Софийски университет ,,Св. Климент Охридски”
Факултет по математика и информатика

Онтология за езици за програмиране

Курсов проект по
Представяне и моделиране на знания

Николай Димов Паев
Факултетен номер: 2MI3400435
Магистърска програма: Изкуствен Интелект

23.01.2024 гр. София

Съдържание

1. Въведение

В последните години езиците за програмиране стават все повече и повече. Всеки нов език се опитва да стане най-популярния, а създателите му претендират, че са взели най-добрите идеи от останалите. Това прави откриването на правилния инструмент за конкретна задача още по-сложно. Една онтология би помогнала за запознаването с езиците и техните характеристики и правилното класифициране на нов език.

Задачата обаче не е лесна, тъй като езиците нямат конкретен родител, а по-скоро са вдъхновени от поредица други. Те обикновено не следват една парадигма, а взимат по-нещо от различните. В допълнение те също имат и много характеристики, което прави подредбата им в йерархия сложна.

Езиците грубо се разделят на такива от високо и от ниско ниво. Тези от високо се характеризират с различни парадигми, типови системи, стил на писане, среди за изпълнение, различна работа с паметта и цели за ползване. Освен това често имат общи цели, сходен синтаксис, общи платформи, между тях има йерархия на влияния.

2. Съществуващи решения и източници

Първото изречение от параграфа таксономия в статията за езиците за програмиране в Уикипедия [1] гласи: ,,There is no overarching classification scheme for programming languages.” В същия параграф, авторът посочва подобни съображения, като тези описани във въведението. Въпреки това в редица други статии в Уикипедия, съществуват сравнения на езиците по една или друга ос. Статията ,,Сравнение на езиците за програмиране” [2] подрежда десетки езици по парадигми. Ще послужат и други статии, една от които сравнява езиците по тип [3] (без да е казано, какво точно е тип), както и сравнение по типова система [4]. Не на последно място ще бъдат използвани и познанията ми относно различни езици, придобити в множество избираеми курсове във факултета.

Налични са предишни решения за създаване на онтологии, но те засягат по-тясно конкретни езици или парадигма [5]. В настоящия проект е предложена онтология, която има по-широк поглед върху езиците, но и достатъчно детайлен, за да се покрият всички особености.

3. Онтология
3.1. Класове езици за програмиране

В онтологията са поместени, няколко йерархии: на езиците, на характеристиките, на типовите системи, на средите за изпълнение, на домейните и на подходите за менажиране с паметта. Централна е йерархията на езиците.

Родител е класът **Programming Language**. Всички класове, които имат за индивиди езици за програмиране са наследници на Programming Language.
Класовете са описани в термините на свойствата на техните индивиди - наличието на дадени характеристики. Сложната йерархия между тях е оставена имплицитна (чрез йерархията между характеристиките) и е необходимо машина на извод да я изведе наяве.
За имплементацията на онтологията е използвана библиотеката OwlReady 2 на Python.

Описание на класовете:
High Level Language е класа на тези програмни езици, които имат поне 1 абстракция.
Low Level Language е класа на тези, които нямат никакви абстракции.
Imperative Language е класа на програмните езици, които се изпълняват ред по ред.
Concurrent Language е класа на езиците, които позволяват конкурентно изпълнение.
Structured Language е класа на езиците, които имат структури в кода - цикли, блокове, функции и т.н.
Assembly Language е наследник на Imperative Language и Low Level Language и освен това има ограничение, че съдържа тези езици, които се изпълняват на Native Environment.
Procedural Language е класа на езиците, които имат процедури, известни още като субрутини или функции. Използвана е думата процедура, за да се прави разлика с функциите във функционалното програмиране, които се определят само от стойността, които връщат.
Declarative Language е класа на езиците, които се изпълняват чрез оценка на заявка или оценка на израз.
Functional Inspired Language е класа на езиците, вдъхновени от функционалното програмиране. Класа е дефиниран, като тези езици, в които функциите са “първокласни жители” и в които още има функции от по-висок ред.
Класът **Functional Language** е дефиниран, като наследник на Functional Inspired Language, но внася ограничението изпълнението на програмите на езика да се свежда до оценка на израз и данните в тях да са неизменяеми по подразбиране. **Pure Functional Language** разширява Functional Language като изисква наличие на чисти функции и стриктна неизменяемост. **Lazy Evaluated Language** разширява Pure Functional Language като добавя и изискване за лениво оценяване.

```
class FunctionalLanguage(ProgrammingLanguage):
    equivalent_to = [
        FunctionalInspiredLanguage &
        has_execution_type.value(evaluation_of_expression) &
        has_feature.some(Immutability)
    ]
```

Парче код 1. Дефиниция на класа Functional Language

Logic Language се дефинира като класа на езиците, които се описват с факти и правила. По подобен начин е дефиниран и **Query Language** - със съответно заявки.
OOP Inspired Language съдържа езиците, които имат ръчно дефинирани типове - структури. **OOP Language** разширява предния клас, като изисква наличие на класове и наследяване.
Multi Paradigm Language е наследник на OOP Inspired Language и Functional Inspired Language. **Statically Typed Language**, **Dynamically Typed Language**, **Strongly Typed Language** и **Weakly Typed Language** са класовете на езиците, които имат съответно, статична, динамична, стриктна и слаба типова система.
Safe Type Checking Language е наследник на Strongly Typed Language и Statically Typed Language.
Safe Language разширява Safe Type Checking Language като внася ограничението грешките да се третират като типизирани стойности.
Has Algebraic Types е класа на езиците, които имат Product Types и Sum Types.
Modern Functional Language обединява Pure Functional Language, Safe Type Checking Language и Has Algebraic Types.
Compiled To Bytecode Language и **Compiled To Machine Code Language** са дефинирани като еквивалентни съответно на езиците, които се изпълняват на Native Environment и на Bytecode интерпретатор.
Compiled Language е обединението на Compiled To Bytecode Language и Compiled To Machine Code Language.
Interpreted Language е език, който се изпълнява на интерпретатор.
Scripting Language е език, който се интерпретира ред по ред.
Shell Language е дефиниран като език, който се използва за работа с операционната система.
JVM Language е език, който се изпълнява на Java Virtual Machine.
Garbage Collected Language е еквивалентен на множеството от езиците, които имат автоматично менажиране на паметта.

```
class GarbageCollectedLanguage(ProgrammingLanguage):
    equivalent_to = [
        has_memory_management.some(AutomaticMemoryManagement)
    ]
```

Парче код 2. Дефиниция на класа Garbage Collected Language

Reference Counted Language е множеството от езиците, които използват техниката за reference counting за грижа за паметта. Т.к. тази техника е инстанция на Automatic Memory Management, Reference Counted Language имплицитно е наследник на Garbage Collected Language.
Multi Purpose Language съдържа езиците, които имат минимум 2 употреби.
General Purpose Language съдържа езиците, които имат поне 3 случаи за употреба.
Influential Language и **Influenced Language** съдържат езиците, които са вдъхновили или съответно вдъхновени от поне 2 езика.
Father Language е език, вдъхновил, поне 3 езика.
Класовете са описани по-формално на езика DL в таблица 1:

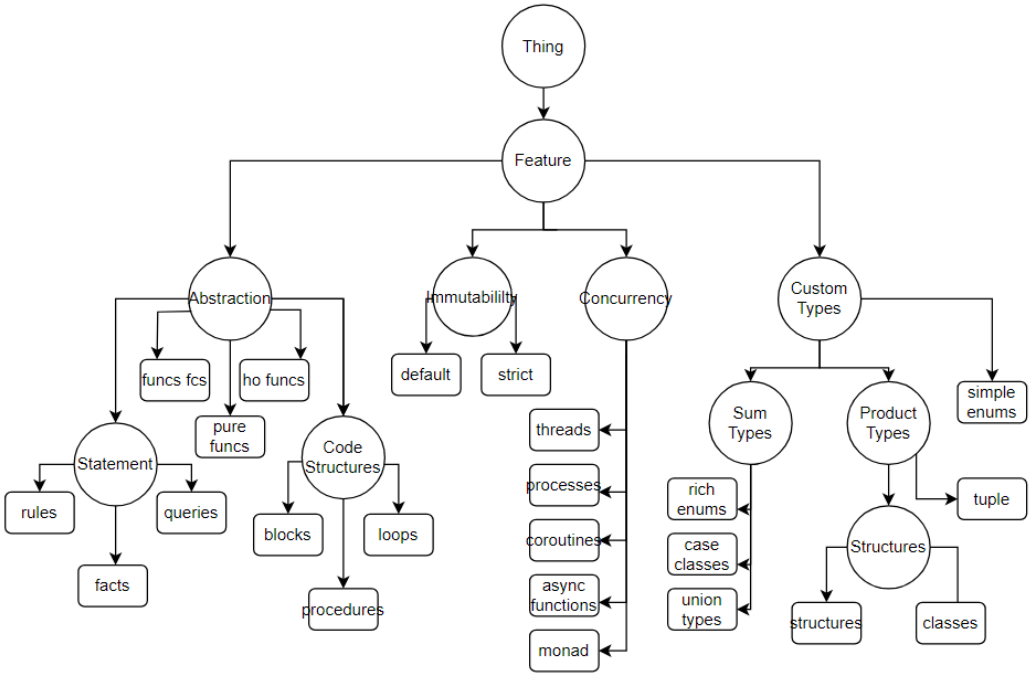
ProgrammingLanguage	⊑ Thing
HighLevelLanguage	[AND ProgrammingLanguage [EXISTS 1 :has_feature Abstraction]]
LowLevelLanguage	[AND ProgrammingLanguage [NOT [EXISTS 1 :has_feature Abstraction]]]
ConcurrentLanguage	[AND ProgrammingLanguage [EXISTS 1 :has_feature Concurrency]]
ImperativeLanguage	[AND ProgrammingLanguage [FILLS :has_execution_type line_by_line_execution]]
StructuredLanguage	[AND ProgrammingLanguage [EXISTS 1 :has_feature Concurrency]]
AssemblyLanguage	[AND ImperativeLanguage LowLevelLanguage [FILLS :runs_on native_environment]]
ProceduralLanguage	[AND ProgrammingLanguage [FILLS :has_feature procedures]]
DeclarativeLanguage	[AND ProgrammingLanguage [ALL :has_execution_type [ONE-OF evaluation_of_query evaluation_of_expression]]]
FunctionalInspiredLanguage	[AND ProgrammingLanguage [FILLS :has_feature functions_as_first_class_citizens] [FILLS :has_feature higher_order_functions]]
FunctionalLanguage	[AND FunctionalInspiredLanguage [FILLS :has_execution_type evaluation_of_expression] [EXISTS 1 :has_feature Immutability]]
PureFunctionalLanguage	[AND FunctionalLanguage [FILLS :has_feature pure_functions] [FILLS :has_feature strict_immutability]]
LazyEvaluatedLanguage	[AND PureFunctionalLanguage [FILLS :has_feature lazy_evaluation]]
LogicLanguage	[AND ProgrammingLanguage [FILLS :has_feature facts] [FILLS :has_feature rules]]
QueryLanguage	[AND ProgrammingLanguage [FILLS :has_feature query]]
OOPInspiredLanguage	[AND ProgrammingLanguage

	[EXISTS 1 :has_feature Structures]]
OOPLanguage	[AND ProgrammingLanguage [FILLS :has_feature classes] [FILLS :has_feature inheritance]]
MultiParadigmLanguage	[AND OOPInspiredLanguage FunctionalInspiredLanguage]
StaticallyTypedLanguage	[AND ProgrammingLanguage [FILLS :has_type_checking static_type_checking]]
DynamicallyTypedLanguage	[AND ProgrammingLanguage [FILLS :has_type_checking dynamic_type_checking]]
StronglyTypedLanguage	[AND ProgrammingLanguage [FILLS :has_type_safety strong_type_safety]]
WeaklyTypedLanguage	[AND ProgrammingLanguage [FILLS :has_type_safety weak_type_safety]]
SafeTypeCheckingLanguage	[AND StaticallyTypedLanguage StronglyTypedLanguage]
SafeLanguage	[AND SafeTypeCheckingLanguage [FILLS :has_error_handling_type errors_as_sum_types]]
HasAlgebraicTypes	[AND ProgrammingLanguage [EXISTS 1 :has_feature ProductTypes] [EXISTS 1 :has_feature SumTypes]]
ModernFunctionalLanguage	[AND PureFunctionalLanguage SafeTypeCheckingLanguage HasAlgebraicTypes]
CompiledToByteCodeLanguage	[AND ProgrammingLanguage [EXISTS 1 :runs_on ByteCodeInterpreter]]
CompiledToMachineCodeLanguage	[AND ProgrammingLanguage [FILLS :runs_on native_environment]]
CompiledLanguage	[ONE-OF CompiledToByteCodeLanguage CompiledToMachineCodeLanguage]
InterpretedLanguage	[AND ProgrammingLanguage [EXISTS 1 :runs_on Interpreter]]
ScriptingLanguage	[AND ProgrammingLanguage [EXISTS 1 :runs_on LineByLineInterpreter]]

ShellLanguage	[AND ProgrammingLanguage FILLS :used_for os_communication]
JVMLanguage	[AND ProgrammingLanguage [FILLS :runs_on jvm]]
GarbageCollectedLanguage	[AND ProgrammingLanguage [EXISTS 1 :has_memory_management AutomaticMemoryManagement]]
ReferenceCountedLanguage	[AND ProgrammingLanguage [FILLS :has_memory_management reference_counting]]
MultiPurposeLanguage	[AND ProgrammingLanguage [EXISTS 2 :used_for]]
GeneralPurposeLanguage	[AND ProgrammingLanguage [EXISTS 3 :used_for]]
InfluencedLanguage	[AND ProgrammingLanguage [EXISTS 2 :inspired_by]]
InfluentialLanguage	[AND ProgrammingLanguage [EXISTS 2 :inspired]]
FatherLanguage	[AND ProgrammingLanguage [EXISTS 3 :inspired]]

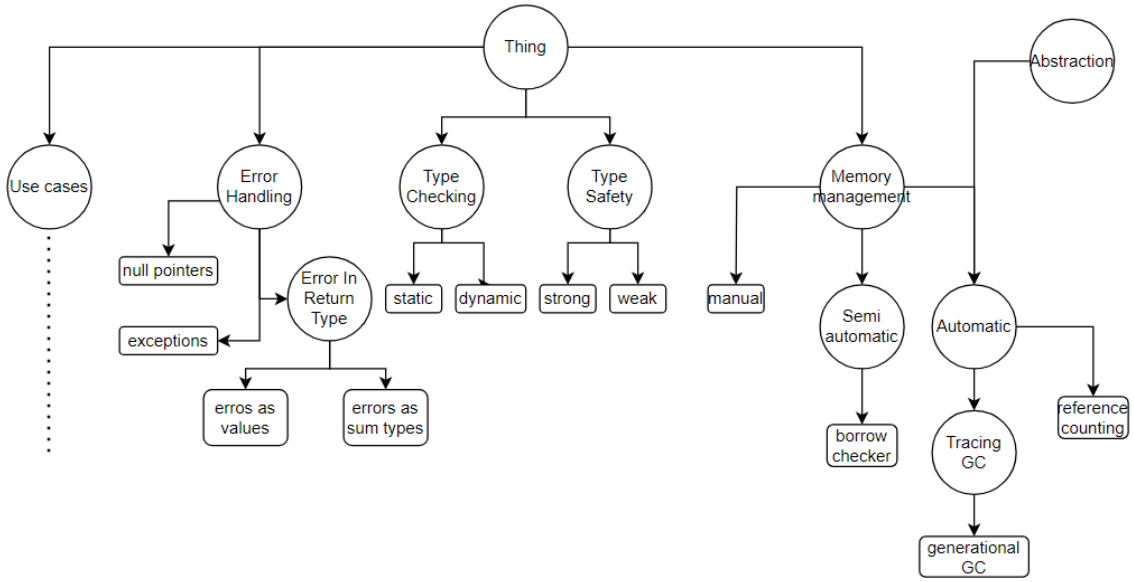
Таблица 1. Дефиниции на класове на езици за програмиране

3.2. Класове на характеристики



Фиг 1. Йерархия на Feature. С кръгчета са изобразени класовете,
а с правоъгълници - индивидите

Feature е наследник на Thing. **Abstraction** наследява Feature и негови наследници са **Statements** и **Code Structures**. Statements има индивиди: query, rules, facts, function definition. Code Structures има индивиди: loops, blocks, procedures. Други наследници на Feature са **Immutability**, **Concurrency** и **Custom Types**.
Immutability има инстанции: default immutability и strict immutability.
Concurrency има инстанции: threads, coroutines, processes, async functions, monad concurrency.
Custom Types се поделят на два вида: **Sum Types** и **Product Types**.
Sum types биват индивидите: rich enums, case classes и union types
Structures наследява Product Types и негови инстанции са: structures и classes.
Инстанция на Product Types е tuples, а на Feature е simple enum.



Фиг 2. Йерархия на част от останалите класове

Memory Management е наследник на Thing.
Automatic Memory Management е наследник на Memory Management и Abstraction.
Инстанция на Automatic Memory Management е reference Counting,
Tracing GC е наследник на Automatic Memory Management.
Негова инстанция е g1.
Semi Automatic Memory Management е наследник на Memory Management и има инстанция: borrow checker.
Накрая manual memory management е инстанция на Memory Management

Type Checking и **Type Safety** са наследници на Thing. Те имат съответно инстанциите: static type checking и dynamic type checking и strong type safety и weak type safety.

Error Handling също е наследник на Thing. Негови инстанции са null pointers и exceptions, а наследник му е **Error In Return Type** с инстанции: errors as sum types и errors as values.

Use cases е наследник на Thing. Негови инстанции са: web, mobile, desktop, enterprise, cloud, highly concurrent applications, scientific computing, ai, os, game, compiler, embedded development, database management, databases, financial software, research

Последните класове са свързани със средата за изпълнение:
Running Environment наследява Thing. Негов наследник е **Interpreter**. Наследници на Interpreter са **Line By Line Interpreter** и **Bytecode Interpreter**.
bash interpreter и python interpreter са инстанции на Line By Line Interpreter.
native environment е инстанция на Running Environment.
jvm, dot net vm, ruby vm, php interpreter са инстанции на ByteCode Interpreter.
v8 engine, scheme repl, prolog interpreter и др. са инстанции на Interpreter.

По-формално:

Feature ⊑ Thing Abstraction ⊑ Feature Statements ⊑ Abstraction CodeStructures ⊑ Abstraction Immutability ⊑ Feature Concurrency ⊑ Feature CustomTypes ⊑ Feature ProductTypes ⊑ CustomTypes Structures ⊑ ProductTypes SumTypes ⊑ CustomTypes
ExecutionType ⊑ Thing
MemoryManagement ⊑ Thing AutomaticMemoryManagement ⊑ MemoryManagement U Abstraction TracingGC ⊑ AutomaticMemoryManagement SemiAutomaticMemoryManagement ⊑ MemoryManagement
TypeChecking ⊑ Thing
TypeSafety ⊑ Thing
ErrorHandling ⊑ Thing ErrorInReturnType ⊑ ErrorHandling
UseCase ⊑ Thing
RunningEnvironment ⊑ Thing Interpreter ⊑ RunningEnvironment LineByLineInterpreter ⊑ Interpreter ByteCodeInterpreter ⊑ Interpreter

Таблица 2. Дефиниции на класове на характеристики

3.3. Свойства:
Свойствата свързват йерархията на езиците с тази на характеристиките. Всички изводи, които онтологията поддържа са получени от тях.

Имената им дават ясна представа за действието им, затова по-подробни обяснения ще бъдат пропуснати.

Свойство	Домейн	Обхват	Ограничения
used_for	Programming Language	UseCase	
has_similar_syntax_to	Programming Language	Programming Language	еквивалентност
inspired_by	Programming Language	Programming Language	транзитивност
inspired	Programming Language	Programming Language	транзитивност обратно на inspired_by
inter_op_with	Programming Language	Programming Language	
runs_on	Programming Language	Running Environment	функционално
has_feature	Programming Language	Feature	
has_execution_type	Programming Language	Execution Type	функционално
has_error_handling_type	Programming Language	Error Handling	функционално
has_type_checking	Programming Language	Type Checking	функционално
has_type_safety	Programming Language	Type Safety	функционално
has_memory_management	Programming Language	Memory Management	функционално

Таблица 3. Свойства

3.4. Индивиди на програмни езици

В онтологията са включени над 20 индивиди. Тук са показани са само някои, които по-късно ще участват в примерните изводи:

```
scala = ProgrammingLanguage("scala",
    has_feature = [
        blocks,
        loops,
        classes,
        functions_as_first_class_citizens,
        higher_order_functions,
        rich_enums,
        case_classes,
        monad_concurrency,
        async_functions,
        tuples,
        default_immutability
    ], has_execution_type = [
        evaluation_of_expression,
        line_by_line_execution
    ], has_memory_management = [generational_GC]
    , runs_on = [jvm]
    , has_type_checking = [static_type_checking]
    , has_type_safety = [strong_type_safety]
    , has_error_handling_type =
[errors_as_sum_types]
    , used_for = [
        cloud_applications,
        highly_concurrent_applications
    ], inspired_by = [
        java,
        haskell
    ])
```

```
c_pp = ProgrammingLanguage("c++",
    has_feature = [
        blocks,
        procedures,
        loops,
        classes,
        functions_as_first_class_citizens,
        higher_order_functions,
        inheritance,
        simple_enums,
        threads
    ], has_execution_type = [line_by_line_execution]
    , has_memory_management =
[manual_memory_management]
    , runs_on = [native_environment]
    , has_type_checking = [static_type_checking]
    , has_type_safety = [weak_type_safety]
    , has_error_handling_type = [
        null_pointers,
        exceptions
    ], used_for = [
        game_development,
        desktop_applications,
        enterprise_applications,
        os_development,
        embedded_development,
        compiler_development,
        database_management
    ], inter_op_with = [c]
    , inspired_by = [c]
    , has_similar_syntax_to = [c]
    )
```

```
rust = ProgrammingLanguage("rust",
    has_feature = [
        blocks,
        loops,
        structures,
        functions_as_first_class_citizens,
        higher_order_functions,
        rich_enums,
        threads,
        async_functions,
        tuples,
        default_immutability
    ], has_execution_type = [line_by_line_execution]
    , has_memory_management = [borrow_checker]
    , runs_on = [native_environment]
```

```
php = ProgrammingLanguage("php",
    has_feature = [
        blocks,
        loops,
        classes,
        functions_as_first_class_citizens,
        higher_order_functions,
        inheritance,
        simple_enums,
        threads
    ], has_execution_type = [line_by_line_execution]
    , has_memory_management =
[generational_GC]
    , runs_on = [php_interpreter]
    , has_type_checking = [dynamic_type_checking]
```

```
  , has_type_checking = [static_type_checking]
  , has_type_safety = [strong_type_safety]
  , has_error_handling_type =
[errors_as_sum_types]
  , used_for = [
     embedded_development,
     ai_and_ml,
     game_development,
     os_development,
     compiler_development,
     database_management
  ], inspired_by = [
     c_pp,
     haskell
  ], has_similar_syntax_to = [c, c_pp]
  )
```

```
  , has_type_safety = [weak_type_safety]
  , has_error_handling_type = [
     null_pointers,
     exceptions
  ], used_for = [web_applications]
  , has_similar_syntax_to = [c]
  )
```

Таблица 4. Дефиниции на някои индивиди

3.5. Автоматични изводи

Направени са изводи с алгоритъма за извод HermiT. Езиците са причислени към нови родителски класове, чрез изводи върху свойствата им. Например:

```
c_pp.is_a
✓ 0.0s
[programming_languages.CompiledToMachineCodeLanguage,
 programming_languages.CurlyBracketLanguages,
 programming_languages.FatherLanguage,
 programming_languages.WeaklyTypedLanguage,
 programming_languages.MultiParadigmLanguage,
 programming_languages.ImperativeLanguage,
 programming_languages.OOPLanguage,
 programming_languages.GeneralPurposeLanguage,
 programming_languages.StaticallyTypedLanguage,
 programming_languages.ProceduralLanguage,
 programming_languages.ConcurrentLanguage]
```

```
scala.is_a
✓ 0.0s
[programming_languages.InfluencedLanguage,
 programming_languages.JVMLanguage,
 programming_languages.SafeLanguage,
 programming_languages.HasAlgebraicTypes,
 programming_languages.MultiParadigmLanguage,
 programming_languages.ImperativeLanguage,
 programming_languages.StructuredLanguage,
 programming_languages.GarbageCollectedLanguage,
 programming_languages.OOPLanguage,
 programming_languages.ConcurrentLanguage,
 programming_languages.MultiPurposeLanguage,
 programming_languages.FunctionalLanguage]
```

```
php.is_a
✓ 0.0s
[programming_languages.CurlyBracketLanguages,
 programming_languages.CompiledToByteCodeLanguage,
 programming_languages.WeaklyTypedLanguage,
 programming_languages.MultiParadigmLanguage,
 programming_languages.ImperativeLanguage,
 programming_languages.StructuredLanguage,
 programming_languages.DynamicallyTypedLanguage,
 programming_languages.GarbageCollectedLanguage,
 programming_languages.OOPLanguage,
 programming_languages.ConcurrentLanguage]
```

```
haskell.is_a
✓ 0.0s
[programming_languages.CompiledToMachineCodeLanguage,
 programming_languages.SafeLanguage,
 programming_languages.FatherLanguage,
 programming_languages.LazyEvaluatedLanguage,
 programming_languages.ModernFunctionalLanguage,
 programming_languages.GarbageCollectedLanguage,
 programming_languages.GeneralPurposeLanguage,
 programming_languages.ConcurrentLanguage]
```

Таблица 5. Изводи върху някои езици

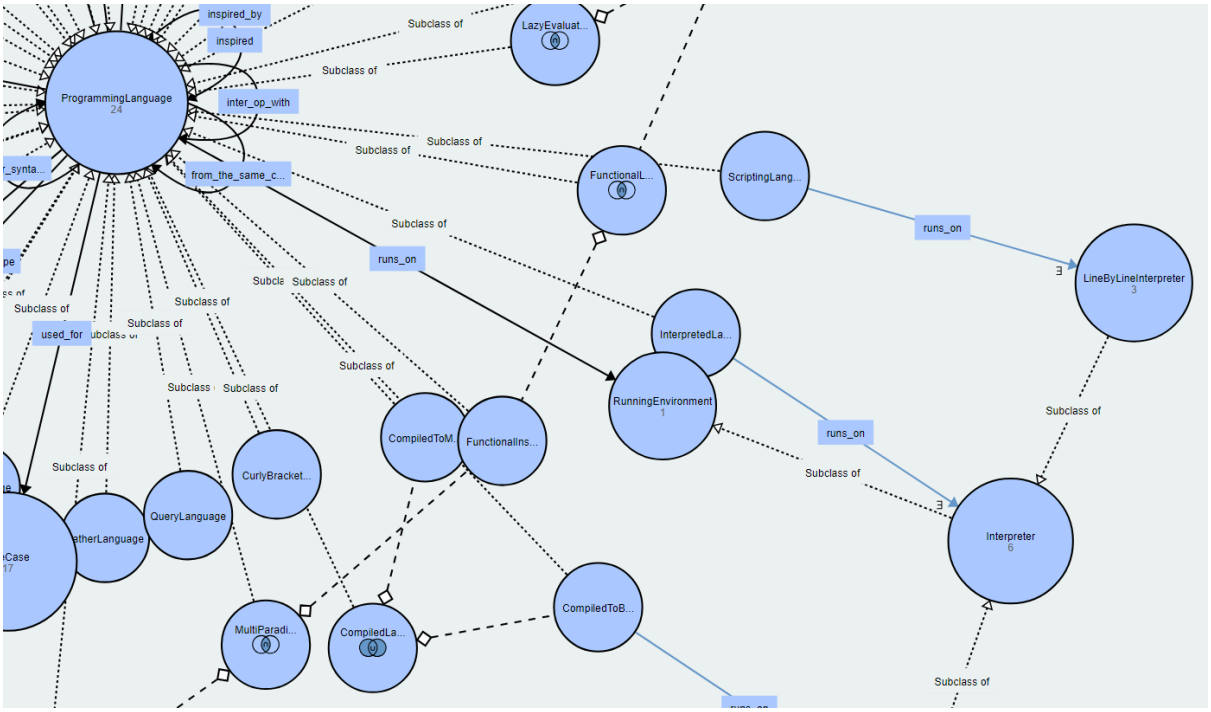
Reasoner-а също променя йерархията на класовете. Класът Functional Language е правилно причислен към Declarative Language.

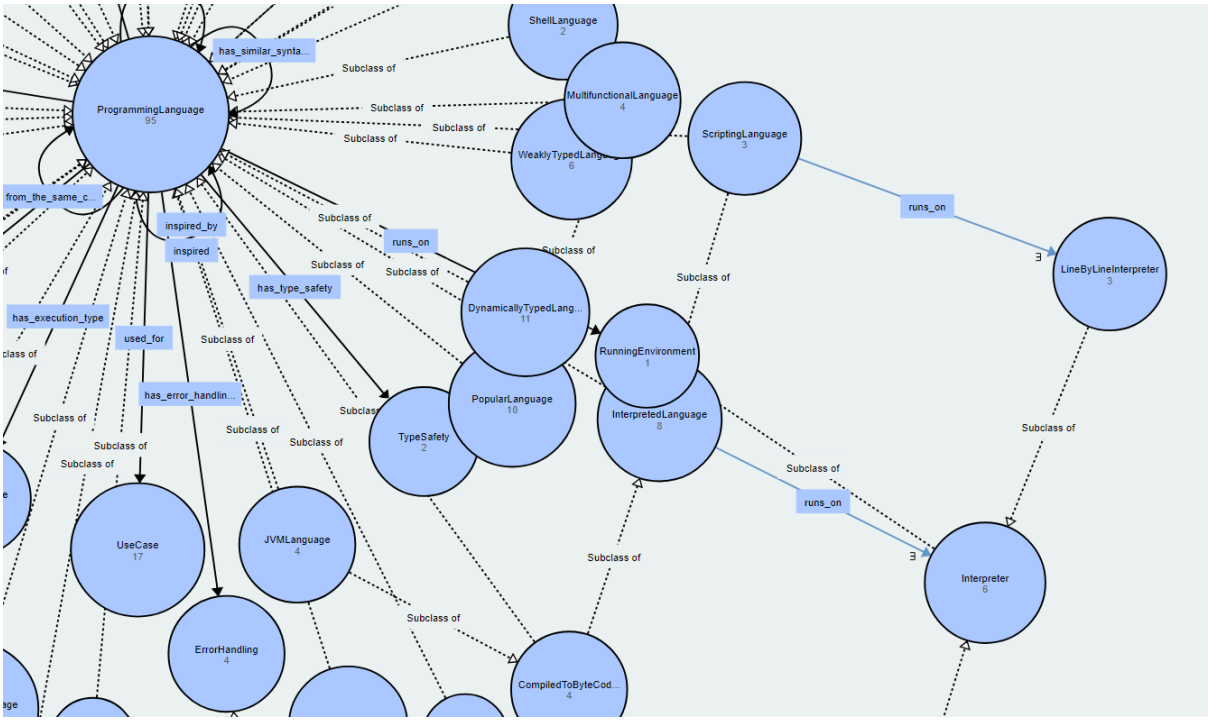
```
onto.get_parents_of(FunctionalLanguage)
```

✓ 0.0s

```
[programming_languages.ProgrammingLanguage,
 programming_languages.DeclarativeLanguage,
 programming_languages.FunctionalInspiredLanguage]
```

Добавени са много нови връзки между класовете:





Фиг 3 и 4. Визуализация с WOWL на част от онтологията преди и след автоматичния извод

На Фиг 3. и 4. може да се проследи, например, че Scripting Language (горе леко вдясно) е станал подклас на Interpreted Language.

Онтологията е проверена успешно за консистентност.

4. Идеи за бъдещо развитие

Възможно разширение на онтологията е освен самите езици като индивиди, да се добавят и различните им версии. Между тях ще има релация на наредба, както и различни стойности по-отношение на релацията inter_op_with, т.к. не всички езици имат обратно съвместимост.

Друго разширение е да се добавят класове, които биха се възползвали от по-сложни изводи. Например да се добави клас Fast Language, който е дефиниран като езиците, които са компилирани и нямат Garbage Collection.

5. Линк към програмна реализация

https://github.com/NikolayDPaev/pl_ontology

6. Референции:

[1] Параграф таксономия от статията в Уикипедия:
https://en.wikipedia.org/wiki/Programming_language#Taxonomies
[2] Сравнение на езиците за програмиране:
https://en.wikipedia.org/wiki/Comparison_of_programming_languages
[3] Списък в Уикипедия на езиците за програмиране по тип:
https://en.wikipedia.org/wiki/List_of_programming_languages_by_type
[4] Сравнение на езиците за програмиране по типова система:
https://en.wikipedia.org/wiki/Comparison_of_programming_languages_by_type_system
[5] Domain Ontology for Programming Languages Mr. Izzeddin A.O. Abuhassan and Akram M.O. AlMashaykhi
https://www.scienpress.com/Upload/JCM/Vol%202_4_4.pdf
[6] Документация на OwlReady2 https://owlready2.readthedocs.io/en/v0.44/index.html