**Praca inżynierska**

**Karol Etrych**

kierunek studiów: **informatyka stosowana**

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

[1 Spis treści 3](#_Toc503408510)

[2 Wstęp 4](#_Toc503408511)

[1.1. Założenia projektu. 4](#_Toc503408512)

[1.2. Wnioskowanie typów. 4](#_Toc503408513)

[1.3. generujący kod zgodny ze specyfikacją ECMA-335 5](#_Toc503408514)

[3 Opis użytych narzędzi 5](#_Toc503408515)

[3.1 Common Language Infrastructure 5](#_Toc503408516)

[3.2 Język F# 5](#_Toc503408517)

[3.2.1 Algebraiczne struktury danych, przykład: 5](#_Toc503408518)

[3.2.2 Dopasowanie do wzorca, przykład: 6](#_Toc503408519)

[3.3 Inne narzędzia 6](#_Toc503408520)

[4 Opis zaprojektowanego języka 7](#_Toc503408521)

[4.1 Moduły 7](#_Toc503408522)

[4.2 Funkcje 7](#_Toc503408523)

[4.3 Klasy 7](#_Toc503408524)

[4.4 Typy 8](#_Toc503408525)

[4.5 Instrukcje 8](#_Toc503408526)

[4.5.1 Deklaracje zmiennych i wartości 8](#_Toc503408527)

[4.5.2 Instrukcja warunkowa if 8](#_Toc503408528)

[4.5.3 Pętla while 8](#_Toc503408529)

[4.6 Wyrażenia 8](#_Toc503408530)

[4.6.1 Operatory binarne 8](#_Toc503408531)

[4.7 Wnioskowanie typów 9](#_Toc503408532)

[5 Realizacja projektu 9](#_Toc503408533)

[5.1 Architektura kompilatora 9](#_Toc503408534)

[5.2 Obsługa błędów kompilacji. 10](#_Toc503408535)

[5.3 Parser 11](#_Toc503408536)

[5.4 Znajdowanie typów 13](#_Toc503408537)

[5.5 Rozwiązywanie typów. 13](#_Toc503408538)

[5.6 Wnioskowanie typów. 14](#_Toc503408539)

[5.6.1 Algorytm działania. 14](#_Toc503408540)

[5.6.2 Algorytm znajdowania least-upper-bound. 14](#_Toc503408541)

[5.7 Sprawdzanie semantyki. 14](#_Toc503408542)

[5.8 Generowanie reprezentacji pośredniej. 15](#_Toc503408543)

[5.8.1 Model generowanej struktury danych 15](#_Toc503408544)

[5.9 Generowanie Common Intermediate Language. 20](#_Toc503408545)

[5.9.1 Przykład: generowanie modułów 20](#_Toc503408546)

[6 Przykłady użycia kompilatora 22](#_Toc503408547)

[6.1 Wymagania 22](#_Toc503408548)

[6.2 Użycie 22](#_Toc503408549)

[6.3 Przykład 1. Liczby pierwsze 23](#_Toc503408550)

[6.4 Przykład 2. Polimorfizm 24](#_Toc503408551)

[6.5 Przykład 3. Użycie zewnętrznej biblioteki. 25](#_Toc503408552)

[7 Podsumowanie. 26](#_Toc503408553)

[7.1 Dalszy rozwój projektu. 26](#_Toc503408554)

[7.2 W ramach dalszego rozwoju języka planowane jest dodanie nienulowalnych typów referencyjnych oraz wersja na .NET Standard (Linux). 26](#_Toc503408555)

[8 Bibliografia 26](#_Toc503408556)

# Wstęp

## Założenia projektu.

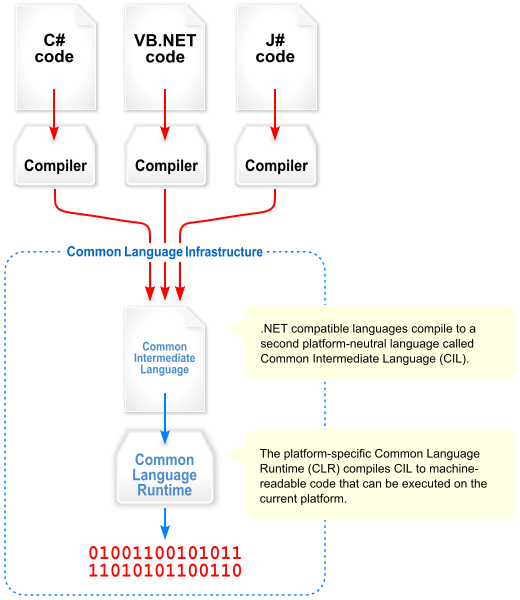
Planowane cechy języka to:

* Statyczne typowanie.
* Paradymaty programowania: imperatywny, proceduralny, obiektowy
* Mechanizm wnioskowania typów.
* Możliwość deklarowania stałych lokalnych wewnątrz funkcji.
* Spełnianie wymagań stawianych przez standard ECMA-335 dotyczący „konsumentów Common Language Specification”. Szczegóły w podsumowaniu.

Planowane cechy kompilatora:

* Generowanie wyjściowego kodu zgodny ze standardem ECMA-335.
* Możliwość używania bibliotek skompilowanych na platformę .NET.
* Możliwość generowania pliku wyjściowego będącego plikiem wykonywalnym .exe lub biblioteką .dll.

## Common Language Infrastructure



1 Źródło:https://en.wikipedia.org/wiki/Common\_Language\_Infrastructure

## Wnioskowanie typów.

# Opis użytych narzędzi

## Język F#

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

1. podmiana 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 cech jak algebraiczne struktury danych i dopasowanie do wzorców symbolicznych (*ang.* *pattern matching*).

F# jest funkcyjnym językiem będącym częścią platformy .NET firmy Microsoft. Możliwe jest używanie w nim .NET Frameworku oraz bibliotek skompilowanych dla CLR.

### Algebraiczne struktury 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 itd.*

| JNumber of float

| JBool of bool

| JNull

| JObject of Map<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])] |> Map.ofList

)

### Dopasowanie do wzorca, przykład:

Rekurencyjna funkcja serializująca wartość typu JValue 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*

|> String.concat ", " *// zserializowane element są łączone z użyciem przecinka*

|> sprintf "[%s]" *// i otaczane nawiasami kwadratowymi*

| JNull -> "null" *// w przypadku JNull zwracany jest string ”null”*

| JBool b -> b.ToString() *// serializacja z użyciem metody implementowanej przez System.Bool*

| JNumber n -> n.ToString()

| JObject o ->

(o

|> Map.toList *// zamiana mapy na listę krotek*

|> List.map(fun (key, value) ->

*// rekurencyja serializacja wartości obiektu*

key + ": " + serialize value)

|> String.concat ", ")

|> sprintf "{%s}"

| JString s -> "\"" + s + "\""

Wynik działania funkcji:

serialize json;;

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

## 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

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

# Opis zaprojektowanego języka

Poniżej przedstawiono możliwości języka oraz zasady jego semantyki.

## 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.

## 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));

}

## 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

{

**val** \_noise : **string**

**var** \_age : **int**

**construct** (noise : **string**) (age : **int**)

{

\_noise = noise;

\_age = age;

}

**fun** MakeNoise

{

System::Console:.WriteLine(\_noise);

}

}

## Specyfikatory typu

Dostępne są następujące specyfikatory typów wbudowanych:

|  |  |
| --- | --- |
| **Specyfikator typu** | **Docelowy typ z .NET Framework** |
| *Float* | System.Single |
| *Bool* | System.Boolean |
| *Int* | System.Int32 |
| *String* | System.String |
| *Void* | System.Void |

W przypadku specyfikowania 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();

## Instrukcje

### Deklaracje zmiennych i wartości

**var** y : **int** = 4; *// deklaracja zmiennej typu System.Int32 z adnotacją typu*

**var** a = 4; *// deklaracja bez adnotacji typu*

**val** s1 = "str"; *// deklaracja stałej typu System.String*

**val** f1 : **float** = 3.14;

**var** arr1 = [1;2;3;4.0;5.2;""six""; new System::Object()]; *// deklaracja listy*

### Instrukcja warunkowa if

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");

}

### Pętla while

Przykład użycia pętli while:

**var** a = 0;

while(a<3)

{

System::Console:.WriteLine(a);

a = a + 1;

}

W języku nie ma instrukcji *break* ani *continue*.

### Wywołania funkcji

Wywołanie funkcji statycznej:

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

## Wyrażenia

### Operatory binarne

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

## Wnioskowanie typów

Pozwala

**var** a = "str";

**var** length = a.Length;

1. 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>.*

1. Jako typ zwracany wywnioskowany zostanie najniższy wspólny przodek typów Cat i Dog - Animal.

**class** Cat : Animal {}

**class** Dog : Animal {}

**fun** createAnimal(animalType : **string**)

{

if(animalType == "dog")

return new Dog();

else

return new Cat();

}

**fun** main

{

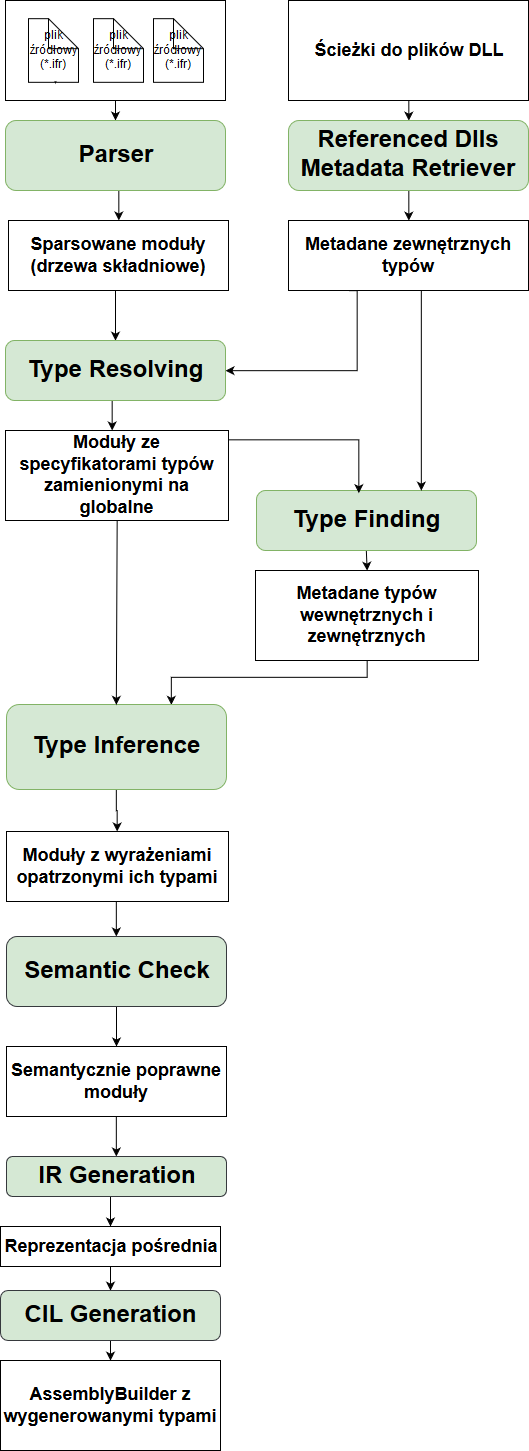
**var** animal = createAnimal("dog");

}

# Realizacja projektu

## Architektura kompilatora

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



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

## 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* zaś jeśli błędne zostanie zwrócona lista błędów.

Niektóre z możliwych błędów:

type Error =

| SyntaxError of string

| FunctionTypeCannotBeInferred of name : string \* arguments : TypeIdentifier list

| UndefinedVariable of string

| TypeNotFound of TypeSpec

.

.

.

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 utrudnione.

## Parser

### Drzewo składniowe

### 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 "[") [>>.](http://www.quanttec.com/fparsec/reference/primitives.html#members.:62::62:..) [pfloat](http://www.quanttec.com/fparsec/reference/charparsers.html#members.pfloat) [.>>](http://www.quanttec.com/fparsec/reference/primitives.html#members...:62::62:) (pstring "]")

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

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

}

)

Powyższy parser sparsuje deklarację funkcji do następującego węzła:

type Function<'Expression> = {

Name : string

Parameters : Parameter list

ReturnType : TypeSpec option

Body : Statement<'Expression> list

}

and Parameter = string \* TypeSpec

### Parser operatorów

Gramatyka operatorów binarnych jest lewostronnie rekurencyjna:

Parser wpadnie w nieskończoną rekurencję.

FParsec posiada narzędzie do parsowania operatorów – klasę *OperatorPrecedenceParser*.

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

opp.AddOperator(

InfixOperator("+", *// Parser for expre*

spaces,

6,

Associativity.Left,

fun x y -> AstExpression(BinaryExpression(x, Plus, y))

)

## Znajdowanie typów

type Type =

{

IsStatic : bool

BaseType : Type option

DeclaredConstructors : Constructor list

Identifier : TypeIdentifier

GenericParameters : GenericParameterInfo list

ImplementedInterfaces : Type list

Methods : Function list

Fields : Field list

NestedTypes : Type list

}

member x.BaseTypes = Option.toList x.BaseType @ x.ImplementedInterfaces

## Rozwiązywanie typów.

Moduł wyszukuje wszystkie specyfikatory typów w drzewie składniowym. Dla każdego specyfikatora generowany jest identyfikator typu (*TypeIdentifier)*, a następnie na bazie zbioru identyfikatorów typów wygenerowanego wcześniej 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 nie zostanie znaleziony zwracany jest błąd:

| TypeNotFound of TypeSpec

and TypeSpec =

| BuiltInTypeSpec of BuiltInTypeSpec

| CustomTypeSpec of string list \* CustomType *// namespace and Custom Type*

| TypeIdentifier of TypeIdentifier *// TypeIdentifier generated by TypeResolving module*

and BuiltInTypeSpec =

| Bool

| Int

| Float

| String

| Void

| Object

and CustomType = {

Name : string

GenericArgs : TypeSpec list *// lista argumenty generycznych*

}

Jak widać jeżeli specyfikator typu nie jest typem wbudowanym (*CustomType*) może on zawierać wewnętrzne typy będące argumentami generycznymi np.: *Dictionary<string, List<int>>*.

W takiej sytuacji funkcja sprawdzająca specyfikator wywoływana jest rekurencyjnie dla zagnieżdżonych specyfikatorów.

## Wnioskowanie typów.

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

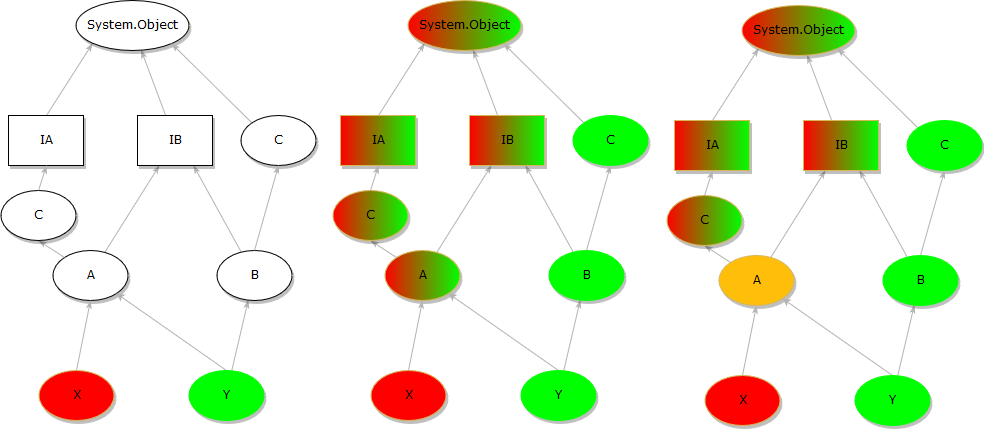
Do każdego węzła z wyrażeniem dołączana jest informacja o typie - TypeIdentifier).

Proces odbywa się od dołu w górę drzewa pozwala to na ustalenie typów wyrażeń wykorzystujących wyrażenia wewnętrzne (wnioskowanie typów)

Informacja o typie używana jest w późniejszych etapach:

* Sprawdzania semantyki - czy operandy instrukcji przypisania mają zgodne typy
* Generatora reprezentacji pośredniej - nadanie typów wyrażeniom pośrednim

### Algorytm znajdowania najniższego wspólnego przodka.



1. Dla każdego typu znajdowani są wszyscy jego rodzice. 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. Jeżeli więcej niż jeden typ ma taką samą odległość od korzenia zwracany jest błąd. (Programista musi samodzielnie wyspecyfikować typ).

## Sprawdzanie semantyki.

Moduł zajmuje się zweryfikowaniem czy dostarczone drzewo składniowe jest zgodne z zasadami semantyku 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 dla tych operatorów. Np. czy operandy alternatywy logicznej są typu *bool*.
* Sprawdzane jest czy nie występują zduplikowane deklaracje zmiennych, funkcji, klas i pól.

## 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.

### 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

* + 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 *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.

## 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.

### Przykład: generowanie modułów

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

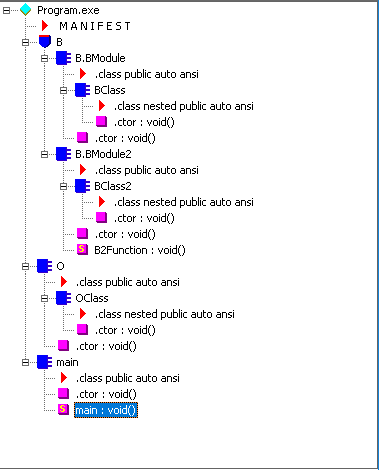
| main.ifr

| O.ifr

\---B

BModule.ifr

BModule2.ifr



3. 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:

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 (obiektów *System.Type*) czy też odwołań do metod (*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 następującej struktury:

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

}

1. Generowane są ciała konstruktorów i metod. Funkcja budująca korzysta z zestawu funkcji wyszukujących typy oraz ich członków z użyciem powyższej struktury.
2. Dla każdego z budowniczych typów wywoływana jest metoda CreateType().

**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);

# Przykłady użycia kompilatora

## Wymagania

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

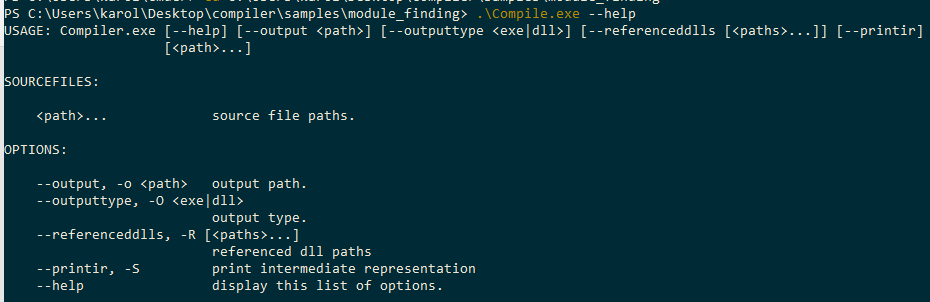
## Użycie

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.



## Przykład 1. Liczby pierwsze

**fun** main

{

**val** primes = [2];

**var** i = 3;

while(i < 100)

{

**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;

}

**val** enumerator = primes.GetEnumerator(); *// pobranie iteratora listy*

while(enumerator.MoveNext()) *// przejście do kolejnego elementu listy*

{

System::Console:.WriteLine(enumerator.Current);

}

enumerator.Dispose(); *// zamknięcie iteratora*

}

## Przykład 2. Polimorfizm

**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 instrukcji 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);

}

## Przykład 3. Użycie zewnętrznej biblioteki.

**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** bitmap = DrawFilledRectangle(640, 480);

    bitmap.Save("rectangle.jpg");

}

# Podsumowanie.

## Dalszy rozwój projektu.

## W ramach dalszego rozwoju języka planowane jest dodanie nienulowalnych typów referencyjnych oraz wersja na .NET Standard (Linux).

* Dodanie dyrektywy using/open/import czyli rozszerzenie modułu rozwiązywania typów o wyszukiwanie typów w zaimportowanych przestrzeniach nazw, tak aby nie było konieczne podawanie globalnych specyfikatorów typów.
* Dodanie instrukcji break oraz continue.
* Lepsze komunikaty o błędach, dostarczanie informacji o miejscu wystąpienia błędu.

# Bibliografia

1. <https://en.wikipedia.org/wiki/Type_inference>
2. <https://fsharpforfunandprofit.com/rop/>
3. <https://en.wikipedia.org/wiki/Top-down_parsing>
4. <https://en.wikipedia.org/wiki/LL_parser>

1. <https://en.wikipedia.org/wiki/Common_Language_Infrastructure>