Nowoczesny C# - Marcin Najder

Instalacja

- https://dotnet.microsoft.com/download/dotnet/5.0 - .NET5, po zainstalowaniu wpisać `dotnet --info`
- https://code.visualstudio.com/ - edytor Visual Studio Code
- https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp - dodatek dla .NET/C# do VS Code

C# 2,3

Stworzenie i konfiguracja projektu

Otwórz linię poleceń, a następnie wykonaj następujące polecenia:

- `mkdir MiniMal` - stwórz nowy folder
- `cd MiniMal` - przejdź do folderu
- uwaga:
  - gdy niżej podczas wykonywania polecenia `dotnet new ...` pojawi się błąd typu
  - `error NU1100: Nie można rozpoznać elementu "Microsoft.NET.Test.Sdk (>= 16.9.4)" dla elementu "net5.0".`
  - to należy wykonać polecenie `nuget add source -n nuget.org https://api.nuget.org/v3/index.json`
- `dotnet new console -n MiniMal` - stwórz projekt aplikacji konsolowej
- `dotnet new mstest -n MiniMal.Tests` - stwórz projekt testów jednostkowych
- `dotnet new sln` - stwórz plik "solution"
- `dotnet sln MiniMal.sln add MiniMal/MiniMal.csproj` - dodaj projekt aplikacji do "solution"
- `dotnet sln MiniMal.sln add MiniMal.Tests/MiniMal.Tests.csproj` - dodaj projekt testów do "solution"
- `dotnet add MiniMal.Tests/MiniMal.Tests.csproj reference MiniMal/MiniMal.csproj` - dodaj referencję z projektu `MiniMal.Tests` do projektu `MiniMal`
- `dotnet build` - zbuduj całe rozwiązanie
- `dotnet run -p ./MiniMal/MiniMal.csproj` - uruchom aplikację konsolową, na ekranie powinien wyświetlić się tekst `Hello World!`
- `dotnet test` - uruchom testy jednostkowe, na ekranie powinna pojawić się informacja `Passed: 1`

- `code .` - uruchom edytor Visual Studio Code dla aktualnego folderu
  - edytor powinien zapytać `Required assets to build and debug are missing from 'MiniMal'. Add them?`, należy wybrać opcję `Yes`, na dysku zostanie stworzony folder `.vscode` z plikami konfigurującymi uruchamianie i debuggowanie rozwiązanie bezpośrednio z edytora
  - ustaw w edytorze "break point" w metodzie `Program.Main`, naciśnij `F5` aby uruchomić oraz debuggować rozwiązanie
- do plików projektów `MiniMal/MiniMal.csproj` i `MiniMal.Tests/MiniMal.Tests.csproj` dodaj opcję kompilator C# "null-owalnych typów referencyjnych" dodając poniższe elementy XML

```
<PropertyGroup>
  ..
  <Nullable>enable</Nullable>
  <WarningsAsErrors>nullable</WarningsAsErrors>
</PropertyGroup>
```

Zapytanie LINQ, testy jednostkowe

- w pliku `Program.cs` dodaj atrybut:
  - `[assembly: System.Runtime.CompilerServices.InternalsVisibleTo("MiniMal.Tests")]`
  - nad deklaracją przestrzeni nazw `namespace MiniMal { ...`
  - dzięki temu projekt testów jednostkowych będzie miał dostęp do składowych (klas, metod, właściwości, ...) z widocznością `internal`
- dodaj `internal` do definicji klasy `internal class Program { ... }`
- do klasy `Program` dodaj metodę `internal static IEnumerable<string> ProcessItems(string[] items) { ... }` która za pomocą zapytania LINQ zwraca:
  - elementy z kolekcji `items` które nie są pustymi ciągami znaków
  - posortuj je po długości
  - skonwertuj do dużych liter
- przykładowo, dla `"", "a", "bbbb", "cc", "x"` zwrócone zostaną `"A", "X", "CC", "BBBB"`
- napisz test jednostkowy sprawdzający poprawność działa metody `ProcessItems`
- uwaga: testy jednostkowe uruchamiać możemy:
  - z lini poleceń `dotnet test` lub `dotnet test -l "console;verbosity=detailed` gdy dodatkowo chcemy wypisać wywołania `Console.WriteLine("...")`
  - bezpośrednio z VS Code debuggując kod testu korzystając z https://code.visualstudio.com/docs/languages/csharp#_codelens nad klasą lub metodą testu

Wybrane elementy języka C# 6

- wprowadzenie teoretyczne w tematykę MAL

Programowanie obiektowe

- uwagi ogólne:
  - **dla każdego etapu ćwiczenia warto pisać testy jednostkowe**, można skorzystać z gotowych dostarczonych z kompletnym ćwiczeniem
  - na pierwszym etapie ćwiczenia celowo pliki z kodem C# nazywamy `Test1.cs`, `Printer1.cs`, `Reader1.cs`, ponieważ zostaną one zamienione na docelowe implementacje i umieszczone w plikach `Test.cs`, `Printer.cs`, `Reader.cs`
  - informując o błędach działania interpretera za pomocą `Exception` należy ustawić stosowną wiadomość

Types

- dodaj nowy plik `Types1.cs` zawierający klasę statyczną `Types1`
- dodaj klasy (jako klasy wewnętrzne w `Types1`) opisujące AST dla języka MAL:
  - pusta abstrakcyjna klasa bazowa `MalType`
  - wszystkie poniżej wypisane klasy reprezentujące elementy drzewa AST:
    - są `public`
    - dziedziczą po bazowej klasie `MalType`
    - posiadają konstruktory inicjujące wartości właściwości
    - są "immutable"
  - dodaj puste klasy `Nil`, `True`, `False`
  - dodaj klasy opakowujące proste wartości `Str { string Value; }`, `Number { double Value; }` oraz `Symbol { string Name; }`
  - dodaj klasę reprezentują listę elementów `List { MalType[] Items, ListType ListType; }` gdzie `enum ListType { List, Vector }`
  - dodaj klasę reprezentującą mapę `Map { Dictionary<string, MalType> Value; }`
  - dodaj klasę reprezentującą funkcję `Fn { FnDelegate Value; }` gdzie `delegate MalType FnDelegate(MalType[] args);`
- dla pustych klas `Nil`, `True`, `False` w klasie `Types1` dodaj statyczne pola "singletony" `static True TrueV = new True();`
- do klasy `Types1` dodaj metodę `public static bool MalEqual(MalType mal1, MalType mal2) { ... }` porównującą elementy AST
  - dla `List` wszystkie elementy powinny zostać porównane za pomocą funkcji `MalEqual`, typ listy `ListType` (`List, Vector`) nie ma znaczenia podczas porównania, czyli `(1 3) == [1 3]`
  - dla `Map` wszystkie pary "klucz-wartość" powinny zostać porównane za pomocą `MalEqual`, kolejność par nie ma znaczenia podczas porównania
  - dla `Fn` porównanie zawsze zwraca `false`
  - dla pozostałych typów, dwie instancje są takie same jeśli są tego samego typu oraz posiadają takie same wartości właściwości
  - uwaga: implementując porównanie elementów AST możemy skorzystać z "obiektowego podejścia" implementując metodę wirtualną `override bool Equals(object? obj) { ...}`

dla każdego z elementów AST, ale nie jest koniecznie, szczególnie że kolejna implementacja powyższych klas stosowała funkcyjne podejście

Printer

- dodaj nowy plik `Printer1.cs` zawierający klasę statyczną `Printer1`
- dodaj statyczną metodę `public static string PrintStr(MalType? mal) { ... }` zwracającą reprezentację tekstową dla elementów AST
  - `null` -> ``
  - `Nil` -> `nil`
  - `True` -> `true`
  - `False` -> `false`
  - `Number` -> `123`
  - `Str` -> `"mama"`
  - `Symbol` -> `abc`
  - `List` -> `(1 2)` lub `[1 2]`
  - `Map` -> `{"name" "adam" "age" 30}`
  - `Fn` -> `#<function>`
- uwaga: implementując "konwersję do string" elementów AST możemy skorzystać z "obiektowego podejścia" implementując metodę wirtualną `public override string ToString() { ... }`, ale nie jest to konieczne
- dodaj pomocniczą metodę `static string JoinWithSpaces(this IEnumerable<MalType> mals, string separator = " ") { ... }` która wykonuje metodę `PrintStr` dla każdego z elementów sekwencji `mals` a następnie łączy je przekazanym `separator`

Reader

- dodaj nowy plik `Reader1.cs` zawierający klasę statyczną `Reader1`
- przekopiuj z dostarczonej implementacji `Reader1.cs` zawartość statycznej prywatne metody `static IEnumerable<string> Tokenize(string str) { ... }`. Metoda ta rozbija dostarczony fragment kodu języka MAL na tokeny, przykładowo dla tekstu `(+ 1 2)` zwróci sekwencję `"(", "+", "1", "2", ")"`
- dodaj klasę wewnętrzną `ReaderObj` która
  - przyjmuje w konstruktorze kolekcję tokenów `ReaderObj(string[] tokens) { }` i zapisuje ją w prywatnym polu klasy. Dodatkowo przechowuje pozycje tokenu `int _position` na którym się aktualnie znajduje (domyślnie jest to `0`)
  - posiada metodę `string? Peek()` zwracającą aktualny token na którym się znajdujemy, `null` gdy przekroczyliśmy ilość dostępnych tokenów
  - posiada metodę `string? Next()` działającą analogicznie do `Peek`, ale dodatkowo "na koniec" inkrementującą aktualną pozycję tokenu
- dodaj metodę `internal static MalType ReadAtom(string token) {...}` która
  - dla tekstów `"true"`, `"false"`, `"nil"` zwraca odpowiednio obiekty typów `True`, `False`, `Nil`

    - dla tekstu który jest poprawną liczbą, zwraca obiekt typu `Number`
        - warto skorzystać z funkcji `Double.TryParse("...", out number)`
    - dla tekstu w którym pierwszy i ostatni znak to `"`, zwraca obiekt typu `Str`
        - jeśli pierwszy znak to `"` a ostatni nie jest `"`, rzucany jest stosowny błąd `new Exception("...")`
    - dla pozostałych testów, zwracany jest obiekt typu `Symbol`
- dodaj metodę `internal static MalType? ReadForm(ReaderObj reader) { ... }` która
    - sprawdza kolejny token za pomocą `reader.Peek()`
    - jeśli token jest znakiem otwierającym listę (`(`, `[`), zwracany jest rezultat wywołania funkcji pomocniczej (więcej o niej będzie niżej) `ReadList(reader, token == "(" ? ")" : "]")`
    - jeśli token jest znakiem otwierającym mapę (`{`), analogicznie wywoływana jest metoda pomocnicza `ReadList(reader, "}")` tworząca listę elementów, parzyste elementy w liście muszą być typu `Str`, nieparzyste mogą być dowolnego typu `MalType`. Są to klucze oraz wartości na podstawie których powstaje element `Map`
    - jeśli ilość elementów nie jest parzysta lub kolejne elementy nie są odpowiednich typów, rzucany jest `Exception` ze stosowną informacją
- dodaj metodą pomocniczą `internal static MalType ReadList(ReaderObj reader, string endOfListToken) { ... }` która
    - czyta kolejne tokeny z pomocą `reader.Next()`, następnie wykonuje dla nich funkcje `ReadForm` i dla zwróconego rezultatu:
    - jeśli zwrócony jest `null` (brak kolejnego tokenu), zwracany jest `Exception` informujący o tym, że lista nie została zamknięta
    - jeśli zwrócony jest `Symol` który jest znakiem zamykającym listę `endOfListToken`, to zwracany jest obiekt `List` odpowiedniego typu `ListType` zawierający aktualnie elementy
    - w pozostałych przypadkach zwrócony obiekt typu `MalType` dodawany jest do listy która zostanie zwrócona gdy dojdziemy do tokenu zamykającego

REPL

- w startowej metodzie `Program.Main` dopisz kod wykonujący REPL (read evaluate print loop) dla kodu MAL:
    - odczytaj kolejną linię z konsoli
    - jeśli tekst nie jest `null` to
        - stwórz drzewo AST wykonując `Reader.ReadText`
        - wypisz na konsole wynik wywołania `Printer.PrintStr` przekazując drzew AST
    - powtarzaj powyższe kroki w pętli nieskończonej
    - jeśli podczas wykonywania iteracji pętli pojawi się błąd (zostanie rzucony wyjątek `Exception`), to obsłuż wyjątek wypisując treść błędu na konsole i nie przerywaj kolejnych iteracji

Programowanie funkcyjne

PowerFP

- uwagi:
  - `PowerFP` (Power Functional Programming) będzie osobną "biblioteką" oferującą zestaw narzędzi użytecznych przy funkcyjnym podejściu
  - w projekcie `MiniMal` załóż folder `PowerFP`, wszystkie pliki C# w tym folderze posiadały będą namespace `namespace PowerFP { ... }`
  - w dalszej części ćwiczenia staraj się (tam gdzie to możliwe) pisać metody, które są pojedynczym expression C#
    - możesz wtedy stosować "expression body members" C#
    - korzystaj z "pattern patching" C#
    - nie stosujesz "statement" C#, a jedynie "expression" C#
    - czasami warto stosować funkcje `Pipe` (więcej informacji o niej będzie podanych niżej)

LList, LListM

- `LList<T>` czyli "Linked List" będzie implementację listy jednokierunkowej
  - lista jednokierunkowa jest kolekcją "immutable", czyli każda operacja zmiany listy (dodanie, usunięcie, ...) zwracana referencję do nowego obiektu listy
- do folderu `PowerFP` dodaj plik `LList.cs` (Linked List Module), zawierał będzie "dane" (klasy/rekordy/struktury/...) oraz "zachowanie" (statyczne metody) czyli operacje które można wykonać w kontekście listy jednokierunkowej
- do pliku `LList.cs` dodaj klasę/rekord C# o nazwie `LList<T>` reprezentującą listę jednokierunkową, a dokładniej pojedynczy węzeł w liście, który posiadał będzie
  - właściwość `T Head` przechowującą element typu `T` oraz właściwość `LList<T>? Tail` przechowującą referencję do pozostałych elementów w liście
  - konstruktor inicjujący wartości właściwości, których nie można zmienić po zainicjowaniu, ponieważ są "readonly" (posiadają jedynie gettery)
  - zaimplementowane metod wirtualnych `Equals` oraz `GetHashCode` porównujących obiekty listy przez wartość ("pole po polu"). Uwaga: w przypadku zastosowania rekordu C# (zamiast klasy) nie musimy tego robić, ponieważ implementacja dostarczona będzie automatycznie
- przykładowa lista elementów `1,2,3` tworzona będzie `new LList<int>(1, new(2, new (3, null)));`, pusta lista elementów reprezentowana jest jako wartość `null`
- do pliku `LList.cs` doda klasę statyczną `ListM` reprezentującą moduł zawierający szereg funkcji operujących na `LList<T>`
  - funkcje do wygodnego tworzenia listy lub konwertowania do innych typów kolekcji:
    - `IEnumerable<T> ToEnumerable<T>(this LList<T>? llist)`
      - konwertowanie listy do leniwej sekwencji (iteratora)
    - `LList<T>? ToLList<T>(this IEnumerable<T> llist)`
      - konwertowanie leniwej sekwencji do listy
      - `new []{1,2,3}.ToLList()` => `(1,2,3)`
    - `LList<T>? LListFrom<T>(params T[] items)`

        - tworzenia listy zawierających znaną listę elementów przekazanych jako kolejne argumenty funkcji
        - `LListFrom(1,2,3)` -> `(1,2,3)`
- funkcje analogiczne do operatorów LINQ:
    - `Count<T>(this LList<T>? llist)`
        - zliczanie ilości elementów listy
        - `LListFrom("mama", "tata", "babcia").Count()` -> `3`
    - `LList<R>? Select<T, R>(this LList<T>? llist, Func<T, R> f)`
        - projekcja, czyli konwertowanie elementów
        - `LListFrom("mama", "tata", "babcia").Select(s => s.Length)` -> `4,4,6`
    - `LList<T>? Where<T>(this LList<T>? llist, Func<T, bool> f)`
        - filtrowanie, czyli zwrócenie elementów spełniających warunek
        - `LListFrom("mama", "tata", "babcia").Select(s => s.Length > 4)` -> `"babcia"`
    - `A Aggregate<T,A>(this LList<T>? llist, A seed, Func<A, T, A> f)`
        - agregowanie elementów do pojedynczej wartości
        - `LListFrom("mama", "tata", "babcia").Aggregate("", (agg, c) => agg + c)` -> `mamatatababcia`
    - `LList<T>? Take<T>(this LList<T>? llist, int count)`
        - zwrócenie listy zawierającej `count` pierwszych elementów
        - `LListFrom("mama", "tata", "babcia").Take(2)` -> `"mama","tata"`
    - `LList<T>? Skip<T>(this LList<T>? llist, int count)`
        - zwrócenie listy z pominięciem `count` pierwszych elementów
        - `LListFrom("mama", "tata", "babcia").Skip(1)` -> `tata,babcia`
    - `LList<T>? Concat<T>(this LList<T>? llist1, LList<T>? llist2)`
        - połączenie obu list
        - `LListFrom("mama", "tata", "babcia").Concat(LListFrom("dziadek"))` -> `"mama","tata","babcia","dziadek"`
    - `LList<R>? Zip<T1, T2, R>(this LList<T1>? llist1, LList<T2>? llist2, Func<T1, T2, R> f)`
        - lista której wynikiem jest wynik wywołanie funkcji `f` dla kolejnych elementów w listach `llist1` oraz `llist2`
        - `LListFrom("mama", "tata", "babcia").Zip(LListFrom("dziadek", "ciocia"), (l, r) => l + "-" + r)` -> `"mama-dziadek","tata-ciocia"`
    - `LList<TT>? SelectMany<T, TT>(this LList<T>? llist, Func<T, LList<TT>?> f)`
        - “spłaszczenie listy”, dla każdego elementu listy `llist` wykonywana jest funkcja `f` zwracająca nową listę, wszystkie zwrócone listy łączone są do jednej
        - `LListFrom("mama", "tata", "babcia").SelectMany(s => s.ToLList())` -> `'m','a','m','a','t','a','t','a','b','a','b','c','i','a'`
        - uwaga: warto skorzystać funkcji z `Concat`
- opcjonalne pozostałe funkcje
    - `T Aggregate<T>(this LList<T>? llist, Func<T, T, T> f)`

        - funkcja analogiczna do poprzedniej, ale `seed` (wartość początkowa) jest pierwszym elementem w liście, dla pustej listy wracany jest `Exception`
        - `LListFrom("mama", "tata", "babcia").Aggregate((agg, c) => agg + c)` -> `mamatatababcia`
    - `LList<R>? SelectMany<T, TT, R>(this LList<T>? llist, Func<T, LList<TT>?> f, Func<T, TT, R> r)`
        - działanie analogiczne dla wcześniejszej funkcji `SelectMany`, ale przekazany jest dodatkowy parametr `r` który jest funkcją wykonywaną dla kolejnego elementu listy `llist` oraz kolejnych elementów listy zwróconej z wywołania funkcji `f`
        - `LListFrom("mama", "tata", "babcia").SelectMany(s => s.ToLList(), (s, c) => s + "-" + c)` -> `"mama-m","mama-a","mama-m","mama-a","tata-t", ...`
    - `bool All<T>(this LList<T>? llist, Func<T, bool> f)`
        - zwraca `true` jeśli wszystkie elementy spełniają warunek `f`
        - `LListFrom("mama", "tata", "babcia").All(s => s.Contains("a"))` -> `True`
    - `bool Any<T>(this LList<T>? llist, Func<T, bool> f)`
        - zwraca `true` jeśli chodź jeden element spełniają warunek `f`
        - `LListFrom("mama", "tata", "babcia").Any(s => s.Contains("x"))` -> `False`
    - `T ElementAt<T>(this LList<T>? llist, int index)`
        - zwraca element na indeksie `index`, rzuca `Exception` gdy elementów jest za mało lub `index` jest wartością mniejszą od zera
        - `LListFrom("mama", "tata", "babcia").ElementAt(1)` -> `"tata"`
    - `bool SequenceEqual<T>(this LList<T>? llist1, LList<T>? llist2, Func<T, T, bool> equals)`
        - zwraca `true` jeśli obie listy zwracają dokładnie takie same elementy, czyli listą mają taką samą ilośc elementów oraz kolejne pary elementów zwracają `true` dla wywołań funkcji `equals`
        - `LListFrom("mama", "tata").SequenceEqual(LListFrom("mama", "tata"), (x, y) => x == y)` -> `True`
    - `LList<T>? Reverse<T>(this LList<T>? llist)`
        - zwraca odwrócona listę
        - `LListFrom("mama", "tata", "babcia").Reverse())` -> `"babcia","tata","mama"`

Map, MapM

- `Map<K,V>` czyli “mapa/słownik/tablica asocjacyjna” jest strukturą danych mapującą klucze typu `K` na wartości typu `V`
    - .net dostarcza typ `Dictionary<Key,Value>`, ale kolekcja tak jest “mutable”,
    - kolekcja `Map<K,V>` będzie “immutable” i będzie dostarczał tylko kluczowe operacje z punktu widzenia potrzeb późniejszego kodu

    - typ `Map<K,V>` będzie opakowaniem listy jednokierunkowej par "klucz-wartość", czyli opakowaniem typu `LList<(K Key, V Value)>?`
- do folderu `PowerFP` dodaj plik `Map.cs`
- do pliku `Map.cs` dodaj klasę/rekord C# o nazwie `Map<K,V>` reprezentującą mapę, która posiadała będzie
    - właściwość `LList<(K Key, V Value)>? Items` która jest readonly (posiada jedynie getter) oraz jest `public`
    - konstruktor ustawiający wartość właściwości
        - konstruktor będzie miał widoczność `internal`, ponieważ nie chcemy aby zewnętrzny kod korzystający z naszej biblioteki mógł tworzyć instancję mapy za pomocą konstruktora
        - do stworzenia instancji mapy wykorzystywana będzie dedykowana metodę `MapM.MapFrom(LList<(K Key, V Value)>? items)`, funkcja ta zadba o to aby tworzona mapa posiada unikatowe wartości kluczy
    - zaimplementuj metody wirtualnych `Equals` oraz `GetHashCode` porównujących obiekty listy przez wartość ("pole po polu")
        - w przypadku zastosowania rekordu C# (zamiast klasy) nie musimy tego robić, ponieważ implementacja generuje się automatycznie
    - dodatkowo do typu `Map<K,V>` warto dodać ograniczenie argumentu generycznego `where K: notnull`, wszystkie funkcje modułu `MapM` operujące na `Map<K,V>` będą musiały posiadać to ograniczenie
    - przykładowa mapa tworzona będzie `MapM.MapFrom(LListFrom((1,"1"),(2,"2")))`, natomiast pusta mapa `MapM.MapFrom(null)`
- do pliku `Map.cs` doda klasę statyczną `MapM` ("Map Module") reprezentującą moduł zawierający szereg funkcji operujących na `Map<K,V>`
    - `Map<K, V> Add<K, V>(this Map<K, V> map, K key, V value)`
        - funkcja dla wartości `key` zapisuje wartość `value`, jeśli wartość `key` już istnieje to jest nadpisywana, tak aby w liście używanej "pod spodem" istniała tylko jedna para dla danego klucza `key`
        - `map = map.Add(1, "1!").Add(2, "2!");`
    - `Map<K, V> MapFrom<K, V>(LList<(K Key, V Value)>? items)`
        - "factory metoda" wykorzystywana do stworzenia instancji `Map<K,V>`, pod spodem wykorzystuje funkcje `Add` zapewniającą unikatowość kluczy
        - `Map<int,string> map = MapM.MapFrom(LListM.LListFrom((1,"1!"),(2,"2!")));`
    - `(bool IsFound, V? Value) TryFind<K, V>(this Map<K, V> map, K key)`
        - funkcja wyszukująca wartości dla zadanego klucza `key`, zwraca `(true, value)` gdy wpis istnieje, oraz `(false, default(V))` gdy nie istnieje
        - `var (isFound, value) = MapFrom(LListFrom((1,"1!"),(2,"2!"))).TryFind(1);`
    - `V Find<K, V>(this Map<K,V> map, K key)`
        - funkcja działa analogicznie do `TryFind`, zwraca wartość dla zdanego klucza `key` lub rzuca `Exception` gdy klucz nie istnieje

Function

- do folderu `PowerFP` dodaj plik `Function.cs` zawierający następujący kod

```
namespace PowerFP
{
    public static class Function
    {
        public static R Pipe<T, R>(this T value, Func<T, R> func) => func(value);
        public static void Pipe<T>(this T value, Action<T> func) => func(value);
    }
}
```

- funkcja `Pipe`, znana z innych języków programowania jako operator `... |> ...`, pozwala wygodnie łączyć wiele "expression" w jedno "expression", np.:
    - `Console.ReadLine().Pipe(text => text!.ToUpper()).Pipe(Console.WriteLine)`

Programowanie funkcyjne 2

Funkcyjna implementacja `Types`, `Printer`, `Reader`

`Types`

- skopiuj plik `Types1.cs` do pliku `Types.cs`, zmień nazwę klasy `Types1` na `Types`
- wprowadź następujące zmiany w typach reprezentujących AST
    - zmień wszystkie klasy na rekordy C#9, skorzystaj z notacji "primary constructors", zwróć uwagę na to że nazwy właściwości zaczynamy od dużej litery
    - zmień typ właściwości na `LList<MalType>? Items` w typie `List`
    - zmień typ właściwości na `Map<string, MalType> Value` w typie `Map`
    - zmień typ argumentu delegatu `FnDelegate` na `LList<MalType>? args`
- zmień implementację metody `MalEqual` uwzględniając powyższe zmiany oraz następujące uwagi
    - wszystkie elementy AST są rekordami C#, metody `Equals` porównują przez wartość
    - zwróć uwagę, że wykorzystywane kolekcje danych (`LList<T>`, `Map<K,V>`) także porównują dane "w głąb" przez wartość
    - w implementacji warto wykorzystać metody `LList<T>`: `SequenceEqual`, `Count`, `All`, `Any`
- dla przypomnienia `MalEqual` porównuje dwie instancje `MalType` w następujący sposób
    - dla `List` wszystkie elementy powinny zostać porównane za pomocą funkcji `MalEqual`, typ listy `ListType` (`List, Vector`) nie ma znaczenia podczas porównania, czyli `(1 3) == [1 3]`
    - dla `Map` wszystkie pary "klucz-wartość" powinny zostać porównane za pomocą `MalEqual`, kolejność par nie ma znaczenia podczas porównania
    - dla `Fn` porównanie zawsze zwraca `false`
    - dla pozostałych typów, dwie instancje są takie same jeśli są tego samego typu oraz posiadają takie same wartości właściwości

`Printer`

- skopiuj plik `Printer1.cs` do pliku `Printer.cs` oraz zmień nazwę klasy `Printer1` na `Printer`
- upewnij się że moduł `Printer` wykorzystuje moduł `Types`, zamiast `Types1`
- zmień typ argumentu metody `JoinWithSeparator` na `LList<MalType>? mals`, zmień implementację funkcji uwzględniając te zmiany
- zmień implementacje metody `PrintStr` aby ciało metody było jednym "expression" C#

`Reader`

- skopiuj plik `Reader1.cs` do pliku `Reader.cs` oraz zmień nazwę klasy `Reader1` na `Reader`
- upewnij się że moduł `Reader` wykorzystuje moduł `Types`, zamiast `Types1`
- zmień implementacje metody `ReadAtom`, aby ciało metody było jednym "expression" C#
- zmień implementację metody `ListToMap`
  - aby przyjmowała `LList<MalType>?`, zamiast `MalType[]`
  - uwaga: możesz spróbować najpierw napisać metodę pomocnicza `LList<(string Key,MalType Value)>? MalsToKeyValuePairs(LList<MalType>? mals)` tworzącą listę par kolejnych elementów listy
    - następnie w metodzie `ListToMap` wywołaj `new Map(new(MalsToKeyValuePairs(mals)));`
- zmień implementacje metod `ReadList` oraz `ReadForm` aby ich ciała były pojedynczymi "expression" C#
  - pamiętaj że rekurencja może zastąpić pętle
- pytanie/zadanie dodatkowe
  - obecna implementacja klasy `ReaderObj` używanej w metodzie `ReadLis` oraz `ReadForm` jest "mutowalna" (posiada "efekty uboczne") tzn. wykonanie metody `Next` zmienia wewnętrzny stan obiektu (zmienia aktualny `_positon`)
  - jak mogłoby wyglądać API oraz implementacja metod `ReadList,ReadForm` aby działały one jako "pure functions" ? (przykładową implementację dostarczono w przykładach)

Wprowadzenie środowiska `Env`

- dodaj plik `Env.cs`, w nim moduł `EnvM` (Environment Module)
- dodaj klasę/rekord wewnętrzną `Env` która posiada właściwości `Map<Symbol, MalType> Data` oraz `Env? Outer`
- `Env` reprezentuje "środowisko" czyli globalny dynamiczny obiekt do którego możemy dowolnie dodawać/usuwać elementy o danych nazwach, czyli jest opakowaniem obiektu mapy, ale dodatkowo każde środowisko posiada referencję do innego środowiska (może to być wartość `null`), gdy nie znajdziemy danego elementy w środowisku to szukamy w powiązanym z nim `Outer` (jeśli istnieje)
- do modułu `EnvM` dodaj funkcje
  - `MalType Set(this Env env,Symbol key, MalType value)`
    - funkcja dodaje `value` pod podanym kluczem `key` modyfikując zawartość `Env`, zwracana jest podana wartość `value`
  - `Env? Find(this Env env, Symbol key)`

        - funkcja szuka obiektu `Env` zawierającego przekazany `key`, jeśli przekazany `env` nie posiada `key` rekurencyjnie sprawdzany jest powiązany obiekt `Env? Outer`
    - `MalType Get(this Env env, Symbol key)`
        - funkcja zwraca wartość `MalType` zapisaną po przekazanym `key`, przeszukiwane są kolejne `Env? Outer`, jeśli `key` nie zostanie znaleziony to zwracany jest `Exception`

Operacje arytmetycznych oraz operacje porównania w `Core` (wbudowane funkcje)

- dodaj plik `Core.cs`, w nim moduł `Core`
- moduł `Core` posiada jedną publiczną właściwość statyczną `Map<Symbol, MalType> Ns` (Namespace) zawierającą definicję wbudowanych w język MAL operacji "globalnych"
    - chodzi o zarejestrowanie w mapie kolejnych par dla operatorów `+`, `-`, ... czyli par `(new Symbol("+"), new Fn(arg => ...)`
    - sygnatura wszystkich operacji zgodna będzie z sygnaturą `delegate MalType FnDelegate(LList<MalType>? args)`, czyli zdaniem powyżej lamndy jest
        - sprawdzenie poprawności argumentów, rzucenie `Exception` w przypadku błędnych
        - wykonanie koniecznych obliczeń
        - zwrócenie obiekty typu `MalType` jako rezultat
- zaczniemy od operacji arytmetycznych (`+, -, /, *`) oraz operacji porównania (`<, <=, >, >=`) gdzie
    - argumenty tych operacji muszą być typu `Number`, w przeciwny wypadku zwracany jest `Exception`
    - przykładowe wywołania operacji wyglądać będą `(+ 1 2)` lub `(< 1 2)`
    - (opcjonalnie) operacje arytmetyczne mogą wspierać więcej jak 2 argumenty czyli `(+ 1 2 3 4)`
- uwaga
    - implementację operacji warto wyciągnąć do osobnych funkcji pomocniczych, ponieważ wygodnie możemy je testować za pomocą testów jednostkowych
    - przykładowo możemy zdefiniować funcję `internal static MalType Plus(LList<MalType>? args) { ... }`, następnie użyć jej tworząc obiekt Namespace `Ns = MapFrom(LListFrom( (new Symbol("+"), new Fn(Plus)), ... ))`

Ewaluacja operacji arytmetycznych

- dodaj plik `Eval.cs`, w nim moduł `EvalM` (Evaluation Module)
- dodaj funkcję `MalType EvalAst(MalType mal, Env env)`
    - jeśli `mal` jest typu `Symbol`, zwracana jest wartość przechowywana w `env` dla danego symbolu
    - jeśli `mal` jest typu `List` (zarówno `ListType.List` jak i `ListType.Vector`)
        - dla każdego elementu tej listy wykonaj poniżej opisaną funkcję `Eval`
        - zwróć nową listę (zachowując `ListType`) składającą się z elementów będących wynikiem wywołania `Eval`
    - jeśli `mal` jest typu `Map`

        - zwróć nowy obiekt typu `Map` którego klucze skopiowane są z `mal`, natomiast dla wartości wykonano funkcję `Eval`
    - w przeciwnym przypadku, zwróć przekazaną wartość `mal`
- dodaj funkcję `MalType Eval(MalType mal, Env env)`
    - jeśli `mal` nie jest listą lub jest listą ale typu `ListType.Vector` (chodzi nam jedynie o listy `(...)`, nie wektory `[...]`)
        - zwróć wynik wywołania funkcji `EvalAst` dla `mal`
    - jeśli `mal` jest listą pustą, zwróć `mal` bez zmian
    - w przeciwny razie (`mal` jest niepustą listą), zakładany że mamy do czynienia z wywołaniem funkcji np `(+ 1 2)`
        - wykonaj `EvalAst` dla `mal`, zwrócony zostanie wynik (także lista) `l`
        - załóż że pierwszy element listy `l` będzie typu `Fn`, a pozostałe elementy są dowolnymi `MalType` (argumenty funkcji)
            - wykonaj delegat przechowywany w obiekcie `Fn` przekazując argumenty funkcji (elementy listy `l` bez pierwszego)
            - wynikiem implementowanej funkcji `Eval` jest wynik wywołania delegatu
        - jeśli lista `l` nie posiada elementów odpowiedniego typu, zwróć `Exception`

Pełna pętla REPL `Reader` -> `Eval` -> `Print`

- zmień główna pętlę aplikacji w `Program.Main`
    - upewnij się, że nie korzystasz z poprzednich implementacji `Types1`, `Reader1`, `Print1`
    - na starcie aplikacji stwórz obiekt `Env` zawierający wszystkie wbudowane operatory z `Core.Ns`
    - główna pętla aplikacja wykonuje `Reader.ReadText` -> `EvalM.Eval` -> `Printer.PrintStr`
- uruchom aplikację i wpisz przykładowe fragmenty kodu MAL: `(+ 1 2)`, `(/ 9 (+ 1 2) )`
- **gratulacje !!** zaimplementowałeś własny interpreter języka Lisp :)

Modyfikowanie oraz tworzenie własnego środowiska `Env` (`def!, let*`)

- uwaga
    - w dalszej części ćwiczenia bardzo często będziemy rozbudowywać implementację funkcji `Eval` o nowe przypadki, zazwyczaj będziemy sprawdzali czy pierwszy element listy to symbol o konkretnej nazwie `def!`, `let`, `do`, `if`, ...
    - kolejne przypadki warto implementować jako dedykowane funkcje np nazywając je `ApplyDef`, `ApplyLet`, ... , pozwoli to nam na wygodne pisanie testów jednostkowych oraz funkcja `Eval` zbytnio się nie rozrośnie
- dodaj nowe warunki w funkcji `Eval`
    - jeśli `mal` jest jest `(def! ... )` (czyli listą gdzie pierwszy element jest `Symbol` z nazwą `def!`)
        - to oznacza że definiujemy nową zmienną w aktualnym środowisku
        - `(def! a 6)` -> `6`, `(def! b (+ a 2))` -> `8`
        - pierwszy argument powinien być typu `Symbol`

        - drugi argument może być dowolnym typem MAL
            - wykonujemy dla niego `Eval` a wynik ustawiamy jako wartość symbolu na aktualnym środowisku `Env env` (drugi parametru funkcji `Eval`), czyli `env.set(pierwszy_arg, Eval(drugi_arg))`
            - wynikiem funkcji `Eval` jest wynik wywołania `env.set`
        - jeśli argumenty nie spełniają powyższych warunków, zwracany jest `Exception`
    - jeśli `mal` jest `(let* ...)`
        - to oznacza że wykonujemy (ostatnie) wyrażenie MAL poprzedzone definicją zmiennych pomocniczych
        - `(let* (a 1 b (+ a 4)) (* a b 10))` -> `50`
        - pierwszy argument jest listą gdzie parzyste elementy to symbole (nazwy zmiennych), a nieparzyste to dowolne wyrażenia MAL (wartości kolejnych zmiennych)
        - drugi argument jest dowolnym wyrażeniem MAL wykonanym w kontekście nowo-stworzonego środowiska zawierającego wszystkie zmienne stworzone wcześniej
        - implementując `let*` należy
            - stworzyć nowe środowisko `Env` którego `Outer` jest tym przekazanym do funkcji `Eval`, czyli `newEnv = new Env(null, env)`
            - kolejne pary elementów w liście parametrów dodawane są do nowego środowiska `newEnv.set(parzysty, Eval(nieparzysty))`
            - na końcu wykonaj `Eval(drugi_arg, newEnv)` co jest rezultatem implementowanej funkcji `Eval`
        - jeśli argumenty nie spełniają powyższych warunków, zwracany jest `Exception`

Blok kodu, If-then-else, własne funkcje (`do, if, fn*`)

- dodaj nowe warunki w funkcji `Eval`
    - jeśli `mal` jest `(do ...)`
        - to oznacza że definiujemy nowy blok składający się w wielu wyrażeń MAL, wykonywane są kolejne wyrażenie a rezultatem `do` jest rezultat ostatniego wyrażenia
        - `(do (println "mama") (println "tata") 6)` -> `6`
        - implementując `do` należy wykonywać funkcję `Eval` dla kolejnych elementów, wynik `Eval` ostatniego elementu jest wynikiem całego `do`
        - jeśli argumenty nie spełniają powyższych warunków, zwracany jest `Exception`
    - jeśli `mal` jest `(if ...)`
        - to oznacza że definiujemy typową konstrukcję “if-then-else”, która w języku MAL jest wyrażeniem (zwraca rezultat)
        - `(if (> 2 1) 10 20)` -> `10`
        - pierwszy argument (“if-”) jest ewaluowany
            - jeśli rezultat jest inny jak `nil` lub `false` (tzn. warunek jest “prawdziwy”), to zwracany jest wynik ewaluacji drugiego argumentu ("-then-")
            - w przeciwnym razie (warunek jest “fałszywy”) zwracany jest wynik ewaluacji trzeciego argumentu ("-else"), jeżeli trzeci argument ("-else") nie został podany to zwracany jest `nil`

        - jeśli argumenty nie spełniają powyższych warunków, zwracany jest `Exception`
    - jeśli `mal` jest `(fn* ...)`
        - to oznacza że definiujemy funkcję anonimową (w wielu językach nazywaną także "wyrażeniem lambda")
        - `( (fn* (a b) (+ a b)) 2 3)` -> `5`
        - pierwszy argument jest listą symboli reprezentujących nazwy parametrów funkcji
        - drugi argument jest dowolnym wyrażeniem MAL reprezentującym ciało funkcji
        - implementując `fn*` faktycznie tworzony jest element języka MAL `Fn` przyjmujący delegat C#, którego wywołanie:
            - tworzy nowy `Env` którego `Outer` ustawiony jest na ten przekazany do funkcji `Eval` (dzięki temu wspierane jest domknięcie/closure !!!)
            - dla każdej pary parametr + argument (- wartość parametru przekazana w momencie wykonania funkcji) dodawana jest zmienna w nowo-stworzonym środowisku `newEnv.set(parametr, argument)`
            - w ostanim kroku ewaluowane jest ciał funkcji z przekazaniem nowo-stworzonego środowiska
        - jeśli argumenty nie spełniają powyższych warunków, zwracany jest `Exception`
        - opcjonalne zadanie
            - wiele języków programowania wspiera funkcje przyjmujące zmienną ilość parametrów, przykładowo w C# możemy napisać `void Fun(string first, string second, params string[] others) { ... }`
            - analogiczną funkcję w języku MAL możemy zapisać `(def! fun1 (fn* (first second & others) ...))`
            - parametr nazwie `&` traktowany jest specjalnie tzn. pomijany jest na etapie bindowania argumentów do parametrów oraz parametr znajdujący się po nim (u nas `other`) jest listą zawierającą dodatkowe argumenty
            - wywołanie naszej funkcji `(fun1 1 2 3 4 5)` przypisze argumenty: `first=1`, `second=2`, `others=(3,4,5)`

Więcej wbudowanych funkcji w `Core`

- w ramach tego ćwiczenia dodamy wiele funkcji pomocniczych do modułu `Core`, nie musimy wszystkich implementować, ale warto zaimplementować główne funkcje operujące na listach

Funkcje działające na kolekcjach (listach)

- kolekcje danych w MAL
    - z punktu widzenia programisty języka MAL jedyną kolekcją jaką mamy jest wektor `[1,2,3]`, napisanie tego samego dla listy `(1,2,3)` powoduje, błąd ponieważ interpreter traktuje listę jako kod programu gdzie pierwszy element powinien być symbolem (wykonywaną operacją)
    - aby wykorzystać listę jako kolekcję danych, możemy np. wywołać funkcje pomocniczą `(list 1 2 3)` -> `(1 2 3)`
    - lista vs wektor:

        - zarówno lista jak i wektor są “immutable” tzn. operacja dodania/usunięcie/modyfikacji kolekcji zwraca nową kolekcję uwzględniającą zmiany
        - lista `(1,2,3)` działa jak lista jednokierunkowa, gdzie wygodnie możemy dodawać elementy na początek
        - wektor `[1,2,3]` działa bardziej jak “po-indeksowana kolekcja” gdzie wygodnie (wydajnie) możemy czytać n-ty element oraz zmieniać n-ty element np. dodawać element na koniec
        - w dalszej części ćwiczenia zaimplementujemy funkcje `(conj ...)` która dodaje elementy, do listy na początek, do wektora na koniec
- ogólne uwagi do poniższych funkcji operujących na listach:
    - większość z poniższych funkcji powinna działać dla obu typów listy `ListType { List, Vector }`, gdy funkcja przyjmuje kolekcję i zwraca nową kolekcję, to zazwyczaj jest to lista (nie wektor, nawet gdy przekazaliśmy wektor)
    - jeśli argumenty poniższych funkcji nie są odpowiedniego typu, zwracany jest `Exception`
- `(list 1 2 3)` - tworzy nowa listę typu `ListType.List` zawierającą `1, 2, 3`
- `(vector 1 2 3)` - tworzy nowa listę typu `ListType.Vector` zawierającą `1, 2, 3`
- `(cons 0 (list 1 2 3))` - dodaje element na początek listy
- `(concat (list 1 2) (list 3 4) [5 6])` - łączy wiele list w jedną
- `(conj (list 1 2) 4 5 6))`, `(conj [1 2] 4 5 6))` - dodaje elementy na początek w przypadku listy oraz na koniec w przypadku wektora
- `(count (list 1 2 3))` - zwraca ilość elementów listy
- `(first (list 1 2 3))` - zwraca pierwszy element, `nil` gdy kolekcja jest pusta
- `(rest (list 1 2 3))` - zwraca listę bez pierwszego elementu, jeśli argument jest `nil` lub pustą listą to zwracana jest pusta lista
- `(nth (list 1 2 3) 0)` - zwraca element o podanym indeksie, `Exception` gdy nie ma wystarczającej ilości elementów
- `(empty? (list 1 2 3))` - zwraca `true` jeśli listą jest pusta, `false` w przeciwnym wypadku
- `(list? (list 1 2 3))` - zwraca `true` jeśli argument jest listą typu `ListType.List`, `false` w przeciwnym wypadku
- `(vec (list 1 2 3))` - funkcja konwertuje listę do wektora zawierającego te same elementy, gdy przekazany zostanie wektor to zwracana jest ta sama instancja wektora

Funkcje działające na map-ach

- `(assoc {} "a" 1 "b" 2)` - dodaje do mapy kolejne pary klucz-wartość, nadpisuje elementy jeśli już istnieją
- `(dissoc {"a" 1 "b" 2 "c" 3} "a" "b")` - usuwa elementy o podanych kluczach
- `(get {"a" 1 "b" 2 } "a")` - zwraca wartość dla podanego klucza, zwraca `nil` gdy element nie istnieje
- `(contains? {"a" 1 "b" 2} "a")` - zwraca `true` jeśli mapa zawiera element o podanym kluczu, `false` w przeciwnym wypadku
- `(keys {"a" 1 "b" 2})` - zwracali listę kluczy
- `(vals {"a" 1 "b" 2})` - zwracali listę wartości

- `(hash-map "name" "marcin" "age" 20)` - tworzy mapę z listy elementów, działa analogicznie

Proste funkcje pomocnicze

- `(str nil false [1 2])` - funkcja może przyjmować wiele argumentów, zwraca `Str` zawierający napis łączący separatorem `" "` rezultaty wywołań funkcji `Printer.Print(...)` dla każdego z elementów MAL
- `(println nil false [1 2])` - wywołuje funkcję `(str)`, rezultat wypisuje na konsolę i zwraca `nil`
- `(= (list 1 2) (list 1 2))` - przyjmuje dwa 2 argumenty i zwraca `true` gdy są tego samego typu i mają taką samą zawartość, wykonuje pod spodem istniejącą już funkcję `Types.MalEqual`
- `(nil? nil)` funkcja `nil?` zwraca `true` gdy argument jest wartością `nil`, w przeciwnym razie zwraca `false`
- `true?, false?, string?, number?, symbol?, list?, vector?, map?, fn?` - funkcje działają analogicznie do `(nil?)`
- `(sequential? [1 2 3])` - zwraca `true` gdy argument jest listą niezależnie od typu, w przeciwnym razie `false`

Funkcje korzystające z interpretera MAL

- `(read-string "(+ 1 2)")` - funkcja przyjmuje fragment kodu języka MAL jako string i zwraca drzewo AST, wykorzystuje pod spodem `Reader.ReadText(...)`, dla pustego tekstu zwracany jest `nil`
- `(eval (list + 1 2))` - przyjmuje argument będący AST języka MAL (fragmentem kodu MAL) i go ewaluuje, wykorzystuje pod spodem `Eval.Eval(...)`, odpowiednik funkcji `eval("...")` w JavaScript
  - gdy w module `Core` implementujemy funkcje wbudowane, to nie mamy dostępu do środowiska `Env`, w przypadku implementacji `(eval...)` chcemy wywołać `Eval.Eval(...)` który przyjmuje instancję `Env`
  - jedno podejście to tworzenie nowego `Env` zawierającego funkcje wbudowane `Core.Ns` za każdy razem gdy wywołujemy `(eval ...)`
  - drugie podejście to przeniesienie "rejestracji" funkcji `(eval ...)` z `Core` do głównej metody `Program.Main`, gdzie tworzony jest globalny obiekt `Env`, więc mamy do niego dostęp

> Because mal programs are regular mal data structures, you can dynamically generate or manipulate those data structures before calling `eval` on them. This isomorphism (same shape) between data and programs is known as "homoiconicity". Lisp languages are homoiconic and this property distinguishes them from most other programming languages.
> https://github.com/kanaka/mal/blob/master/process/guide.md#step-6-files-mutation-and-evil

`quote` i makra

`quote`, `quasiquotequote`

- w tym ćwiczeniu dodamy 4 nowe funkcje: `quote`, `quasiquote`, `unquote`, `splice-unquote`, ale w pierwszej kolejności postarajmy się zrozumieć jak one działają
  - `(def! lst (quote (b c)))` -> `(b c)`
    - operator `quote` zwraca wyrażenie przekazane jako argument bez ewaluowania go, czyli pozwala traktować fragment kodu MAL jako wartość (strukturę danych)
      - `(quote (1 2 3))` zwraca listę `(1 2 3)` i w tym przypadku jest to równoważne z `(list 1 2 3)`
      - `(quote (1 2 (+ 1 2)))` zwraca listę `(1 2 (+ 1 2))`, natomiast `(list 1 2 (+ 1 2))` zwróci listę `(1 2 3)` ponieważ `(+ 1 2)` zostanie wyliczony
      - `(quote (+ 1 2))` zwraca listę `(+ 1 2)`, natomiast `(list + 1 2)` zwraca `(#<function> 1 2)`
      - `(quote +)` zwraca listę `(+)`, natomiast `(list +)` zwraca `(#<function>)`
  - `(quasiquote (a lst d))` -> `(a lst d)`
    - `quasiquote` działa analogicznie do `quote` ale wewnątrz przekazanego argumentu (zazwyczaj listy) mogą znajdować się wyrażenie które chcemy wy-ewaluować przed zbudowaniem finalnej struktury, korzystamy wtedy z `unquote` lub `splice-unquote`
    - `(quasiquote (a (unquote lst) d))` -> `(a (b c) d)`
    - `(quasiquote (a (splice-unquote lst) d))` -> `(a b c d)`
- dodaj nowe warunki w funkcji `Eval`
  - jeśli `mal` jest `(quote arg)`
    - wymagany jest jeden argument, implementacja sprowadza się do zwrócenia wartości przekazanego argumentu `arg`
  - jeśli `mal` jest `(quasiquoteexpand arg)`
    - to wykonana zostanie transformacja argumentu MAL `arg` do nowej postaci MAL, `quasiquoteexpand` jest operacją pomocniczą pomagającą w testowaniu/diagnozowaniu działania głównej operacji `quasiquote` która zostanie zaimplementowana poniżej
    - wymagany jest jeden argument, w zależności od typu argumentu `arg`
      - jeśli jest listą postaci `(unquote un_arg)`, to zwracany jest `un_arg`
      - jeśli jest dowolną inna listą `(...)`
        - zakładając że lista ta zaimplementowana jest jako lista jednokierunkowa czyli `(head ...tail)`, analizowane będą kolejne elementy zaczynając od `head`, dla każdego z nich zwracana będzie nowa lista `(cons ... )` lub `(concat ...)`
        - jeśli `head` jest `(splice-unquote spl_arg)` zwracana jest lista `(concat spl_arg call_quasiquote(...tail))`, gdzie `call_quasiquote` jest rekurencyjnym wywołaniem funkcji którą implementujemy
          - `(quasiquoteexpand (a (splice-unquote lst) d))` -> `(cons (quote a) (concat lst (cons (quote d) ())))`
        - w przeciwnym przypadku zwracany jest lista `(cons call_quasiquote(head) call_quasiquote(...tail))`
          - `(quasiquoteexpand (a lst d))` -> `(cons (quote a) (cons (quote lst) (cons (quote d) ())))`

            - jeśli jest `Map` lub `Symbol`, to zwracana jest nowa lista opakowująca argument w `quote` czyli `(quote arg)`
            - w pozostałych przypadkach zwracany jest argument `arg`
        - uwaga: to jest najbardziej zagmatwana funkcja implementowana do tej pory, może warto po prostu przekleić implementację operacji `quasiquoteexpand` z dostarczonych przykładów :)
    - jeśli `mal` jest `(quasiquote arg)`
        - to wykonana zostanie transformacja argumentu MAL zgodnie z działaniem `quasiquoteexpand`, której rezultat zostanie przekazany do funkcji `Eval`, jej wynik jest wynikiem operacji `quasiquote`

Makra

- makro definiuje się w następujący sposób `defmacro! nazwa (fn* (...) ...)`, czyli bardzo podobnie do definicji funkcji `def! nazwa (fn* (...) ...)`
- o makrach możemy myśleć jak o specyficznym rodzaju funkcji ponieważ
    - zazwyczaj przyjmują argumenty
    - wywoływane są analogicznie do wywołań funkcji
    - zawierają zwykły kod sterujący programu MAL ...
    - ... który powinien zwracać strukturę danych MAL (będącą poprawnym programem MAL) która zostanie następnie wy-ewaluowana
- `(defmacro! execute-op (fn* (op arg1 arg2) (quasiquote ((unquote op) 10 (unquote arg1) (unquote arg2) ))))`
    - jest to przykładowe makro do którego możemy przekazać 3 argumenty (operator i dwie liczby) i zostanie zwrócony kod programu który wykonuje przekazany operator dla przekazanych liczb oraz dodatkowej liczy `10`
    - `(macroexpand (execute-op + 1 2 ))` -> `(+ 10 1 2)` - funkcja pomocnicza `(macroexpand ...)` pozwala na rozwinięcie makra bez jego wywoływania
    - `(execute-op + 1 2)` -> `13` - wykonanie makra
    - `(execute-op - 1 2 )` -> `7`
    - podobny efekt możemy osiągnąć w języku JavaScript
        - `const executeOp = (op, arg1, arg2) => `10 ${op} ${arg1} ${op} ${arg2}`;` - definicja makra
        - `executeOp("+", 1, 2)` -> `"10 + 1 + 2"` - rozwinięcie makra bez wywołania
        - `eval(executeOp("+", 1, 2))` -> `13` - wywołanie makra
- dodaj nową właściwość `bool IsMacro = false` do typu `Fn`, dzięki ustawieniu wartości domyślnej na `false` nie musimy zmieniać wszystkich istniejących wystąpień typu `Fn` w kodzie
- w module `EvalM` zdefiniuj funkcje pomocniczą `MalType MacroExpand(MalType mal, Env env) { ... }`
    - jeśli argument `mal` jest wywołaniem funkcji czyli jest, listą postaci `(funName arg1 arg2)` to
        - to wartość dla symbolu `funName` szukana jest w środowisku `env`
            - jeśli jej nie ma, to zwracany jest `nil`
            - jeśli jest i jest typu funkcji `Fn` której flaga `IsMacro` jest `true` to

            - funkcja ta jest wykonywana dla argumentów `arg1 arg2`
            - rezultat jej wywołania przekazywany jest do rekurencyjnego wywołania funkcji `MacroExpand`, rezultat tego wywołania jest zwracany jako rezultat `MacroExpand`
        - jeśli wartość jest ale innego innego typu jak wyżej, to zwracana wartość jest niezmieniona
    - w przeciwny razie zwracany jest niezmieniony argument `mal`
- w module `EvalM` zmień implementację głównej metody `MalType Eval(MalType mal, Env env)`
    - w pierwszym krok wykonaj funkcje `MacroExpand` jej rezultat przekaż do standardowej obsługi, o tej operacji możemy myśleć jak o `mal = MacroExpand(mal, env)`
- dodaj nowe warunki w funkcji `Eval`
    - jeśli `mal` jest `(macroexpand ...)`
        - to wykonujemy makro w trybie jego podglądu, bez wykonania
        - w implementacji jedynie wywołujemy wcześniej napisaną funkcję `MacroExpand` i zwracamy jej rezultat
    - jeśli `mal` jest `(defmacro! ...)`
        - to definiowane jest makro
        - `(defmacro! one (fn* () (quote 1)))`
        - składnia oraz zachowanie jest bardzo podobne do `def!`, pierwszy argument to symbol, drugi musi być funkcją (w przypadku `def!` MAL może być dowolny), do środowiska dodana jest nowa zmienna która jest makrem
        - implementacja `defmacro!` jest bardzo podobna do implementacji `def!`
            - wykonywana jest funkcja `Eval` dla drugiego argumentu (tutaj zawsze funkcji)
            - powstaje obiekt `Fn` dla którego ustawiamy flagę `IsMacro=true`
            - dodajemy do środowiska nowa zmienną o nazwie pierwszego argumentu i wartości powstałej funkcji

Uwagi końcowe do makr

- do bardziej rozbudowanych warunków “if-then-else” możemy wykorzystać funkcję `cond` (“conditional”)
    - `(cond true 7 true 8)` -> `7`
    - `(def! max (fn* (a b) (cond (> a b) a true b ) ))`
    - `(max 1 2)` -> `2`
    - `cond` przyjmuje parzystą ilość argumentów, element parzysty (licząc od zera) ewaluowany jest do prawdy/fałszu, ewaluacja pierwszego parzystego elementu do wartości prawdziwej kończy wykonanie zwracając element nieparzysty znajdujący się po nim
- funkcję `cond` możemy zdefiniować za pomocą makra

```
(defmacro! cond (fn* (& xs)
    (if (> (count xs) 0)
        (list (quote if) (first xs) (nth xs 1) (cons (quote cond) (rest (rest xs)))) )))
```

- `(macroexpand (cond true 7 true 8))` -> `(if true 7 (cond true 8))`

- “macros” vs “reader macros”
  - makra o których mówiliśmy do tej pory nazywane są po prostu “macros”, ale są także “reader macros”, tzn. na etapie działa readera tworzącego drzewa AST mogą pojawi się specjalne tokeny które zamieniane są “w locie” na inne tokeny.
  - często dla “quote” tworzymy takie makra, przykładowo program `'1` zamieniany jest na `(quote 1)`, analogicznie `` ` `` na `quasiquote`, `~` na `unquote`, `~@` na `splice-unquote`
- poprzenie makro moglibyśmy zapisać krócej

```
(defmacro! cond (fn* (& xs)
    (if (> (count xs) 0)
        (list 'if (first xs) (nth xs 1) (cons 'cond (rest (rest xs)))) )))
```