AGH
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Karol Etrych
kierunek studiów: **informatyka stosowana**

Zaprojektowanie i implementacja kompilatora własnego języka do kodu wynikowego wykonywalnego przez Common Language Runtime

Opiekun: **dr inż. Janusz Malinowski**

Kraków, styczeń 2017

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....

(czytelny podpis)

Merytoryczna ocena pracy przez opiekuna:

Merytoryczna ocena pracy przez recenzenta:

Spis treści

Spis tabel

Spis rysunków

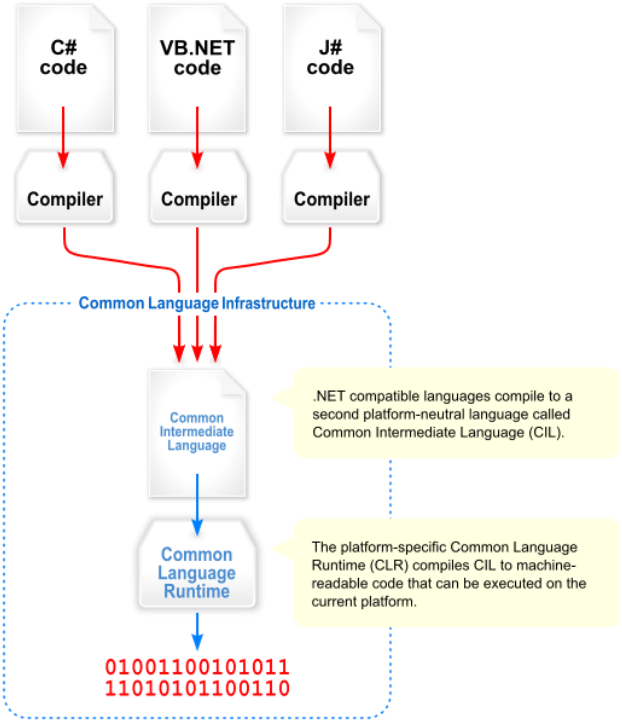
1 Wstęp

Niniejsza praca prezentuje proces projektowania i implementacji statycznie typowanego języka programowania kompilowanego do kodu wykonywalnego zgodnego ze standardem Common Language Infrastructure.
Stworzony język posiada mechanizm wnioskowania typów. Mechanizm ten pozwala programiście pominąć konieczność podawania specyfikatorów typów zachowując jednocześnie statyczne sprawdzanie typów.

1.1 Common Language Infrastructure (z ang. architektura wspólnego języka)

Common Language Infrastructure jest standardem stworzonym przez Microsoft opisującym kod wykonywalny oraz środowisko uruchomieniowe pozwalające różnym językom wysokiego poziomu (jak np. C#, Visual Basic .NET, F# czy C++/CLI) być używanym na różnych platformach bez potrzeby rekompilacji kodu dla konkretnej architektury.

CLI pozwala obiektom stworzonym w jednym z języków być traktowanym na równi przez kod napisany w zupełnie innym języku. Integracja ta jest możliwa dzięki ustandaryzowanemu zestawowi typów, metadanych (informacji o typach) oraz wspólnemu środowisku uruchomieniowemu (CLR).



Rysunek 1 Common_Language_Infrastructure
Źródło:https://en.wikipedia.org/wiki/Common_Language_Infrastructure

2 Założenia projektu

Planowane cechy języka to:

- Statyczne typowanie.
- Wspierane paradygmaty programowania: proceduralny, obiektowy.
- Mechanizm wnioskowania typów.
- Możliwość deklarowania stałych lokalnych wewnątrz funkcji.
- Spełnianie przez język wymogów stawianych przez standard Common Language Infrastructure (ECMA-335) dotyczących „konsumentów Common Language Specification" (sekcja I. 7. 2. 1.) - szczegóły w podsumowaniu.

Planowane cechy kompilatora:

- Generowanie wyjściowego kodu zgodnego ze standardem Common Language Infrastructure.
- Możliwość używania bibliotek skompilowanych na platformę .NET.
- Możliwość generowania pliku wyjściowego będącego plikiem wykonywalnym .exe lub biblioteką .dll.
- Wersja kompilatora dla .NET Framework oraz dla .NET Core (możliwość uruchomienia na systemach operacyjnych Linux).

3 Wykorzystane narzędzia

3.1 Język F#

Proces kompilacji kodu w dużej mierze składa się z operacji na drzewach np.:

1. zamiana specyfikatorów typu
2. opatrywanie wyrażeń ich typem
3. transformacja drzewa składniowego do reprezentacji pośredniej

Implementacja takich operacji jest łatwiejsza z użyciem języka funkcyjnego dzięki wykorzystaniu takich jego cech jak algebraiczne typy danych (unie i rekordy) oraz dopasowanie do wzorców (*ang. pattern matching*).
F# jest funkcyjnym językiem będącym częścią platformy .NET firmy Microsoft. Możliwe jest używanie w nim .NET Frameworka oraz bibliotek skompilowanych dla CLR.

3.1.1 Algebraiczne typy danych, przykład:

Deklaracja unii/sumy rozłącznej (*ang. discriminated union)* reprezentującej wartość w formacie JSON.

```
type JValue = // JValue może być jednym z poniższych przypadków
// jeśli jest przypadkiem JString to składa się z wartości typu string
    | JString of string
// jeśli jest przypadkiem JNumber to składa się z wartości typu float
    | JNumber of float
    | JBool of bool
// jeśl jest JNull to nie zawiera żadnych danych
    | JNull
    | JObject of (string * JValue) list // JObject to lista krotek (string, JValue)
    | JArray of JValue list
```

Inicjalizacja wartości typu JValue. W tym przypadku konkretny typ wartości to JObject (słownik).

```
let json =
    JObject(
             [
             ("name", JString "John");
             ("age", JNumber 30.);
             ("items",
              JArray [JString "A";
                      JBool true;
                      JNull])]
           )
```

3.1.2 Dopasowanie do wzorca, przykład:

Rekurencyjna funkcja przyjmująca wartość typu JValue i serializująca ją do ciągu znaków.

```
let rec serialize jvalue =
    match jvalue with
    | JArray list -> // w przypadku gdy JValue jest tablicą:
        list
        |> List.map serialize // każdy element listy jest serializowany (rekurencja)
        |> String.concat ", " // zserializowane element są łączone z użyciem przecinka
        |> sprintf "[%s]"     // i otaczane nawiasami kwadratowymi
    | JNull -> "null"         // w przypadku JNull zwracany jest ciąg znaków "null"
    | JBool b -> b.ToString() // serializacja z użyciem metody implementowanej przez System.Bool
    | JNumber n -> n.ToString()
    | JObject o ->
        (o
         |> List.map(fun (key, value) ->            // dla każdej pary (klucz, wartość)
                     key + ": " + serialize value) // rekurencyjna serializacja wartości
         |> String.concat ", ")
        |> sprintf "{%s}"
    | JString s -> "\"" + s + "\""
```

Wynik działania funkcji:

```
serialize json;;

"{age: 30, items: ["A", True, null], name: "John"}"
```

3.2 Inne narzędzia

- Visual Studio Code – edytor tekstu
- Ionide – wtyczka do Visual Studio Code dostarczająca wsparcie dla języka F#
- Expecto – framework do testów jednostkowych
- Argu – biblioteka do parsowania argumentów wywołania kompilatora
- Paket – menedżer pakietów
- FAKE – odpowiednik makefile dla .NET
- FParsec – biblioteka do tworzenia parserów

4 Opis zaprojektowanego języka

Poniżej przedstawiono możliwości języka oraz zasady jego semantyki.

4.1 Moduły

Wszelki kod umieszczany jest w modułach. Każdy plik z kodem źródłowym tworzy moduł. Przestrzeń nazw dla modułu może zostać określona z użyciem dyrektywy module:

```
module Project::Component::ExampleModule
```

Deklaracja modułu jest opcjonalna. Jeżeli nie zostanie podana, nazwa wynikowego modułu zostanie określona na bazie nazwy pliku i jego miejsca w systemie plików względem miejsca uruchomienia kompilatora. PRZYKŁAD: GENEROWANIE MODUŁÓW.

4.2 Funkcje

Funkcje definiowane są z użyciem słowa kluczowego *fun*. Parametry funkcji oraz ich typy podawane są w nawiasach. Typ zwracany podawany jest po parametrach. Ciało funkcji składa się z instrukcji i zawiera się w nawiasach klamrowych.

```
fun concat (a : int) (b : int) : string
{
    return a.ToString() + b.ToString();
}

fun main
{
    System::Console:.WriteLine(concat(1,2));
}
```

Nie jest możliwe przeładowywanie funkcji.

4.3 Klasy

Klasy deklarowane są z użyciem słowa kluczowego *class*. Członkowie klas zadeklarowani muszą być w następującej kolejności:

1. Deklaracje pól modyfikowalnych (*var*) oraz tylko do odczytu (*val*).
2. Konstruktory (*construct*).
3. Metody klasy (*fun*).

```
class Animal : BaseAnimal // deklaracja dziedziczenia
{
    val _noise : string = "noise"
    var _age : int = 0

    construct (noise : string) (age : int) : (noise, age) // konstruktor klasy bazowej
    {
        _noise = noise;
        _age = age;
    }

    fun MakeNoise
    {
        System::Console:.WriteLine(_noise);
    }
}
```

4.4 Specyfikatory typu

Tabela 1 Dostępne specyfikatory wbudowanych typów

Specyfikator typu	**Docelowy typ z .NET Framework**
float	System.Single
bool	System.Boolean
int	System.Int32
string	System.String
void	System.Void

W przypadku odwoływania się do innych typów które nie są zdeklarowanie w tym samym module konieczne jest podanie pełnej przestrzeni nazw z użyciem operatora „::”:

```
var dict = new System::Collections::Generic::Dictionary<int, string>();
var dateTime = new System::DateTime();
```

4.5 Wyrażenia

1. Operatory:

Tabela 2 Operatory uporządkowane według priorytetu (malejąco)

Symbol	Operator	Łączność	typ
!, -	Negacja logiczna, negacja arytmetyczna	-	Unarny
*, /, %	Mnożenie, dzielenie, reszta z dzielenia	Lewostronna	Binarny
+, -	Suma, różnica	Lewostronna	Binarny
<=, >=, >, <	Operatory porównania	Lewostronna	Binarny
==, !=	Równy, nierówny	Lewostronna	Binarny
&&	Koniunkcja	Lewostronna	Binarny
||	Alternatywa	Lewostronna	Binarny
=	przypisanie	Prawostronna	Binarny

2. Inicjalizator listy:

Wyrażenia będące elementami listy zawarte są wewnątrz nawiasów kwadratowych i oddzielone średnikami:

```
[1; "A"; 3.2];
```

3. Literały:

Dostępne są literały typu bool, int, float i string np.:

```
true, false, 42, 1.12, "str"
```

4. Stworzenie instancji klasy:

```
new ClassX(1, "str")
```

5. Wywołanie lokalnej funkcji:

```
localFunction(1, "str")
```

6. Wywołanie funkcji członkowskiej obiektu, dostęp do pola:

```
o.Function(1, "str")
```

```
o.Field
```

7. Wywołanie statycznej funkcji i dostęp do statycznego - operator „:.”:

```
System::Console:.WriteLine(System::DateTime:.Now)
```

Wywołania funkcji oraz przypisania mogą być używane również jako instrukcje.

4.6 Instrukcje

4.6.1 Deklaracje zmiennych i wartości

```
var y : int = 4; // deklaracja zmiennej typu System.Int32 z adnotacją typu
val s1 = "str";  // deklaracja stałej typu System.String bez adnotacji typu
val f1 : float = 3.14;
var list = [1; 2; 3; 4.0; 5.2; "six"; new System::Object()]; // deklaracja listy
```

4.6.2 Instrukcja warunkowa if

Przykład użycia instrukcji warunkowej:

```
if(3<2)
        System::Console:.WriteLine("3<2");
else
{
    System::Console:.WriteLine("3>2");
    if (6>4)
        System::Console:.WriteLine("6>4");
    else
        System::Console:.WriteLine("6<4");
}
```

4.6.3 Pętla while

Przykład użycia pętli while:

```
var a = 0;
while(a<3)
{
    System::Console:.WriteLine(a);
    a = a + 1;
}
```

Instrukcje poza instrukcjami
W języku nie ma instrukcji *break* ani *continue*.

4.7 Wnioskowanie typów

4.7.1 Typy w deklaracjach zmiennych

Pozwala nie specyfikować typu zmiennej. Typ zmiennej określany jest na podstawie typu jej inicjalizatora:

```
var a = "str";
var length = a.Length;
```

W przypadku użycia inicjalizatora listy typ listy zostanie wyznaczony na bazie najniższego wpólnego przodka typów przypisywanych do listy. Np. poniżej

```
var objList = [1; "A"; 3.2];
```

Najniższym wspólnym przodkiem typów *int*, *string* oraz *float* jest *System.Object* więc typ zmiennej objList to *System.Collections.Generic.List<System.Object>*.

4.7.2 Typy funkcji

Na podstawie typów wyrażeń w instrukcjach *return* wnioskowany jest typ funkcji.

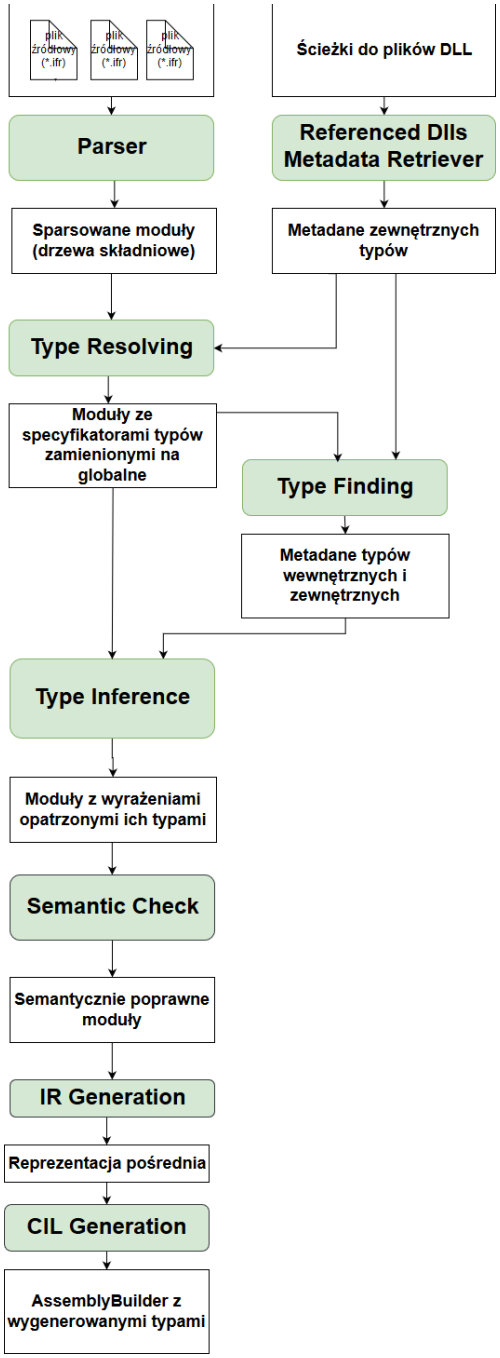
Przykład:
Jako typ zwracany wywnioskowany zostanie najniższy wspólny przodek typów Cat i Dog czyli Animal.

```
class Cat : Animal {}

class Dog : Animal {}

fun createAnimal(animalType : string)
{
    if(animalType == "dog")
        return new Dog();
    else
        return new Cat();
}

fun main
{
    var animal = createAnimal("dog");
}
```

5 Realizacja projektu

5.1 Architektura kompilatora

Poniższy diagram prezentuje budowę kompilatora. Kolorem zielonym oznaczono moduły kompilatora (zachowano nazwy modułów z kodu źródłowego projektu), a białym zasoby na których operują.



Rysunek 2 Diagram przepływu danych w kompilatorze (źródło: własne)

5.2 Obsługa błędów kompilacji.

Każda funkcja kompilatora w której mogą zostać wykryte błędy w dostarczonych danych zwraca wynik generycznego typu:

```
type CompilerResult<'TSuccess> =
| Success of 'TSuccess
| Failure of Error list
```

Jeśli dostarczone dane są poprawne zostanie zwrócony typ generyczny *'TSuccess* (na najwyższym poziomie kompilatora będzie nim lista zadeklarowanych modułów) jeśli błędne zostanie zwrócona lista błędów.
Niektóre z możliwych błędów:

```
type Error =
| SyntaxError of string // komunikat błędu z FParsec
| FunctionTypeCannotBeInferred of name : string * arguments : TypeIdentifier list
| UndefinedVariable of string // nazwa zmiennej
| TypeNotFound of TypeSpec // specyfikator typu którego nie rozpoznano
.
.
.
```

Rozwiązanie takie:

- zapewnia spójne zgłaszanie błędów - unia Error zawiera wszystkie przewidziane błędy które mogą wystąpić w dostarczonym kodzie.
- pozwala na zebranie listy błędów bez przerywania procesu sprawdzania w danej fazie kompilatora. Gdyby do zgłaszania błędów użyto wyjątków byłoby to utrudnione.

5.3 Parser

5.3.1 Drzewo składniowe

Zadaniem parsera jest przetworzenie dostarczonego kodu źródłowego do drzewa składniowego. Wykorzystanie algebraicznych typów danych umożliwia łatwe zdefiniowanie modelu drzewa. Poniżej przedstawiono niektóre węzły drzewa składniowego:

Plik z programem (docelowy moduł) składa się z opcjonalnego identyfikatora modułu (listy segmentów przestrzeni nazw) oraz z deklaracji:

```
type ProgramFile = {
    ModuleIdentifier : string list option
    Declarations : Declaration<AstExpression> list
}
```

Deklaracja może być deklaracją funkcji lub klasy:

```
and Declaration<'Expression> =
| FunctionDeclaration of Function<'Expression>
| ClassDeclaration of Class<'Expression>
```

Funkcja składa się z nazwy, parametrów, opcjonalnego specyfikatora typu i ciała (listy instrukcji):

```
and Function<'Expression> = {
  Name : string
  Parameters : Parameter list
  ReturnType : TypeSpec option
```

```
  Body : Statement<'Expression> list
}
and Parameter = string * TypeSpec // nazwa i specyfikator typu parametru
```

Specyfikator typu może być jednym z wbudowanych lub odwołaniem do klasy zadeklarowanej przez programistę/w zewnętrznej bibliotece:

```
and TypeSpec =
| BuiltInTypeSpec of BuiltInTypeSpec
| CustomTypeSpec of namespace : string list * customTypeSpec : CustomType
```

Typy wbudowane:

```
and BuiltInTypeSpec =
| Bool
| Int
| Float
| String
| Void
| Object
```

Specyfikator niewbudowanego typu składa się z jego nazwy oraz argumentów generycznych będącymi specyfikatorami typów:

```
and CustomType = {
    Name : string
    GenericArguments : TypeSpec list
}
```

Poniżej przedstawiono niektóre z instrukcji:

```
and Statement<'Expression> =
// cel przypisania oraz przypisywane wyrażenie
| AssignmentStatement of Assignee<'Expression> * 'Expression
// instrukcje w nawiasach klamrowych traktowane są jako jedna instrukcja złożona
| CompositeStatement of Statement<'Expression> list
// instrukcja warunkowa składa się z:
// 1. wyrażenia które ma zostać zewaluowane
// 2. instrukcji która ma zostać wykonana jeżeli wyrażenie jest prawdziwe (może to być instrukcja
złożona)
// 3. Opcjonalnej instrukcji else
| IfStatement of 'Expression * Statement<'Expression> * Statement<'Expression> option
// instrukcja powrotu z funkcji składa się z opcjonalnego wyrażenia które ma zostać zwrócone
| ReturnStatement of 'Expression option
| WhileStatement of 'Expression * Statement<'Expression>
```

5.3.2 Kombinatory parserów

Początkowo do zrealizowania parsera rozważano użycie generatorów parserów takich jak FsLex i FsYacc (opartych na Lex i Yacc). Takie rozwiązanie wprowadza jednak konieczność dodatkowego kroku kompilacji co utrudnia proces rozwoju projektu.

Zdecydowano się na użycie biblioteki FParsec. Umożliwia ona stworzenie parsera z użyciem kombinatorów parserów (*ang. parser combinators*). W podejściu tym parsery są funkcjami. Kombinator parserów to funkcja wyższego rzędu przyjmująca kilka parserów (innych funkcji) i

zwracająca jako wyjście nowy parser. Parsery wyższego poziomu są komponowane z parserów niższych poziomów, od symboli terminalnych do całego drzewa składniowego.
Biblioteka dostarcza zestaw parserów najniższego poziomu oraz narzędzia do komponowania ich.

Przykład: Parser deklaracji funkcji:
Deklaracja funkcji w stworzonym języku wygląda następująco:

```
fun concat (a : int) (b : int) : string
{
    return a.ToString() + b.ToString();
}
```

Operatory `.>>` , `>>.` oraz `.>>.` pochodzą z biblioteki FParsec i służą do komponowania parserów. Wynik parsera od strony kropki zwracany jest jako rezultat docelowego parsera np. poniższy parser *floatBetweenBrackets* zaaplikowany na liczbie zmiennoprzecinkowej pomiędzy nawiasami kwadratowymi zwróci wartość tej liczby.

```
let floatBetweenBrackets = (pstring "[") >>. pfloat .>> (pstring "]")
```

Poniższe parsery znaków użyte są w parserze deklaracji funkcji. Konsumują one dany znak oraz białe znaki po nim.

```
module Char =
    let colon = skipChar ':' .>> spaces
    let leftParen = skipChar '(' .>> spaces
    let rightParen = skipChar ')' .>> spaces
```

Parser deklaracji funkcji:

```
    let pFunctionDeclaration =

// Parser parametru. Po jego zaaplikowaniu zwrócona zostanie krotka (string, TypeSpec) czyli
nazwa parametru i specyfikator typu.
        let parameter =
            Char.leftParen >>. pIdentifier  .>>. (Char.colon >>. Types.pTypeSpec) .>>
Char.rightParen

// Parser listy parametrów
        let parametersList =
            many parameter
// Parser zwracanego typu. Zwracany typ jest opcjonalny (opt).
        let returnType = opt (Char.colon >>. Types.pTypeSpec)

// Parser ciała funkcji. Składa się ono z listy instrukcji otoczonej nawiasami klamrowymi.
        let body =
            between
                Char.leftBrace
                Char.rightBrace
                (many Statement.pStatement)
// Właściwy parser deklaracji:
// Funkcja pipe4 komponująca parsery przyjmuje:
//   - 4 parsery składowe
//   - wyrażenie lambda opisujące konstrukcję węzła drzewa składniowego na podstawie
//     wyników parserów składowych
```

```
        pipe4
            (Keyword.pFun >>. pIdentifier)
            parametersList
            returnType
            body
            (fun name parameters returnType body
                -> {
                        Name = name;
                        Parameters = parameters;
                        ReturnType = returnType;
                        Body = body
                    }
            )
```

5.3.3 Parser operatorów

Parser utworzony powyższą metodą jest parserem typu LL produkującym wyprowadzenie metodą zstępującą. Nie jest możliwe zastosowanie takiego parsera dla gramatyk lewostronnie rekurencyjnych.
Gramatyka wyrażeń będących operatorami binarnymi jest lewostronnie rekurencyjna:
$Expression \rightarrow Expression\ Operator\ Expression$

Parser wpadłby więc nieskończoną rekurencję. FParsec udostępnia narzędzie do parsowania operatorów z użyciem metody pierwszeństwa operatorów – klasę *OperatorPrecedenceParser*:

```
let opp = new OperatorPrecedenceParser<AstExpression, _, _>()

opp.AddOperator(
    InfixOperator(
      "+",      // operator plus
      spaces,   // parser który ma zostać zastosowany po znaku '+' (skonsumowanie białych znaków)
      6,        // priorytet operatora
      Associativity.Left, // łączność lewostronna
      fun x y -> AstExpression(BinaryExpression(x, Plus, y)) // transformacja podwyrażeń do węzła
      )
```

5.4 Znajdowanie danych o typach

Moduły **ReferencedDllsMetadataRetriever** i **TypeFinding** z rysunku RYSUNEK 2 odpowiadają za uzyskiwanie informacji o typach z odpowiednio:

- zewnętrznych bibliotek do których ścieżki podane zostały przy uruchomieniu kompilatora (oraz mscorlib.dll – biblioteki zawierającej podstawowe typy .NET Frameworka)
- drzewa składniowego sparsowanego kodu

Dane z zewnętrznych bibliotek uzyskiwane są z użyciem mechanizmu refleksji.
Dla każdego ze znalezionych typów tworzony jest następujący identyfikator typu:

```
and TypeIdentifier = {
    Namespace : string list // przestrzeń nazw
    Name : string
    GenericParameters : GenericParameter list // miejsca deklaracji gen. parametrów
    DeclaringType : TypeIdentifier option // jeżeli klasa jest zagnieżdżoną,  identyfikator typu
w którym jest zagnieżdżona
```

```
}
```

oraz rekord z informacjami o typie:

```
type Type =
    {
        IsStatic : bool

// klasa po której typ dziedziczy.
// W przypadku interfejsów i System.Object będzie to None.
        BaseType : Type option

        DeclaredConstructors : Constructor list // sygnatury konstruktorów
        Identifier : TypeIdentifier
        GenericParameters : GenericParameterInfo list
        ImplementedInterfaces : Type list
        Methods : Function list // sygnatury metod
        Fields : Field list     // nazwy i typy pól
        NestedTypes : Type list // zagnieżdżone typy (np. klasy zagnieżdżone w module)
    }
```

Dane uzyskane z obu powyższych modułów umieszczane są w słowniku typów: *Map<TypeIdentifier, Type>*. Jest on wykorzystywany w późniejszych fazach kompilatora i umożliwia obsługę typów zewnętrznych i zadeklarowanych przez programistę w jednakowy sposób.

5.5 Rozwiązywanie typów.

Moduł wyszukuje wszystkie specyfikatory typów w drzewie składniowym. Dla każdego specyfikatora generowany jest identyfikator typu (*TypeIdentifier)*, a następnie na bazie słownika typów sprawdzane jest czy taki typ istnieje.

Przy dopasowywaniu identyfikatorów uwzględniane są lokalne typy tzn., w poniższym przykładzie *new Animal()* odwoła się do klasy *Animal*. Nie jest konieczne specyfikowanie modułu (*Classes::Animal*).

```
module Classes

class Animal
{
}

fun createAnimal
{
    return new Animal();
}
```

W przypadku gdy identyfikator zostanie znaleziony jest on wstawiany w miejsce specyfikatora. W przeciwnym wypadku zostanie zwrócony błąd:

```
| TypeNotFound of TypeSpec
```

5.6 Wnioskowanie typów.

Moduł wnioskowania typów wyznacza typ dla każdego wyrażenia w drzewie składniowym.

Dla wyrażeń składających się z podwyrażeń proces odbywa się rekurencyjnie w głąb drzewa. Identyfikator wywnioskowanego typu dołączany jest do każdego z węzłów.

Informacja o typach wyrażeń wykorzystywana jest w późniejszych etapach:

- Sprawdzania semantyki – np. czy operand instrukcji przypisania ma zgodny typ
- Generatora reprezentacji pośredniej - nadanie odpowiednich typów zmiennym lokalnych

Ponadto moduł ten odpowiada za wnioskowanie typów zadeklarowanych funkcji. W każdej z funkcji znajdowane są instrukcje *return* i wnioskowane są typy zwracanych wyrażeń, a następnie typ funkcji określany jest jako najniższy wspólny przodek tych typów.

Tabela 3 Reguły ustalania typów dla wyrażeń:

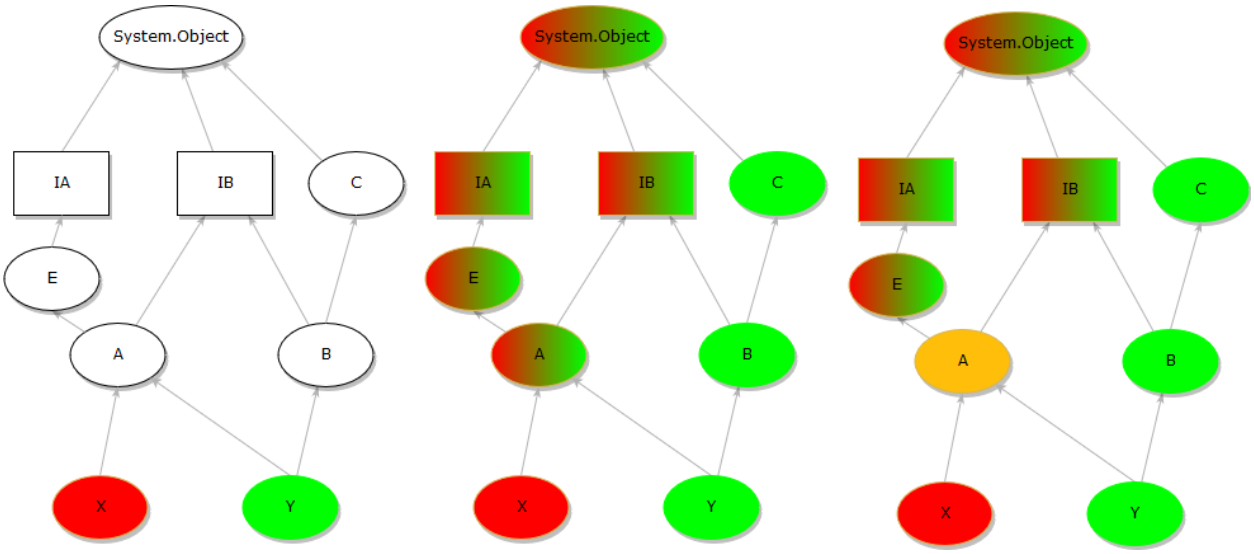
Wyrażenie	**Sposób określania typu**
Operator binarny arytmetyczny (+, -, *, /, %)	Jeżeli typy operandów są takie same zwracany jest typ operandów, w przeciwnym wypadku błąd.
Operator binarny logiczny (||, &&), operator porównania	*bool*
Operator unarny (!, -)	Typ operandu.
Literał	Typ literału np. *true* -> *bool*, 3 -> *int.*
Wyrażenie *new*	Typ podany w wyrażeniu.
Przypisanie	Typ zmiennej lub pola do którego przypisywana jest wartość.
Identyfikator (odwołanie do parametru funkcji, lokalnej zmiennej lub pola)	W przypadku parametru lub pola odczytywany jest typ podany przez programistę. Dla zmiennej zwracany jest typ wywnioskowany wcześniej w deklaracji owej zmiennej.
Lokalna funkcja	Odczytywany jest wyspecyfikowany typ zwracany funkcji lub wywnioskowany wcześniej jeśli funkcja zadeklarowana jest powyżej obecnie sprawdzanej

Członkowska funkcja lub pole
Np. *zmienna.ToString()*
	Najpierw wnioskowany jest typ wyrażenia którego funkcja jest wywoływana/pole jest pobierane. Następnie w słowniku typów sprawdzany jest typ tej funkcji/pola.
	
	Na tym etapie wykonywane jest również podstawianie generycznych argumentów:
	`val dict = new Dictionary<int, string>();`
	`val enumerator = dict.GetEnumerator();`
	`var current = enumerator.Current;`
	W powyższym przykładzie typ zmiennej *enumerator* znaleziony w słowniku typów to: `Enumerator<TKey, TValue>` a typ *current* to `KeyValuePair<TKey, TValue>`.
	Generyczne argumenty wstawiane w miejsce *TKey* i *TValue* są odczytywane z typu zmiennej *dict.*

Statyczna funkcja lub pole	Typ odczytywany jest ze słownika typów.
Inicjalizator listy	*System.Collections.Generic.List<T>* gdzie typ T jest najniższym wspólnym przodkiem typów wyrażeń w inicjalizatorze.

5.6.1 Algorytm znajdowania najniższego wspólnego przodka.

Na poniższym rysunku zaprezentowano znajdowanie najniższego wspólnego przodka typów X i Y.



Rysunek 3 Znajdowanie najniższego wspólnego przodka (źródło: własne)

1. Dla każdego typu znajdowany jest zbiór jego rodziców. Typy oznaczone kolorem czerwonym są rodzicami typu X, a zielonym typu Y.
2. Wyznaczane jest część wspólna zbiorów, na powyższym rysunku są to typy oznaczone kolorem czerwono-zielonym.
3. Wybierany jest typ najbardziej oddalony od korzenia (na rysunku typ A). Jeżeli więcej niż jeden typ ma taką samą odległość od korzenia zwracany jest błąd. (Programista musi samodzielnie wyspecyfikować typ).

5.7 Sprawdzanie semantyki.

Moduł zajmuje się zweryfikowaniem czy dostarczone drzewo składniowe jest zgodne z zasadami semantyki języka.

- Czy zmienne lokalne oznaczone jako tylko do odczytu (modyfikator *val*) nie są nadpisywane po inicjalizacji.
- Czy pola oznaczone modyfikatorem *val* nie są nadpisywane poza konstruktorem.
- Czy wyrażenia wewnątrz instrukcji warunkowych *if* są typu *bool*.
- Jeżeli w deklaracji zmiennej został podany jej typ, to sprawdzane jest czy jest on zgodny z wywnioskowanym typem jej inicjalizatora.
- Jeżeli oczekiwanym wyjściem kompilacji jest plik .exe, to sprawdzane jest czy wśród modułów znajduje się dokładnie jedna funkcja o nazwie *main*.
- Operandy operatorów binarnych są właściwego typu. Np. czy operandy alternatywy logicznej są typu *bool*.
- Sprawdzane jest czy nie występują zduplikowane deklaracje zmiennych.

5.8 Generowanie reprezentacji pośredniej.

Wprowadzenie reprezentacji pośredniej przynosi następujące korzyści:

- Możliwość wprowadzenia modułu optymalizującego kod wynikowy działającego na reprezentacji pośredniej.
- Uproszczenie modułu generującego Common Intermediate Language, co umożliwia łatwą zamianę go na moduł generujący kod wynikowy na inną platformę np. .NET Core lub zupełnie inne środowisko np. kod bajtowy Javy lub asembler x86.
- Możliwość napisania testów jednostkowych sprawdzających czy dla danego wejścia generowana jest odpowiednia reprezentacja pośrednia. Gdyby na tym etapie emitowano już kod wynikowy jego weryfikacja w teście byłaby trudniejsza, jeśli nie niemożliwa.

Moduł odpowiada za:

- Wygenerowanie listy zmiennych dla każdej funkcji i konstruktora. Dla każdej deklaracji zwracana jest nazwa zmiennej oraz identyfikator jej typu.
- Dla każdego odwołania do identyfikatora (*IdentifierExpression*) sprawdzane jest czy jest to odwołanie do zmiennej lokalnej, argumentu funkcji czy pola klasy zawierającej funkcję. Na tej podstawie generowana jest odpowiednia instrukcja.

5.8.1 Model generowanej struktury danych

Poniżej przedstawiono model struktur danych, które generowane w module:
Moduł składa się z identyfikatora, funkcji oraz klas.

```
type Module =
    {
        Identifier : TypeIdentifier
        Functions : Function list
        Classes : Class list
    }
```

Klasa składa się z identyfikatora, pól, metod, identyfikatora klasy bazowej oraz konstruktorów.

```
and Class =
    {
        Identifier : TypeIdentifier
        Fields : Variable list
        Methods : Function list
        BaseClass : TypeIdentifier
        Constructors : Constructor list
    }
```

Funkcja składa się z nazwy, identyfikatora zwracanego typu, listy parametrów, listy lokalnych zmiennych oraz kontekstu (czy jest to funkcja statyczna).

```
and Function =
    {
        Name : string
        ReturnType : TypeIdentifier
        Parameters : Variable list
        Body : Instruction list
        LocalVariables : Variable list
        Context : Context
    }

and Context =
| Static
```

```
| Instance

and Variable =
    {
        TypeId : TypeIdentifier
        Name : string
    }
```

Analogicznie konstruktor:

```
and Constructor = {
    Parameters : Variable list
    Body : Instruction list
    LocalVariables : Variable list
}
```

Instrukcje:

```
and Instruction =
```

- **Instrukcje kontroli przepływu**
  Wstawienie etykiety z podanym identyfikatorem:
  ```
  | Label of int
  ```
  Bezwarunkowy skok do podanej etykiety:
  ```
  | Br of int
  ```
  Skok do podanej etykiety jeżeli ostatnia wartość na stosie jest równa 0:
  ```
  | Brfalse of int
  ```
  Skok do podanej etykiety jeżeli ostatnia wartość na stosie jest różna od 0:
  ```
  | Brtrue of int
  ```

- **Instrukcje wywołania i dostępu do pól**
  Po wykonaniu się tych instrukcji na stos zostaje wypchnięta referencja do stworzonego obiektu/ wartość zwrócona z funkcji.

  - Wywołanie konstruktora. Instrukcja używana w konstruktorach w celu wywołania konstruktora klasy bazowej. Składa się z typu do którego należy konstruktor oraz typów argumentów.
    ```
    | CallConstructor of calleeType : TypeIdentifier * argumentTypes :
    TypeIdentifier list
    ```
  - Stworzenie nowego obiektu zadanego typu.
    ```
    | NewObj of calleeType : TypeIdentifier * argumentTypes : TypeIdentifier
    list
    ```
  - Wywołanie metody. Składa się z:
    - typu do którego należy metoda
    - sygnatury metody czyli:
      ```
      and MethodRef =
      {
      ```

```
    MethodName : string
    Parameters : TypeIdentifier list
    Context : Context
}
```

- instrukcji koniecznych do załadowania referencji do instancji na której wywoływana jest metoda na stos
- instrukcji koniecznych do załadowania argumentów na stos

```
| CallMethod of
    calleeType : TypeIdentifier *
    methodRef : MethodRef *
    calleeInstructions : Instruction list *
    argumentsInstructions : Instruction list
```

Początkowo rozważano rozwiązanie w którym instrukcja ta nie składała się z instrukcji koniecznych do załadowania wywoływanego obiektu i argumentów na stos. Instrukcje te były umieszczane bezpośrednio przed wywołaniem metody.
Obecny kształt tej instrukcji jest jednak niezbędny ze względu na różnicę w wywoływaniu metod na obiektach których wartość jest przechowywana na stosie (ang. *value types* np. struktury w C#). Dla takich obiektów wywołanie metody musi być poprzedzone uzyskaniem referencji do obiektu i wypchnięciem jej na stos. Informację o tym czy typ jest typem którego wartość jest przechowywana na stosie jest jednak uzyskiwana dopiero w następnej fazie kompilatora. W stworzonym języku takich typów nie da się zdefiniować więc pominięto tę informację we wcześniejszych fazach.

- Wywołanie lokalnej metody:

```
| CallLocalMethod of
    method : MethodRef *
    calleeInstructions : Instruction list *
    argumentsInstructions : Instruction list
```

- Pobranie zewnętrznego pola:

```
| GetExternalField of TypeIdentifier * FieldRef * Instruction list
```

- Ustawienie zewnętrznego pola:

```
| SetExternalField of TypeIdentifier * FieldRef * Instruction list * Instruction list
```

- **Instrukcje operujące na stosie**
  - Załadowanie argumentu spod indeksu 0 na stos.
    W przypadku gdy obecna metoda wywołana została na instancji (metoda nie statyczna) pod tym indeksem przechowywana jest referencja do niej.

```
| LdThis
```

  - Załadowanie zmiennej o podanym indeksie na stos.

```
| Ldloc of string
```

  - Wczytanie ostatniego elementu ze stosu do zmiennej o podanej nazwie.

```
| Stloc of string
```

  - Analogicznie dla argumentów.

```
| Ldarg of string
```

```
    | Starg of string
```

- Analogicznie dla pól.

```
    | Ldfld of string
    | Stfld of string
```

- Zduplikowanie ostatniej wartości na stosie (zostanie ona wypchnięta jeszcze raz).

```
    | Duplicate
```

- Załadowanie stałej typu 32-bitowego całkowitego na stos.

```
    | LdcI4 of int
```

- Załadowanie stałej typu 32-bitowego zmiennoprzecinkowego.

```
    | LdcR4 of single
```

- Załadowanie referencji do literału będącego ciągiem znaków.

```
    | Ldstr of string
```

- **Instrukcje arytmetyczne i logiczne**
  - Instrukcja pobiera dwie ostatnie wartości ze stosu i porównuje je. Jeśli są równe na stos zostanie wypchnięta wartość 1, a w przeciwnym wypadku 0.

```
| Ceq
```

  - Jeśli pierwsza wartość jest większa od drugiej zostanie wypchnięte 1, w przeciwnym wypadku 0.

```
| Cgt
```

  - Jeśli pierwsza wartość jest mniejsza od drugiej zostanie wypchnięte 1, w przeciwnym wypadku 0.

```
| Clt
```

    Brak jest instrukcji „większy lub równy” i „mniejszy lub równy”. Operatory te zrealizowane są poprzez zanegowanie wyników instrukcji odpowiednio: „mniejszy” i „większy”.

  - Dodawanie, odejmowanie, mnożenie, dzielenie, reszta z dzielenia.

```
| Add
| Sub
| Mul
| Div
| Rem
```

  - Binarne operacje: negacja, alternatywa, koniunkcja.

```
| Neg
| Or
| And
```

  - Instrukcja powrotu z funkcji. W przypadku gdy jest zwracana wartość przyjmuje identyfikator jej typu w przeciwnym wypadku wartość *None*.

```
| Ret of TypeIdentifier option
```

Przykład:
Implementacja generowania reprezentacji pośredniej dla instrukcji warunkowej *if*:

```
let rec generateIRFromStatement (statement : Statement<InferredTypeExpression>)=
```

```
match statement with
.
.
.
| IfStatement(condition, statement, elseStatement) ->
            let elseLabel = nextLabelId() // wygenerowanie etykiety dla początku else
            let endLabel = nextLabelId()  // wygenerowanie etykiety dla końca if'a

            // transformacja instrukcji z opcjonalnego bloku else
            // do listy instrukcji reprezentacji pośredniej
            let elseStatements =
                elseStatement
                |> Option.map generateIRFromStatement
                |> Option.toList
                |> List.concat

            // zwracane są skonkatenowane następujące instrukcje:
            convertExpression condition  // instrukcje warunku if'a
            @ [Brfalse elseLabel]        // jeśli warunek nieprawdziwy skocz do else
            @ generateIRFromStatement statement  // instrukcje z właściwego bloku kodu
            @ [Br endLabel]              // skok do końca if'a
            @ [Label elseLabel]          // etykieta początku bloku else
            @ elseStatements             // instrukcje z bloku else
            @ [Label endLabel]           // etykieta końca instrukcji if
```

Oprócz przekształcania dostarczonego drzewa składniowego do reprezentacji pośredniej moduł ten wykonuje równiej następujące czynności:

1. Jeżeli klasa nie ma zdefiniowanej klasy bazowej w jej miejsce wstawiany jest typ *System.Object*.
2. Do konstruktorów wstawiane są wywołania konstruktorów klas bazowych.
3. Do konstruktorów wstawiane są przypisania do pól.
4. Jeżeli klasa nie ma żadnego konstruktora generowany jest konstruktor domyślny wywołujący konstruktor klasy bazowej.

5.9 Generowanie Common Intermediate Language.

Moduł ten odpowiada za wyemitowanie kodu wyjściowego na podstawie reprezentacji pośredniej opisanej w rozdziale GENEROWANIE REPREZENTACJI POŚREDNIEJ.

Do działania moduł wykorzystuje klasy z przestrzeni nazw *System.Reflection.Emit*:

- *AssemblyBuilder* – służy do wygenerowania docelowego zestawu (*ang. assembly*) czyli pliku wykonywalnego .exe lub pliku biblioteki .dll.
- *ModuleBuilder* – służy do zbudowania modułu wewnątrz zestawu. W środowisku .NET zestawy składają się z modułów, te zaś z klas.
  Kompilator tworzy tylko jeden moduł .NET o nazwie takiej jak nazwa zestawu.
- *TypeBuilder* – budowanie modułów języka i klas. *TypeBuilder* dziedziczy po *System.Type* co umożliwia odwoływanie się w instrukcjach do niezbudowanych jeszcze typów np. generowanie instrukcji Call:
  ILGenerator.Emit(OpCode, MethodInfo, Type[])

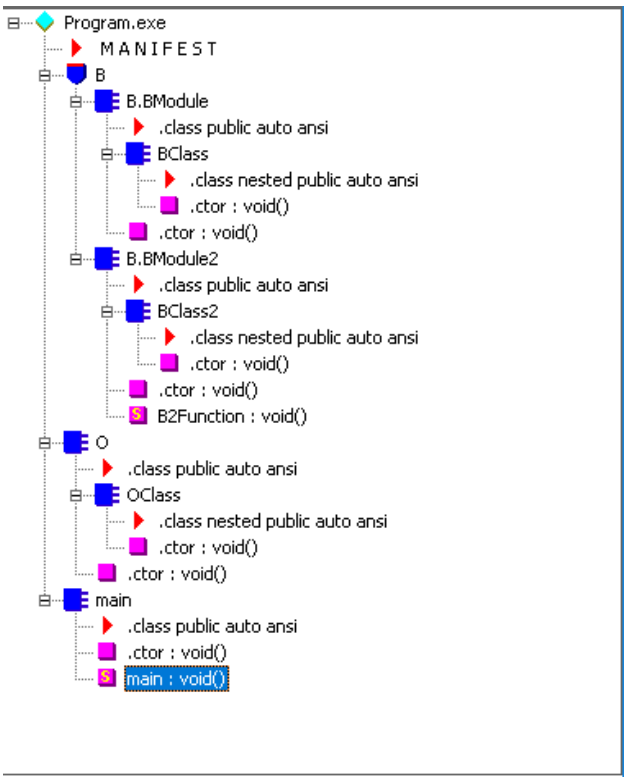
- *MethodBuilder* – budowanie metod. Wszystkie metody należące do klas oznaczane są jako publiczne i wirtualne.
- *FieldBuilder* – budowanie pól. Wszystkie pola oznaczane są jako publiczne.
- *ConstructorBuilder* – budowanie konstruktorów.
- *ILGenerator* – emitowanie instrukcji CIL.

Moduły kompilowane są do klas statycznych. Klasy zdefiniowane wewnątrz modułów kompilowane są do klas zagnieżdżonych (*ang. nested*) wewnątrz klas statycznych.

5.9.1 Przykład: generowanie modułów

Kompilator uruchomiono w folderze o następującej strukturze plików. Moduły nie miały podanych identyfikatorów więc zostały one nadane automatycznie.

```
|   main.ifr
|   O.ifr
\---B
        BModule.ifr
        BModule2.ifr
```



1. Zrzut ekranu z programu ILDasm. Struktura wygenerowanych klas.

Dla folderu B stworzona została przestrzeń nazw w skład której wchodzą moduły (klasy statyczne) B.BModule i B.BModule2.

Sygnatura głównej funkcji modułu generowania CIL:

```
let generateAssembly
    (assemblyBuilder : System.Reflection.Emit.AssemblyBuilder) // assembly builder ze
zdefiniowaną wcześniej nazwą zestawu
    (referencedAssemblies : System.Reflection.Assembly list) // zewnętrzne biblioteki
    (modules : IR.Module list) // moduły z reprezentacją pośrednią
    (setEntryPoint : bool) =   // czy punkt wejścia (funkcja main) ma zostać oznaczony
```

Ze względu na to że wygenerowanie niektórych instrukcji wymaga dostarczenia typów (*System.Type*) czy też informacji o metodach (*System.Reflection.MethodInfo*) procedura emitowania kodu przebiega następująco:

1. Stworzenie instancji *TypeBuilder* dla każdego generowanego modułu i klasy.
2. Stworzenie budowniczych konstruktorów, pól i metod dla każdego z typów.
3. Stworzeni budowniczowie typów wstawiani są do następującego rekordu:

   ```
   type FilledTypeTable = {
       FilledTypeBuilders : Map<TypeIdentifier, TypeBuilderWrapper>
       ExternalTypes : Map<TypeIdentifier, System.Type> // typy z zewnętrznych zestawów
   }
   ```

   Klasa TypeBuilder nie umożliwia wyszukiwania członków klasy (ponieważ nie zostali oni jeszcze zbudowani). Konieczne jest więc przechowywanie ich:

   ```
   and TypeBuilderWrapper = {
       MethodBuilders : Map<IR.MethodRef, MethodBuilder>
       FieldBuilders : Map<string, FieldBuilder>
       ConstructorBuilders : Map<TypeIdentifier list, ConstructorBuilder>
       TypeBuilder : TypeBuilder
   }
   ```
4. Generowane są ciała konstruktorów i metod. Funkcja budująca korzysta z zestawu funkcji wyszukujących typy oraz ich członków z użyciem powyższej struktury.
5. Dla każdego z budowniczych typów wywoływana jest metoda CreateType().
6. Na budowniczym zestawu wywoływana jest metoda Save()

Przykład: Generowanie wywołania funkcji.

```
    let callMethod calleeTypeId (methodRef : MethodRef) calleeInstructions argsInstructions =

        // znajdowanie System.Type na podstawie identyfikatora typu
        // na którym wywołana jest metoda
        let typeInfo = findFilledType calleeTypeId

        // wygenerowanie instrukcji wypychających instancję
        // na stos z uwzględnieniem tego czy jest on przechowywana na stosie
        // (komentarz w generowaniu reprezentacji pośredniej)
        generateCallee methodInfo.Variables il typeInfo calleeInstructions

        // wyemitowanie instrukcji powodujących wypchnięcie argumentów na stos
        argsInstructions |> List.iter (emitInstruction >> ignore)

        // znalezienie Reflection.Emit.MethodInfo wywoływanej metody
        let methodInfo = findMethod calleeTypeId methodRef

        // w zależności od tego czy obiekt wywołania jest przechowywany na stosie
```

```
        // oraz tego czy funkcja jest statyczna użyta jest instrukcja Call
        // lub Callvirt

        if methodRef.Context = Static || typeInfo.IsValueType
        then
            il.Emit(OpCodes.Call, methodInfo)
        else
            il.Emit(OpCodes.Callvirt, methodInfo);
```

6 Sposób użytkowania

6.1 Wymagania

Wymagany jest komputer z zainstalowanym systemem Windows. Do zbudowania projektu potrzebny jest zainstalowany .NET Framework w wersji co najmniej 4.6.1.

6.2 Użycie

Plik wynikowy z kompilatorem - *Compile*.exe znajduje się w katalogu *release*. Zbudowanie całego projektu i wygenerowanie tego pliku jest też możliwe z użyciem skryptu w głównym katalogu - *build.cmd*.

W celu użycia kompilatora należy skopiować plik *Compile.exe* do folderu z kodem źródłowym. Poniżej przedstawiono przykładowe programy napisane w stworzonym języku. Ich kod źródłowy można znaleźć w katalogu *samples*.

Uruchomienie programu z flagą *--help* powoduje wyświetlenie dostępnych opcji.

```
PS C:\Users\karol\Desktop\compiler\samples\module_finding> .\Compile.exe --help
USAGE: Compiler.exe [--help] [--output <path>] [--outputtype <exe|dll>] [--referenceddlls [<paths>...]] [--printir]
                    [<path>...]

SOURCEFILES:

    <path>...             source file paths.

OPTIONS:

    --output, -o <path>   output path.
    --outputtype, -O <exe|dll>
                          output type.
    --referenceddlls, -R [<paths>...]
                          referenced dll paths
    --printir, -S         print intermediate representation
    --help                display this list of options.
```

Flaga	Opis
-o	Ścieżka do pliku docelowego
-O	Typ pliku wynikowego dll lub exe
-R	Ścieżki do bibliotek .dll, które mają zostać wykorzystane
-S	Zapisanie reprezentacji pośredniej w formacie .fsx (rekordy w języku F#)

W przypadku gdy nie zostanie podana lista plików wejściowych kompilator wyszuka w obecnym folderze i folderach zagnieżdżonych wszystkie pliki z rozszerzeniem .ifr.

6.3 Przykład 1. Liczby pierwsze

Poniższy program znajduje liczby pierwsze z przedziału 3-30 i wstawia je do listy. Następnie iteruje po liście i wypisuje jej zawartość na ekran.

```
fun main
{
    val primes = [2]; // lista zainicjalizowana wartością 2

    var i = 3;
    while(i < 30)
    {
        var isPrime = true;
```

```
        var finish = false;
        var j = 2;
        while(j*j < i && !finish)
        {
            if (i % j == 0)
            {
                isPrime = false;
                finish = true;
            }
            j = j + 1;
        }

        if(isPrime)
        {
            primes.Add(i);
        }
        i = i + 1;
    }

    // wypisanie wszystkich elementów listy
    val enumerator = primes.GetEnumerator(); // pobranie iteratora listy
    while(enumerator.MoveNext()) // przejście do kolejnego elementu listy
    {
        System::Console:.WriteLine(enumerator.Current);
    }

    enumerator.Dispose(); // zamknięcie iteratora
}
```

Uruchomienie:

```
C:\Users\karol\Desktop\compiler\samples\prime_numbers>Compile.exe
No input files specified. Searching for .ifr files:
Base directory: C:\Users\karol\Desktop\compiler\samples\prime_numbers\
primes.ifr
Writing output file to: Program.exe

C:\Users\karol\Desktop\compiler\samples\prime_numbers>Program.exe
2, 3, 4, 5, 7, 9, 11, 13, 17, 19, 23, 25, 29,
C:\Users\karol\Desktop\compiler\samples\prime_numbers>
```

6.4 Przykład 2. Polimorfizm

Poniższy program używa metody fabrycznej *createAnimal* w celu stworzenia trzech implementacji klasy *Animal* i wstawienia ich do listy. Następnie prezentowane jest wywołanie wirtualnej metody *MakeNoise()* na każdej z implementacji.

```
module Classes

class Animal
{
    val _noise : string
```

```
    construct (noise : string)
    {
        _noise = noise;
    }

    fun MakeNoise
    {
        System::Console:.WriteLine(_noise);
    }
}

class Dog : Animal
{
    construct : ("Hau!") // wywołanie konstruktora klasy bazowej Animal
    {
    }
}

class Cat : Animal
{
    construct : ("Miau!")
    {
    }
}

class Duck : Animal
{
    construct : ("")
    {
    }

    fun MakeNoise // nadpisanie (ang. override) metody z klasy bazowej
    {
        System::Console:.WriteLine("Kwak!");
        System::Console:.WriteLine("Kwak!");
    }
}

fun createAnimal (animalType : string)
// typ zwracany funkcji jest dedukowany na podstawie typów wyrażeń w instrukcjach return
{
    if(animalType == "dog")
        return new Dog();
    else if(animalType == "cat")
        return new Cat();
    else
        return new Duck();
}

fun printAnimals (animals : System::Collections::Generic::List<Animal>)
{
    var i = 0;
    while(i < animals.Count)
    {
```

```
        val animal = animals.get_Item(i); // pobranie elementu z listy spod indeksu i

        System::Console:.Write("I am: ");
        System::Console:.WriteLine(animal.GetType()); // wypisanie typu klasy pochodnej

        animal.MakeNoise();

        i = i + 1;
    }
}

fun main
{
    val dog = createAnimal("dog");
    val cat = createAnimal("cat");
    val duck = createAnimal("duck");

    val animals = [dog; cat; duck];
    printAnimals(animals);
}
```

Uruchomienie:

```
C:\Users\karol\Desktop\compiler\samples\polymorphism>Compile.exe classes.ifr
Writing output file to: Program.exe

C:\Users\karol\Desktop\compiler\samples\polymorphism>Program.exe
I am: Classes+Dog
Hau!
I am: Classes+Cat
Miau!
I am: Classes+Duck
Kwak!
Kwak!
```

6.5 Przykład 3. Użycie zewnętrznej biblioteki.

Poniższy program tworzy bitmapę o wymiarach podanych przez użytkownika. Następnie wypełnia ją białym prostokątem i tworzy elipsę koloru czerwonego. Wynik zapisywany jest do pliku *ellipse.jpg*.
Program wykorzystuje bibliotekę *System.Drawing.dll*.

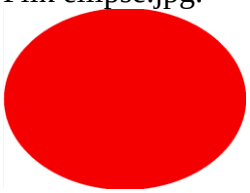
```
fun DrawFilledRectangle (x : int) (y : int)
{
    val bmp = new System::Drawing::Bitmap(x, y);
    val graphics = System::Drawing::Graphics:.FromImage(bmp);

    val imageRectangle = new System::Drawing::Rectangle(0, 0, x, y);
    graphics.FillRectangle(System::Drawing::Brushes:.White, imageRectangle);

    val blackBrush = new System::Drawing::SolidBrush(System::Drawing::Color:.Red);
    graphics.FillEllipse(blackBrush, imageRectangle);

    graphics.Dispose();
    return bmp;
}
```

```
fun main
{
    val width = System::Convert:.ToInt32(System::Console:.ReadLine());
    val height = System::Convert:.ToInt32(System::Console:.ReadLine());
    val bitmap = DrawFilledRectangle(width, height);
    bitmap.Save("ellipse.jpg");
}
```

Uruchomienie:

```
C:\Users\karol\Desktop\compiler\samples\external_library>Compile.exe -R System.Drawing.dll -o Drawing.exe
No input files specified. Searching for .ifr files:
Base directory: C:\Users\karol\Desktop\compiler\samples\external_library\
drawing.ifr
Writing output file to: Drawing.exe

C:\Users\karol\Desktop\compiler\samples\external_library>Drawing.exe
```

Plik ellipse.jpg:



7 Podsumowanie.

7.1 Realizacja założeń

7.1.1 Zrealizowane założenia:

- Statyczne typowanie.

  Stworzony język jest statycznie typowany. Dla poniższego fragmentu kodu:

```
var a = 3;
a = "5";
```

  Zostanie zwrócony błąd:

```
C:\Users\karol\Desktop\Nowy folder (3)>Compile.exe a.ifr
Compilation Failure:
InvalidType: Variable: a Expected System.Int32 but was System.String
```

- Wsparcie dla paradygmatów programowania proceduralnego i obiektowego.

  Możliwe jest dzielenie kodu na funkcje (procedury) oraz klasy przechowujące stan (pola) i zachowania (metody). Wspierane jest dziedziczenie oraz polimorfizm.

- Mechanizm wnioskowania typów

  Zrealizowano wnioskowanie typów dla lokalnych zmiennych oraz funkcji.

- Możliwość deklarowania stałych lokalnych wewnątrz funkcji.

  Istnieje możliwość deklarowania zmiennych (słowo kluczowe *var*) oraz stałych (*val)* lokalnych.

- Generowanie wyjściowego kodu zgodnego ze standardem Common Language Infrastructure.

  Generowany kod wynikowy zgodny jest z powyższym standardem. Pliki wyjściowe wygenerowane w testach integracyjnych kompilatora przechodzą weryfikację narzędziem *Peverify.exe*.

- Możliwość używania bibliotek skompilowanych na platformę .NET.

  Istnieje możliwość odwoływania się zewnętrznych bibliotek przy użyciu flagi *-R*:

```
C:\Users\karol\Desktop\Nowy folder (3)>Compile.exe a.ifr -R A.dll
Writing output file to: Program.exe
```

- Możliwość generowania pliku wyjściowego będącego plikiem wykonywalnym .exe lub biblioteką .dll

  Można wygenerować kod będący biblioteką przy użyciu flagi *-O*:

```
C:\Users\karol\Desktop\Nowy folder (3)>Compile.exe a.ifr -O dll
Writing output file to: Library.dll
```

7.1.2 Niezrealizowane założenia

- Dostarczona wersja projektu uruchamiana jest na platformie .NET Framework 4.6.1. Nie udało się stworzyć wersji kompilatora na platformę .NET Core. Okazało się to obecnie niemożliwe ponieważ klasa *AssemblyBuilder* w .NET Core 2.0 nie posiada wymaganych metod *SetEntryPoint*() oraz *Save*(). Sytuacja ta może ulec zmianie w kolejnych wersjach .NET Core.

- Common Language Specification (część CLI) definiuje wymogi stawiane konsumentom CLS czyli językom zaprojektowanym by używać bibliotek zgodnych z CLS. Wymogi te znajdują się w sekcji I. 7. 2. 1 standardu ECMA-335:

Wymaganie:	**Stopień realizacji:**
Możliwość wywoływania dowolnej metody/delegatu zgodnego z CLS.	istnieje możliwość wywołania dowolnej metody zgodnej z CLS.
Mechanizm wywoływania metod których nazwy są słowami kluczowymi języka.	nie zrealizowano
Możliwość wywoływania różnych metod dostarczanych przez typ które mają taką samą nazwę i sygnaturę ale implementują różne interfejsy.	nie zrealizowano
Możliwość tworzenia instancji dowolnych typów zgodnych z CLS.	zrealizowano
Odczyt i modyfikacja pól zgodnych z CLS.	zrealizowano
Dostęp do zagnieżdżonych typów.	zrealizowano
Dostęp do właściwości (ang. properties) zgodnych z CLS.	zrealizowano – na poziomie stworzonego języka właściwości traktowane są równoważnie z polami
Dostęp do zdarzeń (ang. event) zgodnych z CLS.	nie zrealizowano
Posiadanie mechanizmu importowania, tworzenia instancji i używania generycznych typów i metod.	możliwe jest tworzenie instancji i używanie generycznych typów

7.2 Dalszy rozwój projektu.

- Dodanie dyrektywy *using*/*open* czyli rozszerzenie modułu rozwiązywania typów o dopasowywanie typów z zaimportowanych przestrzeni nazw, tak aby nie było konieczne podawanie globalnych specyfikatorów typów.
- Dodanie instrukcji *break* oraz *continue*.
- Dokładniejsze komunikaty o błędach, dostarczanie informacji o miejscu wystąpienia błędu.
- Wersja na .NET Core.
- Wprowadzenie modułu optymalizującego wygenerowaną reprezentację pośrednią (i tym samym kod wynikowy).
- Realizacja pozostałych wymogów dla konsumentów CLS.

8 Załączniki

Wraz z pracą dostarczana jest płyta CD, na której znajduje się kod źródłowy opisywanego programu.

9 Bibliografia

1. https://en.wikipedia.org/wiki/Type_inference (17.01.2018)
2. https://fsharpforfunandprofit.com/rop/ (17.01.2018)
3. https://en.wikipedia.org/wiki/Top-down_parsing (17.01.2018)
4. https://pl.wikipedia.org/wiki/Parser_LL (17.01.2018)
5. https://pl.wikipedia.org/wiki/Metoda_pierwsze%C5%84stwa_operator%C3%B3w (17.01.2018)
6. https://en.wikipedia.org/wiki/Common_Language_Infrastructure (17.01.2018)
7. https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf (17.01.2018)