AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ ZARZĄDZANIA

KATEDRA INFORATYKI STOSOWANEJ

Praca dyplomowa licencjacka

Projekt kompilatora języka domenowego dla systemu zarządzania regułami biznesowymi Rebit

Domain specific language compiler project for the Rebit business rules management system

Autor: *Oskar Dawid Furlepa*
Kierunek studiów: Informatyka i ekonometria
Opiekun pracy: *dr inż. Andrzej Macioł*

Kraków, 2020

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

Spis treści

1 Wstęp

1.1 Systemy eksperckie

Systemy eksperckie są programami komputerowymi rozwiązującymi złożone problemy poprzez proces rozumowania w oparciu o posiadaną wiedzę, zdefiniowaną zazwyczaj jako zbiór reguł „jeżeli, to", wypełnianych ręcznie przez ekspertów. Były one jedną z pierwszych udanych form sztucznej inteligencji. Pierwsze z nich tworzone były już w latach siedemdziesiątych XX wieku. Z biegiem czasu systemy te zaczęły tracić popularność. Było to głównie spowodowane trudnością w przystosowywaniu systemu do zmieniającego się otoczenia. Za każdym razem, gdy w dziedzinie działania systemu pojawiała się zmiana, w wiedzy systemu musiały zostać ręcznie wprowadzone odpowiednie poprawki. Utrzymywanie systemu było czasochłonne i kosztowne. Obecnie klasyczne systemy eksperckie praktycznie w całości wyparte zostały przez technologie bardziej generalnego użytku takie jak uczenie maszynowe, algorytmy genetycznie i sieci neuronowe. Wiele z nich wciąż korzysta jednak z narzędzi i technik będących częścią systemów eksperckich.[1]

1.2 Cel pracy

Jednym z systemów eksperckich jest system do zarządzania regułami biznesowymi Rebit. Prace nad nim przypadają na okres 2009 – 2011.[2] Technologie tamtego czasu znacznie różnią się od dzisiejszych, powodując trudności w korzystaniu z systemu. Celem pracy jest zaprojektowanie nowego języka domenowego pozwalającego na tworzenie zestawów reguł, napisanie dla niego kompilatora do języka natywnego Rebita i zamknięcie funkcjonalności Rebita w nowoczesnej aplikacji webowej w postaci web api.

1.3 Streszczenie pracy

Niniejsza praca składa się z 9 rozdziałów. W pierwszym rozdziale przybliżona została historia i idea systemów eksperckich. Pokazane zostały do tego ich podstawowe zastosowania, jak też wady będące głównym powodem spadku ich popularności. Przybliżony został też system zarządzania regułami biznesowymi Rebit. Rozdział drugi przedstawia główne technologie wykorzystane podczas realizacji projektu będącego tematem pracy. Trzeci rozdział stanowi przedstawienie i wybranie metody realizacji kroków budowy kompilatora. W czwartym rozdziale przedstawiona jest implementacja kompilatora w języku C# w technologii .Net Core. Rozdział piąty przedstawia połączenie kompilatora i usługi wnioskującej Rebita w jedną abstrakcję, tak żeby możliwe było przeprowadzenie procesu wnioskowania w oparciu o nowy język. Abstrakcja ta udostępniona zostaje w postaci web api, napisanego w Asp.Net Core. Rozdział szósty obejmuje dokumentację dla web api, utworzoną w wykorzystaniem narzędzia Swagger. Rozdział siódmy zawiera podsumowanie zrealizowanego projektu i możliwości dalszego rozwoju aplikacji. W rozdziale ósmym zamieszczony jest spis ilustracji, w rozdziale dziewiątym spis diagramów, w dziesiątym – bibliografia.

[1] https://www.linkedin.com/pulse/ai-now-expert-systems-george-gesior (dostęp 03.07.2020)
[2] http://rebit.zarz.agh.edu.pl/projekt.html (dostęp 03.07.2020)

2 Wykorzystane technologie

2.1 Asp.Net Core

Asp .Net core to darmowa i otwarta (ang. open-source) platforma programistyczna do tworzenia aplikacji podłączonych do Internetu. Należą do nich aplikacje webowe, ale też web api, aplikacje mobilne i IOT w językach takich jak F#, Visual Basic, czy C#. Jedną z jej cech charakterystycznych jest multiplatformowość. Aplikacja napisana w Asp.Net Core bez zmiany w kodzie źródłowym potrafi działać jednocześnie na systemach operacyjnych Windows, Linux i MacOs. Charakteryzuje się też niewielkimi rozmiarami. Poprzez podzielenie swoich funkcjonalności na pakiety Nuget[3] aplikacja korzysta tylko z potrzebnych jej zależności. Pakiety te są często aktualizowane[4]. Server Kestrel, na którym domyślnie uruchamiana jest aplikacja, jest jednym z najszybszych dostępnych serwerów webowych.[5] Aplikacje napisane w Asp.net core działać mogą na środowisku uruchomieniowym .Net Framework (dla Asp.Net Core przed wersją 3.0) i .Net Core.

Domyślnie aplikacje .net core wykorzystują wzorzec projektowy Model-View-Controller. Wzorzec ten ma za zadanie wspierać luźne zależności w logice biznesowej, co powoduje lepszą testowalność i reużywalność kodu. Znajduje się w warstwie prezentacji aplikacji. Składa się z trzech części:

1. Modelu, zajmującego się danymi aplikacji i logiką z nimi związaną. Logika ta przejawia się przede wszystkim w pobieraniu, aktualizacji, usuwaniu i dodawaniu nowych danych.
2. Widoku, przedstawiającego wyniki operacji przeprowadzonych w części Modelu. W przypadku aplikacji internetowych najczęściej przyjmuje formę plików języków znacznikowych HTML (dla aplikacji webowych) lub JSON (dla web apis).
3. Kontrolerze, który zarządza interakcją między Modelem, a Widokiem, a także zajmuję się interakcją z użytkownikiem aplikacji.[6]

2.2 Swagger

Swagger to darmowe i otwarte (ang. open-source) narzędzie pozwalające w oparciu o kod źródłowy Web API na wygenerowanie dokumentu zawierającego wszelkie informacje użytkowe związane z api. W oparciu o ten dokument możliwe jest między innymi wygenerowanie interaktywnej dokumentacji i utworzenie bibliotek klienta w językach takich jakich jak C# i JavaScript.[7]

---

[3] https://docs.microsoft.com/en-us/nuget/ (dostęp 03.07.2020}
[4] https://app.pluralsight.com/course-player?clipId=39b26eff-d35a-4044-9415-ab818394d985 (dostęp 04.07.2020)
[5] https://www.ageofascent.com/2019/02/04/asp-net-core-saturating-10gbe-at-7-million-requests-per-second/ (dostęp 04.07.2020)
[6] https://app.pluralsight.com/course-player?clipId=ded2a0a5-cf51-492e-8d23-4e985096ef82 (dostęp 03.07.2020)
[7] https://swagger.io/docs/specification/2-0/what-is-swagger/ (dostęp 03.07.2020)

2.3 Docker

Docker to narzędzie pozwalające na zapakowanie aplikacji wraz ze wszystkimi zależnościami potrzebnymi do jej uruchomienia w kontener, który uruchomiony może być na każdym systemie, na którym zainstalowane jest środowisko uruchomieniowe Dockera (ang. Docker runtime). Kontenery porównywane są często do maszyn wirtualnych, istnieją jednak znaczne różnice. Zamiast tworzyć w każdym kontenerze nowy system operacyjny (tak, jak jest to w maszynach wirtualnych), Docker dzieli większość zasobów systemu między kontenerami. Powoduje to, że kontenery są znacznie mniejsze, szybsze i bardziej efektywne. Na komputerze zazwyczaj może działać jednocześnie kilka razy więcej kontenerów niż maszyn wirtualnych. Jako, że dzielone przez kontenery zasoby są dla nich tylko do odczytu, nie jest przy tym poświęcane bezpieczeństwo. Kontenery buduje się w oparciu o definicje określone w specjalnie zaprojektowanym, deklaratywnym języku dziedzinowym.[8] Definicje te formalnie nazywa się obrazami (ang. Docker images).[9]

Popularyzacja Dockera poskutkowała nową metodą projektowania aplikacji webowych. Aplikacja podzielona zostaje pod względem funkcjonalności na mniejsze, niezależne aplikacje, komunikujące się w prywatnej sieci wirtualnej za pomocą protokołu HTTP. Aplikacje napisane w ten sposób charakteryzują się lepszą:

1. Testowalnością, jako że każda z aplikacji może być testowana oddzielnie.
2. Odpornością na błędy, jako że w razie problemu z działaniem kontenera, możliwa jest podmiana go na nowy bez zamykania całej aplikacji. Narzędzia do tworzenia sieci kontenerów potrafią monitorować ich stan i automatycznie podjąć odpowiednią akcję w przypadku problemów w działaniu jednego z nich.
3. Skalowalnością poprzez dodanie nowych komputerów. Jako, że kontenery komunikują się poprzez sieć, możliwe jest utworzenie ich na wielu różnych komputerach. Istnieją serwisy, takie jak portal.azure.com, pozwalające na postawienie kontenerów w chmurze i w zależności od, na przykład, obciążenia aplikacji, tworzyć lub usuwać nowe kontenery, tak aby zużywane zasoby zawsze były odpowiednie względem zapotrzebowanie aplikacji.[10]

[8] https://pl.wikipedia.org/wiki/J%C4%99zyk_dziedzinowy (dostęp 04.07.2020)
[9] https://medium.com/swlh/what-exactly-is-docker-1dd62e1fde38 (dostęp 03.07.2020)
[10] https://smartbear.com/solutions/microservices/ (dostęp 05.07.2020)

3 Budowa kompilatora

3.1 Wprowadzenie

Kompilator to program pobierający program napisany w języku źródłowym na taki sam program w innym języku nazywanym językiem docelowym. Jeśli w trakcie kompilacji dojdzie do błędu, ważne jest, żeby kompilator o nim poinformował. Kompilacja, czyli proces wykonywany przez kompilator, może zostać podzielona na podprocesy. Ich ilość i rodzaje różnią się w zależności od zastosowania. [11] Dla kompilatora przygotowanego w związku z pracą dyplomową są to:

1. Analiza leksykalna.

   Tekst źródłowy otrzymany w postaci strumienia znaków dzielony jest na najmniejsze posiadające znaczenie łańcuchy znaków nazywane leksemami. Dla każdego leksemu, Lexer – abstrakcja zajmująca się analizą leksykalną, tworzy token składający się z nazwy, nazywanej również typem, identyfikującej token, wartości – leksemu, któremu token odpowiada i metadanych – informacji nie koniecznych do prawidłowego przeprowadzenia kompilacji, takich jak pozycja leksemu w tekście źródłowym, jego długość itd. Wynikiem analizy leksykalnej jest strumień tokenów.[12]

2. Analiza syntaktyczna.

   Z tokenów, będących wynikiem analizy leksykalnej, zostaje utworzona struktura drzewa reprezentująca gramatyczną strukturę tekstu źródłowego. Strukturę tę nazywa się drzewem parsowania (ang. parse tree). Drzewo parsowania, poprzez usunięcie tokenów niepotrzebnych do dalszej kompilacji (takich jak nawiasy, przecinki itd.), zostaje przekształcone w abstrakcyjne drzewo syntaktyczne (ang. abstract syntax tree), nazywane też drzewem syntaktycznym (ang. syntax tree). Drzewo to zawierające tylko elementy niezbędne do utrzymania gramatycznego sensu parsowanego tekstu. Każdy węzeł drzewa reprezentuje operację, której parametrami są jego dzieci.[13]

3. Analiza semantyczna.

   W oparciu o abstract syntax tree tekst źródłowy walidowany jest pod względem znaczenia. Proces ten dzieli się na sprawdzanie poprawności syntaktycznej, nieobjętej przez zdefiniowaną gramatykę i analizę kompatybilności typów. W przypadku błędu kompilacja powinna zostać zatrzymana i powinna zostać zwrócona odpowiednia informacja. Typy zostają przekazane dalej, jako część syntax tree.[14]

4. Generowanie kodu XML na podstawie drzewa syntaktycznego.

[11] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s. 4 - 5
[12] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 5 – 6
[13] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 8
[14] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 8 - 9

Na podstawie abstract syntax tree budowany i zwracany jest zestaw reguł w postaci kodu XML, który wykorzystać można w usłudze wnioskowania oferowanej przez system regułowy Rebit.

Każdy z podprocesów posiada dostęp do struktury danych, nazywanej tablicą symboli (ang. symbol table), z której pobierać mogą potrzebne w danej chwili informacje lub zamieszczać informacje potrzebne w kolejnych krokach parsowania.[15] W tym przypadku są to nazwy zdefiniowanych zmiennych, typów wyliczeniowych, przekazywane podczas analizy syntaktycznej.

3.2 Definicja języka

Przed rozpoczęciem budowy kompilatora należy zdefiniować język, który zostanie poddany procesowi kompilacji. Język przygotowany do zaspokajania szczególnej potrzeby, rozwiązywania określonej dziedziny problemów nazywany jest formalnie językiem dziedzinowym lub domenowym.[16] Celem języka jest umożliwienie zdefiniowania zbioru reguł przyjmowanego przez system ekspercki Rebit. Rebit posiada do tego natywny język w postaci struktury danych XML. Język domenowy powinien umożliwiać wszystko, co język natywny. W tym celu w pseudokodzie przygotowany został przykładowy zbiór reguł, z myślą o przetwarzaniu którego będzie pisany kompilator.

[15] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 11
[16] https://www.youtube.com/watch?v=eF9qWbuQLuw (dostęp 04.07.2020)

```
Enum Days
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
Enum WhichDay
{
    Workday,
    Weekend
}

Single Days day;
Single real time;
Multiple Ordered int orderedValues;
Multiple Unordered int orderedValues;
Single bool isTimeToWork;
Single string isTimeToSleep;

if (day in [Days.Monday, Days.Tuesday, Days.Wednesday, Days.Thursday, Days.Friday])
    today = WhichDay.Workday;
if (today == WhichDay.Workday && time > 9 && time < 17)
    isTimeToWork = true;
if (today != WhichDay.Workday)
    isTimeToWork = true;
if (time < 9)
    isTimeToWork = false;
if (time > 17)
    isTimeToWork = false;
if (isTimeToWork == true)
    isTimeToSleep = "yes";
if (isTimeToWork == false)
    isTimeToSleep = "no";
if(LogN((1 + 2) * 4)/5 * Sum({1, 2, 3}) == 5)
    isTimeToWork = true;
```

Rysunek 1 Przykład zestawu reguł napisanego w nowym języku domenowym Rebita

W języku możliwe są 3 główne rodzaje komunikatów (Statements):

1. Zdefiniowanie typu wyliczeniowego (Enum).

Definicja takiego typu składa się ze słowa kluczowego Enum, nazwy typu i wartości, jakie zmienne za pomocą niego stworzone mogą przyjmować.

2. Definicji zmiennej.

Zdefiniowanie zmiennej zaczyna się od podania jej typu. Na typ składają się 3 cechy: krotność, uporządkowanie i typ bazowy. Ze względu na krotność wydziela się:

- zmienne jednokrotne (Single)
- zmienne wielokrotne (Multiple)

Podczas definiowania zmiennej wielokrotnej należy później podać uporządkowanie zmiennej. Cecha ta przyjmuje 2 wartości:

- nieuporządkowana (Unordered),
- uporządkowana (Unordered),

Kolejny do podania jest typ bazowy. Może być to: string, bool, int, real, lub typ wyliczeniowy zdefiniowany wcześniej. Typy bazowe odpowiadają typom o identycznej nazwie w języku C#. Typ „real" odpowiada typowi „double". Zmienne nieuporządkowane odpowiadają tablicom, uporządkowane – listom. Na końcu podawana jest nazwa zmiennej, do której można odnosić się w dalszej części kodu.

3. Komunikatu logicznego (If Statement)

Komunikat logiczny zawsze zaczyna się od słowa kluczowego „if" i lewego nawiasu okrągłego. W nawiasie znajduje się jedna lub więcej przesłanek, które łączyć ze sobą można poprzez operator logiczny i (&&).

Każda z przesłanek musi składać się z lewej strony, prawej strony i operatora porównania. Każda ze stron jest dowolną kombinacją referencji do zmiennych, stałych i ewaluacji funkcji, które łączyć ze sobą można operatorami matematycznymi i operatorami porównania. Operatory matematyczne muszą podlegać zasadom pierwszeństwa, przy czym pierwszeństwo można wymuszać za pomocą nawiasów okrągłych. Zarówno lewa jak i prawa strona muszą przyjmować formę wyrażeń. Typy wyrażeń muszą być ze sobą zgodne.

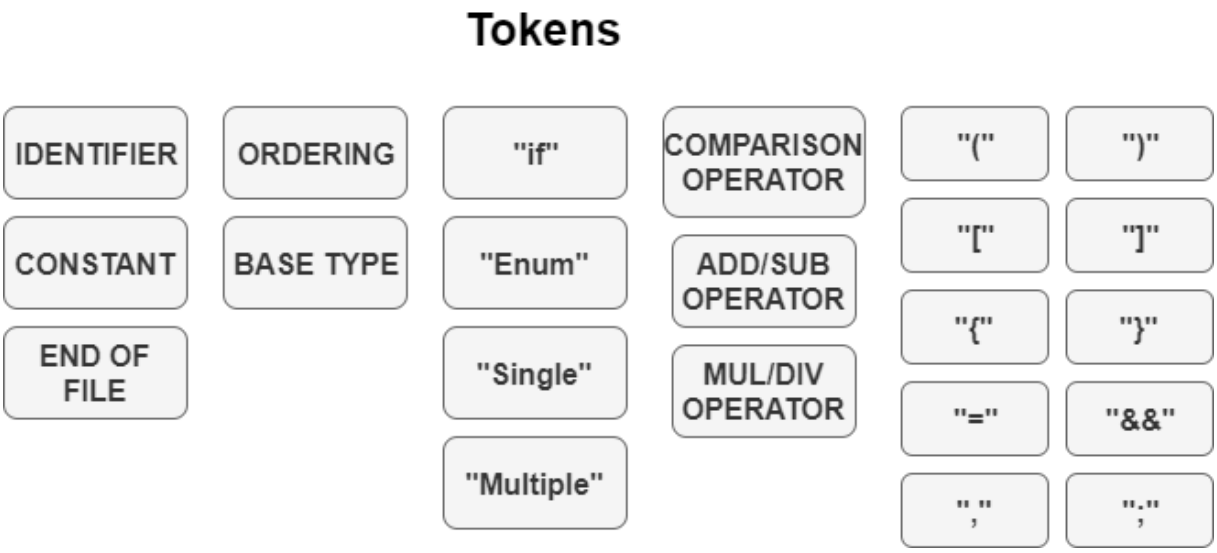
Referencję do zmiennej przedstawia nazwa zmiennej. Stałe jednokrotne przyjmują formę literałów z języka C#. Stałe wielokrotne zapisuje się poprzez podanie symbolu „[„ lub „{" w zależności od uporządkowania zmiennej, po podaniu wartości stałych jednokrotnych oddzielonych przecinkami i podanie na końcu odpowiednio symbolu „]" lub „}". Jako, że silnik Rebita nie wspiera zagnieżdżania dla stałych wielokrotnych (np. utworzenia listy list liczb całkowitych), nie będzie to również wspierane przez kompilator.

W większości języków programowania każdy element języka zwracający pewną wartość nosi nazwę wyrażenia (ang. Expression). Za to procedura, wykonana dla określonego efektu (na przykład zmiany stanu zmiennej), nie zwracająca żadnej żadnej wartości, nazywana jest komunikatem (ang. Statement). Wyrażenia mogą być w sobie zagnieżdżane, dozwolona jest więc między nimi rekurencja.

Po zakończeniu przesłanki należy podać konkluzję, która składa się z nazwy zmiennej, operatora przypisania („=") i wyrażenia zwracającego wartość zgodną ze zmienną o wcześniej podanej nawie.

3.3 Analiza leksykalna

Analiza leksykalna języka rozpoczyna się od zdefiniowania wszystkich atomów, formalnie nazywanych tokenami. Są to elementarne składowe, z których zbudowany będzie każdy zbiór reguł napisany w tym języku.[17] Poniżej przedstawione zostały wszystkie zdefiniowane tokeny.



Rysunek 2 Wszystkie typy tokenów języka domenowego

Za każdym razem, kiedy w strumieniu znaków pojawi się podzbiór (leksem) spełniający warunki do zostania jednym z tokenów, jest on wyodrębniany i zwracany jako nowy token. Tokeny o większej ilości znaków mają pierwszeństwo. Przykładowo, jeśli natrafiony zostanie znak „=", nie zostanie on od razu zwrócony jako nowy token „=". Pobrany zostanie jeszcze jeden znak. Jeśli okaże się, że on również jest znakiem „=", zostanie zbudowany i zwrócony token „==". W przeciwnym wypadku zwrócony zostanie token „=", a drugi znak zostanie wykorzystany do budowy nowego tokena.

Na piątą i szóstą kolumnę składają się jedno lub dwu-znakowe operatory przeznaczone do różnych zastosowań. Cudzysłów w nazwie tokena oznacza ciąg znaków jakiemu on odpowiada. Na czwartą kolumnę składają się agregaty operatorów. Odpowiadające tokenom leksemy wyglądają następująco:

1. COMPARISON OPERATOR: „>", „<", „>=", „<=", „==", „!=",", „in", „notin"
2. ADD/SUB OPERATOR: „+", „-"
3. MUL/DIV OPERATOR: „*", „/"

Trzecia kolumna składa się ze słów kluczowych, druga z agregatów słów kluczowych. Odpowiadające tokenom leksemy wyglądają następująco:

1. ORDERING: „Ordered", „Unordered"
2. BASE TYPE: „string", „int", „bool", „real"

W pierwszej kolumnie znajdują się tokeny specjalne, przeznaczone dla zastosowań bardziej złożonych. Pierwszym tokenem jest „IDENTIFIER". Przedstawia on nazwę typu

[17] https://www.youtube.com/watch?v=eF9qWbuQLuw (dostęp 04.07.2020)

wyliczeniowego, zmiennej lub funkcji. Kolejny token, „Constant” przedstawia literały dla dostępnych przez parser typów podstawowych. Mogą być to:

- literał ciągu znaków, zaczynający się i kończący na znaku: „””
- literał liczby całkowitej, przedstawiający liczbę całkowitą
- literał liczby rzeczywistej, składający się z liczby całkowitej, symbolu „.” i liczby całkowitej
- literał typu wyliczeniowego, składający się na ciąg znaków, symbol „.” i ciąg znaków.

Za każdym razem, kiedy w tekście źródłowym pojawi się łańcuch znaków odpowiadający tokenom z trzech ostatnich kolumn (nie licząc łańcuchów „in” i „notin”), z tekstu źródłowego wydzielony zostanie dany łańcuch i utworzony zostanie token. Ciągi znaków związane z tymi tokenami pełnią dodatkowo funkcję separatorów. Separatorem nazywa się ciąg znaków lub znak dzielący tekst w kontekście analizy leksykalnej. Separatorami dodatkowo są znak końca linii i whitespace. Dla tokenów z kolumn drugiej i trzeciej (, i ciągów znaków „in”, „notin”) oprócz pojawienia się określonego ciągu znaków potrzebne jest również, żeby występował po nim separator. Przykładowo ciąg znaków „OrderedSingle” nie spowoduje powstania ani tokena ORDERING („Ordered”), ani „Multiple. Za to „Ordered Single”, dlatego że między literałami „Ordered” i „Single” jest separator, spowoduje utworzenie obydwu tych tokenów.

Ostatni token oznacza koniec tekstu źródłowego.

3.4 Specyfikacja gramatyki

Gramatyką nazywamy zestaw reguł nazywany regułami produkcyjnymi (ang. production Rules). Każda z reguł składa się z lewej strony, określającej co jest definiowane, i prawej, określającej jak wygląda definicja. Reguły produkcyjne nazywa się również regułami zamiennymi (ang. replacement Rules), jako że każde odniesienie do zdefiniowanego non-terminala możemy zamienić z jego definicją, bez zmiany w znaczeniu gramatyki.

Podstawowymi elementami budującymi reguły są Terminale i Non-Terminale. Terminale to inaczej atomy, tokeny. Mają one sens same w sobie i charakteryzują się brakiem możliwości podzielenia ich na części składowe. Każdy token jest Terminalem. Non-terminale to elementy składające się z non-terminali i terminali. Definicja non-terminali i określenie zależności między terminalami i non-terminalami są podstawowym zadaniem gramatyki. [18]

[18] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 42 – 46

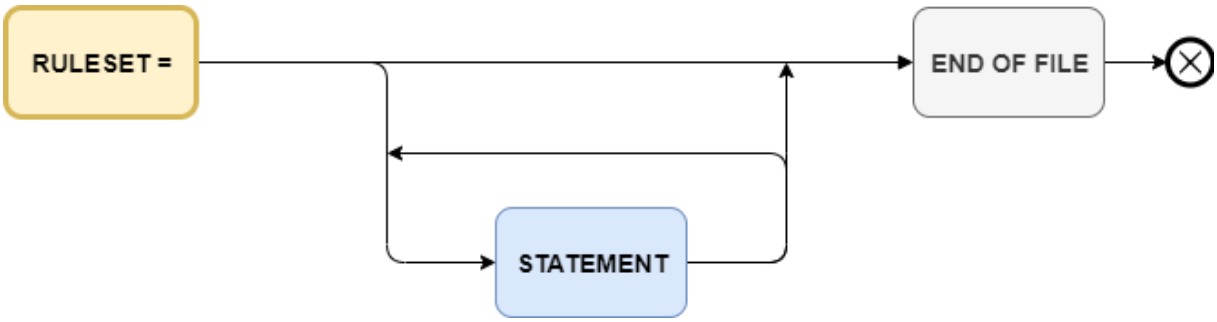


Diagram 1 Gramatyka non-terminala RULESET

Powyżej przedstawiony jest diagram jednej z reguł. Żółty element po lewej stronie oznacza definicję non-terminala, niebieski – odniesienie się do non-terminala, szary – odniesienie się do terminala. Reguła przedstawiona tutaj określa najbardziej ogólny element języka nazywany zestawem reguł. Wskazuje ona, że RULESET może składać się z dowolnej ilości non-terminali STATEMENT, po których pojawić musi się terminal END OF FILE.

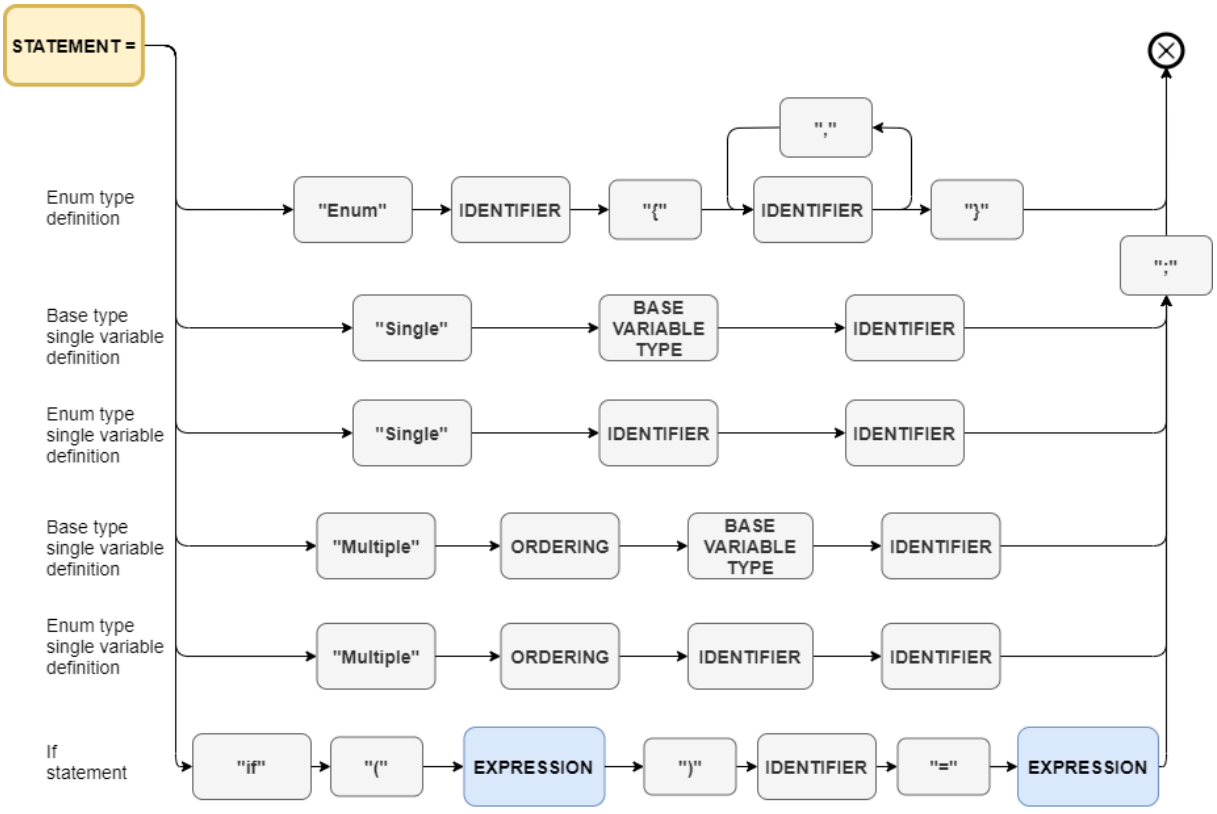


Diagram 2 Gramatyka non-terminala STATEMENT

Następnym non-terminalem wymagającym definicji jest STATEMENT. Jego definicja składa się z 6 reguł. Określają one kolejno definicję typu wyliczeniowego Enum, 2 definicje zmiennej pojedynczej, 2 definicje zmiennej wielokrotnej, komunikat warunkowy. Spełnienie dowolnej z nich oznacza, że ciąg znaków tworzy non-terminal STATEMENT.

Reguły te można zinterpretować:

1. Żeby z ciągu leksemów możliwe było zbudować definicję typu wyliczeniowego, jako pierwszy token powinien pojawić się „Enum”, po nim IDENTIFIER, określający nazwę typu, po której będzie się można odnosić w dalszej części kodu, „{”, 1 lub więcej tokenów IDENTIFIER, oddzielonych od siebie przecinkami, oznaczających wartości dopuszczalne przez Typ i na końcu „}”.
2. W przypadku definicji zmiennej pojedynczej typu prostego pierwszym tokenem musi być token „Single”, po którym musi nastąpić BASE VARIABLE TYPE, IDENTIFIER, określający nazwę zmiennej i „;”. Dla kolejnego non-terminala, będącego podmiotem reguły trzeciej, a więc definicji zmiennej pojedynczej, zamiast BASE VARIABLE TYPE spodziewany jest token IDENTIFIER, określający typ wyliczeniowy, który definiowany jest w trakcie działania programu.
3. (W przypadku definicji zmiennej wielokrotnej) Pierwszym wczytanym tokenem musi być token „Multiple”. Po nim muszą nastąpić kolejno ORDERING, BASE VARIABLE TYPE lub IDENTIFIER, IDENTIFIER, określający nazwę zmiennej i średnik.
4. (W przypadku komunikatu warunkowego) Fraza musi zacząć się od tokena „if”, po którym nastąpić muszą „(”, 1 lub więcej tokenów EXPRESSION dzielonych „&&”, „)”, IDENTIFIER określający nazwę zmiennej, do której zostanie przypisana wartość, operator przypisania, Expression zwracające wartość, która zostanie przypisana.

Utworzona gramatyka określa jedynie zależności między położeniem tokenów, nie nadając tokenom żadnego znaczenia. Przykładowo, informacja z reguły pierwszej, mówiąca, że token IDENTIFIER na drugim miejscu oznaczać ma nazwę typu wyliczenionwego nie jest określona przez gramatykę, a zostanie dopiero w kroku analizy semantycznej w dalszej części pracy. Wartym zauważenia jest, że do określenia którą z reguł należy aktualnie wykorzystać, wystarczy znajomość 1 tokena w strumieniu wejściowym, którego porównać należy do pierwszych tokenów każdej z reguł. Reguła, dla której tokeny będą takie same jest odpowiednią regułą. Ostatnim non-terminalem do zdefiniowania jest EXPRESSION.

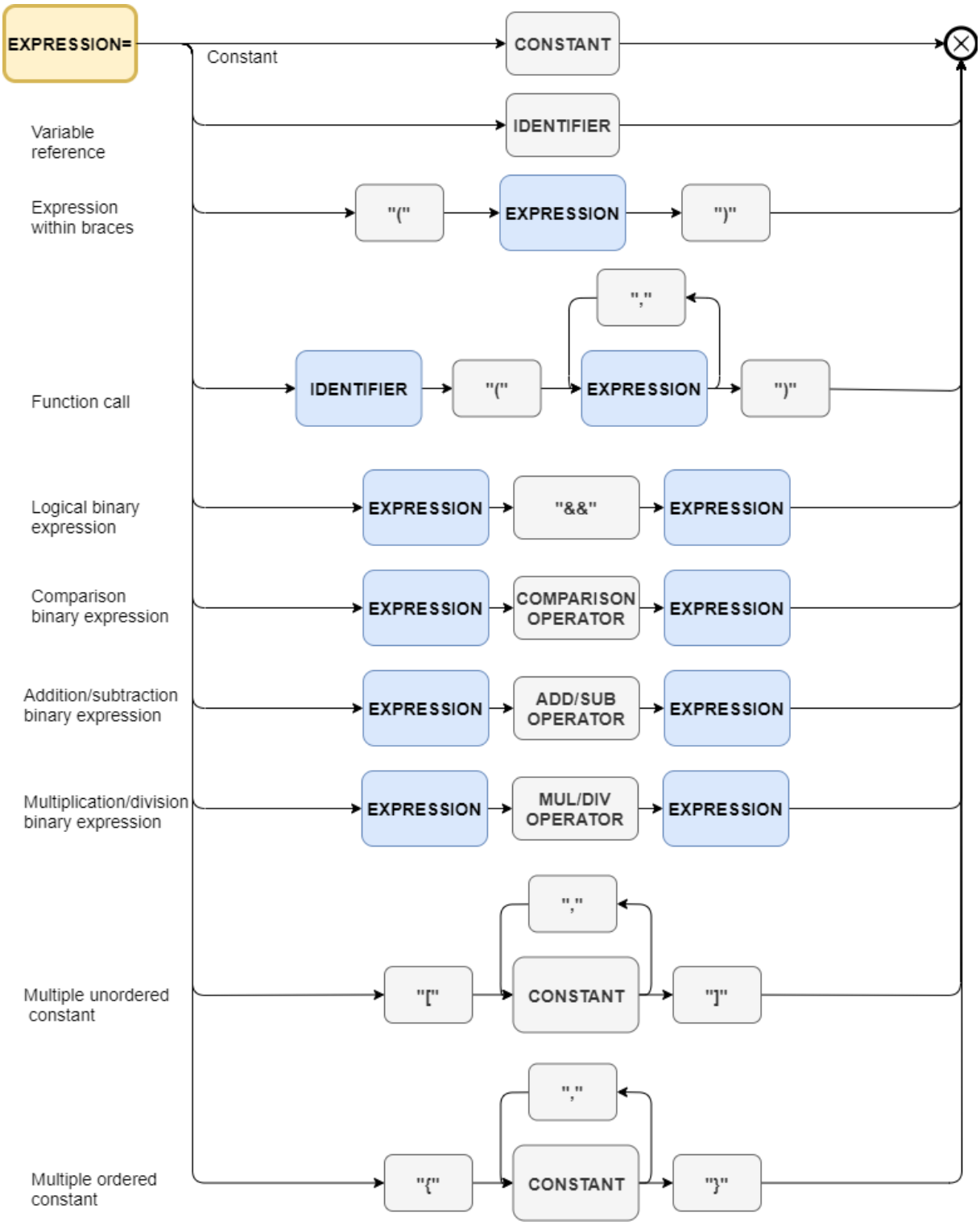


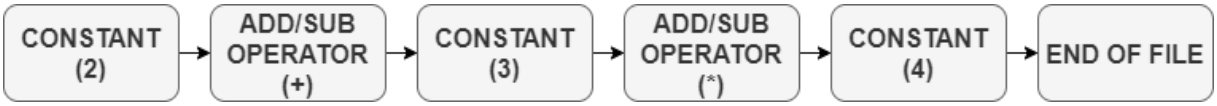
Diagram 3 Gramatyka non-terminala EXPRESSION

Kolejne reguły przedstawiają:

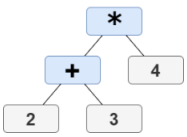
1. Stałą pojedynczą, np. 5, „aaa”, 27.5, Days.Monday
2. Referencję do zmiennej, np. month, variable1
3. Wyrażenie w nawiasie, np. („aaa”), (5 + 3.2), (8 * (2 + 3))
4. Ewaluację funkcji, np. ToString(„50”), Add(100, 10.5)
5. Wyrażenie binarne z operatorem logicznym, np. variable1 == 5 && false, 1 != 1 && true
6. Wyrażenie binarne z operatorem porównania, np. 5 > 7, 1 == 4 – 3, Add(2,3) <= 5
7. Wyrażenie binarne z operatorem dodawania lub odejmowania, np. 1 + 2, 2 + 4 * 3, 8 - ToString (5)
8. Wyrażenie binarne z operatorem mnożenia lub dzielenia np. 7 * 3, Add(2,3)/ 2
9. Stałą wielokrotną nieuporządkowaną, np. [1, 2, 3], [„a”, „b”], [Days.Monday]
10. Stałą wielokrotną uporządkowaną, np. {1, 2, 3}, {„a”, „b”}, {Days.Monday}

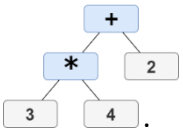
3.5 Wieloznaczność gramatyki

Przykładem frazy (ciągu znaków) obsługiwanej przez gramatykę Expression jest: 2 + 3 * 4. Wynikiem analizy leksykalnej tej frazy będzie strumień tokenów:



*Rysunek 3 Strumień tokenów odpowiadający wyrażeniu "2 + 3 * 4"*

Poprawnym gramatycznie abstract syntax tree uzyskanym w wyniku jej parsowania może być jednocześnie 

i 

.

Gramatykę, w której na podstawie jednego ciągu tokenów możliwe jest utworzenie więcej niż jednego drzewa syntaktycznego nazywa się gramatyką dwuznaczną (lub ambiwalentną) (ang. ambiguous grammar).[19] Dwuznaczna gramatyka nie zawsze jest problemem. W tym przypadku jednak jest. Język dziedzinowy posiadający różne typy operatorów powinien uwzględniać pierwszeństwo między nimi. Operatory w porządku od tego o największym do tego o najmniejszym pierwszeństwie powinny wyglądać następująco.

1. Operatory mnożenia/dzielenia (token MUL/DIV OPERATOR).
2. Operatory dodawania/odejmowania (token ADD/SUB OPERATOR).
3. Operatory porównania (token COMPARISON OPERATOR).
4. Operatory logiczne (token “&&”).

Jedną z technik pozbywania się ambiwalencji, a w konsekwencji wprowadzania pierwszeństwa jest nakładanie warstw (Layering)[20]. Przejawia się ona w krokach:

1. Z non-terminala wydzielone zostają wszystkie reguły powodujące dwuznaczność.

[19] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 203 – 204
[20] https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/02/Slides02.pdf (dostęp 03.07.2020)

2. Reguły te grupowane są pod względem pierwszeństwa.
3. Z grup zostają utworzone
4. non-terminale, tak żeby non-terminal o niższym pierwszeństwie był budowany z non-terminala o wyższym, a jednocześnie możliwie najniższym pierwszeństwie. [21]
5. Non-terminal o najniższym pierwszeństwie zbudowany zostaje z non-terminala, z którego zostały wydzielone reguły w kroku 1.

W przypadku zdefiniowanej gramatyki problem wieloznaczności występuje w non-terminalu EXPRESSION. Jednocześnie znajdują się w nim reguły tworzenia wyrażeń porównywania, dodawania i odejmowania . Wszystkie te reguły zostają więc wydzielone. Dla każdego stopnia pierwszeństwa istnieje po 1 regule, nie ma więc potrzeby grupowania reguł. Utworzone zostają nowe non-terminale. Wyrażenie logiczne nazwane wyrażeniem(nowe EXPRESSION) posiada w sobie jedno lub więcej wyrażeń porównania (COMPARISON EXPRESSION), które mają w sobie jedno lub więcej wyrażeń dodawania/odejmowania(ADD/SUB EXPRESSION), które z kolei posiadają w sobie wyrażenia mnożenia i dzielenia(MUL/DIV EXPRESSION). Zgodnie z zasadą czwartą wydzielone wyrażenie o najniższym pierwszeństwie. Wyrażenie mnożenia i dzielenia(MUL/DIV EXPRESSION) zbudowane więc zostaje z pozostałości EXPRESSION i nazwane FACTOR. Po wprowadzeniu layeringu gramatyka EXPRESSION wygląda następująco:

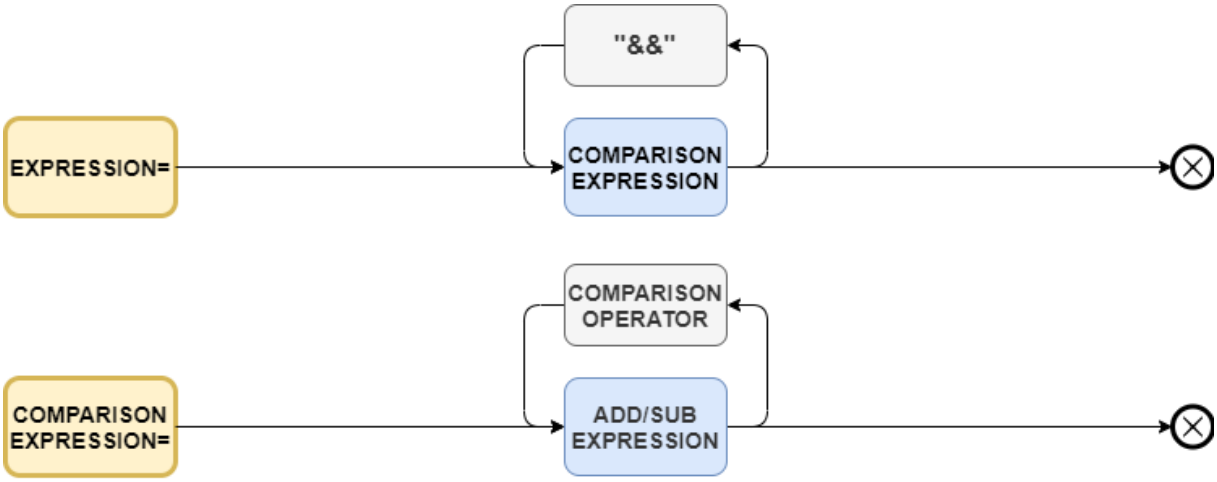


Diagram 4 Gramatyka non-terminala EXPRESSION po zastosowaniu techniki layeringu w celu usunięcia wieloznaczności (1)

[21] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 210 - 212

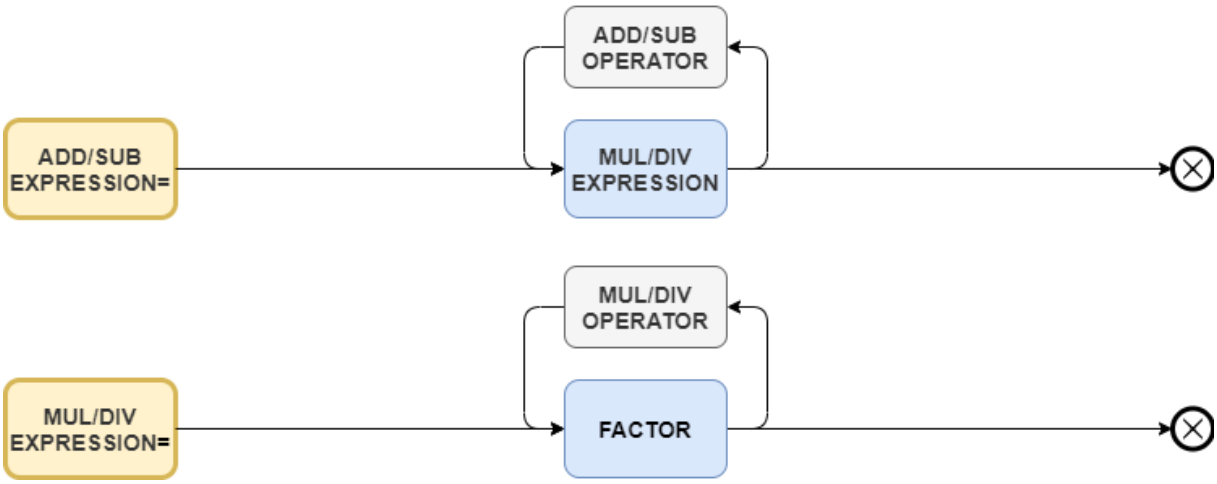


Diagram 5 Gramatyka non-terminala EXPRESSION po zastosowaniu techniki layeringu w celu usunięcia wieloznaczności (2)

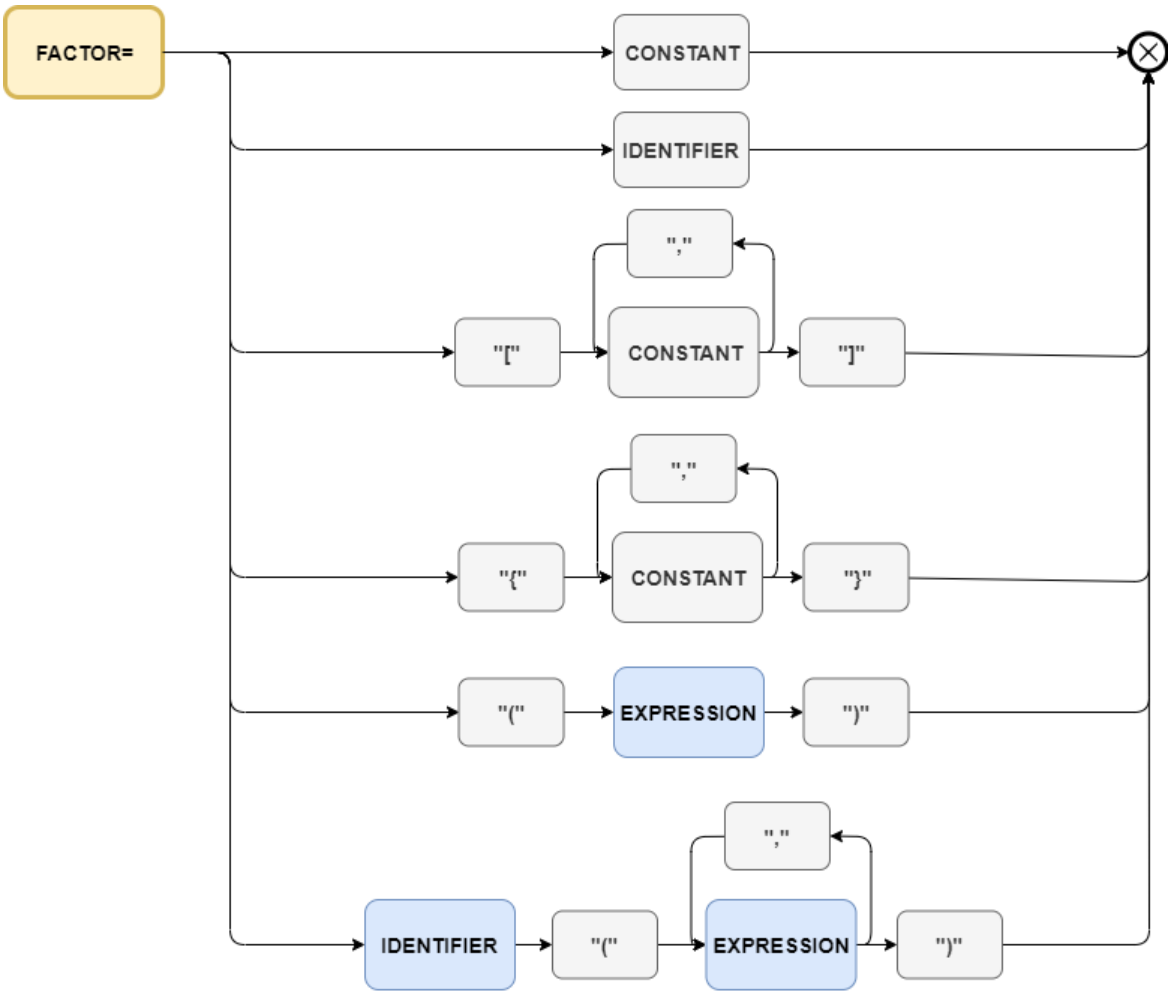


Diagram 6 Gramatyka non-terminala FACTOR po zastosowaniu techniki layeringu

3.6 Wybór odpowiedniej reguły gramatyki dla non-terminali złożonych

Istnieje wiele reguł w oparciu których możliwe jest tworzenie non-terminali STATEMENT i FACTOR. Potrzebny jest sposób na zdecydowanie jakiej reguły użyć w danej chwili. Jednym ze sposobów jest podejrzenie jednego lub więcej najbliższych tokenów i porównanie ich do kolejnych potrzebnych tokenów w każdej z reguł. Ostatnia reguła w której wszystkie do tej

pory tokeny wykazywały zgodność jest regułą w oparciu której należy utworzyć non-terminal. Jeśli w dowolnym momencie żadna z reguł lub do końca tekstu źródłowego więcej niż jedna reguła zostanie prawidłowa, w tekście źródłowym występuje błąd syntaktyczny. Do wybrania odpowiedniej reguły dla non-terminala STATEMENT potrzebne są tokeny:

1. „Enum”
2. „Single”
3. „Multiple”
4. „if”.

Do wybrania odpowiedniej reguły non-terminala FACTOR potrzebne są tokeny:

1. CONSTANT
2. IDENTIFIER
3. „[”
4. „{”
5. „(”
6. IDENTIFIER, „(”,

W przypadku reguły szóstej dla non-terminala FACTOR potrzebne są 2 tokeny. We wszystkich innych przypadkach – 1. Grupę tokenów, której wartość można sprawdzić w trakcie parsowania nazywa się lookup bufferem. Maksymalna ilość tokenów jaką można podejrzeć naraz nazywana jest rozmiarem lookup buffera. W tym przypadku wynosi ona 2 tokeny. Parsery korzystające z lookup buffera nazywa się parserami przewidującymi (ang. Predictive parsers).

Utworzona gramatyka posiada cechy charakterystyczne:

1. Parsowanie tekstu źródłowego dokonuje się od lewej do prawej.
2. Lewa strona każdej z reguł składa się z pojedynczego non-terminala. Gramatyka posiadająca tą cechę nazywana jest gramatyką bez kontekstu (context-free grammar).
3. Do określenia jaka reguła wykonana ma zostać w danej chwili potrzeba maksymalnie 2 tokenów.

Gramatyki posiadające podane cechy nazywa się gramatykami LL(2). Pierwsze L wynika z pierwszej reguły, drugie L – z drugiej, 2 – z trzeciej.[22]

3.7 Algorytm parsowania

Jedną z metod parsowania wykorzystującą zdefiniowaną gramatykę jest recursive-descent top-down parsing. Parser wykonany tym sposobem składa się ze zbioru procedur dla każdego non-terminala. Jeśli dana definicja non-terminala składa się z więcej niż jednej reguły, każda z podreguł przeniesiona zostaje do oddzielnej procedury. Wszystkie procedury podreguł agreguje jedna nadrzędna, która w oparciu o tokeny z symbol table, określa którą z podreguł wykorzystać w danej chwili. Proces parsowania zaczyna się od ewaluacji procedury związanej z non-terminalem startowym. Kończy się on sukcesem, jeśli w trakcie jego działania

[22] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 222 – 226

parsowaniu podlegnie cały strumień wejściowy, to jest napotkany zostanie token END OF FILE.[23]

Poniżej pokazana jest napisana w pseudokodzie procedura dla definicji non-terminala o nazwie nonTerminal, składającego się z 2 zasad produkujących nonTerminal1 i nonTerminal2.

```
nonTerminal ProcedureForNonTerminal()
{
    lookupTokens = GetLookupTokens();
    if(lookup tokens match rule for nonTerminal1)
        return ProcedureForNonTerminal1();
    else if(lookup tokens match rule for nonTerminal2)
        return ProcedureForNonTerminal2();
    else return syntax error
}
```

Rysunek 4 Pseudokod odpowiadający procedurze reguły non-terminala „nonTerminal”

3.8 Analiza semantyczna

Analiza semantyczna polega na sprawdzeniu sensowności kodu pod względem znaczenia. Pozwala ona na weryfikację czy dany kod źródłowy może być skompilowany, jak też wyłapanie znacznej części błędów programistycznych zanim program zostanie uruchomiony. Analiza semantyczna dzieli się na:

1. Analizę syntaktyczną (ang. syntactic checking).

   Jest to walidacja kodu pod względem poprawności syntaktycznej wychodząca poza definicję gramatyki. Na przykład: sprawdzenie, czy dana zmienna lub typ został zdefiniowany co najwyżej raz.

2. Sprawdzanie typów (ang. type checking).

   Określenie, czy typy wyrażeń są zgodne ze sobą w kontekście języka. Niezależne od gramatyki. Przykładowo, wyrażenie 5 == ”10”, mimo poprawności gramatycznej, powinno zostać uznane za błędne, jako że (w kontekście zdefiniowanego języka) nie jest dozwolone porównanie liczby całkowitej z literałem typu string. Obejmuje to właściwości języka takie jak:

   1. Koercja (ang. coercion) – konwersja wartości o danym typie na jej ekwiwalent w innym typie, jeśli wpłynie to na poprawność semantyczną. Przykładowo, wyrażenie 5.0 == 5 po lewej stronie posiada opeand typu real (5.0), po prawej – int(5). Dozwolone jest tylko porównywanie wartości o tych samych typach. Typ int, może zostać jednak skonwertowany na typ real. 5 zostaje zamienione na 5.0. Wyrażenie staje się poprawne semantycznie.
   2. Przeciążanie (ang. overloading) – legalizacja wielu różnych zestawów typów przez dany operator lub funkcję. Przykładowo legalnym wyrażeniem może być jednocześnie „text” == „text”, jak też 5 == 5. Operator ”==” jest w takim wypadku

[23] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 219

przeciążonym operatorem, bo może być wykorzystany z opeandami typu „string”, jak też „int”.[24]

3.9 Serializacja drzewa syntaktycznego do natywnego języka Rebita.

Ostatnim krokiem kompilacji jest zamiana drzewa syntaktycznego, będącego wynikiem procesu parsowania na język XML, którym posługuje się silnik wnioskujący Rebit. Każdemu non-terminalowi i terminalowi odpowiadać musi jeden lub więcej bloków kodu. Poniżej przedstawione zostały bloki kodu odpowiadające składowym drzewa syntaktycznego.

```
<!-- EnumTypeDefinition -->
<Type>
    <EnumType "Id = [[Name]]">
      [[AllowedValues]]
    </EnumType>
</Type>;

<!-- VariableDefinition -->
<Variable Multiplicity = "[[Multiplicity]]" Id = "[[definition.Name]]">
    <[[BaseXmlType]] IdRef = "[[XmlType]]"/>
</Variable>

<!-- IfStatement -->
<Rule Id = "[[RuleId]]">
    <If>
        [[Condition]]
    </If>
    <Then>
        [[Body]]
    </Then>
</Rule>

<!-- BinaryExpression -->
<Atom Operator = "[[Operator]]">
    <LeftHand >
        [[Left]]
    </LeftHand >
    <RightHand >
        [[Right]]
    </RightHand >
</Atom>
```

Rysunek 5 Przedstawienie kodu XML odpowiadające wszystkim elementom gramatyki (1)

[24] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, s 8 – 9

```
<!-- BinaryExpression -->
<Term Content = "Function">
    <Function IdRef ="[[FunctionName]]">
        [[Left]]
        [[Right]]
    </Function >
</Term >

<!-- FunctionCall -->
<Term Content = "Function"> +
    <Function IdRef ="[[Name]]">
        [[Inputs]]
    </Function >
</Term >

<!-- CollectionOfConstants -->
<Term Content = "Constant">
    <Constant Multiplicity ="Multiple[[Ordering]]">
        [[Values]]
    </Constant>
</Term>

<!-- Constant -->
<Term Content = "Constant">
        <Constant Multiplicity = "Single">
            <[[TypeName]]>[[Value]]</[[TypeName]]>
    </Constant>
</Term>

<!-- VariableReference -->
<Term Content = "Variable">
    <Variable IdRef = "[[Name]]"/>
</Term >
```

Rysunek 6 Przedstawienie kodu XML odpowiadające wszystkim elementom gramatyki (2)

Nazwy elementów drzewa, którym dany blok kodów odpowiada znajdują się w komentarzu nad blokiem kodu. Symbolem „[[]]” oznaczone zostały parametry, których wartości różnić się będą w wartości tokenów użytych do zbudowania elementu drzewa i przypisanego typu.

Znaczenie parametrów:

1. EnumTypeDefinition
    a. Name – nazwa zdefiniowanego typu, wartość pola Name
    b. AllowedValues – dopuszczalne wartości wypisane kolejno jako znaczniki XML
2. VariableDefinition
    a. Multiplicity – krotność, przyjmuje wartości „Single” lub „Multiple”
    b. Name – nazwa zdefiniowanej zmiennej, wartość pola Name
3. IfStatement
    a. Condition – blok kodu XML odpowiadający BinaryExpression zawartemu w polu Condition
    b. Body – Blok kodu XML odpowiadający BinaryExpression zawartemu w polu Body. Operatorem w tym BinaryExpression jest zawsze „=”.
4. BinaryExpression

    a. Operator – zawartość pola Operator
    b. Left – Blok kodu XML odpowiadający Expression zawartemu w polu Left
    c. Right – Blok kodu XML odpowiadający Expression zawartemu w polu Right
5. BinaryExpression
    a. FunctionName – nazwa dwuargumentowej funkcji jakiej odpowiada Operator danego BinaryExpression.
    b. Left – Blok kodu XML odpowiadający Expression zawartemu w polu Left
    c. Right – Blok kodu XML odpowiadający Expression zawartemu w polu Right
6. FunctionCall
    a. Name – nazwa funkcji
    b. Inputs – Bloki kodu XML. odpowiadające wejściom funkcji. Wszystkie wejścia są typu Expression.
7. CollectionOfConstants
    a. Zawartość pola Ordering. Przyjmuje wartości „Ordered” lub „Unordered”.
    b. Values – Wypisane kolejno bloki kodu XML odpowiadające elementom pól Values. Wszystkie te elementy są typu Constant.
8. Constant
    a. TypeName – Typ każdego elementu należącego do kolekcji. Ten sam dla każdego elementu.
    b. Value – Zawartość pola Value.
9. VariableReference
    a. Name – Zawartość pola Name.

Non-terminal BinaryExpression jako jedyny serializowany jest do różnego w zależności od kontekstu, w jakim został użyty. Powodem tego jest gramatyka języka Rebita. Język natywny Rebita wspiera wyłącznie wyrażenia binarne nie będące częścią żadnego innego wyrażenia. Inaczej mówiąc, wyrażenia binarne będące korzeniem struktury danych, której są częścią. Kod XML odpowiadający temu wyrażeniu znajduję się we wcześniej zdefiniowanym BinaryExpression (*Rysunek 15*). W celu obejścia tego ograniczenia pozostałe wyrażenia binarne zamieniane są na dwuargumentowe funkcje (*Rysunek 16*). Przykładowo wyrażenie „operand1 + operand2” zserializowane będzie jako funkcja Add(operand1, operand2). Język wymaga również, aby wszystkie EnumTypeDefinition znajdowały się w znaczniku <Types>, VariableDefinitions w <Variables> i IfStatements w <Rules>.

4 Implementacja kompilatora w języku C#

4.1 Implementacja lexera.

W oparciu o informacje zawarte w rozdziale 3.3, w języku C# napisany został lexer.

```
public enum TokenType
{
    CurlyBraceLeft,
    CurlyBraceRight,
    SquareBracketLeft,
    SquareBracketRight,
    RoundBracketLeft,
    RoundBracketRight,
    Colon,
    Semicolon,
    MathOpartorMultiplyOrDivide,
    MathOpartorAddOrSubtract,
    LogicalOperator,
    AssignmentOperator,
    ComparisonOperator,
    If,
    Else,
    BaseVariableType,
    Enum,
    Multiplicity,
    Constant,
    Ordering,
    Identifier,
    EndOfFile
}
```

Rysunek 7 Definicja typu wyliczeniowego przedstawiającego wszystkie typy tokenów obsługiwane przez lexer.

```
13 references
public struct Metadata
{
    3 references
    public int Position { get; set; }
}
```

Rysunek 8 Definicja struktury przedstawiającej metadane tokenów.

```
public class Token
{
    74 references | 6/6 passing
    public TokenType Type { get; set; }
    45 references | 2/2 passing
    public string Value { get; set; }
    8 references | 24/24 passing
    public Metadata Metadata { get; set; }
}
```

Rysunek 9 Definicja klasy przedstawiającej token.

Na początku napisana została klasa Token. Zawiera ona 3 właściwości:

1. Właściwość Type, przedstawiającą typ tokenu, będący zmienną typu wyliczeniowego TokenType. Każda z możliwych do przyjęcia przez zmienną TokenType wartość odpowiada jednemu z typów tokena określonych w rozdziale 3.3.
2. Value, będącą wartością leksemu, wyodrębnienie którego z tekstu źródłowego spowodowało powstanie tokena.
3. Metadata, przedstawiająca metadane tokena. Należy do nich wyłącznie absolutna pozycja tokena w tekście źródłowym, wliczając znaki końca linii, paragrafu i tym podobne. Nie jest ona niezbędna do poprawnego działania kompilatora. Zwrócona wraz z wiadomością o błędzie w przypadku błędu parsowania ma znaczną wartość dla użytkownika, jako że pozwala mu na szybsze zlokalizowanie błędu w kodzie.

Klasę tą wykorzystuje lexer. Przyjmuje on tekst źródłowy w konstruktorze. Główną, zarządzającą całą logiką analizy leksykalnej metodą jest GetToken. Wydziela i zwraca ona pojedynczy token z tekstu źródłowego. Lexer zapamiętuje pozycję ostatniego zwróconego tokena. Wielokrotne zawołanie GetToken powoduje zwrócenie kolejnych tokenów. Jeśli okaże się, że analizie podległ cały tekst źródłowy, zwrócony zostanie token EndOfFile. W celu ułatwienia korzystania z lexera dodana została publiczna metoda GetPhrase, zwracająca frazę, to jest zbiór tokenów przestawiających jedną myśl (lub zdanie) w kontekście parsowania. W przypadku zdefiniowanego języka zawsze kończy się ona symbolem „;”. Poniżej przedstawiony został konstruktor lexera.

```
public Lexer(string tokenizationSource)
{
    _tokenizationSource = tokenizationSource;
    _tokenizationSourceLength = _tokenizationSource.Length;
    _tokenTypesByTokenSeparator = new Dictionary<char[], TokenType>()
    {
        {new []{',' }, TokenType.Colon },
        {new []{'(' },  TokenType.RoundBracketLeft},
        {new []{')' },  TokenType.RoundBracketRight},
        {new []{'{' },  TokenType.CurlyBraceLeft},
        {new []{'}' },  TokenType.CurlyBraceRight},
        {new []{'[' },  TokenType.SquareBracketLeft},
        {new []{']' },  TokenType.SquareBracketRight},
        {new []{';' },  TokenType.Semicolon},

        {new []{'*' },  TokenType.MathOpartorMultiplyOrDivide},
        {new []{'\\' },  TokenType.MathOpartorMultiplyOrDivide},
        {new []{'+' },  TokenType.MathOpartorAddOrSubtract},
        {new []{'-' },  TokenType.MathOpartorAddOrSubtract},

        {new []{'=' },  TokenType.AssignmentOperator},

        {new []{'>' },  TokenType.ComparisonOperator},
        {new []{'<' },  TokenType.ComparisonOperator},
        {new []{'>', '='},  TokenType.ComparisonOperator},
        {new []{'<', '='},  TokenType.ComparisonOperator},
        {new []{'=', '=' },  TokenType.ComparisonOperator},
        {new []{'!', '=' },  TokenType.ComparisonOperator},

         {new []{'&', '&' },  TokenType.LogicalOperator}
    };
    _tokenSeparators = _tokenTypesByTokenSeparator.Select(pair => pair.Key).ToList();
    _uniqueTokenSeparatorCharacters = _tokenSeparators.SelectMany(group => group).Distinct();
```

Rysunek 10 Konstruktor klasy Lexer (1)

```
    var tokenTypesByKeyword = new Dictionary<string, TokenType>()
    {
        {"in",  TokenType.ComparisonOperator},
        {"notin",  TokenType.ComparisonOperator},

        {"if",  TokenType.If},

        { "Enum", TokenType.Enum},

        { "Single", TokenType.Multiplicity},
        { "Multiple", TokenType.Multiplicity},

        { "Ordered", TokenType.Ordering},
        { "Unordered", TokenType.Ordering},

        { "int", TokenType.BaseVariableType},
        { "real", TokenType.BaseVariableType},
        { "string", TokenType.BaseVariableType},
        { "bool", TokenType.BaseVariableType},

        {"true", TokenType.Constant },
        {"false", TokenType.Constant }
    };
    var tokenTypesByPredicate = new[]
    {
        new Tuple<Func<string, bool>, TokenType>(IsLexemeInteger, TokenType.Constant),
        new Tuple<Func<string, bool>, TokenType>(IsLexemeNumeric, TokenType.Constant),
        new Tuple<Func<string, bool>, TokenType>(IsLexemeEnumValue, TokenType.Constant),
        new Tuple<Func<string, bool>, TokenType>(tokenContent => true, TokenType.Identifier)
    };
    _tokenFactory = new TokenFactory(tokenTypesByKeyword, tokenTypesByPredicate,
        _tokenTypesByTokenSeparator, _tokenizationSourceLength);
}
```

Rysunek 11 Konstruktor klasy Lexer (2)

W konstruktorze określona i zapisana w postaci prywatnych pól jest większość informacji potrzebnych podczas analizy leksykalnej. Są to pola:

1. _tokenizationSource – Tekst źródłowy, który podlegać będzie analizie.
2. _tokenizationSourceLenght – Długość tekstu źródłowego.
3. _tokenTypesByTokenSeparator – Słownik, w którym typom tokenów przypisane zostały leksemy pełniące funkcję separatorów. Nie należą do nich separatory nie będące leksemami, to znaczy znaki końca linii, paragrafu i tym podobne. Leksemy takie w kontekście tego lexera posiadają nazwę „tokenSeparators”, w myśl tego, że są to separatory, na podstawie których zostaną stworzone tokeny.
4. tokenTypesByKeyword – Słownik, w którym typom tokenów przyporządkowane zostały leksemy nie będące separatorami, które wyodrębnić można poprzez porównanie tekstu źródłowego do odpowiednich ciągów znaków. W kontekście lexera leksemy te noszą nazwę „keywords”.
5. tokenTypesByPredicate – Kolekcja tupli[25], w której predykatom określającym, czy na podstawie danego leksemu może zostać utworzony token o odpowiednim typie, przyporządkowany jest typ tokena. Przykładowo, jako pierwszy element, metodzie

[25] https://docs.microsoft.com/pl-pl/dotnet/api/system.tuple?view=netcore-3.1 (dostęp 03.07.2020)

sprawdzającej, czy leksem jest liczbą całkowitą przyporządkowana jest metoda tworząca token CONSTANT. Metody te wykorzystywane są kiedy nie da się określić tokenu poprzez porównanie leksemu do odpowiedniego ciągu znaków, a przez określenie specjalnych warunków dla leksemu. Warunki określone zostały w rozdziale 3.3. Warte zauważenia jest, że predykat ostatniej metody zawsze zwraca prawdę. Oznacza to, że token IDENTIFIER tworzony jest z każdego leksemu, którego nie udało się wykorzystać do stworzenia tokenu żadnego innego typu.

6. _tokenSeparators – Leksemy będące separatorami tokenów.
7. _uniqueTokenSeparatorCharacters – Pierwsze znaki, z których składają się leksemy będące separatorami.
8. _tokenFactory – fabryka tokenów napisana z wykorzystaniem wzorca projektowego Factory[26].

```
public Token GetToken()
{
    if (SkipInsignificantCharactersAndReturnIfEndOfFile())
    {
        _extractedTokenStartingPositions.Push(_tokenizationSourceLength);
        return _tokenFactory.CreateEndOfFileToken();
    }
    var startingPosition = _currentPosition;

    var (tokenSeparator, positionAfterTokenSeparator) = TryGetTokenSeparator(startingPosition);
    if (tokenSeparator.Count() > 0)
    {
        _extractedTokenStartingPositions.Push(startingPosition);
        _currentPosition = positionAfterTokenSeparator;
        return _tokenFactory.CreateTokenBasedOnTokenSeparator(tokenSeparator, startingPosition);
    }

    var (stringLiteral, positionAfterStringLiteral) = TryGetStringLiteral(startingPosition);
    if (stringLiteral.Count() > 0)
    {
        _extractedTokenStartingPositions.Push(startingPosition);
        _currentPosition = positionAfterStringLiteral;
        return _tokenFactory.CreateToken(TokenType.Constant, stringLiteral, startingPosition);
    }

    var (keyWord, positionAfterKeyword) = TryGetKeyword(startingPosition);
    _extractedTokenStartingPositions.Push(startingPosition);
    _currentPosition = positionAfterKeyword;
    return _tokenFactory.CanCreateTokenBasedOnKeyword(keyWord) ?
        _tokenFactory.CreateTokenBasedOnKeyword(keyWord, startingPosition) :
        _tokenFactory.CreateTokenUsingPredicates(keyWord, startingPosition);
}
```

Rysunek 12 Metoda GetToken lexera

Podstawową metodą działania lexera jest GetToken. Oprócz informacji określonych w konstruktorze, korzysta ona jeszcze z pola _currentPosition, którego używa do zapisywania i wczytywania między kolejnymi jej ewaluacjami pozycji znaku w tekście źródłowym od którego

[26] https://medium.com/lean-in-women-in-tech-india/understanding-the-factory-method-design-pattern-using-instagram-story-50aa1b56f5b0 (dostęp 03.07.2020)

należy kontynuować analizę. Sposób działania metody GetToken przedstawić można za pomocą kroków:

1. Za pomocą metody SkipInsignificantCharactersAndReturnIfEndOfFile pomijane są wszystkie znaki nieistotne dla analizy, to znaczy znaki końca linii, paragrafu itd. Dzięki temu między leksemami znajdować może się dowolna ilość tych znaków, a nie zmieni to wyniku działania lexera. Jeśli w trakcie pomijania znaków natrafiony zostanie koniec tekstu źródłowego, metoda zwraca prawdę, a lexer zwraca token EndOfFile.
2. Z wykorzystaniem metody TryGetTokenSeparator następuje próba wyodrębnienia leksemu będącego separatorem. Metoda ta zwraca tuplę, której pierwszym elementem jest sam leksem, drugim – nowa pozycja, od której kontynuowany zostanie proces analizy leksykalnej. Jeśli na podstawie leksemu nie da utworzyć takiego tokena, zwrócony leksem jest pusty.
3. Jeśli okaże się, że TryGetTokenSeparator zakończyła się sukcesem, w oparciu o otrzymany leksem za pomocą CreateTokenBasedOnTokenSeparator tworzony i zwracany jest token. Argumentami tej metody są leksem i jego absolutna pozycja w tekście źródłowym.
4. W przeciwnym wypadku za pomocą metody TryGetStringLiteral następuje próba wyodrębnienia leksemu będącego literałem ciągu znaków. Metoda ta zwraca tuplę, której pierwszy elementem jest oczekiwany leksem, drugim – nowa pozycja, od której kontynuowany zostanie proces analizy leksykalnej. Jeśli na podstawie leksemu nie da utworzyć takiego tokena, zwrócony leksem jest pusty.
5. Jeśli okażę się, że TryGetStringLiteral zakończyła się sukcesem, w oparciu o otrzymany leksem za pomocą Create tworzony i zwracany jest token typu CONSTANT.
6. W przeciwnym wypadku za pomocą metody TryGetKeyword wyodrębniony zostaje leksem nie będący separatorem. Jako, że SkipInsignificantCharactersAndReturnIfEndOfFile zwróciła fałsz, do analizy leksykalnej został co najmniej 1 poprawny znak. Metoda ta zawsze więc zwróci poprawny leksem. Najpierw, z użyciem CreateTokenBasedOnKeyword wykonana zostaje próba utworzenia tokenu poprzez znalezienie jego odpowiednika wśród leksemów nie będących separatorami. Jeśli token nie zostanie utworzony (a metoda zwróci null), ewaluowana jest CreateTokenUsingPredicates, która sprawdza predykaty z _tokenTypesByPredicate w kolejności, w jakiej zostały one zdefiniowane i jeśli któryś z predykatów zwróci prawdę, tworzy token typu odpowiadającego predykatowi.

```
public IEnumerable<Token> GetPhrase()
{
    var tokens = new List<Token>();
    while (true)
    {
        var token = GetToken();
        tokens.Add(token);
        if (token.Type == TokenType.Semicolon || token.Type == TokenType.EndOfFile)
            break;
    }

    return tokens;
}
```

Rysunek 13 Metoda GetPhrase lexera

Kolejna metoda, GetPhrase pozwala na pobranie tokenów stanowiących jedną frazę, a więc najmniejszy zestaw tokenów stanowiących samodzielny sens w procesie parsowania. Jeśli podczas procesu parsowania, w tekście źródłowym zauważony zostanie błąd, z wykorzystaniem tej metody możliwe jest pominięcie tokenów będących częścią frazy w której znaleziono błąd.

```
public Token PeekToken()
{
    var token = GetToken();
    PushLastToken();
    return token;
}

3 references
public (Token, Token) PeekTwoTokens()
{
    var token0 = GetToken();
    var token1 = GetToken();
    PushLastToken();
    PushLastToken();
    return (token0, token1);
}
```

Rysunek 14 Metoda PeekToken i PeekTokens lexera

PeekToken pozwala na „podejrzenie” tokena, a więc pobranie go tak, żeby możliwe było pobranie go jeszcze raz za pomocą GetToken lub PeekToken. PeekTwoToken pozwa na podejrzenie 2 tokenów.

```
public void PushLastToken()
{
    _currentPosition = _extractedTokenStartingPositions.Pop();
}
```

Rysunek 15 Metoda PushLastToken lexera

PushLastToken pozwala na zwrócenie ostatniego tokena. Po jej ewaluacji, ostatni token pobrany za pomocą GetToken, będzie najbliższym token do pobrania.

4.2 Implementacja algorytmu parsowania dla non-terminala Expression.

W oparciu o zdefiniowaną gramatykę i opisany algorytm parsowania, w języku C# został napisany parser. Poniżej znajdują się metody odpowiadające za parsowanie non-terminala Expression i jego składowych.

```
private VariableReference ParseToVariableReference()
{
    Token variableName = _lexer.GetToken();
    var output = new VariableReference() { Name = variableName.Value };

    return _typeAssigner.AssignTypeOrThrow(output, variableName);
}
```

Rysunek 16 Metoda odpowiadająca za parsowanie terminala VariableReference

```
1 reference
private Expression ParseToExpressionUnderBrackets()
{
    _lexer.GetToken().ThrowIfNotOfType(TokenType.RoundBracketLeft);
    var output = ParseToNonRootExpression();
    _lexer.GetToken().ThrowIfNotOfType(TokenType.RoundBracketRight);

    return output;
}
```

Rysunek 17 Metoda odpowiadająca za parsowanie non-terminala ExpressionWithinBrackets

```
1 reference
private Constant ParseToConstant()
{
    var constantToken = _lexer.GetToken();
    var output = new Constant() { Value = constantToken.Value };

    return _typeAssigner.AssignTypeOrThrow(output, constantToken);
}
```

Rysunek 18 Metoda odpowiadająca za parsowanie terminala Constant

```
private Expression ParseToFunctionEvaluation()
{
    Token functionName = _lexer.GetToken().ThrowIfNotOfType(TokenType.Identifier);
    _lexer.GetToken().ThrowIfNotOfType(TokenType.RoundBracketLeft);

    var output = new FunctionEvaluation()
    {
        Inputs = new List<Expression>(),
        Name = functionName.Value,
    };

    if(_lexer.PeekToken().Type == TokenType.RoundBracketRight)
    {
        _lexer.GetToken();
        return _typeAssigner.AssignTypeOrThrow(output, functionName);
    }

    var next = new Token() { Type = TokenType.Colon };
    while (next.Type == TokenType.Colon)
    {
        output.Inputs.Add(ParseToNonRootExpression());
        next = _lexer.GetToken();
    }
    next.ThrowIfNotOfType(TokenType.RoundBracketRight);

    return _typeAssigner.AssignTypeOrThrow(output, functionName);
}
```

Rysunek 19 Metoda odpowiadająca za parsowanie non-terminala FunctionEvaluation

```
private CollectionOfConstants ParseToCollectionOfConstants()
{
    Token squareBracketLeftOrCurlyBraceLeft = _lexer.GetToken()
        .ThrowIfNotOfTypes(TokenType.SquareBracketLeft, TokenType.CurlyBraceLeft);
    Token constant = _lexer.GetToken().ThrowIfNotOfType(TokenType.Constant);

    var outputValues = new List<string>() { constant.Value };
    var shouldExpectConstantTokenAsNext = false;

    var next = _lexer.GetToken();
    while (next.Type != TokenType.SquareBracketRight && next.Type != TokenType.CurlyBraceRight)
    {
        if (shouldExpectConstantTokenAsNext)
        {
            next.ThrowIfNotOfType(TokenType.Constant);
            outputValues.Add(next.Value);
        }
        else
            next.ThrowIfNotOfType(TokenType.Colon);

        shouldExpectConstantTokenAsNext = !shouldExpectConstantTokenAsNext;
        next = _lexer.GetToken();
    }

    if (squareBracketLeftOrCurlyBraceLeft.Type == TokenType.SquareBracketLeft)
        next.ThrowIfNotOfType(TokenType.SquareBracketRight);
    else
        next.ThrowIfNotOfType(TokenType.CurlyBraceRight);

    var output = new CollectionOfConstants()
    {
        Values = outputValues
    };

    return _typeAssigner.AssignTypeOrThrow(output, squareBracketLeftOrCurlyBraceLeft);
}
```

Rysunek 20 Metoda odpowiadająca za parsowanie non-terminala CollectionOfConstants

Krok analizy syntaktycznej i analizy semantycznej zostały połączone w jeden. Abstrakcjami pełniącymi funkcję tablicy symboli są ParserMemory i ParsingContext. ParserMemory odpowiada za zarządzanie informacjami na temat typów i zmiennych zdefiniowanych w trakcie procesu parsowania. ParsingContext zarządza informacjami na temat dostępnych funkcji i operatorów, które przekazywane są przy starcie aplikacji. Obie te klasy zaimplementowane zostały z wykorzystaniem wzorca projektowego serwisu[27]. Wykorzystywane są one przez klasę TypeAssigner, której zadaniem jest przypisanie typu dowolnej składowej Expression przyjętej jako pierwszy argument lub jeśli to niemożliwe, wyrzucenie InvalidTokenException, do utworzenia którego wykorzystuje token przyjęty jako drugi argument. Metoda ta ewaluowana jest na końcu każdej z metod tworzących składowe Expression.

Nazwa każdej z metod parsowania zaczyna się od „ParseTo” i kończy na nazwie reguły z jaką jest związana. Warte zauważenia jest podobieństwo między gramatyką, a kodem napisanym do jej realizacji. Przykładowo, metoda ParseToVariableReference składa tylko z pobrania pierwszego tokena ze strumienia i zwrócenia go jako Expression. Metoda

[27] https://medium.com/@davislevine/service-design-patterns-930203c8df37 (dostęp 04.07.2020)

ParseToFunctionEvaluation pobiera token ”(”, po czym do momentu pobrania tokena ”)” w celu uzyskania argumentów funkcji, ewaluuje na zmianę metodę ParseToExpression i wczytuje tokeny ”,”, będące separatorami dla argumentów.

```
private Expression ParseToFactor()
{
    var (first, second) = _lexer.PeekTwoTokens();

    if (IsFunctionEvaluation(first, second))
        return ParseToFunctionEvaluation();
    else if (IsCollectionOfConstants(first))
        return ParseToCollectionOfConstants();
    else if (IsVariableReference(first))
        return ParseToVariableReference();
    else if (IsConstant(first))
        return ParseToConstant();
    else if (IsExpressionUnderBrackets(first))
        return ParseToExpressionUnderBrackets();
    else
        throw new InvalidTokenException(first);
}
```

Rysunek 21 Metoda odpowiadająca za parsowanie non-terminala Factor

```
private bool IsFunctionEvaluation(Token identifier, Token roundBracketLeft)
{
    return identifier.Type == TokenType.Identifier
        && roundBracketLeft.Type == TokenType.RoundBracketLeft;
}
1 reference
private bool IsVariableReference(Token identifier)
{
    return identifier.Type == TokenType.Identifier;
}
1 reference
private bool IsConstant(Token identifier)
{
    return identifier.Type == TokenType.Constant;
}
1 reference
private bool IsCollectionOfConstants(Token bracketLeft)
{
    return bracketLeft.Type == TokenType.SquareBracketLeft
        || bracketLeft.Type == TokenType.CurlyBraceLeft;
}
1 reference
private bool IsExpressionUnderBrackets(Token token)
{
    return token.Type == TokenType.RoundBracketLeft;
}
```

Rysunek 22 Predykaty służące określeniu terminali lub non-terminali z których składa się non-terminal Factor

Metodą łączącą w jedno wszystkie powyższe metody jest metoda ParseToFactor. Zaczyna ona od „podejrzenia” 2 tokenów, po czym wybór jednej z wcześniej opisanych metod w oparciu o predykaty. Predykatem w językach programowania nazywamy dowolną funkcję zwracającą wartość boolowską. Jeśli żaden z predykatów nie zwróci prawdy, w tekście źródłowym jest błąd syntaktyczny, w wyniku czego wyrzucone zostanie InvalidTokenException.

```
private Expression ParseToBinaryExpressionWithOperators(TokenType operatorType,
    Func<Expression> underlyingExpressionCreationMethod)
{
    BinaryExpression output = new BinaryExpression()
    {
        Left = underlyingExpressionCreationMethod()
    };

    var next = _lexer.GetToken();
    while (next.Type == operatorType)
    {
        var right = underlyingExpressionCreationMethod();
        if (output.Right == null)
        {
            output.Operator = next.Value;
            output.Right = right;
        }
        else
        {
            output = new BinaryExpression()
            {
                Left = output,
                Operator = next.Value,
                Right = right
            };
        }

        _typeAssigner.AssignTypeOrThrow(output, next);
        next = _lexer.GetToken();
    }

    _lexer.PushLastToken();
    return output.Right == null ? output.Left : output;
}
```

Rysunek 23 Funkcja odpowiadająca za parsowanie non-terminala BinaryExpression

```
11 references | 47/47 passing
public Expression ParseToExpression()
{
    return
        ParseToBinaryExpressionWithOperators(TokenType.LogicalOperator,
        () => ParseToBinaryExpressionWithOperators(TokenType.ComparisonOperator,
        () => ParseToBinaryExpressionWithOperators(TokenType.MathOpartorAddOrSubtract,
        () => ParseToBinaryExpressionWithOperators(TokenType.MathOpartorMultiplyOrDivide,
        ParseToFactor))));
}
```

Rysunek 24 Funkcja odpowiadająca za parsowanie non-terminala Expression

Dla definicji non-terminali EXPRESSION, ADD/SUB EXPRESSION i MUL/DIV EXPRESSION, ze względu na podobieństwo między nimi, została przygotowana jedna metoda ParseToBinaryExpressionWithOperators. Pierwszym parametrem, jaki przyjmuje metoda jest typ tokena będącego operatorem wyrażenia binarnego, drugim – funkcja (formalnie delegat, callback) tworząca non-terminal o wyższym poziomie pierwszeństwa. W porządku od najwyższego do najniższego poziomu pierwszeństwa non-terminalami tymi są: FACTOR, MUL/DIV EXPRESSION, ADD/SUB EXPRESSION, COMPARISON EXPRESSION, EXPRESSION. Wszystkie metody tworzące non-terminale połączone zostały w metodzie ParseToExpression. Drzewo syntaktyczne będące wynikiem działania algorytmu składa się z klas przedstawionych na diagramie poniżej:

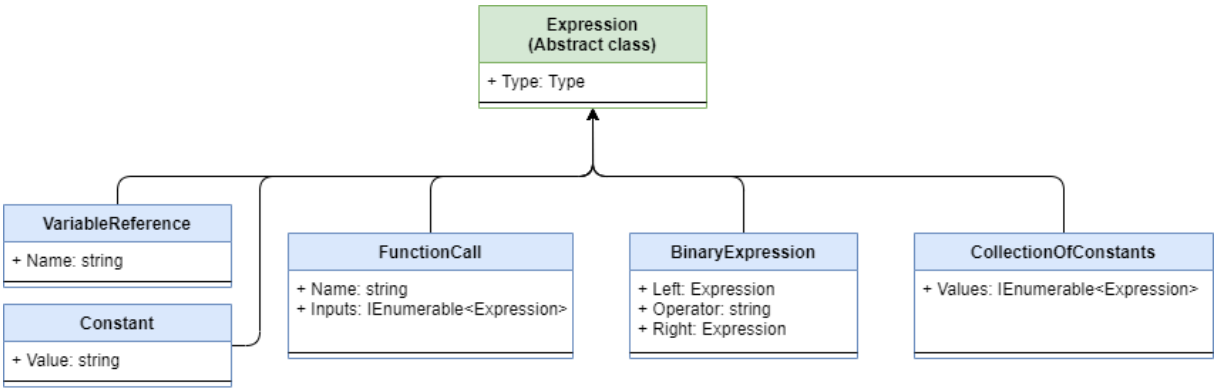


Diagram 7 Diagram klas non-terminali wchodzących w skład non-terminala Expression

Cała logika parsowania zamknięta została w klasie ExpressionParser. Tym samym sposobem napisane zostały parsery dla każdego pozostałego non-terminala.

4.3 Implementacja algorytmu parsowania dla pozostałych non-terminali

Wszystkie klasy realizujące logikę parsowania przedstawia diagram poniżej. Strzałki z niewypełnionym grotem oznaczają asocjację, z wypełnionym, realizację interfejsu. Zależności między klasami odpowiadają zależnościom między regułami w gramatyce. Jeśli w regule pierwszego non-terminala znalazł się drugi, zależnością parsera pierwszego, będzie parser drugiego.

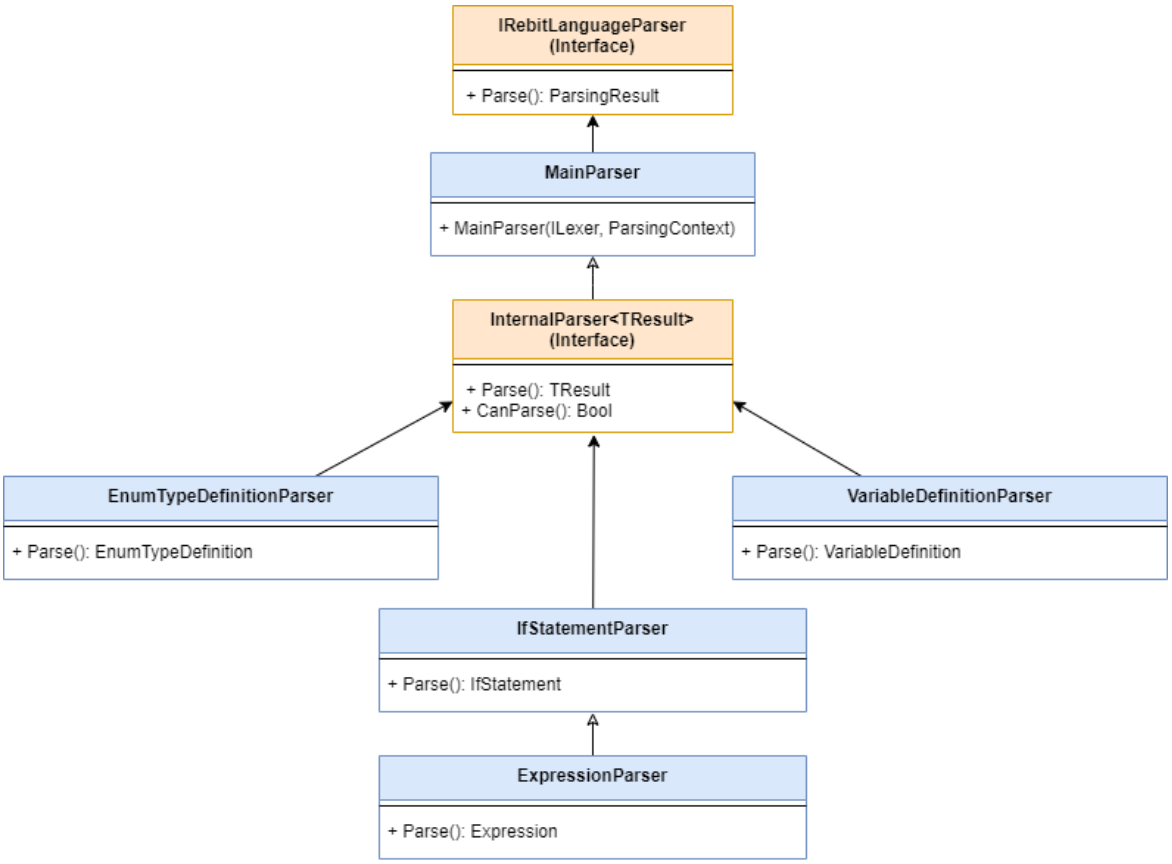


Diagram 8 Diagram klas parserów języka domenowego Rebita

4.4 Sprawdzanie typów – implementacja

Zgodnie z tym, co zostało opisane wcześniej, typy posiadają tylko i wyłącznie wyrażenia. Poniżej przedstawiony jest diagram klas wszystkich typów.

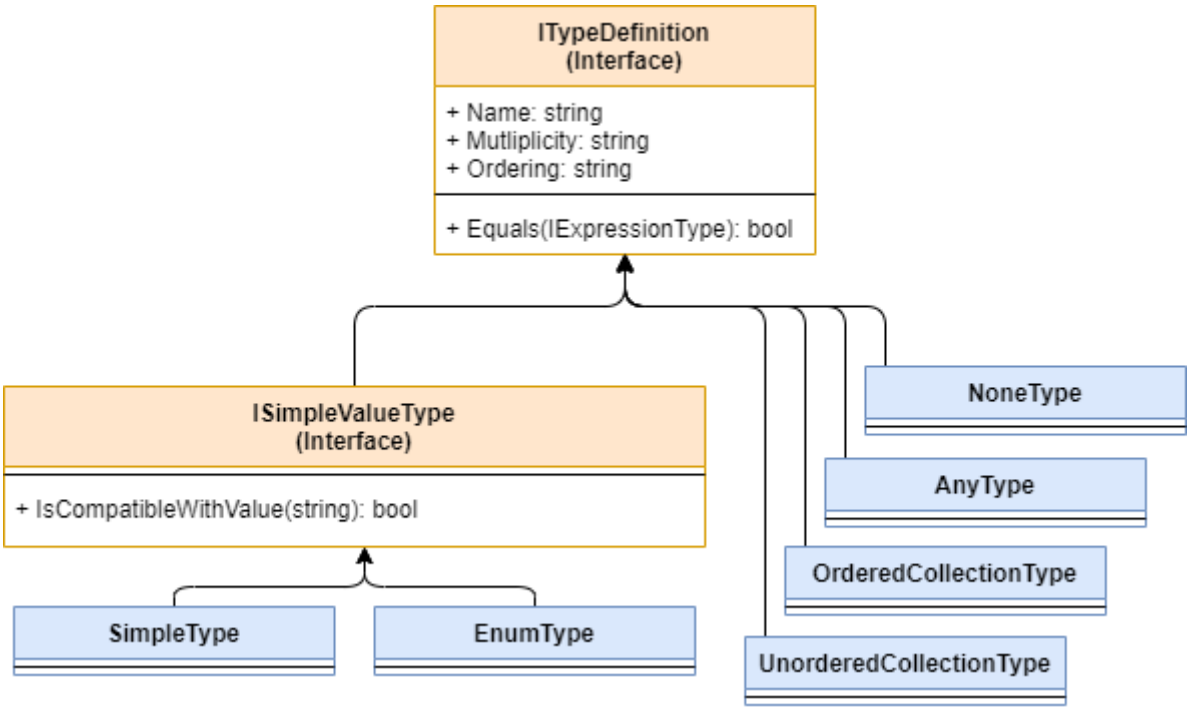


Diagram 9 Diagram klas typów wyrażeń (Expressions).

Każdy z typów posiada w sobie 3 publiczne cechy potrzebne do późniejszych kroków kompilacji. Są to:

1. Name – nazwa typu
2. Multiplicity - krotność
3. Ordering – uporządkowanie

Wszystkie te cechy odpowiadają cechom typów natywnego języka Rebita. Porównywanie typów zrealizowane zostało za pomocą publicznej metody Equals. Wykorzystywana jest ona do sprawdzania czy dany typ równoznaczny jest innemu, co oznacza, że może być z nim wykorzystany. Przy czym:

1. Obiekt typu zawsze równoznaczny jest samemu sobie.
2. Typ dowolny (AnyType) równoznaczny jest każdemu innemu typowi.
3. Typ żaden (NoneType) nie jest równoznaczny żadnemu typowi, nawet samemu sobie.

Po lewej stronie diagramu znajdują się typy dziedziczące po ISimpleValueType. Ich cechą wspólną jest metoda IsCompatibleWithValue, określająca czy dany leksem mógłby zostać skonwertowany do typu, na którym metoda jest wykonywana. Przykładowo, dla typu Int, określającego typ pojedynczej liczby całkowitej, metoda zwróciłaby prawdę dla literału „5”, ale nie dla „word”, czy „10.5”.

Wydziela się wiele typów. Pierwszym z nich SimpleType, którego obiekty przestawiają bazowe typy z natywnego języka Rebit. Dzieli się on na 4:

1. Int, określający liczbę całkowitą.
2. Bool, określający wartość prawdy lub fałszu.
3. Real, oznaczający liczbę rzeczywistą.
4. String, określający ciąg znaków.

Kolejny jest AnyType, przedstawiający typ kompatybilny z każdym innym typem. Wprowadzony został w celu umożliwienia istnienia funkcji, posiadającej dowolne argumenty. Wykorzystany w ten sposób porównany może być do klasy Object z języka C#. Inny typ, NoneType, nie jest kompatybilny z żadnym typem. Jego odpowiednikiem w języku C# jest void.

OrderCollectionType i UnorderedCollectionType reprezentują kolekcje typów prostych. W zależności od uporządkowania i typu ich elementów utworzonych zostało 8 obiektów. Każdy z nich przedstawia oddzielny typ:

1. UnorderedCollectionOfString – nieuporządkowana kolekcja elementów typu string.
2. UnorderedCollectionOfInt – nieuporządkowana kolekcja elementów typu int.
3. UnorderedCollectionOfReal – nieuporządkowana kolekcja elementów typu real.
4. UnorderedCollectionOfBool – nieuporządkowana kolekcja elementów typu bool.
5. OrderedCollectionOfString – uporządkowana kolekcja elementów typu string.
6. OrderedCollectionOfInt – uporządkowana kolekcja elementów typu int.
7. OrderedCollectionOfReal – uporządkowana kolekcja elementów typu real.
8. OrderedCollectionOfBool – uporządkowana kolekcja elementów typu bool.

Wszystkie obiekty tych typów tworzone są w chwili uruchomienia aplikacji. Są tylko do odczytu. Nie ma możliwości tworzenia na ich podstawie innych typów. Jako, że jednym z non-terminali Expression jest ewaluacja funkcji (Function call), potrzebne jest sposób na określenie jakie funkcje są dostępne, jakie są ich argumenty i co zwracają. Służą do tego klasy przedstawione na diagramie poniżej:

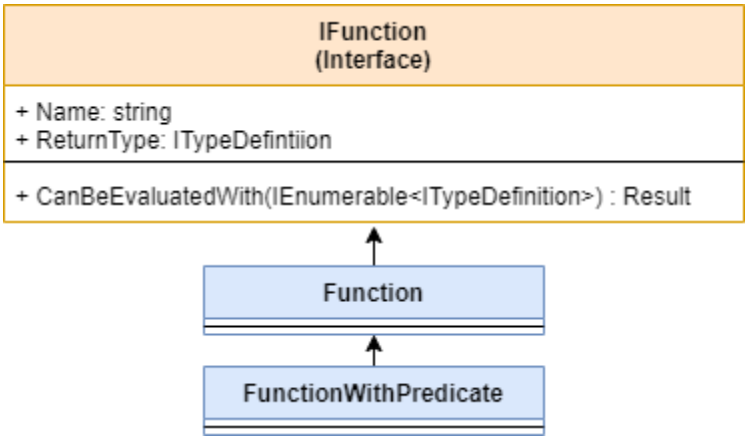


Diagram 10 Diagram klas funkcji wykorzystywanych do sprawdzania typów.

Każda funkcja posiada nazwę, zwracany typ i metodę CanBeEvaluatedWith. Metoda ta określa, czy możliwa jest ewaluacja danej funkcji ze zmiennymi o podanych jako parametry typach. Najbardziej podstawową klasą funkcji jest Function. Każdy jego obiekt reprezentuje nową funkcję, przyjmująca parametry o odpowiednich typach i wartość zwracaną o odpowiednim typie. Przeciążanie funkcji zrealizowane jest poprzez utworzenia kilku obiektów funkcji o tej samej nazwie i różnych parametrach. Kolejnym typem jest FunctionWithPredicate. Jest to rozszerzenie Function o możliwość dodania predykatu, jaki muszą spełniać wartości podane jako argumenty. Pozwala to na zdefiniowanie bardziej ogólnych funkcji spełniających swoje zadanie dla różnych typów. Przykładowo, chcąc zdefiniować funkcję AddToCollection, mającą pozwolić na dodanie elementu do kolekcji, zamiast definiować ją oddzielnie dla każdego bazowego typu (Int, String, Bool, Real), możemy określić, że przyjmuje ona dwa argumenty typu dowolnego (AnyType), przy czym muszą być one tego samego typu bazowego, a do tego jeden z nich musi być pojedynczy, a drugi wielokrotny.

Informacje o dostępnych funkcjach przekazane są w chwili uruchomienia aplikacji. Dzięki temu dostępne funkcje odzwierciedlać mogą stan silnika wnioskującego. Definicje funkcji przekazywane są konstruktorze klasy ParsingContext. Oprócz bezpośredniego dodawania funkcji pozwala ona również na określenie ich jako odpowiedników operatorów. Poniżej przedstawiony jest przykład definiowania funkcji.

```
new ParsingContext(
    functions: new List<Function> { },
    functionsByOperators: new Dictionary<string, IEnumerable<IFunction>>()
    {
        { "&&", new[] {
            new Function("And", new []{ Types.Bool, Types.Bool }, Types.Bool),
        }},
        { "+", new[] {
            new Function("Add", new []{ Types.Real, Types.Real }, Types.Real),
            new Function("Add", new []{ Types.Int, Types.Int }, Types.Int),
            new Function("Add", new []{ Types.Int, Types.Real }, Types.Real),
            new Function("Add", new []{ Types.Real, Types.Int }, Types.Real)
        }},
        { "-", new[]{
            new Function("Subtract", new []{ Types.Real, Types.Real }, Types.Real),
            new Function("Subtract", new []{ Types.Int, Types.Int }, Types.Int),
            new Function("Subtract", new []{ Types.Int, Types.Real }, Types.Real),
            new Function("Subtract", new []{ Types.Real, Types.Int }, Types.Real)
        }},
        { "*", new[]{
            new Function("Multiply", new []{ Types.Real, Types.Real }, Types.Real),
            new Function("Multiply", new []{ Types.Int, Types.Int }, Types.Int),
            new Function("Multiply", new []{ Types.Int, Types.Real }, Types.Real),
            new Function("Multiply", new []{ Types.Real, Types.Int }, Types.Real)
        }},
```

Rysunek 25 Przykład definicji funkcji które traktowane będą jako poprawne przez kompilator (1)

```
        { "==", new[]
        {
            new FunctionWithPredicate("Equal", new []{ Types.Any, Types.Any }, Types.Bool,
            inputs => inputs.First().Equals(inputs.Last()),
            "Type on the right side of opeator: \"=\" has to equal type on the left side.")
        }},
        { "!=", new[]
        { new FunctionWithPredicate("NotEqual", new []{ Types.Any, Types.Any }, Types.Bool,
            inputs => inputs.First().Equals(inputs.Last()),
            "Type on the right side of opeator: \"=\" has to equal type on the left side.") } },
        { "in", new[]
        {
            new FunctionWithPredicate("Contains", new []{ Types.Any, Types.Any }, Types.Bool,
            inputs =>
            {
                var type = inputs.First();
                var collectionOfType = inputs.Last();

                return type.Mutliplicity == "Single" && collectionOfType.Mutliplicity == "Multiple"
                && type.Name == collectionOfType.Name;
            }, "Type on right side of opeator: \"in\" has to be multiple of type on the left side.")}
        },
        { "notin", new []{ new FunctionWithPredicate("DoesNotContain",
        new []{ Types.Any, Types.Any }, Types.Bool,
            inputs =>
            {
                var type = inputs.First();
                var collectionOfType = inputs.Last();

                return type.Mutliplicity == "Single" && collectionOfType.Mutliplicity == "Multiple"
                && type.Name == collectionOfType.Name;
            }, "Type on right side of opeator: \"in\" has to be multiple of type on the left side.") }
        },
    }
)
```

Rysunek 26 Przykład definicji funkcji które traktowane będą jako poprawne przez kompilator (2)

Każda z dodanych funkcji odpowiada jednemu z wyrażeń binarnych. Wyrażeniu z operatorem „+” odpowiadają 4 przeciążenia funkcji Add. Pierwsza z nich przyjmuje 2 liczby rzeczywiste i zwraca liczbę rzeczywistą, druga przyjmuje 2 liczby całkowite i zwraca liczbę całkowitą, trzecia i czwarta przyjmują po jednej liczbie całkowitej i rzeczywistej i zwracają liczbę rzeczywistą. Przykładem funkcji z predykatem jest funkcja Equal będąca odpowiednikiem operatora „==”. Przyjmuje ona dwa argumenty typu dowolnego i zwraca typ Boolowski, przy czym wymaga, żeby lewy i prawy operand były tego samego typu.

4.5 Serializacja drzewa syntaktycznego do kodu XML – implementacja

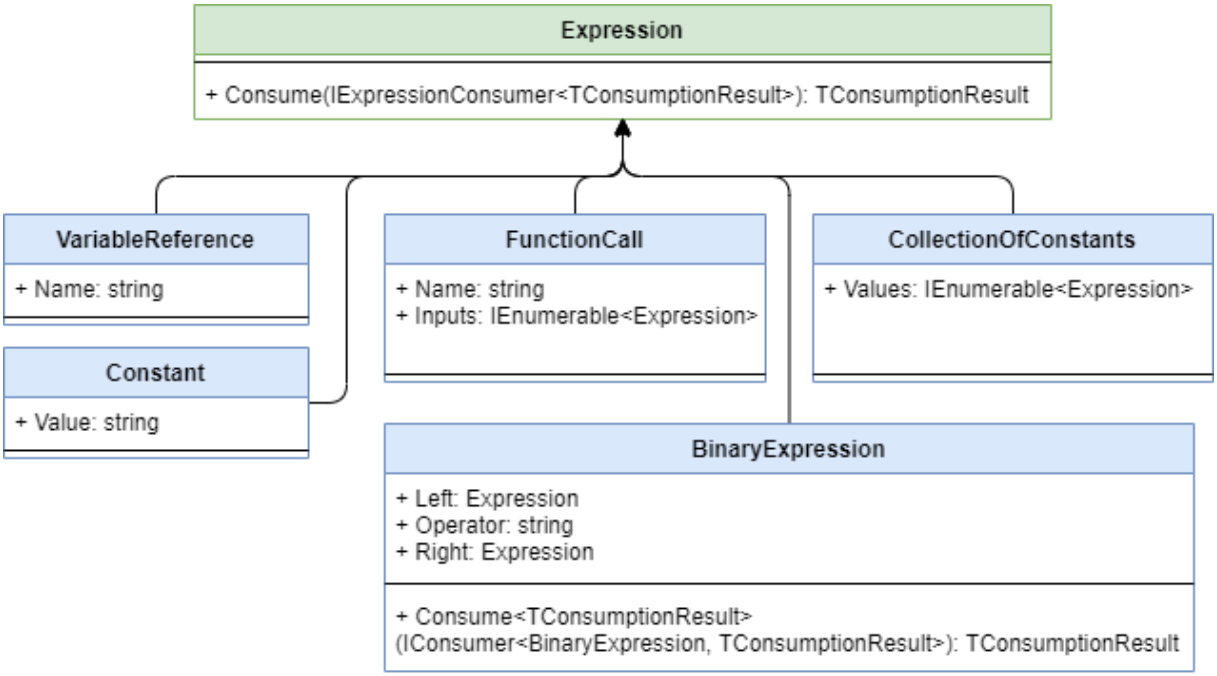


Diagram 11 Metody Konsumpcji poszczególnych elementów drzewa parsowania Expression

Klasie abstrakcyjnej Expression przypisana została abstrakcyjna metoda Consume. Metoda ta jako parametr przyjmuje realizację interfejsu IExpressionConsumer. Interfejs ten, dzięki metodom Consume, po jednej dla każdego elementu Expression, umożliwia konsumpcję wszystkich elementów drzewa z zachowaniem zależności między nimi. Posiada on argument generyczny[28] TConsumptionResult, określający wynik konsumpcji. Dziedziczy po nim klasa ExpressionXmlSerializer, odpowiadająca za serializację drzewa do natywnego zapisu zbioru reguł Rebita w języku XML. Jako, że jest to serializer, wynikiem konsumpcji jest string. Jeśli w przyszłości język XML Rebita zostanie zmieniony, nie będzie potrzeby zmian w kodzie kompilatora, a wystarczy napisanie nowej klasy dziedziczącej po IExpressionConsumerze. Z wykorzystaniem tego interfejsu możliwe będzie też rozszerzanie funkcjonalności kompilatora. Ze względu na specyfikę języka natywnego Rebita, potrzebne jest rozróżnienie między wyrażeniem binarnym będącym korzeniem drzewa Expression, wyrażeniami będącymi gałęziami tego drzewa. W tym celu BinaryExpression otrzymało oddzielną metodę Consume.

[28] https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/generics/ (dostęp 04.07.2020)

Poniżej przedstawiony jest kod ExpressionXmlSerializera, dziedziczącego po IExpressionConsumerze i RootBinaryExpressionXmlSerializera, dziedziczącego po IConsumerze<BinaryExpression, string>.

```
public class ExpressionXmlSerializer : IExpressionConsumer<string>
{
    2 references
    public string Consume(BinaryExpression source)
    {
        var functionName = source.ReturnType.Name;

        return
            "<Term Content =\"Function\">" +
                $"<Function IdRef =\"{functionName}\">" +
                     source.Left.Consume(this) +
                     source.Right.Consume(this) +
                "</Function >" +
            "</Term >";
    }
    2 references
    public string Consume(FunctionEvaluation source)
    {
        return
            "<Term Content =\"Function\">" +
                $"<Function IdRef =\"{source.Name}\">" +
                     string.Join("", source.Inputs.Select(input => input.Consume(this))) +
                "</Function >" +
            "</Term >";
    }
    2 references
    public string Consume(CollectionOfConstants source)
    {
        var type = source.ReturnType;
        var typeName = GetVariableTypeXmlEquivalent(type.Name);

        return $"<Term Content=\"Constant\"><Constant Multiplicity=\"Multiple{type.Ordering}\">" +
            string.Join("", source.Values.Select(value => $"<{typeName}>{value}</{typeName}>")) +
            "</Constant></Term>";
    }
```

Rysunek 27 Implementacja serializera dla gramatyki Expression (1)

```
public string Consume(Constant source)
{
    var typeName = GetVariableTypeXmlEquivalent(source.ReturnType.Name);

    return
        "<Term Content = \"Constant\">" +
         "<Constant Multiplicity = \"Single\">" +
          $"<{typeName}>{source.Value}</{typeName}>" +
           "</Constant>" +
          "</Term>";
}
2 references
public string Consume(VariableReference source)
{
    return
        "<Term Content=\"Variable\">" +
          $"<Variable IdRef = \"{source.Name}\"/>" +
         "</Term > ";
}
```

Rysunek 28 Implementacja serializera dla gramatyki Expression (2)

```
2 references
private string GetVariableTypeXmlEquivalent(string variableType)
{
    switch (variableType)
    {
        case "real":
            return "Real";
        case "bool":
            return "Boolean";
        case "int":
            return "Integer";
        default:
            return "String";
    }
}
```

Rysunek 29 Implementacja serializera dla gramatyki Expression (3)

```
public class RootBinaryExpressionXmlSerializer : IConsumer<BinaryExpression, string>
{
    1 reference
    public RootBinaryExpressionXmlSerializer(ExpressionXmlSerializer expressionParserSerializer)
    {
        _expressionParserSerializer = expressionParserSerializer;
    }

    private readonly ExpressionXmlSerializer _expressionParserSerializer;

    9 references
    public string Consume(BinaryExpression binaryExpression)
    {
        var rootExpressions = FlattenLogicalRootExpression(binaryExpression);
        return string.Join("", rootExpressions
            .Select(binaryExp => SerializeAsRootExpression(binaryExp)));
    }

    1 reference
    public string SerializeAsRootExpression(BinaryExpression source)
    {
        return
            $"<Atom Operator=\"{GetXmlEquivalentOfComparisonOperator(source.Operator)}\">" +
                "<LeftHand >" +
                     source.Left.Consume(_expressionParserSerializer) +
                "</LeftHand >" +
                "<RightHand >" +
                    source.Right.Consume(_expressionParserSerializer) +
                "</RightHand >" +
            "</Atom>";
    }
```

Rysunek 30 Implementacja serializera dla gramatyki BinaryExpression w sytuacji w której jest ono korzeniem drzewa syntaktycznego, w którym się znajduje (1)

```
private IEnumerable<BinaryExpression> FlattenLogicalRootExpression(Expression logcalExpression)
{
    var binaryExpression = logcalExpression as BinaryExpression;
    if (binaryExpression == null)
        return new BinaryExpression[0];

    if (binaryExpression.Operator == "&&")
        return FlattenLogicalRootExpression(binaryExpression.Left)
            .Concat(FlattenLogicalRootExpression(binaryExpression.Right));
    else
        return new[] { binaryExpression };
}
```

Rysunek 31 Implementacja serializera dla gramatyki BinaryExpression w sytuacji w której jest ono korzeniem drzewa syntaktycznego, w którym się znajduje (2)

```
private string GetXmlEquivalentOfComparisonOperator(string comparisonOperator)
{
    switch (comparisonOperator)
    {
        case "==":
            return "eq";
        case "!=":
            return "ne";
        case "<":
            return "lt";
        case "<=":
            return "le";
        case ">":
            return "gt";
        case ">=":
            return "ge";
        case "in":
            return "in";
        case "notin":
            return "notin";
        case "subset":
            return "subset";
        case "notsubset":
            return "notsubset";
        case "between":
            return "between";
        case "notbetween":
            return "not between";
        default:
            throw new ArgumentException($"Unexpected comparison operator: {comparisonOperator}");
    }
}
```

Rysunek 32 Implementacja serializera dla gramatyki BinaryExpression w sytuacji w której jest ono korzeniem drzewa syntaktycznego, w którym się znajduje (3)

Tym samym sposobem wykonana została serialiacja wszystkich pozostałych non-terminali. Całą logikę serializacji w metodzie Serialize łączy klasa ParseTreeXmlSerializer. Metoda ta przyjmuje jako argument drzewo syntaktyczne będące wynikiem parsowania, wydziela z nich odpowiednie non-terminale i serializuje kolejno w odpowiednich znacznikach. Po serializacji wszystkich elementów drzewa, z powstałego tekstu usuwa niepotrzebne znaki, to jest znaki końca linii, paragrafu i tym podobne. Jako wynik otrzymany zostaje zestaw reguł w postaci kodu XML, na którym można wykonać proces wnioskowania za pomocą silnika wnioskującego Rebit.

```
public string Serialize(ParsingResult parsingResult)
{
    string output =
        @"<?xml version=""1.0"" encoding=""UTF-8""?>
        <?xml-stylesheet type='text/xsl' href='rebit_xslt.xslt'?>
        <!-- Last modified 16-03-2011-->
        <Package xmlns:xsi=""http://www.w3.org/2001/XMLSchema-instance"" xsi:noNamespaceSchemaLocation
            <PackageHeader>
                <KnowledgeBaseName/>
                <Description/>
                <Version/>
                <Author/>
                <CreationDate>2010-02-21</CreationDate>
                <LastModifiedDate>2010-02-21</LastModifiedDate>
            </PackageHeader>";

    output += @"<RuleSets>
                <RuleSet Id=""ThermostatSettingRules"" Source=""UserDefined"">
                    <Description>Suggested thermostat settings</Description>
                    <Dialect>Complex</Dialect>";

    //enum definitions
    output += "<Types>";
    output += string.Concat(parsingResult.ParsedEnumTypeDefinitions
        .Select(enumDefiniton => enumDefiniton
            .Consume<EnumTypeDefinitionXmlSerializer, string>(EnumTypeDefinitionSerializer))
        .ToList());
    output += "</Types>";
```

Rysunek 33 Implementacja metody Serialize głównego serializera, łącząca całą logikę serializacji w jednej klasie (1)

```
    //variableDefinitons
    output += "<Variables>";
    output += string.Concat(parsingResult.ParsedVariableDefinitions
        .Select(variableDefinition => variableDefinition
            .Consume<VariableDefinitionXmlSerializer, string>(VariableDefinitionSerializer))
        .ToList());
    output += "</Variables>";

    //ifElseStatements
    output += "<Rules>";
    output += string.Concat(parsingResult.ParsedIfStatements
        .Select(IfStatement => IfStatement
            .Consume<IfStatementXmlSerializer, string>(IfStatementSerializer))
        .ToList());
    output += "</Rules>";

    output += @"</RuleSet>
            </RuleSets>
        </Package>";

    return Sanitize(output);
}

1 reference
private string Sanitize(string xml)
{
    return xml
        .Replace("\n", "")
        .Replace("\r", "")
        .Replace("\t", "")
        .Replace("  ", "");
}
```

Rysunek 34 Implementacja metody Serialize głównego serializera, łącząca całą logikę serializacji w jednej klasie (2)

Po połączeniu funkcjonalności Lexera, Parsera i Serializera otrzymany został sprawny Kompilator. Posiada on tylko jedną metodę Compile, za pomocą której zamienia kod języka domenowego na kod języka natywnego Rebita.

5 Budowa web api

5.1 Specyfikacja funkcjonalności Rebita wspieranych przez api

Zaprojektowany język nie przyniesie żadnego pożytku jeśli nie będzie możliwości wykorzystania go w procesie wnioskowania. W celu umożliwienia konsumpcji aplikacji, w technologii Asp.Net Core w standardzie REST wykonane zostało web api.

Możliwości wnioskujące Rebita udostępnione zostały w kodzie C# za pomocą interfejsu IReasoningService. Wybrane z nich, które wspierać ma api, to:

1. Utworzenie i rozpoczęcie nowej sesji wnioskowania. Podane muszą zostać przy tym: zestaw reguł, nazwy zmiennych wynikowych (w przypadku wnioskowania w tył lub mieszanego) i typ wnioskowania. Zmienne wynikowe to zmienne, których określenie wartości jest celem wnioskowania. Jeżeli wszystkich zmiennych wynikowych zostanie określona wartość, proces wnioskowania kończy się sukcesem. Istnieją 3 typy wnioskowania:
    a. Wnioskowanie w przód – W oparciu o aktualny stan zmiennych (nazywanych faktami) określić stan wszystkich innych, o ile możliwe jest to z posiadaną wiedzą. Wnioskowanie to ma więc na celu otrzymanie jak największej ilości istniejących w kontekście wnioskowania informacji.
    b. Wnioskowanie wstecz i wnioskowanie mieszane – Określić stan jednej lub więcej wybranych zmiennych (prawdziwość jednego lub więcej faktów) w oparciu o posiadaną wiedzę. Wynikiem wnioskowania są jedynie wcześniej określone jako potrzebne informacje, jak też informacje, których otrzymanie było niezbędne do wywnioskowania tych potrzebnych.

    Stan wnioskowania zapisywany jest w sesji, dzięki czemu możliwe jest zatrzymywanie, wznawianie, zapisywanie i edycja stanu wnioskowania będącego w toku. Przed rozpoczęciem procesu wnioskowania możliwe jest również określenie stanu innych zmiennych.
2. Wylistowanie istniejących sesji wnioskowania.
3. Podejrzenie stanu wybranej sesji wnioskowania. W trakcie wnioskowania może okazać się istnieją zmienne, których stan musi zostać ręcznie określony, żeby wnioskowanie mogło kontynuować lub doszło do błędu, który uniemożliwia dalsze wnioskowanie.
4. Zatrzymanie wybranej sesji wnioskowania.
5. Wznowienie wybranej sesji wnioskowania.
6. Usunięcie wybranej sesji wnioskowania.
7. Zmiana stanu wybranej sesji wnioskowania.

5.2 Implementacja web api

Popularnym standardem tworzenia api w technologii Asp.Net Core jest REST. Przejawia on się przede wszystkim w abstrakcji konsumpcji aplikacji jako procesu interakcji z zasobami, jakie posiada i udostępnia aplikacja. W tym przypadku zasobami tymi będą:

1. Sesja wnioskująca, odpowiadająca sesji wnioskującej serwisu wnioskującego Rebita. Interakcje przejawiać się będą w tworzeniu, zmianach stanu i usuwaniu sesji wnioskujących.
2. Translacja, będąca wynikiem kompilacji języka domenowego Rebita na język XML. Możliwe będzie wyłącznie tworzenie nowych translacji, które natychmiastowo będą zwracane użytkownikowi i nie będą zapisywane nigdzie po stronie serwera.

Web api w Asp.Net Core budowane jest w oparciu o opisany w podrozdziale 2.1 wzorzec projektowy Model-View-Controller. Definicja tych interakcji i obsługa komunikacji ze środowiskiem na zewnątrz aplikacji należy do zadań Controllera. Dla każdego zasobu tworzona jest jedna klasa Controllera. Za sesje wnioskowania odpowiada ReasoningSessionsController, za translacje – TranslationsController. Interakcje z zasobami realizowane są przez zawarte w nich publiczne metody. Poniżej przedstawiona jest implementacja Controllerów.

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class TranslationsController : ControllerBase
{
    0 references
    public TranslationsController(IRebitLanguageCompiler compiler, IFormFileReader formFileReader)
    {
        _compiler = compiler;
        _formFileReader = formFileReader;
    }

    private readonly IRebitLanguageCompiler _compiler;
    private readonly IFormFileReader _formFileReader;

    [HttpPost]
    1 reference
    public async Task<ActionResult<CreateTranslationResponseDTO>> CreateTranslation(
        [FromForm] CreateTranslationRequestDTO createTranslationRequestDTO)
    {
        var fileContent = await _formFileReader.ReadAsync(createTranslationRequestDTO.DslReasoningPackageFile);

        return CreatedAtAction(nameof(CreateTranslation),
            new CreateTranslationResponseDTO()
            {
                Result = _compiler.Compile(fileContent)
            });
    }
}
```

Rysunek 35 Implementacja Controllera odpowiedzialnego za translacje

TranslationController posiada 2 zależności. Pierwszą (IRebitLanugageCompiler) jest kompilator, drugą (IFormFileReader) czytnik plików IFormFile, pozwalający na otrzymanie zawartości dowolnego pliku typu IFormFile. IFormFile to abstrakcja przedstawiająca standardowy typ pliku przesyłany w zapytaniach http, wykorzystywany głównie do plików o niewielkich rozmiarach[29]. Zestaw reguł w języku domenowym przesyłany jest w postaci takiego pliku. Kontroler posiada tylko jedną metodę, jaką jest CreateTranslation. Działanie jej opisać można w 3 krokach:

[29] https://docs.microsoft.com/pl-pl/aspnet/core/mvc/models/file-uploads?view=aspnetcore-3.1 (dostęp 04.07.2020)

1. Odczyt zawartości pliku z zestawem reguł w języku domenowym z wykorzystaniem obiektu IFormFileReader.
2. Kompilacja zestawu reguł w języku domenowym na język XML.
3. Odesłanie wyniku kompilacji jako odpowiedź.

Komunikacja z Api pod tą metodą możliwa jest za pomocą zapytania HTTP POST na adres składający się z przedrostka „http://” adresu IP, portu, pod którym dostępna jest aplikacja i literału „/api/Translations”. W sieci lokalnej na porcie 8080 będzie to: http://localhost:8080/api/Translations.

```
[ApiController]
[Route("api/[controller]")]
1 reference
public class ReasoningSessionsController : ControllerBase
{
    0 references
    public ReasoningSessionsController(IRebitLanguageCompiler rebitLangueCompiler,
        IRebitReasoningService reasoningService, IFormFileReader formFileReader)
    {
        _rebitLanguageCompiler = rebitLangueCompiler;
        _reasoningService = reasoningService;
        _formFileReader = formFileReader;
    }

    private readonly IRebitLanguageCompiler _rebitLanguageCompiler;
    private readonly IRebitReasoningService _reasoningService;
    private readonly IFormFileReader _formFileReader;

    [HttpPost]
    1 reference
    public async Task<ActionResult<SessionDTO>> CreateReasoningSession(
        [ModelBinder(BinderType = typeof(JsonModelBinder))]
    CreateReasoningSessionRequestDTO createReasoningSessionRequestDTO, IFormFile ruleSetPackageFile)
    {
        var newSession = await _reasoningService.CreateReasoningSessionAsync(
            await GetXmlRulesetPackageContentAsync(createReasoningSessionRequestDTO, ruleSetPackageFile),
            createReasoningSessionRequestDTO.ReasoningMode,
            createReasoningSessionRequestDTO.FinalVariableName,
            createReasoningSessionRequestDTO.ValuesToBeAssignedByVariableNames);

        return CreatedAtAction(nameof(CreateReasoningSession), newSession.Id, newSession);
    }
```

Rysunek 36 Fragment implementacji Controllera odpowiedzialnego za sesje wnioskowania

ReasoningSessionController oprócz zależności posiadanych przez TranslationControllera posiada serwis wnioskujący Rebita (IRebitReasoningService). Głowną metodą Controllera jest CreateReasoningSession, służąca tworzeniu i uruchomianiu nowych sesji wnioskowania. Jako argumenty przyjmuje ona zestaw reguł w języku domenowym lub natywnym Rebita i obiekt klasy CreateReasoningSessionRequestDTO, na który składają się tryb wnioskowania, nazwę zmiennej wynikowej, wartości zmiennych do przydzielenia przed rozpoczęciem sesji wnioskowania, jak też informacja w którym z języków zapisany jest zestaw reguł. Działanie metody opisać można w krokach:

1. Odczytana zostaje zawartości pliku z zestawem reguł. Jeśli zestaw reguł zapisany jest w języku domenowym, za pomocą metody GetXmlRulesetPackageContentAsync następuje jego kompilacja na język XML.

2. Za pomocą metody serwisu wnioskowania CreateReasoningSessionaAsync utworzona i uruchomiona zostaje nowa sesja wnioskowania. Jako parametry tej metody przekazane są:
    a. zestaw reguł w języku XML
    b. tryb wnioskowania (w przód, w tył, mieszane)
    c. nazwa zmiennej wynikowej
    d. wartości zmiennych do przydzielenia przed rozpoczęciem sesji wnioskowania

   Wynikiem metody jest stan sesji po zatrzymaniu procesu wnioskowania. Zatrzymanie może zostać spowodowane udanym zakończeniem procesu wnioskowania lub błędem w trakcie trwania tego procesu.
3. Zwrócony zostaje wynik wnioskowania.

Komunikacja z Api pod tą metodą możliwa jest za pomocą zapytania HTTP POST na adres składający się z przedrostka „http://” adresu IP, portu, pod którym dostępna jest aplikacja i literału „/api/ReasoningSessions”. Tym samym sposobem wykonane zostały pozostałe metody tego Controllera. Są to:

1. GetReasoningSession do odczytania stanu sesji.
2. ListReasoningSessions do wylistowania wszystkich sesji.
3. UpdateReasoningSession do zmiany stanu sesji, w tym do zrestartowania lub zatrzymania.
4. DeleteReasoningSession do usunięcia sesji.

6 Dokumentacja użytkowa z wykorzystaniem narzędzia Swagger.

6.1 Dodanie dokumentacji do projektu

Popularnym sposobem na dodanie dokumentacji użytkowej dla .net Core web api jest wykorzystanie narzędzia Swagger. W przypadku asp.net Core 3.0 dodanie dokumentacji do projektu sprowadza się do 3 kroków:

1. Pobrania i dodania do projektu pakietu Nuget o nazwie NSwag.AspNetCore.
2. Ewaluacji metody services.AddOpenApiDocument w metodzie ConfigureServices w klasie Startup. Metoda ta umożliwia specyfikację danych takich jak wersja Api, tytuł, opis i wiele innych.
3. Ewaluacji app.UseOpenApi i app.UseSwaggerUi3 w metodzie Configure w klasie Startup [30].

W wyniku tego pod adresem http://<adres IP aplikacji>:<port>/swagger pojawi się aplikacja webowa opisująca i pozwalająca na wykonanie każdego zapytania obsługiwanego przez api.

6.2 Dokumenacja Controllerów

Po utworzeniu dokumentacji za pomocą komentarzy XML udokumentowany został każdy z Controllerów i z wykorzystaniem atrybutu ProducesResponseType określone zostały typy odpowiedzi zwracanych przez Controllery. Poniżej znajduje się udokumentowana metoda CreateTranslation TranslationControllera. Metoda ta zwrócić może 3 różne odpowiedzi, z których każda posiada inne ciało[31].

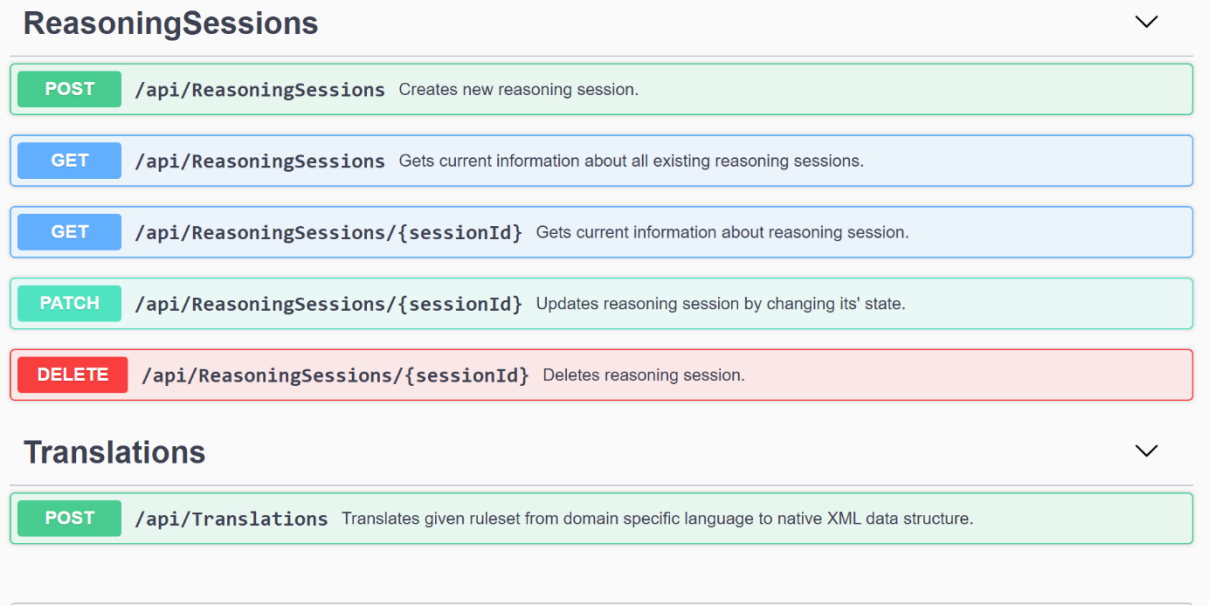
```
/// <summary>
/// Translates given ruleset from domain specific language to native XML data structure.
/// </summary>
[ProducesResponseType(typeof(CreateTranslationResponseDTO), StatusCodes.Status201Created)]
[ProducesResponseType(typeof(IEnumerable<CompilationErrorResponseDTO>), StatusCodes.Status400BadRequest)]
[ProducesResponseType(typeof(ApiErrorResponseDTO), StatusCodes.Status500InternalServerError)]
[HttpPost]
1 reference
public async Task<ActionResult<CreateTranslationResponseDTO>> CreateTranslation(
    [FromForm] CreateTranslationRequestDTO createTranslationRequestDTO)
{
    var fileContent = await _formFileReader.ReadAsync(createTranslationRequestDTO.DslReasoningPackageFile);

    return CreatedAtAction(nameof(CreateTranslation),
        new CreateTranslationResponseDTO()
        {
            Result = _compiler.Compile(fileContent)
        });
}
```

Rysunek 37 Udokumentowana metoda CreateTranslation TranslationControllera

Gotowa dokumentacja dostępna jest pod adres podanym w podrozdziale 6.1.

[30] https://github.com/RicoSuter/NSwag/wiki/AspNetCore-Middleware (dostęp 03. 07. 2020)
[31] https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages (dostęp 03. 07. 2020)



Rysunek 38 Interaktywna dokumentacja Swagger wygenerowana dla api

7 Podsumowanie

W ramach pracy udało się zrealizować postawione cele. Zaprojektowany został język domenowy pozwalający na tworzenie zestawów reguł w systemie regułowym Rebit z zachowaniem wszystkich funkcjonalności języka natywnego. Język pozwala między innymi na definiowanie typów wyliczeniowych, tworzenie zmiennych w oparciu o typy wyliczeniowe i typy proste, tworzenie reguł w postaci komunikatów logicznych, korzystając ze zmiennych, stałych o różnej krotności, funkcji, operatorów logicznych, operatorów porównania i operatorów matematycznych.

W celu umożliwienia wykorzystywania tego języka do pracy z Rebitem, napisany został kompilator. Podzielony on został na klasy, z których każda odpowiada jednemu lub więcej podprocesom kompilacji.

1. Lexer, zamieniający tekst źródłowy na ciąg tokenów, elementarnych składowych potrzebnych w procesie kompilacji. Każdy z tokenów posiada swój typ, ciąg znaków wyodrębniony z tekstu źródłowego podczas tworzenia tokena i metadane, takie jak pozycja tokena w tekście, niekonieczne w procesie kompilacji, jednak przydatne użytkownikowi.
2. Parser, tworzący na podstawie tokenów abstrakcyjne drzewo syntaktyczne, przedstawiające relację między tokenami i ich znaczenie jako całości. Stworzony on został w oparciu o gramatykę jednoznaczną, dzięki czemu i poprawny sposób wspiera pierwszeństwo między operatorami. Parser przyporządkowuje każdemu z elementów drzewa syntaktycznego odpowiedni typ, po czym ocenia zgodność między typami. Dzięki temu realizowane jest sprawdzanie typów. W przypadku błędu w trakcie parsowania, czy to z powodu błędu syntaktycznego, czy też niezgodności typów, proces parsowania zatrzymuje się i zwracana jest zrozumiała dla użytkownika wiadomość o błędzie.
3. Deserializer, zamieniający abstrakcyjne drzewo syntaktyczne na kod XML zrozumiały przez system regułowy Rebit. Drzewo nie jest bezpośrednio zależne od parsera, a jego zależność jest wstrzykiwana. Dzięki temu jeśli język natywny Rebita ulegnie zmianom, nie będzie potrzeby bezpośredniej zmiany kodu kompilatora, a wystarczy napisanie nowego Deserializera.

W celu umożliwienia korzystania z funkcjonalności systemu eksperckiego Rebit, w oparciu o nowy język w technologii Asp.Net Core w standardzie REST wykonane zostało web api. Proces wnioskowania z jego użyciem może zostać określony jako tworzenie i zarządzanie sesjami wnioskowania. Istnieje wiele możliwość rozwoju aplikacji. Niektóre z nich to:

1. Aplikacja client, pozwalająca na korzystanie z aplikacji poprzez GUI.
2. Linter do podświetlania składni nowego języka. Ze względu na podobieństwa do języka C#, do czasu powstania dedykowana lintera, wykorzystany może zostać linter C#.
3. Baza danych do przetrzymywania wiedzy, z jakiej korzysta Rebit.
4. Przydzielanie sesji wnioskowania i wiedzy użytkownikom. Jeden użytkownik nie powinien być w stanie usunąć sesji wnioskowania utworzonej przez drugiego.

8 Spis ilustracji

9 Spis diagramów

10 Bibliografia

[1] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986

[2] Praseed Pai K.T, *The Art of Compiler Construction using C#,* https://archive.codeplex.com/?p=slangfordotnet (dostęp 03.07.2020)

[3] Parser and Lexer — How to Create a Compiler part 1/5 — Converting text into an Abstract Syntax Tree, https://www.youtube.com/watch?v=eF9qWbuQLuw (dostęp 05.07.2020)

[4] Luke I. Wilson, *Understanding Compilers — For Humans,* https://medium.com/@thelukaswils/understanding-compilers-for-humans-ba970e045877 (dostęp 03. 07. 2020)

[5] Wykłady na temat kompilatorów, https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/ (dostęp 03. 07. 2020)

[6] George Gesion, *AI Then and Now – Expert Systems,* https://www.linkedin.com/pulse/ai-now-expert-systems-george-gesior (dostęp 03. 07. 2020)

[7] Dokumentacja użytkowa systemu eksperckiego Rebit

[8] Asp.Net Core web api – wprowadzenie*,* https://app.pluralsight.com/course-player?clipId=39b26eff-d35a-4044-9415-ab818394d985 (dostęp 03. 07. 2020)

[9] Dokumentacja platformy Asp.Net Core, https://docs.microsoft.com/pl-pl/aspnet/core/?view=aspnetcore-3.0 (dostęp 03. 07. 2020)

[10] Swagger – dokumentacja użytkowa, https://swagger.io/docs/ (dostęp 03. 07. 2020)

[11] Język dziedzinowy, https://pl.wikipedia.org/wiki/J%C4%99zyk_dziedzinowy (dostęp 03. 07. 2020)

[12] Docker – dokumentacja użytkowa, https://docs.docker.com/ (dostęp 03. 07. 2020)

[13] Menadżer pakietów Nuget – dokumentacja użytkowa, https://docs.microsoft.com/en-us/nuget/ (dostęp 03. 07. 2020)

[14] Delegaty w języku C# - dokumentacja, https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/delegates/ (dostęp 04.07.2020)

[15] Wprowadzenie do delegatów w języku C#, https://medium.com/@guoyuzhou004/delegates-in-c-why-we-need-that-a8f6519ef951 (dostęp 04.07.2020)

[16] Wprowadzenie do mikroserwisów, https://smartbear.com/solutions/microservices/ (dostęp 04.07.2020)

[17] Wzorzec projektowy serwisu, https://medium.com/@davislevine/service-design-patterns-930203c8df37, (dostęp 04.07.2020)

[18] Wzorzec projektowy MVC, https://app.pluralsight.com/course-player?clipId=ded2a0a5-cf51-492e-8d23-4e985096ef82 (dostęp 04.07.2020)

[19] Zasada inwersji zależności, https://medium.com/@mglover/dependency-inversion-principle-c0264a405d57 (dostęp 04.07.2020)

[20] Zasada pojedynczej odpowiedzialności, https://medium.com/@ipapikas/solid-series-1-5-single-responsibility-principle-5a93f235f0c4 (dostęp 04.07.2020)

[21] Wstrzykiwanie zależności w ASP.Net Core, https://docs.microsoft.com/pl-pl/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1 (dostęp 04.07.2020)

[22] Wprowadzenie do języków domenowych, https://tomassetti.me/domain-specific-languages/ (dostęp 04.07.2020)

[23] Swagger w projektach Asp.Net Core 3.0, https://medium.com/@didourebai/add-swagger-to-asp-net-core-3-0-web-api-874cb265854c (dostęp 04.07.2020)

[24] Docker – wprowadzenie dla web developerów, https://app.pluralsight.com/player?name=docker-web-development-m0&mode=live&clip=0&course=docker-web-development (dostęp 04.07.2020)

[25] Programowanie asynchroniczne w języku C# - dokumentacja, https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/concepts/async/ (dostęp 04.07.2020)

[26] Budowa asynchronicznego api w Asp.Net.Core, https://www.pluralsight.com/courses/building-async-api-aspdotnet-core (dostęp 08.07.2020)

[27] Wyrażenia lambda w języku C# - dokumentacja, https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions (dostęp 08.07.2020)

[28] Używanie generycznych delegatów w języku C#, https://dotnettutorials.net/lesson/generic-delegates-csharp/ (dostęp 08.07.2020)

[29] Zaawansowane programowanie asynchroniczne w języku C#, https://www.youtube.com/watch?v=ZTKGRJy5P2M (dostęp 08.07.2020)

[30] Budowanie .Net Core web Api zgodnego ze standardem REST, https://www.pluralsight.com/courses/asp-dot-net-core-3-restful-api-building (dostęp 08.07.2020)

[31] Definicja standardu architektonicznego REST, https://restfulapi.net/ (dostęp 08.07.2020)