ],
varDeclaration | ["public" | "private"],  methodDefinition} "}";
usingStmnt = "using", name,";"
delegateType = ("Action", ["<", typeList, ">"]) | ("Func", "<", type,
["," typeList]">")
lambdaExpression = "("[argList | nameList]")" "=>" "{" stmnt | expression""}";
methodDefinition = type | "void", reference, "(", parameterList, ")",
blockStmnt;
typeList = [type], | type, {",", type}
nameList = [name], | name, {",", name}
parameterList = [type , reference] | type, reference, {",", type reference};
blockStmnt = "{", {stmnt}, "}";
stmnt = {varDeclaration} | {assignStmnt} | {methodCallStmnt} | {blockStmnt}
| {methodDefinitionStmnt} | {printStmnt} | {lambdaExpression} | returnStmnt

varDeclaration = type, reference, ["=" expr] ";"
assignStmnt = reference, "=", expression | methodCallStmnt ";"
returnStmnt = "return", expression,";"
printStmnt = "print", "(","Console.WriteLine" | (" " ", txt, " " "), ")", ";"

expression = "("expression")" | simpleExpr, mathOp, expression |
simpleExpr;
simpleExpr = [ "-" ], ( number | reference | methodCallStmnt);
methodCallStmnt = reference, "(", argList, ")", ";";
argList = [reference] | reference, {",", reference};

text = " " ", { letter | digit | whiteSpace}, " " ";
type = "int" | delegateType | name;
name = letter, {letter | digit};
commentMark = "/", "/";
mathOp = "+" | "-", "*", "/"
number = ["-"], nonZeroDigit, { digit } | "0";
reference = letter, { letter | digit } ["."reference ];
whiteSpace = " " | "\n" | "\t";
nonZeroDigit = "1" .. "9";
```

- Słowa kluczowe:
  "class", "public", "private", "static", "using", "return", "new", "int", "void", "Action", "Func"
- Pozostałe tokeny:
  "(", ")", "{", "}", ";", "=", "==", ".", "+", "*", "/", "-", """

Przykłady dopuszczalnych konstrukcji

Najprostszym przykładem konstrukcji obsługiwanej przez podzbiór języka C# jest poniższy przykład:

```
class Program
{
    static void Main()
    {
    }
}
```

Przykład 1.

Zawiera on deklarację klasy Program, w której znajduje się statyczna funkcja main, w której wykonywana jest główna część programu – aktualnie pusta instrukcja.

```
using System;

class Program
{
    static void Main()
    {
        int a;
        int b = a*3+a;

        Console.WriteLine(b);
    }
}
```

Przykład 2.

Przykład 2. Ilustruje możliwość definiowania zmiennych(a) oraz ich inicjalizacji(b) oraz operacje arytmetyczne na zmiennych i cyfrach. Dodatkowo została dodana klauzula using, która umożlliwia dodanie referencji do innego pliku oraz wywołanie metody Console.WriteLine wypisującej wartość b na ekran.

```
using System;

class MyClass
{
   int a;
   int b = 3;

   public int fun(int param)
   {
      return param;
   }
}

class Program
{
   static void Main()
   {
      MyClass myClass = new MyClass();

      Console.WriteLine(myClass.fun(10));
   }
}
```

Przykład 3.

Przykład 3. Ilustruje możliwość tworzenia własnych klas, które zawierają pola(a,b) oraz metody (fun). Pokazana została również możliwość zastosowania wartości zwracanej przez metodę jako parametr funkcji.

```
using System;

class Program
{
   static void Main()
   {
      Action foo = () =>
      {
         Console.WriteLine(3);
      };
      Func<int, int> foo2 = (x) =>
      {
         return x;
      };

      foo();

      Console.WriteLine(foo2(3));
   }
}
```

Przykład 4.

Przykład 4. pokazuje najważniejszą funkcjonalność wybranego podzbioru języka c#, natomiast możliwość definiowania zmiennych o typie delegatów Action i Func, które mogą przyjmować wskazanie na implementację funkcji oraz je wywoływać. Jedną z możliwości przekazania implentacji funkcji jest wykorzystanie funkcji lambda jak pokazano w przypadku foo oraz foo2.

```
using System;

class Program
{
    public static void fooImpl()
    {
        Console.WriteLine(3);
    }

    public static int foo2Impl(int x)
    {
        return x;
    }


    static void Main()
    {
        Action foo = fooImpl;
        Func<int, int> foo2 = foo2Impl;

        foo();

        Console.WriteLine(foo2(3));
    }
}
```
Przykład 5.

Przykład 5. pokazuje zaś możliwość definiowania metod statycznych w klasie oraz przypisywania ich referencji do odpowiednich delegatów. Tę samą funkcjonalność można wykonać poprzez napisanie implementacji metody za pomocą wyrażeń lambda.

Przykład 4. i 5. Ilustruje jeden ze sposobów refaktoryzacji kodu z wyrażeniami lambda na kod bez wyrażeń lambda.

Sposób uruchomienia

Program będzie implementował interfejs użytkownika, w którym będzie znajdować się pole tekstowe, do którego będzie można wkleić kod, na którym zostanie wykonana transformacja oraz przycisk rozpoczynający refaktoryzację. W wyniku powodzenia zrefaktoryzowany kod zostanie wyświetlony w drugim polu tekstowym, a w przypadku błedu w polu informacyjnym pojawią się odpowiednie komunikaty błędów – leksykalne, składniowe itd. wraz z ich lokalizacją.

Architektura

Program będzie podzielony na poniższe moduły zaimplementowane w języku C#:

- Analizator leksykalny (Lexer.cs)

  Kod programu zawiera opisaną powyżej składnię, w celu jej przeanalizowania najpierw konieczna jest analiza leksykalna. Analizator pozwoli na uprzedni podział wczytanego ciągu znaków na elementarne tokeny, dzięki czemu będą one mogły być wykorzystane w dalszej analizie struktury kodu. Analizator zostanie zaimplementowany samodzielnie bez wykorzystania gotowych rozwiązań typu FLEX. Moduł ten będzie wykorzystywał tylko prymitywne operacje na stringach.

- Analizator składniowy (Parser.cs)

  Moduł ten będzie zajmować się analizą struktury kodu na podstawie tokenów otrzymanych z lexera w celu ookreślenia struktury gramatycznej w związku z określoną powyżej gramatyką formalną. Umożliwi on przetworzenie struktury czytelnej dla człowieka z poziomu kolejnych tokenów na utworzenie struktury drzewa rozbioru, która umożliwia znacznie łatwiejszą dalszą analizę kodu i umożliwia ona wykonywanie różnorodnych operacji przez oprogramowanie.

- Aplikacja okienkowa (RefactorWindow.cs)

  Moduł aplikacji w której znajdować się będzie graficzny interfejs użytkownika umożliwiający komunikację ze wszystkimi komponentami, najpierw użytkownik wprowadzi kod do pola tekstowego, a następnie rozpocznie proces refaktoryzacji, dzięki czemu RefactorWindow przekaże kod najpierw do analizatora leksykalnego, jeśli refaktoryzacja przejdzie pomyślnie zwrócony zostanie kod wynikowy, który zostanie wyświetlony w oknie, w przeciwnym wypadku do okna zwrócone zostaną informacje o błędzie, które zostaną odpowiednio obsłużone

- Moduł refaktoryzacyjny (RefactorEngine.cs)

  Moduł odpowiedzialny za przeprowadzanie refaktoryzacji kodu – na podstawie analizy drzewa rozbioru otrzymanego z analizatora składniowego moduł ten będzie generował kod po refaktoryzacji, który nie będzie miał wyrażeń lambda. Wstępny algorytm działania tej części programu został opisany poniżej.

Algorytm refaktoryzacji

Algorytm refaktoryzacji będzie polegał na wyszukiwaniu sekwencyjnym wyrażeń lambda w drzewie rozbioru, a następnie analizował w jakim kontekście znajdują się owe wyrażenia przeglądając najbliższe połączenia struktur czyli będzie poszukiwał klasy wewnątrz której są one implementowane i ewentualnych zmiennych oraz funkcji, w których zostały one przekazane.

Miejsca te zostaną zapamiętane, a następnie refaktoryzator przetworzy funkcję lambda znak po znaku na implementację funkcji statycznej i umieści jej definicję na początku definicji klasy, w której znajdowała się definicja wyrażenia lambda. Kolejnym etapem będzie zamiana wszystkich przypisań powyższego wyrażenia na nazwę nowo powstałej funkcji statycznej.

Szczególnym przypadkiem do obsłużenia będzie możliwość wystąpienia identycznego wyrażenia w wielu miejscach, w celach optymalizacji refaktoryzator powinien zaimplementować tę samą metodę tylko raz i przypisać ją do każdego miejsca, w której została wykorzystana.

Sposób testowania

Aplikacja będzie testowana na dwa sposoby:

1. Testy jednostkowe zaimplementowane w .Net'owej bibliotece MsTest. Każdy test zajmować będzie się jednym kodem w podzbiorze języka C#, który będzie zahardkodowany w zmiennej lokalnej testu, a następnie poddawany analizie leksykalnej, składniowej i w razie akceptacji przez wszystkie analizatory – refaktoryzacji, następnie wynik będzie porównywany z zahardkodowanym oczekiwanym wyjściem.
2. Przeprowadzone zostaną testy manualne, w trakcie których przygotowane wcześniej kawałki kodu będą wklejane do programu refaktoryzującego, a jego wyjście porównywane z oczekiwanym.

Zbiór kodów na potrzeby obu testów będzie ten sam, lecz w przypadku testów manualnych ze względu na dłuższy czas ich przeprowadzania może on zostać odpowiednio skrócony.

Przykładowe przypadki testowe:

Przypadek 1.

Wejście:

```
class Program
{
    static void Main()
    {
    }
}
```

Wyjście:

```
class Program
{
    static void Main()
    {
    }
}
```

Oczekiwany komentarz: "Refactorization successful. There was nothing to refactor."

Przypadek 2.

Wejście:

```
class Program
{
    static void Main()
    {
        int a;
        it b = a*3+a;

        Console.WriteLine(b);
    }
}
```

Wyjście: brak

Oczekiwany komentarz: "Refactorization failed, undefined type it, line 6."

Przypadek 3.

Wejście:

```
using System;

class Program
{
    static void Main()
    {
        Action foo = () =>
        {
            Console.WriteLine(3);
        };
        Func<int, int> foo2 = (x) =>
        {
            return x;
        };

        foo();

        Console.WriteLine(foo2(3));
    }
}
```

Wyjście:

```
using System;

class FunctionImplementations
{
    public static void fooImpl()
    {
        Console.WriteLine(3);
    }

    public static int foo2Impl(int x)
    {
        return x;
    }
}

class Program
{
    static void Main()
    {
        Action foo = FunctionImplementations.fooImpl;
        Func<int, int> foo2 = FunctionImplementations.foo2Impl;

        foo();

        Console.WriteLine(foo2(3));
    }
}
```

Oczekiwany komentarz: "Refactorization successful"

Przypadek 4.

Wejście:

```
using System;

class FunctionImplementations
{
    public static void fooImpl()
    {
        Console.WriteLine(3);
    }
}

class Program
{
    static void Main()
    {
        Action foo = FunctionImplementations.fooImpl;
        Func<int, int> foo2 = (x) => { return x; };

        foo();
        Console.WriteLine(foo2(3));
    }
}
```

Wyjście:

```
using System;

class FunctionImplementations
{
    public static void fooImpl()
    {
        Console.WriteLine(3);
    }
}

class FunctionImplementations2
{
    public static int foo2Impl(int x)
    {
        return x;
    }
}

class Program
{
    static void Main()
    {
        Action foo = FunctionImplementations.fooImpl;
        Func<int, int> foo2 = FunctionImplementations2.foo2Impl;

        foo();
        Console.WriteLine(foo2(3));
    }
}
```

Oczekiwany komentarz: "Refactorization successful"