Wydział
Informatyki

Przemysław Gawlas

numer albumu: gp36035

kierunek studiów: Informatyka

specjalność: Inżynieria oprogramowania

forma studiów: studia niestacjonarne

PROJEKT I IMPLEMENTACJA KOMPILATORA JĘZYKA JAVASCRIPT NA PLATFORMĘ .NET.

DESIGN AND IMPLEMENTATION OF THE JAVASCRIPT COMPILER INTO THE .NET PLATFORM.

praca dyplomowa magisterska

napisana pod kierunkiem:

dr inż. Piotr Błaszyński

Katedra Inżynierii Oprogramowania i Cyberbezpieczeństwa

Data wydania tematu pracy: 05.10.2020

Data dopuszczenia pracy do egzaminu: .....

(uzupełnia pisemnie Dziekanat)

Szczecin, 2021

Oświadczenie
autora pracy dyplomowej

Oświadczam, że praca dyplomowa magisterska pn. *Projekt i implementacja kompilatora języka JavaScript na platformę .NET.* napisana pod kierunkiem dr inż. Piotr Błaszyński jest w całości moim samodzielnym autorskim opracowaniem sporządzonym przy wykorzystaniu wykazanej w pracy literatury przedmiotu i materiałów źródłowych. Złożona w dziekanacie Wydziału Informatyki treść mojej pracy dyplomowej w formie elektronicznej jest zgodna z treścią w formie pisemnej.

Oświadczam ponadto, że złożona w dziekanacie praca dyplomowa ani jej fragmenty nie były wcześniej przedmiotem procedur procesu dyplomowania związanych z uzyskaniem tytułu zawodowego w uczelniach wyższych.

Podpis autora: . . . . . .

Szczecin, dnia: . . . . . .

Streszczenie

W tym miejscu trzeba napisać streszczenie pracy w języku polskim. Zawiera krótką charakterystykę dziedziny, przedmiotu i wyników zaprezentowanych w pracy. Maksymalnie 1/2 strony.

słowa kluczowe: np. informatyka, sterowanie, grafika komputerowa

Abstract

The abstract's purpose, which should not exceed 150 words, is to provide sufficient information to allow potential readers to decide on the thesis's relevance—a maximum of half the page.

keywords: e.g.: computer science, control, computer graphics

Spis treści

Wstęp

W branży programistycznej istnieje bardzo dużo języków programowania oraz różnych środowisk uruchomieniowych dla nich przeznaczonych. Podczas tworzenia oprogramowania niezbędny jest dla programisty kompilator. Zamienia on kod programu napisanego w konkretnym języku programowania, na zrozumiały dla środowiska uruchomieniowego ciąg instrukcji. Warto podkreślić, że języki programowania, które są proste do zrozumienia i opanowania dla programistów oraz pozwalają na pisanie programów, które można uruchomić na różnych urządzeniach i umożliwiają tworzenie bardzo zróżnicowanych rodzajów oprogramowania, są o wiele częściej używane niż inne. Powoduje to, że powstaje dużo różnych bibliotek i frameworków ułatwiających i automatyzujących tworzenie oprogramowania.

Jednym z popularnych języków wśród programistów jest język JavaScript, który może być uruchomiany w przeglądarkach internetowych lub w maszynie wirtualnej takiej jak Node.js. Innym z popularnych środowisk uruchomieniowych jest platforma .NET, dla której istnieje wiele kompilatorów różnych języków. Wykorzystanie istniejących bibliotek, frameworków czy modułów napisanych w języku JavaScript w niezmienionej formie na platformie .NET, umożliwi programistom na tworzenie bardziej uniwersalnego kodu.

Celem niniejszej pracy jest **zaprojektowanie** oraz **implementacja kompilatora** języka **JavaScript** na kod **IL Assembler** uruchamiany na **platformie .NET**. Następnie w celu sprawdzenia poprawności działania kompilatora, zostaną przeprowadzone testy przy pomocy prostych implementacji kodu JavaScript. Zostaną również zaprojektowane oraz zaimplementowane dwa bardziej skomplikowane testy, do **porównania maszyn wirtualnych Node.js oraz .NET**. Zostanie także porównany kod assemblera generowanego przez kompilator .Net Core języka C# z kodem generowanym przez implementowany kompilator.

Praca podzielona jest na cztery części. Pierwsza część opisuje pojęcia i technologie wykorzystywane do realizacji tego projektu oraz technologie pokrewne, które w pewnym stopniu realizują cel pracy lub realizują podobne założenia. Druga część poświęcona jest zaprojektowaniu tworzonego kompilatora. Kolejna część opisuje sposób implementacji tego kompilatora na podstawie zdefiniowanych założeń. Ostatnia część opisuje przeprowadzone testy realizowane w ramach pracy oraz przedstawia wyniki ich działania.

1. Pojęcia i technologie

1.1 Zakres projektu

Projekt będzie realizował zaprojektowanie i implementację kompilatora języka JavaScript na platformę uruchomieniową .NET core. W tym rozdziale zostaną opisane pojęcia oraz technologie, które będą wykorzystywane w realizacji tego projektu. Na początku opisane będą takie pojęcia jak *kompilator* oraz *maszyna wirtualna*. Następie zostaną omówione technologie wiodące w projekcie takie jak *JavaScript*, *Node.js*, *.NET Core* oraz *IL Assembler*.

1.1.1 Kompilator

W dzisiejszych czasach niezbędnym narzędziem programisty jest kompilator. Jest to narzędzie, którego zadaniem jest tłumaczenie programu napisanego przez programistę, na program który będzie można uruchomić na konkretnym środowisku uruchomieniowym.

Mówiąc ściślej kompilator to program napisany w języku implementacyjnym, który odczytuje język źródłowy i tłumaczy go na język wynikowy. Proces zamiany kodu źródłowego na wynikowy nazywany jest **kompilacją**. Kodem wynikowym procesu kompilacji może być od razu kod maszynowy, który interpretowany jest bezpośrednio przez procesor lub maszynę wirtualną, albo do kodu pośredniego, który też może zostać skompilowany przez inny kompilator.

Kompilatory mogą być napisane w dowolnym języku programowania. Istnieje kilka specjalnie zaprojektowanych do tego zadania języków, takie jak *Pascal* czy *Algol 68*. Nie mniej jednak, wybór języka do implementacji kompilatora przez twórcę, powinien się opierać na założeniu, że powinien on zminimalizować wysiłek implementacyjny i zmaksymalizować jakość kompilatora.

Język źródłowy który przetwarzany jest przez kompilator prawie zawsze jest oparty na wcześniej zdefiniowanej gramatyce. Dzięki temu program kompilatora potrafi odróżnić od siebie kolejne instrukcje i zamienić je na równoważne ciągi instrukcji w języku docelowym.[1]

Podobnym w działaniu jest **interpreter**, który tak jak kompilator, jest pisany w jednym implementacyjnym oraz odczytuje język kodu źródłowego, ale nie produkuje kodu wynikowego, tylko odczytany kod jest od razu wykonywany. Niektóre języki przyjmują

[1] W. M. Mckeeman. "Compiler Construction". W: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 1974, s. 1–36. ISBN: 9783540069584. DOI: 10.1007/978-3-662-21549-4_1. URL: http://link.springer.com/10.1007/978-3-662-21549-4_1, s. 1–4.

schematy zawierające wykorzystanie kompilatora oraz interpretera w procesie wytwarzania oprogramowania. Jednym z przykładów jest język **Java**, który kompilowany jest do postaci nazywanej *bytecode*, a następnie interpretowany jest przez maszynę wirtualną Java (Java Virtual Machine, JVM).[2]

1.1.2 Maszyna wirtualna

Wirtualizacja stała się ważnym narzędziem w projektowaniu systemów komputerowych, a maszyny wirtualne używane są w wielu obszarach informatycznych, od systemów operacyjnych po architektury procesorów języków programowania.

Dla programistów oraz użytkowników wirtualizacja likwiduje tradycyjne interfejsy oraz ograniczenia zasobów związanych z różnymi urządzeniami. Maszyny wirtualne zwiększają interoperacyjność oprogramowania oraz wszechstronność platformy, dlatego też często się je wykorzystuje.

Maszyna wirtualna to nic innego jak program uruchamiany na prawdziwej maszynie, który potrafi obsługiwać pożądaną architekturę. W ten sposób można obejść rzeczywistą kompatybilność maszyny i ograniczenia zasobów sprzętowych. Pozwala to, między innymi na równoczesne tworzenie oprogramowania dla wielu platform, bez konieczności stosowania bezpośrednio interfejsów rzeczywistej maszyny, a jedynie wykorzystanie tych udostępnianych przez maszynę wirtualną.[3]

1.1.3 JavaScript

Jest to skryptowy język programowania, dzięki którego można realizować aplikacje w paradygmacie imperatywnym, obiektowym oraz funkcyjnym. Najczęściej jest wykorzystywany w stronach internetowych, gdzie kolejne instrukcje wykonywane są przez przeglądarkę sieci Web, ale również zyskuje popularność w innych środowiskach.[4]

JavaScript został wdrożony w roku 1995 roku, jako sposób dodawania programów do stron internetowych. Jako pierwszą przeglądarką obsługującą JavaScript to Netscape Navigator. Następnie inne, głównie graficzne przeglądarki wprowadzały możliwość uruchamiania kodu napisanego w JavaScript. Umożliwiło to tworzenie nowoczesnych stron internetowych z którymi można było bezpośrednio współpracować, bez konieczności ponownego pobierania strony po każdej wykonanej akcji.

W momencie kiedy zaczęto używać JavaScript poza Netscape, został stworzony dokument standaryzujący, który opisuje sposób działania języka. Utworzono go, aby wszystkie nowo tworzone oprogramowanie mające wykorzystywać JavaScript, faktycznie używały tego samego języka. Dokument ten nazywany jest standardem **ECMAScript**, który został nazwany po organizacji Ecma International, twórców tego dokumentu.

JavaScript jest językiem bardzo elastycznym, przez co ma też swoje wady i zalety. Przez swoją elastyczność pozwala na wykorzystywanie wielu technik i praktyk programi-

[2] *Engineering a Compiler*. Elsevier, 2012. ISBN: 9780120884780. DOI: 10.1016/C2009-0-27982-7. URL: https://linkinghub.elsevier.com/retrieve/pii/C20090279827, s. 3, 4.

[3] J.E. Smith i Ravi Nair. "The architecture of virtual machines". W: *Computer* 38.5 (maj 2005), s. 32–38. ISSN: 0018-9162. DOI: 10.1109/MC.2005.173. URL: http://ieeexplore.ieee.org/document/1430629/.

[4] MDN contributors. *About JavaScript*. Maj 2020 (dostępny Maj 28, 2020). URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.

stycznych które mogą być niemożliwe w innych językach.

Jako, że jest to język skryptowy, to tak jak podobne tego typu języki posiada dynamiczne typowanie zmiennych. Oznacza to, że każda ze zmiennej jest definiowana poprzez słowo kluczowe `var`, a w nowszej wersji można to zrobić już przy pomocy dwóch różnych słów `const` oraz `let`. Kolejną z podstawowych rzeczy w JavaScript są funkcje. Dzięki nim można pisać programy we wspomnianych wcześniej paradygmatach. Pozwalają one nie tylko na wydzielenie kodu na mniejsze części ale również na definiowanie bardziej złożonych struktur czy klas.[5]

1.1.4 Node.js

Jest to asynchroniczne środowisko uruchomieniowe dla języka JavaScript. Node.js został zaprojektowany do tworzenia skalowalnych aplikacji sieciowych. Pozwala na jednoczesne przetwarzanie wielu połączeń. Przy każdym połączeniu następuje wywołanie zwrotne, a w przypadku jeśli nie będzie żadnej pracy do wykonania, Node.js przejdzie w tryb uśpienia.[6]

Środowisko Node.js oparte jest na implementacji silnika "V8" stworzonego przez Google. Zaimplementowany głownie jest w języku C i C++, koncentrując się na wydajności i niskim zużyciu pamięci. Różnica polega na tym, że silnik "V8" obsługuje głównie JavaScript w przeglądarkach internetowych, a Node.js został stworzony z myślą o obsłudze długotrwałych procesów serwerowych.

W celu obsługi jednoczesnego wykonywania logiki biznesowej, Node.js opiera się na asynchronicznym modelu zdarzeń wejścia i wyjścia, w przeciwieństwie do większości innych współczesnych środowisk, które oparte są na wielowątkowości. Model zdarzeń jest obsługiwany na poziomie języka, a jest to możliwe ponieważ JavaScript obsługuje wywołania zwrotne zdarzeń oraz funkcjonalny charakter JavaScript sprawia, że niezwykle łatwo jest tworzyć anonimowe obiekty funkcji, które można zarejestrować jako programy obsługi zdarzeń.[7]

1.1.5 .NET Core

Jest to platforma programistyczna ogólnego zastosowania z otwartymi kodami źródłowymi. Pozwala na tworzenie aplikacji dla systemów Windows, macOS oraz Linux przy wykorzystaniu różnych języków programowania.[8]

Architektura środowiska .NET składa się z takich komponentów jak:

– CTS (Common Type System) - opisuje wszystkie wspierane przez platformę typy. Definiuje zasady korzystania z danych typów, dostarcza zorientowany obiektowo model dla różnych języków implementowanych w .NET oraz zapewnia bibliotekę zawierającą prymitywne typy danych (takich jak `Boolean`, `char` itp.).

[5] Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. 2011. ISBN: 1593272820. DOI: 10.1190/1.9781560801597.

[6] Node.js. *About | Node.js*. 2017.

[7] Stefan Tilkov i Steve Vinoski. "Node.js: Using JavaScript to Build High-Performance Network Programs". W: *IEEE Internet Computing* 14.6 (list. 2010), s. 80–83. ISSN: 1089-7801. DOI: 10.1109/MIC.2010.145. URL: http://ieeexplore.ieee.org/document/5617064/.

[8] Paweł Łukasiewicz. *.NET Core vs .NET Framework*. (dostępny Lipiec 6, 2020). URL: https://www.plukasiewicz.net/Artykuly/NetFrameworkVsNetCore.

- CLS (Common Language Specyfication) - definiuje w jaki sposób mają być definiowane obiekty i funkcje, w języku przeznaczonym na platformę .NET. CLS jest podzbiorem CTS, co oznacza, że wszystkie opisane zasady CTS dotyczą również CLS.[9]
- FCL (Framework Class Library) - jest to standardowa biblioteka zawierająca podstawę implementacji klas, interfejsów, typów wartości czy usług, które wykorzystywane są do tworzenia aplikacji.[10]
- CLR (Common Language Runtime) - jest to środowisko uruchomieniowe, które uruchamia kod i zapewnia usługi ułatwiające proces programowania. Środowisko wykonawcze automatycznie obsługuje układ obiektów i zarządza referencjami no nich, zwalniając je w przypadku kiedy już nie są używane.[11]

1.1.6 IL Assembler

Każdy z kompilatorów przeznaczonych na platformę .NET, bez względu na wybrany język, kompiluje kod do postaci pośredniej, jakim jest kod IL.

Wykonywalny kod IL jest w formacie binarnym i nie jest czytelny dla człowieka. Oczywiście jak inne wykonywalne kody binarne, mogą zostać przedstawione w postaci assemblera, tak i kod IL może zostać zaprezentowany w postaci IL Assemblera. Zestaw instrukcji jest taki jak w przypadku tradycyjnego assemblera. Przykładowo, aby dodać dwie liczby należy użyć instrukcji `add`, a w przypadku odejmowania, należy użyć instrukcji `sub`.

Środowisko uruchomieniowe .NET nie potrafi jednak odczytywać bezpośrednio IL Assemblera. Aby kod napisany w IL Assemblerze można było uruchomić, trzeba skompilować go do postaci binarnej IL.[12]

1.2 Technologie pokrewne

Aktualnie istnieje wiele rozwiązań przetwarzających język JavaScript jak i narzędzi, dzięki którymi można budować, jak i uruchamiać aplikacje dedykowane na platformę .NET. W tym rozdziale zostaną opisane niektóre z tych technologii.

1.2.1 ActionScript

Obiektowy język oparty na ECMAScript używany w Adobe Flash/

[9] MDN contributors. *Common Type System & Common Language Specification*. Czer. 2016 (dostępny Lipiec 6, 2020). URL: `https://docs.microsoft.com/en-us/dotnet/standard/common-type-system`.

[10] HOW TO ASP.NET. *.NET FCL (Framework Class Library)*. (dostępny Lipiec 6, 2020). URL: `https://www.howtoasp.net/net-fcl-framework-class-library/`.

[11] MDN contributors. *Common Language Runtime (CLR) overview*. Kw. 2019 (dostępny Lipiec 6, 2020). URL: `https://docs.microsoft.com/pl-pl/dotnet/standard/clr`.

[12] Sameers Javed. *Introduction to IL Assembly Language*. Kw. 2003 (dostępny Lipiec 8, 2020). URL: `https://www.codeproject.com/Articles/3778/Introduction-to-IL-Assembly-Language`.

1.2.2 JScript

Jest to implementacja JavaScript przez Microsoft która jest uruchomiana w środowisku .NET. Istnieją pewne różnice w porównaniu do JavaScript.

1.2.3 TypeScript

Język programowania stworzony przez Microsoft. Uruchamiany na Node.js lub Deno. Może być przekompilowany do js es5 przez Babel.

1.2.4 ?CoffeeScript

Język programowania kompilowany do JavaScript. Nie wiem czy warto wspominać.

Inne języki kompilowane do JavaScript: Roy, Kaffeine, Clojure, Opal

1.2.5 Babel

Kompiluje przykładowo TypeScript do JavaScript lub JavaScript ES6 do ES5.

1.2.6 Deno

Maszyna wirtualna dla języka JavaScript oraz TypeScript.

1.2.7 asm.js

Jeśli dobrze zrozumiałem jest to okrojony JavaScript który tak samo może być uruchamiany w przeglądarkach czy Node.js/Deno. Wykorzystywany przy kompilacji kodu c++ do uruchamiania na tych maszynach.

1.2.8 Emscripten

Kompilator kodu LLVM do JavaScript.

1.2.9 WebAssembly

Język niskopoziomowy, który działa z szybkością zbliżoną do rozwiązań natywnych i pozwala na kompilację kodu napisanego w C/C++ do kodu binarnego działającego w przeglądarce internetowej. (Pomija JavaScript).

1.2.10 C#

Czy opisywać rodzinę .NET?

Na maszynę .NET istnieją implementacje języków takich jak Python, Java, C++ i inne.

1.2.11 .NET Framework

Platforma programistyczna.

1.2.12 Mono

Implementacja open source platformy .NET Framework.

1.2.13 DotGNU

Alternatywa dla Mono.

1.2.14 ?Roslyn

.NET Compiler Platform

2. Projekt kompilatora

2.1 Środowisko i narzędzia

Kod kompilatora zostanie zrealizowany w języku *C#* na platformie *.NET Core 3.1*. W projekcie zostanie również wykorzystana platforma *.NET Framework* w celu dekompilacji skompilowanego kodu testów, ich kompilacji z kodu *IL Assembler* oraz kompilacji kodu *JScript*. W celu weryfikacji poprawności kodu *JavaScript* zostanie wykorzystana platforma *Node.js* w wersji 10.15. Zostanie również wykorzystany skrypt pomocniczy w *PowerShell* służący do łatwego przeprowadzenia testów oraz do zebrania informacji o wielkości plików wykonywalnych i pomiaru czasu wykonania programów testowych. Drugim z dekompilatorów jaki zostanie wykorzystany w projekcie jest program *JetBrains dotPeek 2020.2.1*. Ostatnim z programów, służący do pomiarów zużycia pamięci, jest *JetBrains dotMemory 2020.2.1*.

Wszystkie testy oraz pomiary zostaną przeprowadzone na komputerze o następujących parametrach:

- System operacyjny: Windows 10 Home
- Typ systemu: 64-bitowy
- Procesor: Intel(R) Core(TM) i7-4700MQ CPU @ 2.40Hz
- Pamięć RAM: 16 GB

2.2 Analiza języka JavaScript i określenie zakresu implementacji

W tym rozdziale zawarty zostanie zakres implementacji oraz opis poszczególnych elementów języka JavaScript. W projekcie zostanie zaimplementowana jedynie część standardu ECMAScript, a niektóre mechanizmy zostaną uproszczone.

2.2.1 Wyrażenia

Składnia języka JavaScript zapożycza wiele rozwiązań użytych w Javie, jednak na konstrukcję miały też wpływ takie języki jak: Awk, Perl i Python. W języku JavaScript instrukcję nazywane są wyrażeniami, które rozdzielane są znakiem średniaka. Znaki białe takie jak spacja, tabulator czy znak końca linii nie mają wpływu na sposób działania kolejnych elementów wyrażeń, stanowią jedynie sposób ich oddzielenia. W kodzie źródłowym JavaScript rozróżnialna jest wielkość liter oraz wspierany jest standard znaków Unicode. ECMAScript definiuje również zestaw słów kluczowych i literałów oraz zasady automatycznego umieszczania średników ASI (Automatic semicolon insertion).

2.2.2 Komentarze

Rozróżniane są dwa typy komentarzy:

1. Jednoliniowy - definiowany jest przy pomocy znaku "//" oraz umieszczany jest na końcu linii.

```
console.log(); // komentarz
```

Algorytm 2.1: Przykład komentarza jednoliniowego

2. Wieloliniowy - zawarty jest pomiędzy dwoma elementami "/*" oraz "*/"

```
console.log();
/*
  komentarz na
  wiele linii
*/
```

Algorytm 2.2: Przykład komentarza wieloliniowego

2.2.3 Deklaracje zmiennych i stałych

Zmienne deklaruje się przy pomocy słów kluczowych `var`, `let` oraz `const`. Deklaracja przy pomocy `var` jest podstawowym sposobem tworzenia zmiennych w JavaScript. Zasięg takiej zmiennej nie może być ograniczony przez blok w którym jest zawarta, przez co może powodować błędy przy pisaniu kodu. W celu uściślenia zasięgu i przeznaczenia zmiennych powstały dwa inne sposoby deklaracji `let` oraz `const`. Oba te rodzaje deklaracji powodują, że zakres dostępności zmiennej jest ograniczony do bloku w którym została zadeklarowana. Różnicą między tymi dwoma deklaracjami jest taka, że przy pomocy `const` definiujemy stałą która musi być od razu zadeklarowana, a `let` działa podobnie jak `var`.

```
var zmienna1;
let zmienna2;
const stala = true;
```

Algorytm 2.3: Przykład deklaracji zmiennych

Przy deklaracji zmiennych przy użyciu `var` lub `let`, dla których nie przypiszemy żadnej wartości, przyjmują one wartość `undefined`

2.2.4 Typy danych

W najnowszym standardzie ECMAScript zdefiniowanych jest siedem typów danych:

1. `Boolean` - może przybierać dwie wartości `true` lub `false`.
2. `null` - słowo kluczowe oznaczające wartość zerową.
3. `undefined` - wartość nieokreślona.
4. `Number` - tym przeznaczony dla literałów całkowitych jak i zmiennoprzecinkowych.
5. `String` - typ przeznaczony dla literałów łańcuchowych reprezentujących zero lub więcej pojedynczych znaków ujętych w podwójny lub pojedynczy cudzysłów.
6. `Symbol` - wprowadzony w ECMAScript 6 typ danych, który pozwala na tworzenie unikalnych i nie zmiennych wartości.
7. `Object` - typ złożony do którego zaliczają się funkcje, tablice, słowniki oraz instancje klas.

2.2.5 Operacje arytmetyczne

Operatory arytmetyczne przyjmują jako operandy wartości liczbowe w postaci literałów lub zmiennych i jako wynik zwracają pojedynczą wartość liczbową. Standardowe operatory arytmetyczne to:

- dodawanie "+"
- odejmowanie "-"
- mnożenie "*"
- dzielenie "/"

```
1 var v1 = 10 + 15
2 var v2 = 100 + 100 / 2 * 3 + 10
3 var v3 = 100.7 + 10.40 / 1.67 * 3.1 + 10.23
```

Algorytm 2.4: Przykład użycia operatorów arytmetycznych

2.2.6 Operacje porównania

Operatory porównania porównuje swoje operandy czego wynikiem jest wartość logiczna, określająca czy dane stwierdzenie jest prawdziwe. Operandy mogą być wartościami liczbowymi, łańcuchami znaków, logicznymi lub wartościami obiektu. W przypadku kiedy dwa operandy są różnego typu, JavaScript próbuje przekonwertować je na odpowiedni typ do porównania.

Operatory porównania w języku JavaScript to:

- równość "=="
- nierówność "!="
- ścisła równość "==="
- ścisła nierówność "!=="
- większy ">"
- większy lub równy ">="
- mniejszy "<"
- mniejszy lub równy "<="

Tutaj warto zauważyć, że JavaScript posiada dwa operatory równości i nierówności. Różnica pomiędzy ścisłym porównaniu a zwykłym polega na tym, że w przypadku kiedy operandy są różnego typu, to dla zwykłego porównywania, JavaScript próbuje przekonwertować wartości do jednego typu. Przy porównywaniu ścisłym nie zachodzi konwersja.

```
1 var v1 = 2 == 2
2 var v2 = 1 + 1 != 2
3 var v3 = zmienna1 > zmienna2
```

Algorytm 2.5: Przykład użycia operatorów porównania

2.2.7 Instrukcje warunkowe

Instrukcje warunkowe to zbiór instrukcji, których wykonanie jest zależne od zdefiniowanego warunku którego wynikiem jest wartość logiczna "true" lub "false". JavaScript wspiera dwa rodzaje instrukcji warunkowych:

- `if .. else`
- `siwtch`

Instrukcja `if` wykonuje blok instrukcji w przypadku, kiedy podany warunek zwróci wartość “true”. Jeśli trzeba obsłużyć przypadek kiedy warunek nie został spełniony, można posłużyć się instrukcją `else` lub instrukcją `else if` podając inny warunek.

```
if (20 > number1) {
  console.log("number1 is less than 20")
} else if (25 > number1) {
  console.log("number1 is less than 25 and greater than 20")
} else {
  console.log("number1 is greater than 25")
}
```

Algorytm 2.6: Przykład użycia instrukcji `if .. else`

Instrukcja `switch` wykonuje blok instrukcji w przypadku, kiedy podane wyrażenie zgadza się z identyfikatorom danego bloku. Po dopasowaniu identyfikatora, wykonywane są wszystkie bloki instrukcji poniżej dopasowania, chyba że zostanie użyte słowo kluczowe `break`, które zakańcza wykonywanie instrukcji switch. Jeśli dane wyrażenie nie zostanie dopasowane do żadnego z identyfikatora, wykonywany jest kod z bloku o identyfikatorze `default`.

```
switch (color) {
case "Red":
  console.log("Chosen color: Red");
  break;
case "Blue":
  console.log("Chosen color: Blue");
  break;
case "Green":
  console.log("Chosen color: Green");
  break;
default:
 console.log("Chosen color: Default");
}
```

Algorytm 2.7: Przykład użycia instrukcji `switch`

2.2.8 Pętle

Przy pomocy pętli można w łatwy sposób powtarzać wykonywanie bloków instrukcji. W języku JavaScript występują różne rodzaje pętli. Można rozróżnić następujące konstrukcje:

- `for`
- `for .. in`
- `for .. of`
- `do .. while`
- `while`

Pętla `for` przyjmuje jako parametry trzy elementy: wyrażenie inicjalizacji, warunek zakończenia oraz wyrażenie inkrementacji. Wyrażenie inicjalizacji wykonywane jest tylko raz, na samym początku, jeszcze przed sprawdzeniem instrukcji warunkowej. Zazwyczaj wykorzystuje się je do zadeklarowania lub wyzerowania zmiennej iterującej. Wyrażenie warunkowe sprawdzane jest przed każdym wywołaniem bloku instrukcji. W przypadku kiedy wyrażenie warunkowe jest prawdziwe to zostanie wykonany blok instrukcji, a jeśli jest fałszywe to zakończy się działanie pętli. Wyrażenie inkrementacji wywoływane jest po zakończeniu wykonywania bloku instrukcji. Wykorzystuje się je do modyfikacji zmiennej iterującej.

```
for (var index = 0; index < 10; index = index + 1) {
  var element = x + index; // blok instrukcji
  console.log(element)
}
```

Algorytm 2.8: Przykład użycia instrukcji `for`

Pętla `for .. in` jest instrukcją pozwalającą na przeiterowanie się po elementach obiektu. Przyjmuje jako parametry deklarację zmiennej, do której wpisywana będzie wartość iteratora w danym przebiegu pętli oraz jako drugi argument, podaje się obiekt po którym chcemy się przeiterować.

```
for (var key in obj){
  console.log(obj[key])
}
```

Algorytm 2.9: Przykład użycia instrukcji `for .. in`

Pętla `for .. of` pozwala na iterowanie się po obiektach iterowalnych. Podobnie jak pętla `for .. in` przyjmuje dwa parametry, z których pierwszy również jest deklaracją zmiennej, jednak przechowuje ona wartość elementu w danej iteracji, a drugi parametr jest obiekt iterowalny.

```
for (var value of myArray){
  console.log(value)
}
```

Algorytm 2.10: Przykład użycia instrukcji `for .. of`

Pętla `do .. while` wykonuje blok instrukcji po którym sprawdzane jest wyrażenie warunkowe. Jeżeli wyrażenie jest prawdziwe wykonuje ponownie blok instrukcji, w przeciwnym wypadku wykonywanie pętli zostanie zakończone. Warto zwrócić uwagę, że blok instrukcji wykona się zawsze co najmniej raz.

```
do {
  console.log(zmienna)
} while (zmienna > 10)
```

Algorytm 2.11: Przykład użycia instrukcji `do .. while`

Pętla `while` działa w taki sam sposób jak pętla `do .. while` z tą różnicą, że warunek jest sprawdzany przed wykonaniem bloku instrukcji. oznacza to, że jeśli w pierwszej iteracji warunek będzie fałszywy, to blok instrukcji nie wykona się ani razu.

```
while (zmienna > 10) {
  console.log(zmienna)
}
```

Algorytm 2.12: Przykład użycia instrukcji `while`

2.2.9 Tablice

W języku JavaScript prawie wszystko jest obiektem. Każdy obiekt posiada swoje właściwości (atrybuty) oraz może posiadać swoje metody. Takimi obiektami mogą być między innymi takie konstrukcje jak tablice, obiekty słownikowe, funkcje czy też instancje klas.

Tablice (`Array`) pozwalają na przechowywanie grup danych i utrzymywaniu porządku w danych. Aby utworzyć obiekt tablicy można posłużyć się konstrukcją `new` lub też użyć nawiasów kwadratowych `[]`.

```
// Wykorzystanie operatora 'new'
var tablica1 = new Array();
// Wykorzystanie nawiasów
var tablica2 = [];
```

Algorytm 2.13: Przykład utworzenia tablicy

Tablice posiadają metody pozwalające na zarządzanie nimi. Przykładowo aby przypisać wartości można użyć metody `push`, wtedy dodawany element zapisywany jest na ostatnie miejsce w tablicy, a żeby pobrać ostatni element można użyć metody `pop`.

```
var tablica = []
// Dodanie elementu przy pomocy 'push'
tablica.push('element')
// Pobranie elementu przy pomocy 'pop'
var element = tablica.pop()
```

Algorytm 2.14: Przykład zarządania elementami tablicy przy pomocy 'push' i 'pop'

Przypisywanie oraz pobieranie elementów można również wykonywać za pomocą indeksów. Elementy tablic numerowane są od zera.

```
var tablica = []
// Dodanie elementu
tablica[0] = 'element'
// Pobranie elementu
var element = tablica[0]
```

Algorytm 2.15: Przykład zarządania elementami tablicy przy pomocy nawiasów

Tablice są obiektami dynamicznymi i w każdej chwili można zmienić ilość znajdujących się w nich elementów. W związku z tym tablica posiada atrybut `length` przechowujący ilość elementów znajdujący się w danej tej tablicy.

```
var tablica = []
tablica[0] = 'element1'
tablica[1] = 'element2'
// Pobranie ilości elementów
var len = tablica.length
```

Algorytm 2.16: Przykład pobrania ilości elementów tablicy

Warto również wspomnieć, że tablice pozwalają na przechowywanie elementów o różnych typach, jak i mogą posiadać elementy puste.

```
var tablica = []
tablica[0] = 'element1'
tablica[1] = 2
tablica[3] = true
var len = tablica.length
// tablica zawiera 4 elementy
// element o indeksie 2 jest pusty
```

Algorytm 2.17: Przykład przypisania elementów o różnych typach

2.2.10 Funkcje

Funkcja jest zbiorem instrukcji, które wykonują określone zadanie. Przed użyciem funkcji trzeba najpierw ją zdefiniować. Aby to zrobić należy użyć instrukcji `function`. Instrukcja ta wymaga podania: nazwy funkcji, listy argumentów oraz blok instrukcji, które mają być wykonywane w ramach tejże funkcji.

```
function square(number) {
  return number * number;
}
```

Algorytm 2.18: Przykład deklaracji funkcji

Aby wywołać funkcję podać jej nazwę oraz listę argumentów. Argumenty mogą być dowolnego typu, jednak trzeba pamiętać, że przekazywanie typów prostych takich jak liczby lub ciągi znaków, przekazywane są przez wartości. Natomiast przekazywanie obiektów, takich jak tablice, odbywa się poprzez referencje do danego obiektu. Oznacza to, że zmiana w obiekcie spowoduje zmianę obiektu poza funkcją.

```
var result = square(5)
```

Algorytm 2.19: Przykład uzycia funkcji

2.2.11 Zakres implementacji projektu

W niniejszym projekcie zostaną zaimplementowane następujące elementy:

- Komentarze jednoliniowe oraz wieloliniowe
- Proste typy danych:
  - `Boolean` - wartości *true* oraz *false*
  - `Number` - wartości całkowite oraz rzeczywiste
  - `String` - łańcuchy znaków
- Tworzenie zmiennych typu `var`, z opcją deklaracji kolejnych zmiennych po przecinku
- Uproszczona implementacja funkcji `console.log()`
- Konwersja typów danych
- Operacje matematyczne takie jak dodawanie, odejmowanie, mnożenie oraz dzielenie
- Konkatenacja łańcuchów znaków
- Operacje porównania
- Negacja wartości `Boolean`
- Instrukcja warunkowa `if`
- Pętle `while` oraz `for`
- Uproszczona tablica elementów - przechowywanie tylko wartości liczbowych całkowitych
- Deklaracja oraz wywoływanie funkcji

2.3 Parser

Do celów analizy składni języka JavaScript zostanie wykorzystane narzędzie ANTLR, które dostępne jest na wielu platformach oraz dla wielu języków. Wraz z narzędziem dostępna jest gotowa gramatyka dla języka JavaScript. Jednak, mimo dostępności pełnej gramatyki, zostanie na jej podstawie na nowo, tylko w obrębie planowanego zakresu. Dzięki temu zostaną uniknięte błędy, związane z brakiem pełnej implementacji funkcjonalności, oraz pozwoli to na stworzenie uproszczeń implementacyjnych.

Przy wyborze parsera, były również rozważane następujące narzędzia:

- LEX & YACC - standardowe narzędzia w systemach UNIX pozwalające na analizę plików tekstowych oraz generowanie na ich podstawie kodu w języku C.

- Coco / R - generator kompilatorów, który dla podanej gramatyki kodu źródłowego generuje parser dla tego języka. Implementacja dostępna w językach C#, Java, C++ i innych.
- gppg & gplex - implementacja LEX & YACC w języku C#.
- Owl - generator parserów, generujący do użycia plik nagłówkowy w języku C.

2.4 Struktura projektu

Projekt kompilatora został podzielony na następujące warstwy:

- Definicja gramatyki - zostanie tutaj zdefiniowana gramatyka dla języka JavaScript oraz zostanie wykorzystane narzędzie ANTLR do wygenerowania interfejsów dla tej gramatyki.
- Moduły - będą to klasy odpowiedzialne za obsługę grupy funkcjonalności takie jak:
  - Obsługa standardowego wyjścia
  - Obsługa zmiennych
  - Obsługa działań arytmetycznych
  - Obsługa wyrażeń warunkowych
  - Obsługa pętli
  - Obsługa tablic
  - Obsługa funkcji
- Zarządzanie stosem i definicjami
- Generator kodu assemblera

Zostanie również stworzona sekcja testów w której zawarte będą programy testowe w języku JavaScript oraz odpowiadające im implementacje w języku C#. Testy będą obejmować cały planowany zakres implementacji kompilatora. Dla testów zostanie wykorzystany skrypt pomocniczy napisany PowerShell, pozwalający na szybkie przeprowadzenie pełnych testów.

3. Implementacja aplikacji kompilatora

3.1 Parser

W projekcie zostanie wykorzystane narzędzie ANTLR, które można z następującego linku `https://www.antlr.org/download.html` (dostępne: 28.05.2021). Należy wydzielić osobny folder w którym będą znajdować się pliki z definicjami leksemów oraz gramatyki. Do danego folderu należy przenieść plik wykonywalny (*antlr.jar*) narzędzia ANTLR. Aby uruchomić narzędzie wymagane jest posiadanie środowiska wykonawczego Java.

W celu wygenerowania plików parsera dla środowiska .NET należy wywołać komendę przedstawioną w algorytmie 3.1. W folderze w którym zostanie wykonana komenda znajdują się aktualnie dwa pliki:

- *JavaScriptLexer.g4* - plik zawierający definicję leksemów
- *JavaScriptParser.g4* - plik zawierający definicję gramatyki

Po wykonaniu komendy generowania, powstaną cztery pliki wynikowe:

- *JavaScriptLexer.cs*
- *JavaScriptParser.cs*
- *JavaScriptParserBaseListener.cs*
- *JavaScriptParserListener.cs*

Są to pliki kodu źródłowego parsera dla zdefiniowanych wcześniej leksemów oraz gramatyki. W celu użycia parsera, należy utworzyć własną klasę, która będzie dziedziczyć po klasie *JavaScriptParserBaseListener* zawierająca definicje nagłówków funkcji.

```
1 | java -jar antlr.jar -Dlanguage=CSharp *.g4
```

Algorytm 3.1: Komenda uruchamiająca narzędzie ANTLR

3.1.1 Analiza leksykalna

W celu przeanalizowania pliku wejściowego kodu JavaScript pod kątem leksykalnym i utworzeniu tokenów, należy zdefiniować odpowiednie reguły w pliku *JavaScriptLexer.g4*.

Na samym początku pliku musi się znajdować deklaracja, że ten plik zawiera reguły do analizy leksykalnej.

```
1 | lexer grammar JavaScriptLexer;
```

Algorytm 3.2: Deklaracja pliku do analizy leksykalnej

Następnie w pliku powinny się znajdować reguły tokenów. Definiuje się je, nadając nazwę tokenu, a po dwukropku wpisuje się jedno lub wiele wyrażeń dopasowania oddzielonymi znakami '|'. Nazwy tokenu muszą zaczynać się wielką literą.

```
LESS_THAN:              '<';
MORE_THAN:              '>';
LESS_THAN_EQUALS:       '<=';
GREATER_THAN_EQUALS:    '>=';
EQUALS:                 '==';
NOT_EQUALS:             '!=';
IDENTITY_EQUALS:        '===';
IDENTITY_NOT_EQUALS:    '!==';
```

Algorytm 3.3: Przykład definicji tokenów dla znaków operacji porównania

```
CONSOLE_LOG:    'console.log';
VAR:            'var';
CONTINUE:       'continue';
FOR:            'for';
WHILE:          'while';
FUNCTION:       'function';
LENGTH:         'length';
RETURN:         'return';
```

Algorytm 3.4: Przykład definicji tokenów dla znaków słów kluczowych

Można również zdefiniować regułę pomocniczą, która nie będzie interpretowana jako token, ale jej dopasowanie będzie uwzględniane, przy wykorzystaniu jej w innej regule. Aby stworzyć taką regułę pomocniczą, należy użyć instrukcji `fragment` przed nazwą tokenu.

```
NUMBER
  : INTEGER_NUMBER '.' [0-9] [0-9_]*
  | '.' [0-9] [0-9_]*
  | INTEGER_NUMBER
  ;

fragment INTEGER_NUMBER
  : '0'
  | '-'?[1-9] [0-9_]*
  ;
```

Algorytm 3.5: Przykład definicji tokenu dla wielu dopasowań

Można również zdefiniować regułę, która zamiast tworzyć token, będzie dane dopasowanie pomijać. Aby tego dokonać, trzeba dołączyć do danej reguły, wywołanie komendy leksera `skip`. Istnieją również inne komendy takie jak: `channel(n)` - służący do zmiany kanału emitowanych tokenów; `type(n)` - zmieniający typ tokenu; `mode(n)`, `pushMode(n)`, `popMode`, `more` - kontrolujące tryb działania leksera.[1]

```
MULTI_LINE_COMMENT:   '/*' .*? '*/'   -> skip;
```

Algorytm 3.6: Przykład użycia komendy `skip`

3.1.2 Gramatyka

Po zdefiniowaniu tokenów, można rozpocząć definicję gramatyki. Tak jak było w przypadku pliku z leksemami, należy w pliku *JavaScriptParser.g4* zadeklarować, że plik zawiera reguły gramatyki oraz również wskazać plik zawierający tokeny.

```
parser grammar JavaScriptParser;

```

[1]Stack Overflow contributors. *Lexer rules in v4*. (dostępny Maj 30, 2021). URL: `https://sodocumentation.net/antlr/topic/3271/lexer-rules-in-v4`.

```
3   options {
4     tokenVocab=JavaScriptLexer;
5   }
```

Algorytm 3.7: Deklaracja pliku do analizy gramatyki

Definicje gramatyk deklaruje się w identyczny sposób jak tworzyło tokeny. Najpierw należy podać nazwę reguły a pod dwukropku, podajemy jedna lub wiele definicji oddzielonymi znakiem '|'. Zdefiniowana gramatyka może być wykorzystana w innych definicjach.

```
1   booleanValue
2   : BOOLEAN
3   ;
4
5   constantValue
6   : booleanValue
7   | stringValue
8   | numberValue
9   ;
```

Algorytm 3.8: Przykład definicji reguły gramatyki

Dla każdej zdefiniowanej reguły, przy pomocy narzędzia ANTLR, zostaną utworzone dwie funkcję w wygenerowanym pliku parsera. Pierwsza będzie się wykonywać przed, a druga po analizie reguł. Nazwy wygenerowanych funkcji będą identyczne jak nazwa reguły z dopisanym przedrostkiem *Enter* dla funkcji wywoływanej przed analizą reguły oraz *Exit* dla funkcji po analizie. Jako argument funkcji jest przekazywany kontekst parsera, dzięki któremu można pobrać wartość leksemu, lub też sprawdzić która definicja jest analizowana.

```
1   /// <summary>
2   /// Enter a parse tree produced by <see
        cref="JavaScriptParser.constantValue"/>.
3   /// <para>The default implementation does nothing.</para>
4   /// </summary>
5   /// <param name="context">The parse tree.</param>
6   public virtual void EnterConstantValue([NotNull]
        JavaScriptParser.ConstantValueContext context) { }
7   /// <summary>
8   /// Exit a parse tree produced by <see
        cref="JavaScriptParser.constantValue"/>.
9   /// <para>The default implementation does nothing.</para>
10  /// </summary>
11  /// <param name="context">The parse tree.</param>
12  public virtual void ExitConstantValue([NotNull]
        JavaScriptParser.ConstantValueContext context) { }
```

Algorytm 3.9: Przykład wygenerowanych funkcji gramatyki parsera

Domyślna implementacja wygenerowanych metod, nie posiadają żadnych instrukcji, dlatego też nie trzeba pisać własnych implementacji dla każdej z reguł. W przypadku kiedy chcemy napisać implementacje tych metod, jak było wcześniej wspominane, trzeba stworzyć własną klasę dziedziczącą po wygenerowanej klasie i podać jej instancję, jako klasa nasłuchująca (*listner*) dla parsera.

3.2 Kompilator

Program kompilatora wymaga podania jednego argumentu, który jest ścieżką do pliku źródłowego, zawierającego kod JavaScript. Jego wynikiem jest plik o tej samej na-

zwie z rozszerzeniem "*.il*" zawierający kod assemblera platformy .NET. Zawartość pliku źródłowego jest analizowana przez parser, który wywołuje odpowiednie metody przy dopasowaniu reguł gramatyki.

W funkcji głównej *Main* kompilatora znajduje się: otwarcie pliku źródłowego, utworzenie instancji modułu do generowania pliku wyjściowego, konfiguracja wcześniej wygenerowanego parsera oraz jego uruchomienie. Po zakończeniu analizy pliku źródłowego, plik jest zamykany, a następnie tworzony jest plik wyjściowy.

```
static void Main(string[] args) {
  if (args.Length < 1) {
    throw new ArgumentException("Needs file name as argument");
  }

  using (AsmGenerator.Create(args[0])) {
    using (FileStream fs = File.OpenRead(args[0])) {
      AntlrInputStream imputStream = new AntlrInputStream(fs);
      JavaScriptLexer lexer = new JavaScriptLexer(imputStream);
      CommonTokenStream commonTokenStream = new CommonTokenStream(lexer);
      JavaScriptParser parser = new JavaScriptParser(commonTokenStream);
      parser.BuildParseTree = true;
      JavaScriptParser.ParseContext context = parser.parse();

      JavaScriptListner listner = new JavaScriptListner();
      ParseTreeWalker walker = new ParseTreeWalker();
      walker.Walk(listner, context);
    }
  }
  Console.WriteLine("Done.");
}
```

Algorytm 3.10: Funkcja *Main* kompilatora

3.2.1 Wywołania parsera

Do przechwytywania reguł gramatyki została utworzona klasa o nazwie *JavaScriptListner* dziedzicząca po wygenerowanej klasie *JavaScriptParserBaseListener*. Zawiera ona nadpisane metody parsera w których wywoływane są odpowiednie funkcje z modułów obsługujących funkcjonalności. Funkcjonalności zostały podzielone na moduły i przeniesione do osobnych klas, ze względu na czytelność kodu.

Niektóre z funkcji wywołują jedynie funkcjonalność modułu, a inne sprawdzają dodatkowe warunki na podstawie dostarczonego kontekstu. Są przypadki gdzie, trzeba sprawdzić która z definicji reguł jest analizowana i w zależności od tego wywołać odpowiednią metodę z konkretnego moduł.

```
public override void ExitIfStatement(JavaScriptParser.IfStatementContext
    context)
{
  conditionModule.EndIfStatement();
}
```

Algorytm 3.11: Przykład prostej funkcji listnera

Klasa *JavaScriptListner* ma również za zadanie przy pomocy kontekstu, pobierać nazwy oraz wartości znajdujące się w pliku źródłowym dla konkretnych reguł. Przykładowo musi pobrać nazwę deklarowanej zmiennej lub też przypisywanej do niej wartości.

```
public override void
    ExitIdentifierValue(JavaScriptParser.IdentifierValueContext context) {
  string value = context.GetChild(0).GetText();
```

```
3       if(context.ChildCount > 1) {
4         arrayModule.CreateTableVariable(value);
5       } else {
6         variableModule.CreateVariable(value);
7       }
8     }
```

Algorytm 3.12: Przykład funkcji listnera z sprawdzeniem kontekstu

3.2.2 Zarządzanie stosem zmiennych

W celu zarządzania oraz przechowywania informacji o wartościach kodu źródłowego, została utworzona statyczna klasa `Store`. Zawiera ona następujące elementy:

- `Stack` - stos główny kompilatora
- `LabelStack` - stos indexów dla nazw instrukcji warunkowych oraz pętli
- `FunctionCallArgCounterStack` - stos służący do zliczania argumentów funkcji
- `_labelStackCounter` - licznik indexów etykiet
- `Variables` - słownik zdefiniowanych zmiennych
- `Functions` - słownik informacji definiowanych funkcji
- `ProcessingFunction` - nazwa aktualnie przetwarzanej funkcji.

Elementem który przechowuje główny stos oraz słownik zmiennych jest klasa `StoreItem`. Zawiera on niezbędne informacje o przetwarzanym elemencie. Dzięki tej klasie można przechować następujące informacje:

- `ItemType` - definiuje czy dany element jest zmienną o danym typie, zmienną tablicową, elementem tablicy, argumentem funkcji czy znakiem arytmetycznym lub porównania.
- `IsVariable` - definiuje czy element jest zmienną
- `IsTemporary` - definiuje czy element jest zmienną tymczasową
- `IsInitialized` - definiuje czy zmienna jest dopisana do listy do zainicjalizowania
- `IsFunctionParam` - definiuje czy zmienna jest parametrem funkcji
- `ParamPosition` - definiuje pozycję w liście argumentów funkcji
- `Parent` - element z którego powstał dany element
- `TableIndex` - index zmiennej tablicowej

Dla słownika informacji dla zdefiniowanych funkcji została stworzona klasa `FunctionStore`. Przechowuje ona:

- `Variables` - słownik zmiennych lokalnych funkcji
- `Params` - słownik parametrów funkcji
- `Name` - nazwę funkcji
- `IsUsed` - informacje, czy dana funkcja została wywołana co najmniej raz
- `ReturnValue` - wartość zwracana przez funkcję

3.2.3 Moduły

Każdy z modułów wykorzystuje `Store` do przetwarzania instrukcji kodu źródłowego i wywołuje odpowiednie funkcje klasy generującej kod assemblera. Moduły zostały podzielone ze względu na grupy implementowanych funkcjonalności.

Obsługa standardowego wyjścia

W JavaScript funkcją wyświetlającą tekst na standardowym wyjściu (konsoli) służy funkcja `log()` z standardowej klasy `console`. Jako że założony zakres implementacji kompilatora nie uwzględnia implementacji klas, wywołanie tej funkcji zostało zdefiniowane jako słowo kluczowe. Dzięki temu zabiegowi można zasymulować wywołanie tej metody, jakby rzeczywiście została wywoływana ze standardowej klasy.

```
writeStdOutput
: CONSOLE_LOG OPEN_PAREN constantValue CLOSE_PAREN
| CONSOLE_LOG OPEN_PAREN identifierValue CLOSE_PAREN
| CONSOLE_LOG OPEN_PAREN arithmeticOperation CLOSE_PAREN
| CONSOLE_LOG OPEN_PAREN functionCall CLOSE_PAREN
;
```

Algorytm 3.13: Definicja gramatyki dla funkcji standardowego wyjścia

Zostały zdefiniowane cztery gramatyki dla różnych rodzajów parametrów: wartości stałej (takich jak liczby, czy ciągi znaków), zmiennej, operacji arytmetycznej oraz wywołania funkcji.

Po realizacji definicji gramatyki parametru, zostanie zawsze odłożona na główny stos wartość którą chcemy wyświetlić na konsoli. W tym celu używana jest funkcja o nazwie `WriteStdOutput` z modułu `ConsoleModule`. Ściąga ona ze stosu wartość, a następnie w zależności od tego, czy jest to zmienna tablicowa wywołuje odpowiednią funkcję generatora assemblera.

```
public void WriteStdOutput(){
  StoreItem item = Store.PopStack();
  if(item.IsType(StoreItemType.ARRAY)){
    asmGenerator.WriteArrayToStdOutput(item);
  }else {
    asmGenerator.WriteToStdOutput(item);
  }
  asmGenerator.EmptyLine();
}
```

Algorytm 3.14: Definicja gramatyki dla funkcji standardowego wyjścia

Wyświetlenie pojedynczej wartości wymaga jedynie wygenerowania kodu ładującego daną wartość na stos oraz wywołanie odpowiedniej funkcji do wyświetlenia. W przypadku tablic został zastosowany zabieg, polegający na połączenie wszystkich elementów w ciąg znaków i wyświetlenie otrzymanej wartości na ekranie.

```
// Wyświetlenie pojedynczej wartości
ldstr "tekst 123"
call void [mscorlib]System.Console::WriteLine(string)

// Wyświetlenie tablicy
ldstr    "["
ldstr    ", "
ldloc v_tab
call string [mscorlib]System.String::Join<int32>(string, class
    [mscorlib]System.Collections.Generic.IEnumerable`1 <!!0/*int32*/>)
ldstr    "]"
call string [mscorlib]System.String::Concat(string, string, string)
call void [mscorlib]System.Console::WriteLine(string)
```

Algorytm 3.15: Przykład wygenerowanego kodu assemblera dla funkcji wyjścia standardowego

Obsługa zmiennych

Moduł obsługi zmiennych obejmuje funkcjonalności związanych z definicją elementów stałych, zmiennych, obsługi deklaracji zmiennych oraz obsługi rzutowania typów zmiennych.

Przy analizowaniu reguły gramatyki identyfikującej liczbę, ciąg znaków, wartość liczbową, czy nazwę identyfikatora, wywoływane są dla nich odpowiednie metody które utworzą element opisujący dane wartości i dopiszą je na szczyt głównego stosu. Deklaracja zmiennych przebiega w ten sam sposób, ponieważ przypisanie do listy zmiennych następuje przy pierwszym wystąpieniu przypisania wartości do danej zmiennej. Przy tych operacjach nie jest tworzony kod asemblerowy.

```
1  public void CreateNumber(string value) {
2    StoreItem item;
3    if (value.Contains(".")) {
4      item = StoreItem.CreateDouble(value);
5    } else {
6      item = StoreItem.CreateInteger(value);
7    }
8    Store.PushStack(item);
9  }
```

Algorytm 3.16: Implementacja funkcji tworzącej element opisujący stałą liczbową

W tym module znajduje się jeszcze funkcjonalność rzutowania typów zmiennych. Jest ona potrzebna, ponieważ język JavaScript jest językiem z dynamicznym typowaniem zmiennych i pozwala na operacje na różnych typach zmiennych. Funkcjonalność jest wykorzystywana przykładowo przy operacjach arytmetycznych, kiedy chcemy dodać do siebie wartość całkowitą do wartości rzeczywistej. W takim przypadku następuje rzutowanie zmiennej całkowitej na zmienną rzeczywistą.

Funkcja rzutowania przyjmuje jako argument element opisujący zmienną oraz typ docelowy. Na podstawie dostarczonego elementu tworzona jest jego kopia, następnie nowa zmienna jest inicjalizowana poprzez dopisanie jej do listy zadeklarowanych zmiennych. Na końcu wywoływana jest funkcja generująca kod assemblera, służąca do rzutowania wartości.

```
1  public StoreItem CastVariable(StoreItem item, StoreItemType itemType){
2    if(item.ItemType == itemType){
3      return item;
4    }
5
6    StoreItem vessel = item.CreateCastVariable(itemType);
7    asmGenerator.InitializeVariable(vessel);
8    asmGenerator.CastVariable(item, vessel);
9
10   asmGenerator.Comment($"{item.Print} -> {vessel.Print}\n");
11   return vessel;
12 }
```

Algorytm 3.17: Implementacja funkcji rzutującej zmienne

Funkcja generująca asemblera do rzutowania zmiennych, początkowo sprawdza, czy dany element jest stałą czy zmienną. W przypadku kiedy podany zostanie element opisujący wartość stałą, generator najpierw utworzy zmienną, a następnie przypisze do niej wartość stałej. Jest to konieczne, ponieważ rzutowanie typów zmiennych nie może odbywać się na wartościach stałych. Następnie w zależności od tego czy będzie odbywać

się rzutowanie na wartość rzeczywistą czy łańcuch znaków, zostanie wygenerowany odpowiedni kod.

```
// załadowanie stałej wartości całkowitej "5"
ldc.i4 5
// przypisanie wartości "5" do zmiennej "v_tmp_36"
stloc v_tmp_36

// załadowanie wrtości "v_tmp_36"
ldloc v_tmp_36
// konwersja wartości do typu rzeczywistego
conv.r4
// zapisanie wyniku konwersji do zmiennej "v_tmp_35"
stloc v_tmp_35
```

Algorytm 3.18: Kod assemblera rzutowania wrtości całkowitej na wartość rzeczywistą

W przypadku rzutowania zmiennych na łańcuch znaków, należy wywołać procedurę konwertującą wartość `ToString()`. Przy wywołaniu trzeba podać typ źródłowy wartości jaką chcemy rzutować. Są to odpowiednio:

- `Int32` dla typu całkowitego
- `Single` dla typu rzeczywistego
- `Boolean` dla typu logicznego

```
// załadowanie stałej wartości całkowitej "4.7"
ldc.r4 4.7
// przypisanie wartości "5" do zmiennej "v_tmp_34"
stloc v_tmp_34
// załadowanie adresu "v_tmp_34"
ldloca.s v_tmp_34
// wywołanie procedury konwersji typu rzeczywistego na wartość łańcucha
    znaków
call instance string [mscorlib]System.Single::ToString()
// zapisanie wyniku konwersji do zmiennej "v_tmp_33"
stloc v_tmp_33
```

Algorytm 3.19: Kod assemblera rzutowania wrtości rzeczywistej na wartość łańcucha znaków

Przy generowaniu kodu assemblera dla tej konwersji, w przypadku kiedy znajduje się ona w ciele funkcji, są wstawiane znaczniki. Będą one zamieniane na instrukcje asemblerowe dopiero po przeanalizowaniu całego kodu pliku wejściowego. Taki zabieg wynika z konieczności, podania typu wartości źródłowej przy wywołaniu funkcji `ToString()`, a przy analizowaniu funkcji dany typ może być nie znany.

```
// Znacznik opisuje, że w funkcji o nazwie "convert" należy obsłużyć
    rzutowanie zmiennej o nazwie "a"
ldarga v_a
#CAST_VARIABLE#a@convert
stloc v_tmp_3

// Kod asemblera po zamianie znacznika
ldarga v_a
call instance string [mscorlib]System.Single::ToString()
stloc v_tmp_3
```

Algorytm 3.20: Przykład kodu assemblera rzutowania wrtości w funkcji, przed podmianą znacznika i po podmianie

Obsługa działań arytmetycznych

Moduł obsługi działań arytmetycznych zawiera definicje elementu znaku działań matematycznych, obsługę funkcjonalności działań matematycznych, operacji przypisania oraz inkrementacji zmiennej.

Definiowanie elementu znaku działań matematycznych odbywa się w identyczny sposób jak przy definicji wartości stałych i zmiennych. Na główny stos aplikacji zostaje wrzucony element opisujący działanie, a raczej znak działania matematycznego.

Przy definiowaniu gramatyki dla działań matematycznych należy zdefiniować osobne reguły dla działań dodawania i odejmowania oraz mnożenia i dzielenia, w celu zachowania zasady kolejności wykonywania działań.

```
arithmeticOperation
: arithmeticOperation arithmeticAdditiveSign arithmeticOperationHigher
| arithmeticOperationHigher
;

arithmeticOperationHigher
: arithmeticOperationHigher arithmeticMultiplpicativeSign value
| value
;

arithmeticAdditiveSign
: PLUS
| MINUS
;

arithmeticMultiplpicativeSign
: MULTIPLY
| DIVIDE
;
```

Algorytm 3.21: Definicja gramatyki dla funkcji działań matematycznych

Przy wywołaniu funkcji przetworzenia działania matematycznego, na stosie głównym znajdować się będą trzy elementy uczestniczące w działaniu. Będzie to kolejno dwa elementy opisujące wartości oraz element opisujący znak operacji. Następnie weryfikowane jest jakiego typu są elementy. W przypadku różnych typów danych wykonywane są funkcje rzutowania wartości na wspólne typy danych.

```
public void ProcessArithmeticOperation() {
  StoreItem arg2 = Store.PopStack();
  StoreItem sign = Store.PopStack();
  StoreItem arg1 = Store.PopStack();

  bool stringOperation = false;

  if (StoreItem.IsAnyType(StoreItemType.STRING, arg1, arg2)) {
    if (sign.Value != "+") {
      throw new InvalidOperationException($"Operation is not allowed for
          strings.");
    }
    stringOperation = true;

    arg1 = variableModule.CastVariable(arg1, StoreItemType.STRING);
    arg2 = variableModule.CastVariable(arg2, StoreItemType.STRING);
  }
  if(sign.Value == "/" || StoreItem.IsAnyType(StoreItemType.DOUBLE, arg1,
      arg2)){
    arg1 = variableModule.CastVariable(arg1, StoreItemType.DOUBLE);
    arg2 = variableModule.CastVariable(arg2, StoreItemType.DOUBLE);
  }

  ...
```

Algorytm 3.22: Implementacja funkcji obsługującej działania matematyczne cz.1

Następnie wywoływane są funkcje generujące kod asemblera. W pierwszym kroku obie wartości są ładowane na stos, a następnie w zależności od tego, czy operacja jest

wykonywana na typach łańcuchów znaków, czy na wartościach liczbowych, wykonywane są funkcje do łączenia wartości typu łańcuchowego lub też do wykonania operacji na liczbach.

```
...

asmGenerator.Load(arg1);
asmGenerator.Load(arg2);
if (stringOperation) {
  asmGenerator.ConcatStrings();
} else {
  asmGenerator.ExecuteArithmeticOperation(sign);
}

...
```

Algorytm 3.23: Implementacja funkcji obsługującej działania matematyczne cz.2

Na końcu tworzona jest nowa zmienna do której przypisywana będzie wartość wyniku powyższej operacji oraz wynik zostanie wrzucony na główny stos kompilatora. W przypadku kiedy jedna z wartości była już elementem tymczasowym, zamiast tworzenia nowego elementu tymczasowego, wynik zostanie przypisany już do istniejącego.

```
...

StoreItem resultItem;
if (arg1.IsTemporary) {
  resultItem = arg1;
} else if (arg2.IsTemporary) {
  resultItem = arg2;
} else {
  resultItem = StoreItem.CreateTemporaryVariable(arg1.ItemType);
  resultItem.Parent = arg1;
  asmGenerator.InitializeVariable(resultItem);
}

Store.PushStack(resultItem);
asmGenerator.StoreVariable(resultItem);
```

Algorytm 3.24: Implementacja funkcji obsługującej działania matematyczne cz.3

Aby wykonać działanie matematyczne w kodzie asemblerowym, należy jak już wcześniej było opisywane, załadować dwie wartości na stos i wywołać jedną z następujących instrukcji:

- `add` - dodawanie
- `sub` - odejmowanie
- `mul` - mnożenie
- `div` - dzielenie

Wynik operacji zostanie zapisany na stosie, który następnie można przypisać do zmiennej.

```
// załadowanie liczby stałej o wartości 10
ldc.i4 10
// załadowanie liczby stałej o wartości 15
ldc.i4 15
// wywołanie operacji mnożenia
mul
// zapisanie wyniku do zmiennej "v_tmp_1"
stloc v_tmp_1
```

Algorytm 3.25: Kod assemblera przedstawiający mnożenie dwóch liczb

Natomiast dodawanie do siebie dwóch łańcuchów polega na wywołaniu procedury. Tak jak w przypadku operacji matematycznych liczb, należy załadować na stos dwie wartości łańcuchowe a następnie wywołać procedurę `Concat`.

```
// Załadowanie ciągu "Hello "
ldstr "Hello "
// Załadowanie ciągu "World!"
ldstr "World!"
// Wywołanie procedury łączenia ciągów
call string [mscorlib]System.String::Concat(string,string)
// Zapisanie wyniku do zmiennej "v_tmp_1"
stloc v_tmp_1
```

Algorytm 3.26: Kod assemblera przedstawiający łączenie dwóch tekstów

Kolejną z funkcjonalności w module obsługi działań arytmetycznych jest instrukcja przypisania wartości do zmiennej. W tej operacji bierze udział element źródłowy, z którego pobrana będzie wartość oraz element docelowy, do którego zostanie przypisana wartość.

Funkcja realizująca przypisanie wartości, na początku pobiera ze stosu głównego kompilatora element źródłowy i docelowy. Następnie jest sprawdzany element docelowy, czy jest to element opisujący zmienną należącą do tablicy, jeśli tak wywoływana jest funkcja generująca kod asemblerowy dla tego przypadku.

```
public void Assign(){
  StoreItem source = Store.PopStack();
  StoreItem dist = Store.PopStack();
  if(dist.IsType(StoreItemType.ARRAY_ELEMENT)){
    assignArrayElement(source, dist);
    return;
  }
  asmGenerator.Load(source);
  asmGenerator.RemoveLastDuplicate();

  ...
```

Algorytm 3.27: Implementacja funkcji obsługującej przypisanie wartości cz.1

```
private void assignArrayElement(StoreItem source, StoreItem dist) {
  asmGenerator.LoadVariableWithoutCheck(dist);
  asmGenerator.Load(dist.TableIndex);
  asmGenerator.Load(source);
  asmGenerator.SetElementForList();
  asmGenerator.Comment($"{dist.Print} = {source.Print}\n");
}
```

Algorytm 3.28: Implementacja funkcji pomocniczej, obsługującej przypisanie wartości do elementu tablicowego

Spowodowane jest to tym, że przypisywanie wartości dla tablic, wymaga podania zmiennej przechowującej adres tablicy, następnie indeksu elementu, a na końcu wartości przypisywanej. Następnie można wykonać procedurę przypisania wartości `set_Item`.

```
// Załadowanie zmiennej przechowującej tablicę
ldloc v_myArray
// Załadowanie zmiennej przechowującej indeks elementu tablicy
ldloc v_tmp_2
// Załadowanie wartości 100
ldc.i4 100
// Wywołanie procedury przypisania
```

```
8   callvirt   instance void class
        [mscorlib]System.Collections.Generic.List`1<int32>::set_Item(int32,
        !0)
```

Algorytm 3.29: Kod assemblera przedstawiający przypisanie wartości do elementu tablicowego

W przypadku przypisania wartości dla zwykłej zmiennej, wystarczy załadować na stos element źródłowy, a następnie załadować ze stosu wartość do elementu docelowego. W funkcji obsługującej przypisanie wartości, sprawdzane są dodatkowo informacje odnośnie typów wartości, jak i dopisanie elementów do listy wykorzystywanych zmiennych.

```
1   ...
2
3   if (!dist.IsInitialized) {
4       StoreItemType distOrginalType = dist.ItemType;
5       if(source.IsType(StoreItemType.ARRAY_ELEMENT)){
6         dist.ItemType = StoreItemType.INTEGER;
7       } else {
8         dist.ItemType = source.ItemType;
9       }
10      if(distOrginalType == StoreItemType.UNDEFINED){
11        asmGenerator.InitializeVariableFromUndefined(dist);
12      } else {
13        asmGenerator.InitializeVariable(dist);
14      }
15    }
16    dist.Parent = source.Parent;
17    asmGenerator.StoreVariable(dist);
18    asmGenerator.Comment($"{dist.Print} = {source.Print}\n");
19  }
```

Algorytm 3.30: Implementacja funkcji obsługującej przypisanie wartości cz.2

```
1   // Załadowanie na stos wartości zmiennej "v_tmp_1"
2   ldloc v_tmp_1
3   // Przypisanie wartości ze stosu zmiennej "v_tmp_2"
4   stloc v_tmp_2
```

Algorytm 3.31: Kod assemblera przedstawiający przypisanie wartości

Ostatnim z funkcjonalności implementowanych przez moduł obsługi działań arytmetycznych jest inkrementacja wartości zmiennej. Można to zrealizować poprzez dodanie stałej wartości “1” dla podanego elementu.

```
1   public void IncrementVariable() {
2     StoreItem item = Store.PopStack();
3     StoreItem one = StoreItem.CreateInteger("1");
4     asmGenerator.Load(item);
5     asmGenerator.Load(one);
6     StoreItem addSign = StoreItem.CreateArithmeticSign("+");
7     asmGenerator.ExecuteArithmeticOperation(addSign);
8     asmGenerator.StoreVariable(item);
9     asmGenerator.Comment($"{item.Print}++");
10  }
```

Algorytm 3.32: Implementacja funkcji obsługującej inkrementację wartości

Generowany kod asemblera jest identyczny jak przy zwykłej operacji wykonywania działania matematycznego.

```
1   // Załadowanie na stos wartości zmiennej "v_i"
2   ldloc v_i
3   // Załadowanie na stos stałej wartości "1"
4   ldc.i4 1
5   // wywołanie operacji dodawania
```

```
6   add
7   // Zapisanie wyniku do zmiennej "v_i"
8   stloc v_i
```

Algorytm 3.33: Kod assemblera przedstawiający inkrementacje wartości

Obsługa wyrażeń warunkowych

Moduł obsługi wyrażeń warunkowych odpowiedzialny jest za przetworzenie instrukcji porównania, instrukcję negacji oraz obsługi instrukcji warunkowej. Instrukcja porównania jest bardzo podobna instrukcji działania matematycznego. Uczestniczą w niej dwa elementy reprezentujące wartości oraz jeden element odpowiedzialny za operację porównania.

```
1   public void ProcessCondition() {
2     StoreItem arg2 = Store.PopStack();
3     StoreItem sign = Store.PopStack();
4     StoreItem arg1 = Store.PopStack();
5
6     bool stringOperation = false;
7     if (StoreItem.IsAnyType(StoreItemType.STRING, arg1, arg2)) {
8       if (!Array.Exists(new[] {"==", "===", "!=", "!=="}, x => x ==
            sign.Value) ) {
9         throw new InvalidOperationException($"Operation is not allowed for
              strings.");
10      }
11      stringOperation = true;
12
13      arg1 = variableModule.CastVariable(arg1, StoreItemType.STRING);
14      arg2 = variableModule.CastVariable(arg2, StoreItemType.STRING);
15    }
16
17    ...
```

Algorytm 3.34: Implementacja funkcji obsługującej wyrażenia warunkowe cz.1

Wynik operacji, tak ja to było przy działaniach matematycznych, zapisywany jest do zmiennej tymczasowej i jeśli jeden z elementów już był zmienną tymczasową, zamiast tworzyć nowy element, zostanie ponownie wykorzystany już istniejący.

```
1     ...
2
3     asmGenerator.Load(arg1);
4     asmGenerator.Load(arg2);
5
6     if (stringOperation) {
7       asmGenerator.CompareStrings(sign);
8     } else {
9       asmGenerator.ExecuteConditionOperation(sign);
10    }
11
12    StoreItem resultItem;
13    if (arg1.IsTemporary && arg1.IsType(StoreItemType.BOOLEAN)) {
14      resultItem = arg1;
15    } else if (arg2.IsTemporary && arg2.IsType(StoreItemType.BOOLEAN)) {
16      resultItem = arg2;
17    } else {
18      resultItem = StoreItem.CreateTemporaryVariable(StoreItemType.BOOLEAN);
19      asmGenerator.InitializeVariable(resultItem);
20    }
21    Store.PushStack(resultItem);
22    asmGenerator.StoreVariable(resultItem);
23
24    ...
```

```
25  }
```

Algorytm 3.35: Implementacja funkcji obsługującej wyrażenia warunkowe cz.2

Przy operacjach na łańcuchach znaków jest również wykonywane rzutowanie zmiennych na ciąg łańcuchowy, ponieważ dla tego typu danych należy wywołać procedury `op_Equality` lub `op_Inequality`, które pozwalają na sprawdzenie równości lub nierówności dwóch ciągów.

```
1  // Załadowanie na stos wartości zmiennej "v_tmp_1"
2  ldloc v_tmp_1
3  // Załadowanie na stos stałej wartości wartości "text"
4  ldstr "text"
5  // Wywołanie procedury "op_Equality" sprawdzającą równośc
6  call bool [mscorlib]System.String::op_Equality(string,string)
7  // Zapisanie wyniku do zmiennej "v_tmp_2"
8  stloc v_tmp_2
```

Algorytm 3.36: Kod assemblera przedstawiający porównanie dwóch łańcuchów zanków

Dla porównania wartości liczbowych wykorzystywane są proste instrukcje asemblerowe:

- `cgt` - większy od
- `clt` - mniejszy od
- `ceq` - równy

Dla operacji takich jak: większy równy, mniejszy równy lub nierówny, nie ma odpowiednich dla nich instrukcji w asemblerze. Aby móc je wykonać, należy w kodzie assemblera porównać wynik wywołania powyżej wypisanych instytucji z wartością zerową. Przykładowo, jeśli chcemy wykonać porównanie “większy lub równy”, należy wykonać operację “mniejszy” (`clt`) a następnie wynik porównać z wartością zerową.

```
1  // Załadowanie na stos stałej wartości "5"
2  ldc.i4 5
3  // Załadowanie na stos wartości zmiennej "v_v1"
4  ldloc v_v1
5  // Wywołanie operacji porwónania "większy od"
6  cgt
7  // Załadowanie na stos stałej wartości "0"
8  ldc.i4.0
9  // Wywołanie operacji porwónania "równy"
10 ceq
11 // Zapisanie wyniku do zmiennej "v_tmp_1"
12 stloc v_tmp_1
```

Algorytm 3.37: Kod assemblera przedstawiający porównanie “mnieszy lub równy”

Następnym z elementów modułu jest implementacja funkcji dla instrukcji negacji. Funkcja pobiera ze stosu element, generuje dla niego kod asemblerowy i wynik zapisuje w zmiennej tymczasowej, która z powrotem jest wrzucana na stos.

```
1  public void NotStatement() {
2    StoreItem item = Store.PopStack();
3    asmGenerator.Load(item);
4    asmGenerator.Negation();
5    StoreItem resultItem =
         StoreItem.CreateTemporaryVariable(StoreItemType.BOOLEAN);
6    asmGenerator.InitializeVariable(resultItem);
7    asmGenerator.StoreVariable(resultItem);
8    Store.PushStack(resultItem);
9    asmGenerator.Comment($"{resultItem.Print} = !{item.Print}");
10 }
```

```
}
```

Algorytm 3.38: Implementacja funkcji obsługującej negację wartości

W kodzie asemblerowym, negacja wartości polega na jej przyrównaniu do stałej wartości zerowej. Dzięki takiemu zabiegowi, w przypadku kiedy mamy wartość “true” i porównamy ją do wartości zerowej “false” to otrzymamy jako wynik wartość “false”. W przypadku kiedy jest to wartość “false” to wynikiem będzie wartość “true”.

```
// Załadowanie na stos wartości zmiennej "v_t1"
ldloc v_t1
// Załadowanie na stos stałej wartości "0"
ldc.i4.0
// Porównanie wartości
ceq
// Zapisanie wyniku do zmiennej "v_t2"
stloc v_t2
```

Algorytm 3.39: Kod assemblera przedstawiający porównanie “mnieszy lub równy”

W module obsługi wyrażeń warunkowych, nie może zabraknąć funkcji obsługujących wyrażenie warunkowe `if`. Dla tej instrukcji zostały zdefiniowane trzy reguły: dla konstrukcji wyrażenia, operacji porównania oraz instrukcji `else`.

```
ifStatement
  : IF '(' ifStatementConditionOperation ')' instructionBlock
      elseStatement?
  ;

ifStatementConditionOperation
  : conditionOperation
  ;

elseStatement
  : ELSE instructionBlock
  ;
```

Algorytm 3.40: Gramatyka dla instrukcji `if`

Instrukcje warunkowe opierają się głównie o etykiety oraz skoki warunkowe. W tym celu w klasie `Store` stworzony jest oddzielny stos oraz licznik indexów dla operacji wykorzystujące etykiety.

Pierwszą z funkcji wywoływanych przy przetwarzaniu instrukcji `if`, jest funkcja wywoływana na rozpoczęcie przetwarzania reguły `ifStatement`. Ma ona za zadanie utworzyć index instrukcji i umieścić go na stosie indeksów. Funkcja nie generuje kodu asemblera.

```
public void BeginIfStatement(){
  int ifIndex = Store.NextLabelIndex();
  Store.PushLabelStack(ifIndex);
}
}
```

Algorytm 3.41: Implementacja funkcji wywoływanej przy rozpoczęciu analizy instrukcji `if`

Drugą z kolei jest wywoływana funkcja, która ma za zadanie wygenerować kod asemblera sprawdzający warunek instrukcji `if`. Pobierany jest element z głównego stosu, a następnie wywoływana jest instrukcja skoku warunkowego. Skok ten wykonywany jest do wskazanej etykiety, tylko w przypadku kiedy wartość elementu jest fałszywa. Etykieta generowana jest na podstawie wartości znajdującej się na stosie indeksów.

```
1   public void IfStatementCondition() {
2     StoreItem item = Store.PopStack();
3     asmGenerator.Load(item);
4
5     int ifIndex = Store.TopLabelStack();
6     asmGenerator.JumpIfFalse($"IF_{ifIndex}");
7
8     asmGenerator.Comment($"if ({item.Print} == false) JUMP IF_{ifIndex}");
9   }
10 }
```

Algorytm 3.42: Implementacja funkcji obsługująca warunek instrukcji `if`

Następną z funkcji jest obsługa instrukcji `else`. Jej celem jest utworzenie etykiety w kodzie asemblera dla wcześniej utworzonego skoku, oraz utworzenie kolejnej instrukcji skoku dla sekcji instrukcji `else` przy pomocy nowego indeksu.

```
1   public void ElseStatement() {
2     int elseIndex = Store.NextLabelIndex();
3     int ifIndex = Store.PopLabelStack();
4     asmGenerator.Jump($"IF_{elseIndex}");
5     asmGenerator.CreateLabel($"IF_{ifIndex}");
6     Store.PushLabelStack(elseIndex);
7
8     asmGenerator.Comment($"ELSE");
9   }
```

Algorytm 3.43: Implementacja funkcji obsługująca sekcje `else` instrukcji `if`

Ostatnią z funkcji jest obsługa zakończenia przetwarzania instrukcji `if`. Służy jedynie do utworzenia etykiety kończącej konstrukcję.

```
1   public void EndIfStatement() {
2     int ifIndex = Store.PopLabelStack();
3     asmGenerator.CreateLabel($"IF_{ifIndex}");
4   }
```

Algorytm 3.44: Implementacja funkcji wywoływanej przy zakończeniu analizy instrukcji `if`

```
1   // Załadowanie na stos wartości zmiennej "v_tmp_0"
2   ldloc v_tmp_0
3   // Instrukcja skoku. Jeśli "v_tmp_0" jest wartością "false" zostatnie
        wykonany skok do etykeyty "IF_0"
4   brfalse IF_0
5
6   // Instrukcje bloku warunkowego "if"
7   ...
8
9   // Skok bezwarunkowy do etytkiety "IF_1"
10  br IF_1
11  // Deklaracja etytkiety "IF_0"
12  IF_0:
13
14  // Instrukcje bloku warunkowego "else"
15  ...
16
17  // Deklaracja etytkiety "IF_1"
18  IF_1:
```

Algorytm 3.45: Kod assemblera przedstawiający konstrukcję `if ... else`

Obsługa pętli

Moduł obsługi pętli zawiera szereg funkcji obsługujących instrukcje `for` oraz `while`. Definicja reguł gramatyki dla instrukcji `while` zawarta jest jedynie w dwóch regułach. Pierwsza jest ogólną konstrukcją instrukcji, a druga służy do obsłużenia warunku pętli.

```
whileLoop
  : WHILE '(' whileStatementConditionOperation ')' instructionBlock
  ;

whileStatementConditionOperation
  : conditionOperation
  ;
```

Algorytm 3.46: Gramatyka dla instrukcji `while`

Dla powyższej gramatyki wykorzystywane są trzy funkcji parsera. Pierwszą z nich jest funkcja wywoływana przed analizą reguły. Jej zadaniem jest utworzenie etykiety początku pętli.

```
public void BeginWhile() {
  int labelIndex = Store.NextLabelIndex();
  Store.PushLabelStack(labelIndex);
  asmGenerator.CreateLabel($"WL_{labelIndex}");
}
```

Algorytm 3.47: Implementacja funkcji obsługująca rozpoczęcie analizy instrukcji `while`

Kolejną z wywoływanych funkcji jest obsługa wyrażenia warunkowego. Pobierana jest ze stosu kompilatora element wyniku wyrażenia warunkowego, a następnie generowany jest dla niego skok warunkowy do etykiety kończącej instrukcję `while`.

```
public void CheckWhileCondition() {
  StoreItem item = Store.PopStack();
  asmGenerator.Load(item);
  int labelIndex = Store.TopLabelStack();
  asmGenerator.JumpIfFalse($"WL_EXIT_{labelIndex}");
}
```

Algorytm 3.48: Implementacja funkcji obsługująca wyrażenie warunkowe instrukcji `while`

Ostatnią z funkcji przeznaczonych dla instrukcji `while` w tym module, jest funkcja wywoływana na zakończenie analizy reguły. Generuje ona skok bezwarunkowy do etykiety na początku pętli, oraz tworzy etykietę kończącą pętlę `while`.

```
public void EndWhile() {
  int labelIndex = Store.PopLabelStack();
  asmGenerator.Jump($"WL_{labelIndex}");
  asmGenerator.CreateLabel($"WL_EXIT_{labelIndex}");
}
```

Algorytm 3.49: Implementacja funkcji obsługująca kończącej analize instrukcji `while`

```
// Utworzenie etykiety początku pętli
WL_0:

// Instrukcja warunkowa "x < 10"
ldloc v_x
ldc.i4 10
clt
stloc v_tmp_0

// Sprawdzenie warunku i wykonanie skoku do etykiety końca pętli
ldloc v_tmp_0
brfalse WL_EXIT_0

// Blok instukcji
...

// Skok bezwarunkowy do etykiety początku pętli
br WL_0
```

```
19  // Utworzenie etykiety końca pętli
20  WL_EXIT_0:
```

Algorytm 3.50: Kod assemblera przedstawiający konstrukcję `while`

Definicja reguł gramatyki dla instrukcji `for` jest bardziej złożona. Dla tej instrukcji, poza sprawdzeniem wyrażenia warunkowego, występuje również inicjalizacja wartości zmiennych oraz wyrażenie inkrementacji wartości.

```
1  forLoop
2    : FOR '(' forAssignSection? ';' forStatementConditionOperation ';'
         forExpression? ')' instructionBlock
3    ;
4
5  forAssignSection
6    : assignOperation (COMMA+ assignOperation)*
7    ;
8
9  forStatementConditionOperation
10   : conditionOperation
11   ;
12
13 forExpression
14   : forExpressionAssign
15   | incrementVariable
16   ;
17
18 forExpressionAssign
19   : identifierValue ASSIGN assignValue
20   ;
```

Algorytm 3.51: Gramatyka dla instrukcji `while`

Funkcje rozpoczynająca oraz zakańczająca analizę instrukcji `for` jest taka sama. Różnią się jedynie nazwą generowanych etykiet. Funkcja sprawdzająca warunek, też jest bardzo podobna. Posiada jedynie dodatkowo skok bezwarunkowy pomijający kod wyrażenia inkrementacji.

```
1   public void BeginForExpression() {
2     int labelIndex = Store.TopLabelStack();
3     asmGenerator.CreateLabel($"FL_EX_{labelIndex}");
4   }
5
6   public void EndForExpression () {
7     int labelIndex = Store.TopLabelStack();
8     asmGenerator.Jump($"FL_{labelIndex}");
9     asmGenerator.CreateLabel($"FL_BODY_{labelIndex}");
10  }
```

Algorytm 3.52: Implementacja funkcji obsługująca wyrażenie inkrementacji

Funkcje obsługujące wyrażenie inkrementacji tworzą jedynie etykiety oraz skok bezwarunkowy do początku pętli. Wyrażenie inicjalizacji jest przetwarzane przed rozpoczęciem generowania kodu dla pętli.

```
1  // Inicjalizacja wartości zmiennej "index"
2  ldc.i4 0
3  stloc v_index
4
5  // Utworzenie etykiety początku pętli
6  FL_0:
7
8  // Instrukcja warunkowa "index < 10"
9  ldloc v_index
10 ldc.i4 10
```

```
11    clt
12    stloc v_tmp_0
13
14    // Sprawdzenie warunku i wykonanie skoku do etykiety końca pętli
15    ldloc v_tmp_0
16    brfalse FL_EXIT_0
17    br FL_BODY_0
18
19    // Utworzenie etykiety początku wyrażenia inkrementacji
20    FL_EX_0:
21
22    // Instrukcje inkrementacji
23    ...
24
25    // Skok bezwarunkowy do poczatku pętli
26    br FL_0
27    // Utworzenie etykiety początku ciała pętli
28    FL_BODY_0:
29
30    // Blok instukcji
31    ...
32
33    // Skok bezwarunkowy do wyrażenia inkrementacji
34    br FL_EX_0
35    // Utworzenie etykiety końca pętli
36    FL_EXIT_0:
```

Algorytm 3.53: Kod assemblera przedstawiający konstrukcję `while`

Obsługa tablic

Moduł obsługi tablic zawiera implementacje funkcji obsługujących deklarowanie tablic, tworzenie elementów tablicowych, dodawanie elementów do tablicy oraz obsługa funkcji pobierającą ilość elementów znajdujących się w podanej tablicy.

Funkcjonalność tablic została uproszczona do przechowywania jedynie elementów liczb całkowitych. Realizowane jest to przy wykorzystaniu generycznej klasy `List` typu `int` wbudowanej biblioteki środowiska .NET.

Dla deklaracji tablicy elementów zostały wykorzystane trzy reguły pozwalające na zdeklarowanie tablicy o dowolnej ilości elementów przy pomocy nawiasów kwadratowych.

```
1  arrayLiteral
2    : (OPEN_BRACKET elementList CLOSE_BRACKET)
3    ;
4
5  elementList
6    : COMMA* arrayElement? (COMMA+ arrayElement)* COMMA*
7    ;
8
9  arrayElement
10   : singleExpression
11   ;
```

Algorytm 3.54: Definicja gramatyki dla deklaracji tablicy z wartościami

Funkcja obsługująca deklaracje tablic tworzy zmienną tymczasową dla której tworzona jest instancja listy elementów. Następnie element opisujący zmienną jest wrzucany na główny stos. Kolejnym krokiem przy deklaracji tablic jest dodanie elementów do tablicy. Dla każdej z liczb podanych w nawiasach kwadratowych w kodzie źródłowym, generowany jest kod wywołujący funkcję `Add`.

```
1    public void CreateTempArray() {
```

```
    StoreItem array =
        StoreItem.CreateTemporaryVariable(StoreItemType.ARRAY);
    asmGenerator.CtorVariable(array);
    asmGenerator.Comment($"CREATE ARRAY {array.Value}");
    Store.PushStack(array);
  }

  public void AddElementToArray() {
    StoreItem item = Store.PopStack();
    StoreItem array = Store.TopStack();
    asmGenerator.Load(item);
    asmGenerator.AddElementToList(array);
  }
```

Algorytm 3.55: Implementacja funkcji obsługujących deklaracje tablicy

```
// Utworzenie nowej istancji klasy "List" typu "int32"
newobj instance void class
    [mscorlib]System.Collections.Generic.List`1<int32>::.ctor()
stloc v_tmp_0
ldloc v_tmp_0

// Załadowanie na stos liczby "10"
ldc.i4 10
// Dodanie liczby do listy
callvirt   instance void class
    [mscorlib]System.Collections.Generic.List`1<int32>::Add(!0)
ldloc v_tmp_0
```

Algorytm 3.56: Kod assemblera przedstawiający deklarację tablicy

Kolejną z funkcji w module obsługującym tablice, jest funkcja obsługująca tworzenie zmiennych będących elementami tablicy. Funkcja ta nie generuje kodu asemblerowego, a jedynie tworzy element, który będzie obsługiwany w innych modułach.

```
public void CreateTableVariable(string value) {
  StoreItem index = Store.PopStack();
  variableModule.CreateVariable(value);
  StoreItem array = Store.PopStack().Clone();
  array.TableIndex= index;
  array.ItemType = StoreItemType.ARRAY_ELEMENT;
  asmGenerator.Comment($"{array.Value}[{index.Value}]");
  Store.PushStack(array);
}
```

Algorytm 3.57: Implementacja funkcji obsługująca wystąpienie elementu tablicowego

Ostania z funkcji w tym module jest obsługa instrukcji pobrania ilości elementów tablicy. Początkowo jest tworzona zmienna tymczasowa która ma przechować wynik operacji. Następnie generowany jest kod asemblera, który pobiera długość listy przy pomocy funkcji `get_Count`, a następnie zapisuje wartość we wcześniej utworzonej zmiennej. Wynik jest wrzucany na stos główny aplikacji.

```
public void GetArrayLength(string arrayName) {
  StoreItem array = Store.GetVariableIfExist(arrayName);
  if(!array.IsInitialized){
    throw new InvalidOperationException($"Variable {array.Print} is
        undfined");
  }
  if(array.IsNotType(StoreItemType.ARRAY, StoreItemType.FUNCTION_ARG)){
    throw new InvalidOperationException($"Variable {array.Print} must be
        an array");
  }
  StoreItem arrLen =
      StoreItem.CreateTemporaryVariable(StoreItemType.INTEGER);
```

```
10      asmGenerator.InitializeVariable(arrLen);
11      asmGenerator.Load(array);
12      asmGenerator.GetListSize();
13      asmGenerator.StoreVariable(arrLen);
14      asmGenerator.Comment($"{arrLen.Print} = {array.Print}:LENGTH");
15      Store.PushStack(arrLen);
16  }
```

Algorytm 3.58: Implementacja funkcji obsługująca pobieranie ilości elementów tablicy

```
1  // Załadowanie zmiennej tablicowej
2  ldloc v_myArray
3  // Pobranie ilości elementów tablicy
4  callvirt   instance int32 class
       [mscorlib]System.Collections.Generic.List`1<int32>::get_Count()
5  // Zapisanie wyniku do zmiennej tymczasowej
6  stloc v_tmp_1
```

Algorytm 3.59: Kod assemblera przedstawiający pobieranie ilości elementów tablicy

Obsługa funkcji

Ostatnim z modułów odpowiedzialny jest za obsługę funkcji. Zawiera obsługę deklaracji funkcji, interpretowanie parametrów funkcji, wywoływanie oraz obsługę wartości zwracanej z funkcji. Deklaracja funkcji jest polega na utworzeniu osobnej listy instrukcji w generatorze asemblera. Dodatkowo klasa `Store` jest przestawiany na kontekst obsługi instrukcji wewnątrz funkcji. Kontekst jest przywracany po zakończeniu obsługi instrukcji wewnątrz funkcji.

```
1  public void BeginFunctionDeclaration(string functionName){
2    asmGenerator.Comment($"FUNCTION {functionName}");
3    asmGenerator.EmptyLine();
4    asmGenerator.CreateFunction(functionName);
5  }
6
7  public void EndFunctionDeclaration(string functionName) {
8    asmGenerator.EndFunction();
9  }
```

Algorytm 3.60: Implementacja funkcji do deklaracji funkcji

```
1  public void CreateFunction(string name) {
2    FunctionAsmLines functionAsmLines = new FunctionAsmLines(name);
3    if(asmFunctionsLines.ContainsKey(name)){
4      throw new ArgumentException($"Function `{name}` already exist.");
5    }
6    asmFunctionsLines.Add(name, functionAsmLines);
7    Store.SetFunction(name);
8  }
```

Algorytm 3.61: Implementacja funkcji w klasie `Store` do deklaracji funkcji

Po deklaracji funkcji następuje interpretacja parametrów tej funkcji. Dla każdego z parametrów tworzony jest element opisujący dany parametr. Parametry mają przydzielany specjalny typ `StoreItemType.FUNCTION_ARG`, dzięki któremu podczas generowania kodu asemblera bloku instrukcji funkcji, będzie można zostawić znaczniki które zostaną podmienione w postprocesingu. Jest to konieczne, ponieważ typy parametrów, podczas analizy bloku instrukcji funkcji nie są znane do momentu wywołania danej funkcji.

```
1  public void AddParameter(string name){
2    StoreItem param = StoreItem.CreateVariable(name);
```

```
    param.IsFunctionParam = true;
    param.IsInitialized = true;
    param.ItemType = StoreItemType.FUNCTION_ARG;
    Store.AddVariable(param);
  }
```

Algorytm 3.62: Implementacja funkcji obsługującej parametry deklarowanej funkcji

Kolejną z funkcji w tym module jest funkcja odpowiedzialna za obsługę wywołania instrukcji `return`. Funkcja pobiera element ze stosu głównego, a następnie zapisuje informacje dla danej funkcji, że ten element jest wartością zwracaną funkcji. Generowany kod to jedynie instrukcja załadowania zmiennej na stos oraz instrukcja powrotu.

```
  public void ProcessReturn() {
    StoreItem item = Store.PopStack();
    Store.SetFunctionReturnValue(item);
    asmGenerator.Load(item);
    asmGenerator.Return();
  }
```

Algorytm 3.63: Implementacja funkcji obsługującej wywołanie `return`

```
  // Załadowanie zmiennej "v_returnValue"
  ldloc v_returnValue
  // Wywołanie instrukcji powrotu
  ret
```

Algorytm 3.64: Kod assemblera przedstawiający zwracanie elementu

Ostatnią z obsługiwanych funkcjonalności w module jest obsługa wywoływania wcześniej deklarowanych funkcji. Gramatyka zawiera jedynie trzy reguły, które kolejno definiują wywołanie funkcji, listę argumentów funkcji oraz pojedynczy argument. Przy analizie argumentów funkcji, wywoływane są metody do wyzerowania licznika argumentów, a następnie zliczana jest ilość podanych argumentów. Dzięki temu można określić ile należy pobrać elementów ze stosu aplikacji.

```
functionCall
  : IDENTIFIER arguments
  ;

arguments
  : '('(argument (',' argument)* ','?)?')'
  ;

argument
  : assignValue
  ;
```

Algorytm 3.65: Definicja gramatyki dla wywołania funkcji z arguemntami

```
  public static void FunctionCallInitialize() {
    FunctionCallArgCounterStack.Push(0);
  }

  public static void FunctionCallArgumentIncrement(){
    int value = FunctionCallArgCounterStack.Pop();
    FunctionCallArgCounterStack.Push(value+1);
  }
```

Algorytm 3.66: Implementacja funkcji w klasie `Store` do zliczania argumentów funkcji

Po zakończeniu analizy argumentów wywoływana jest metoda do analizy i wygenerowania kodu asemblerowego dla wywołania podanej funkcji. Na początku są pobierane

ze stosu głównego wszystkie argumenty podane przy wywołaniu funkcji i zapisywane do stosu tymczasowego, w celu odwrócenia kolejności argumentów.

```
public void CallFunction(string name){
  int argCount = Store.FunctionCallArgumentPop();
  Stack<StoreItem> tmpStack = new Stack<StoreItem>();
  List<StoreItem> args = new List<StoreItem>();
  for(int i = 0; i< argCount; i++){
    tmpStack.Push(Store.PopStack());
  }

  ...
```

Algorytm 3.67: Implementacja funkcji obsługującej wywołanie funkcji cz.1

Następnie dla każdego z argumentów jest generowany kod asemblera ładujący je do stosu aplikacji. Dodatkowo zapisywana jest informacja w klasie `Store` o pozycji danego argumentu. Następnie generowany jest kod wywołania danej funkcji z wykorzystaniem listy argumentów. Przy wywoływaniu funkcji wymagane jest podanie typów podanych argumentów.

```
  ...

  for(int i = 0; i< argCount; i++){
    StoreItem elem = tmpStack.Pop();
    args.Add(elem);
    Store.SetFunctionParamType(name, elem, i);
    asmGenerator.Load(elem);
  }
  asmGenerator.Comment($"FUNC CALL {name}");
  asmGenerator.CallFunction(name, args);

  ...
```

Algorytm 3.68: Implementacja funkcji obsługującej wywołanie funkcji cz.2

Ostatnią z części funkcji obsługującej wywołanie funkcji, jest obsługa wartości zwracanej z tej funkcji. W przypadku kiedy funkcja zwraca jakąś wartość, to tworzona jest zmienna tymczasowa, która przechowa wynik wywołania funkcji. Zmienna jest inicjalizowana oraz wrzucana na główny stos aplikacji.

```
  ...

  FunctionStore functionStore = Store.Functions[name];
  functionStore.IsUsed = true;
  if(!functionStore.IsReturnVoid) {
    StoreItem rootItem = functionStore.ReturnValue.RootItem;
    if(null == rootItem){
      rootItem = functionStore.ReturnValue;
    }
    StoreItem retItem =
        StoreItem.CreateTemporaryVariable(rootItem.ItemType);
    retItem.Parent = functionStore.ReturnValue;
    asmGenerator.InitializeVariable(retItem);
    asmGenerator.StoreVariable(retItem);
    Store.PushStack(retItem);
  }
}
```

Algorytm 3.69: Implementacja funkcji obsługującej wywołanie funkcji cz.3

```
// Załadowanie stałej "4.7"
ldc.r4 4.7
// Wywołanie funkcji "f_convert" z jednym argumentem typu "float32"
```

```
4   call string Program.Program::f_convert(float32)
5   // Zapisanie wartości zwracanej do zmiennej "v_tmp_1"
6   stloc v_tmp_1
```

Algorytm 3.70: Kod assemblera przedstawiający wywołanie funkcji

3.2.4 Generator asemblera

Klasą odpowiedzialną za generowanie assemblera jest klasa `AsmGenerator`, która wykorzystywana jest w każdym z modułów przetwarzających kod źródłowy. Instancja generatora jest tworzona przy starcie aplikacji kompilatora, następnie przez całe działanie kompilatora zbiera generowane linie kodu asemblera, a po zakończeniu przetwarzania pliku źródłowego, tworzony jest plik wynikowy. Generator posiada następujące elementy:

- `fileName` - nazwę pliku źródłowego
- `tabSize` - wielkość wcięcia kodu
- `asmLines` - lista instrukcji funkcji głównej
- `asmFunctionsLines` - słownik z listą instrukcji deklarowanych funkcji

Generowany kod asemblerowy dla instrukcji wywoływanych bez deklaracji funkcji zapisywany jest do listy `asmLines`. Dla instrukcji wywoływanych wewnątrz deklarowanych funkcji, zapisywany jest razem z nazwą funkcji w słowniku `asmFunctionsLines`. Dzięki temu będzie można w prostszy sposób wygenerować osobny kod dla każdej z funkcji.

Generowanie pliku wynikowego zaczyna się od jego utworzenia pod tą samą nazwą co plik źródłowy. Następnie generowany jest nagłówek pliku oraz deklaracja funkcji głównej aplikacji. Dla funkcji głównej tworzone są wszystkie zmienne wykorzystywane w instrukcjach, a następnie przepisywana jest zawartość listy `asmLines`. Po przepisaniu wszystkich instrukcji funkcja główna jest zamykana. Na końcu tworzone są zdeklarowane funkcje.

```
1  private void createAsmFile() {
2    StreamWriter outFile = new StreamWriter(Path.Combine(DIST_DIR,
         $"{this.fileName}.il"));
3    this.writeFileHeader(outFile);
4    this.initializeAllVariables(outFile, Store.Variables);
5    foreach(string line in this.asmLines){
6      outFile.WriteLine(line);
7    }
8    this.writeMainFunctionFooter(outFile);
9    this.createAsmFunctions(outFile);
10   this.writeFileFooter(outFile);
11   outFile.Dispose();
12 }
```

Algorytm 3.71: Implementacja funkcji generująca plik wynikowy

W nagłówku przy pomocy instrukcji `.assembly` importowana jest standardowa biblioteka `mscorlib` zawierająca wszystkie podstawowe funkcjonalności platformy .NET. Tą samą instrukcją definiowana jest nazwa asemblerowa pliku oraz w klamrach podawane są dodatkowe informacje o pliku wynikowym. Kolejną z instrukcji nagłówka jest `.module` która mówi jak nazywać się będzie moduł wynikowy.

```
1  .assembly extern mscorlib {}
2  .assembly Program {}
3  .module Program.exe
```

Algorytm 3.72: Kod nagłówka pliku wygenerowanego pliku asemblera

Następnie generowana jest klasa główna programu przy pomocy instrukcji `.class` oraz zawarta w niej funkcja główna `Main`. Funkcja główna deklarowana jest instrukcją `.method` jako funkcja statyczna nie zwracająca żadnych wartości. Funkcja przyjmuje jako parametr listę elementów typu `string`, która jest argumentami przyjmowanymi przy uruchomieniu aplikacji. Na końcu deklaracji znajduje się jeszcze słowa kluczowe `cil managed`, które oznaczają, że dana funkcja zawiera kod zarządzalny. Wewnątrz funkcji głównej znajduje się również instrukcja `.entrypoint`, która wskazuje miejsce z którego ma rozpocząć się wykonywanie kodu po uruchomieniu aplikacji.

```
.class Program.Program
extends [mscorlib]System.Object
{
 .method static void Main(string[] args)
 cil managed
 {
  .entrypoint

  ...
```

Algorytm 3.73: Kod deklaracji funkcji głównej

Po wygenerowaniu nagłówka oraz deklaracji głównej funkcji, generowany jest kod inicjalizujący wszystkie używane zmienne w funkcji `Main`. Inicjalizacja realizowana jest przy pomocy instrukcji `.locals init()`, dla której jako argument podaje się typy oraz nazwy zmiennych.

```
...
.locals init(
    int32 v_a,
    int32 v_b,
    int32 v_c,
    int32 v_tmp_1,
    string v_tmp_2)
...
```

Algorytm 3.74: Kod deklaracji zmiennych lokalnych funkcji głównej

Po zakończeniu generowania funkcji głównej, w podobny sposób generowane są funkcje tworzone przez użytkownika. Do tego również używana jest instrukcja `.method` dla funkcji statycznej. W przypadku kiedy użytkownik nadał wartość zwracaną oraz argumenty funkcji, to używane są odpowiednie typy dla tych elementów. Tak jak w przypadku funkcji głównej, na samym początku również są inicjalizowane zmienne lokalne funkcji przy pomocy instrukcji `.locals init()`, a następnie przepisywane są wcześniej wygenerowane instrukcje. Podczas przepisywania instrukcji jest wykonywany postprocesing.

```
.method static int32 f_mul(
  int32 v_a,
  int32 v_b) cil managed
{
.locals init(
      int32 v_sum,
      int32 v_tmp_0)
...
```

Algorytm 3.75: Kod deklaracji przykładowej funkcji

4. Testy

Podczas procesu tworzenia kompilatora, były również tworzone programy testujące dla poszczególnych funkcjonalności, w celu zweryfikowania prawidłowego działania programu. Dla każdego z modułów został utworzony program testujący w języku JavaScript, który obejmuje zakres funkcjonalności modułu. Zostały również utworzone tożsame programy w języku C# w celu porównania do kodu języka JavaScript.

Zostały również wykorzystane dwa gotowe programy napisane w JavaScript odnalezione w Internecie. Również dla tych programów zostały utworzone odpowiedniki w języku C# oraz zostały stworzone programy w języku JScript. Dla programów zostały przeprowadzone dodatkowe testy: został zmierzony czas wykonywania programu, zużycie pamięci oraz wielkości pliku wynikowego.

4.1 Testy modułów

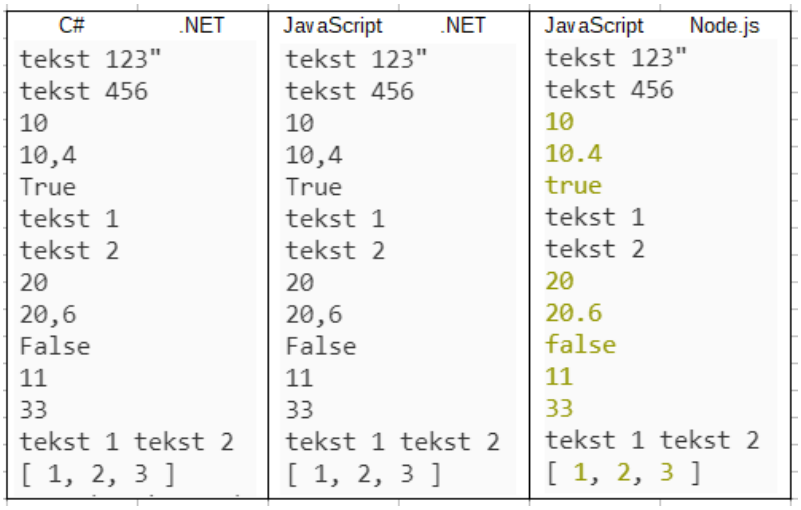
Programy testujące moduły, tworzone były równocześnie z kompilatorem, w celu sprawdzenia, poprawności działania poszczególnych funkcjonalności. Były one modyfikowane i przystosowywane w taki sposób, aby zweryfikować w jak największym stopniu poprawność generowanego kodu i wykluczenia pojawiających się błędów. Testy modułów obejmują planowany zakres implementacji kompilatora.

4.1.1 Porównanie wyniku wykonania programów

Pierwszym z testów został wykonany program testujący wyświetlanie elementów na ekranie. Program wyświetla różne wartości różnych typów, takie jak wartości tekstowe w cudzysłowu podwójnym jak i pojedynczym, wartości liczbowe całkowite i rzeczywiste, wartości logiczne oraz tablice wartości. Na załączonym rysunku 4.1 przedstawione są zrzuty ekranu wyniku działania programów w różnych wariantach.

Jak można zauważyć, wynik programu w języku C# oraz JavaScript kompilowanych na wspólną platformę .NET są identyczne. Tutaj należy również wspomnieć, że standardowa biblioteka C# nie posiada, żadnej z funkcji pozwalającej na wyświetlenie listy elementów tablicy przy pomocy jednej instrukcji. Tak jak podczas implementacji kompilatora JavaScript została zastosowana tutaj funkcja łącząca elementy tablicy w jeden ciąg znaków oraz zostały doklejone nawiasy otwierające oraz zamykające.

Porównując wynik programu uruchomionego na platformie .NET z programem uruchomionym na platformie Node.js, można zauważyć nie wielkie różnice w wyświetlanych wynikach. Pierwszą z nich rzucającą się w oczy, jest zmiana koloru wyświetlanego tekstu dla liczb oraz wartości liczbowych. Jednak ten efekt zależny jest od formatowania kolorów



C# .NET	JavaScript .NET	JavaScript Node.js
tekst 123"	tekst 123"	tekst 123"
tekst 456	tekst 456	tekst 456
10	10	10
10,4	10,4	10.4
True	True	true
tekst 1	tekst 1	tekst 1
tekst 2	tekst 2	tekst 2
20	20	20
20,6	20,6	20.6
False	False	false
11	11	11
33	33	33
tekst 1 tekst 2	tekst 1 tekst 2	tekst 1 tekst 2
[1, 2, 3]	[1, 2, 3]	[1, 2, 3]

Rysunek 4.1: Wyniki wykonania programu testowego dla modułu obsługi standardowego wyjścia. Źródło: własne

w danej konsoli. Konsola wykryła wywołanie platformy Node.js, dzięki czemu zmieniła kolorystykę.

Drugim z różniącym się elementem jest sposób wyświetlania liczb rzeczywistych. Różnią się tym, że dla platformy .NET wyświetlany jest przecinek, kiedy na platformie Node.js wyświetlana jest kropka. Ostatnią różnicą w prezentowanych wynikach jest wyświetlanie wartości logicznych. Dla platformy .NET wartości są wyświetlane z wielką literą, a w przypadku Node.js wartości wyświetlane są małymi literami.

Kolejnym z testów dla którego wynik wykonania programu był różny, został przeprowadzony dla modułu działań arytmetycznych. Zrzuty wyników zostały przedstawione na rysunku 4.2. Głównie różnice występują przy wyświetlaniu wyniku operacji dzielenia.

Pierwszą z różnic między programem napisanym w języku C# a JavaScript na platformie .NET jest wynik dzielenia dwóch liczb całkowitych. Wykonywane działanie ma postać $y = 10/15$, więc wynikiem jest liczba rzeczywista. W programie w języku C# jest językiem silnie typowanym, co oznacza, że jeżeli przynajmniej jedna z liczb nie zostanie przekonwertowana na typ liczb rzeczywistych, to wynik będzie typu liczby całkowitej. W tworzonym kompilatorze została zaimplementowana automatyczna konwersja typów, w momencie kiedy zostanie wykryta operacja dzielenia dwóch liczb.

Drugą z widocznych różnic jest precyzja wartości zmiennych. W tworzonym kompilatorze JavaScript został wykorzystany typ pojedynczej precyzji, a w przypadku kompilatora C# została zastosowana typ podwójnej precyzji. Porównując wyniki platformy .NET do platformy Node.js można zauważyć, że wartości na platformie .NET są zaokrąglane.

Przy uruchomieniu programu JScript jednego z algorytmów testujących, ukazała się jeszcze jedna różnica w prezentowanych na konsoli wynikach. Różnicą jest sposób wyświetlania tablicy elementów. W tworzonym kompilatorze, jak i na platformie .NET, elementy są otoczone nawiasem kwadratowym, oraz oddzielone są poza przecinkiem dodatkową spacją. W przypadku wyniku programu JScript, prezentowana tablica nie posiada nawiasów kwadratowych oraz liczby są oddzielone przecinkami bez spacji. Wynik programów zaprezentowany jest na rysunki 4.3

C# .NET	JavaScript .NET	JavaScript Node.js
-2	-2	-2
25	25	25
-5	-5	-5
150	150	150
0	0,6666667	0.6666666666666666
125	125	125
200	200	200
260	260	260
Test concat	Test concat	Test concat
1 concat	1 concat	1 concat
Test 8	Test 8	Test 8
22,4	22,4	22.4
4,5	4,5	4.5
23,1	23,1	23.1
47,8095238095238	47,80953	47.80952380952381
125,2	125,2	125.19999999999999
191,42	191,42	191.42000000000002
130,235389221557	130,2354	130.2353892215569
Real 7,4	Real 7,4	Real 7.4
4,7 real.	4,7 real.	4.7 real.
1	1	1
z2	z2	z2
z3	z3	z3
2	2	2
9,5	9,5	9.5

Rysunek 4.2: Wyniki wykonania programu testowego dla modułu obsługi działania arytmetyczne. Źródło: własne

JavaScript .NET	JScript .NET
[-75, 1, 2, 3, 4, 5, 6]	-75,1,2,3,4,5,6

Rysunek 4.3: Wyniki wykonania programu dla algorytmu testowego nr 1. Źródło: własne

Skrypty testowe dla pozostałych modułów nie wykazują różnic w prezentowanych wynikach na konsoli.

4.1.2 Porównanie generowanego kodu assemblera

Generowany kod assemblera z języka JavaScript będzie porównany z kodem dezasemblowanym kodem programu napisanego w języku C#. Dezasemblacja jest wykonywana przy pomocy programu *ildasm.exe* znajdujący się w pakiecie **.NET Framework**.

Pierwszą z widocznych różnic jest ilość deklaracji metadanych w nagłówku pliku. Kolejną z różnic jest ilość modyfikatorów przy deklaracji klasy jak i metody `Main`. Dla deklaracji klasy programu C# zostały użyte następujące słowa kluczowe `private`, `auto`, `ansi`, `beforefieldinit`, a dla funkcji `Main` zostały użyte: `private` oraz `hidebysig`.

Kolejnym z różnic jest ilość deklarowanych zmiennych, która spowodowana jest słabą optymalizacją w tworzonym kompilatorze. Następną rzeczą jest nadawanie etykiet dla każdej z instrukcji w obrębie deklarowanych funkcji.

Kod asemblera programu napisanego w JavaScript

```
.class Program.Program
extends [mscorlib]System.Object
{
  .method static void Main(string[] args)
  cil managed
  {
```

Kod asemblera programu napisanego w C#

```
.class private auto ansi beforefieldinit dotnet.Program
        extends [mscorlib]System.Object
{
  .method private hidebysig static void Main(string[] args)
  cil managed
  {
```

Rysunek 4.4: Porównanie kodu asemblera dla deklaracji klasy oraz funkcji `Main`. Źródło: własne

Analizując generowane instrukcje kodu asemblera z dezasemblowanym kodem programu C# można zauważyć różnicę przy zapisywaniu oraz odczytywaniu wartości zmiennych na stosie. W utworzonym kompilatorze odbywa się to zawsze poprzez nazwę zmiennej, a w .NET Framework wykorzystywane są instrukcje wykorzystujące indeksowanie zmiennych od wartości 0 do 3. Dodatkowo odwołanie się do kolejnych zmiennych wykorzystywana jest instrukcja w formie skróconej.

```
1
2   ...
3   IL_0047:  ldloc.0
4   IL_0048:  call       void [mscorlib]System.Console::WriteLine(string)
5   IL_004d:  ldloc.1
6   IL_004e:  call       void [mscorlib]System.Console::WriteLine(string)
7   IL_0053:  ldloc.2
8   IL_0054:  call       void [mscorlib]System.Console::WriteLine(int32)
9   IL_0059:  ldloc.3
10  IL_005a:  call       void [mscorlib]System.Console::WriteLine(float32)
11  IL_005f:  ldloc.s    V_4
12  IL_0061:  call       void [mscorlib]System.Console::WriteLine(bool)
13
14  ...
```

Algorytm 4.1: Fragment kodu deasemblerowanego testu programu C#, przedstawiający ładowanie wartości zmiennych na stos

Następne różnice generowanego kodu widoczne są przy operacjach arytmetycznych, wykonywanych na liczbach stałych. W przypadku wykorzystania *.NET Framework*, obliczenia wykonywane są przy generowaniu kodu, a nie podczas uruchomienia skompilowanego programu. Przykładowo wykonanie działania $x = 1 + 2$, wygeneruje instrukcje jedynie do przypisania wartości 3 do zmiennej x. W tworzonym kompilatorze, wygenerowanie zostanie instrukcji do załadowania liczby 1 oraz 2 na stos, następnie wykonanie operacji dodawania i na końcu przypisanie wyniku do zmiennej x.

Przy dodawaniu do siebie wielu łańcuchów znaków, w kodzie assemblera, programu napisanego w języku C#, można zauważyć, że wszystkie łańcuchy są łączone przy pomocy

jednej instrukcji `Concat`, gdzie w tworzonym kompilatorze, zawsze łączone są jedynie dwa łańcuchy na raz. Co więcej, można zauważyć też różnicę przy konwersji wartości zmiennych na łańcuchy znaków przy dodawaniu ich do elementów typu łańcuchowego. W tworzonym kompilatorze konwersja przebiega przy pomocy instrukcji `ToString()`, jednak w kompilatorze środowiska *.NET Framework* przeprowadzana jest przy pomocy instrukcji `box`, która sprowadza zmienną do typu `object`. Instrukcja `Concat` przyjmuje wtedy elementy typu `object`.

```
...
IL_006a:  ldloc.s   V_1
IL_006c:  ldc.i4.2
IL_006d:  box       [mscorlib]System.Int32
IL_0072:  ldstr     "Tekst"
IL_0077:  call      string [mscorlib]System.String::Concat(object,
                                                           object,
                                                           object)

...
```

Algorytm 4.2: Fragment kodu deasemblerowanego testu programu C#, przedstawiający łączenie łańcuchów znaków

Dalej analizując poszczególne moduły, można zauważyć, że w generowanym kodzie assemblera dla programów kompilowanych przy pomocy narzędzi środowiska *.NET Framework*, naturalnie wykorzystywane są wszystkie dostępne instrukcje. Szczególnie widoczne jest to przy instrukcjach warunkowych. W tworzonym kompilatorze warunki zawsze są sprowadzane do pojedynczej wartości logicznej, a następnie na jej podstawie wykonywany jest skok. W przypadku kompilatora *.NET*, przykładowo wykorzystywane są instrukcje które od razu porównują dwie wartości na jej podstawie wykonują skok.

```
...
IL_0060:  ldc.i4.s  20
IL_0062:  ldloc.s   number1
IL_0064:  ble.s     IL_0072

IL_0066:  ldstr     "number1 is less than 20"
IL_006b:  call      void [mscorlib]System.Console::WriteLine(string)
IL_0070:  br.s      IL_007c

IL_0072:  ldstr     "number1 is greater than 20"
IL_0077:  call      void [mscorlib]System.Console::WriteLine(string)
IL_007c:

...
```

Algorytm 4.3: Fragment kodu deasemblerowanego testu programu C#, przedstawiający instrukcje `if ... else`

```
...
ldc.i4 20
ldloc v_number1
cgt
stloc v_tmp_11
// BOOLEAN:tmp_11 = INTEGER:20 > INTEGER:number1

ldloc v_tmp_11
brfalse IF_2
// if (BOOLEAN:tmp_11 == false) JUMP IF_2
```

```
12    ldstr "number1 is less than 20"
13    call void [mscorlib]System.Console::WriteLine(string)
14
15    br IF_3
16    IF_2:
17    // ELSE
18    ldstr "number1 is greater than 20"
19    call void [mscorlib]System.Console::WriteLine(string)
20
21    ...
```

Algorytm 4.4: Fragment wygenerowanego kodu asemblerowanego z języka JavaScript, przedstawiający instrukcje `if ... else`

4.2 Testy algorytmów

W celu przetestowania działania tworzonego kompilatora, zostały wykorzystane algorytmy odnalezione w Internecie. Zostaną dla nich utworzone odpowiednie kody w języku C# w celu porównania szybkości działania, zużycia pamięci czy wykorzystania przestrzeni dyskowej. Dla kodu programu w JavaScript zostaną również utworzone wersje przystosowane do kompilatora JScript, którego wyniki również zostaną porównane.

4.2.1 Algorytm sortowania

Pierwszym z odnalezionych kodów jest algorytm sortowania bąbelkowego zamieszczony w artykule zamieszczonego pod adresem `https://www.educba.com/sorting-algorithms-in-javascript/`, którego autorem jest Priya Pedamkar. Kod przedstawia się następująco:

```
1  function swap(arr, firstIndex, secondIndex){
2    var temp = arr[firstIndex];
3    arr[firstIndex] = arr[secondIndex];
4    arr[secondIndex] = temp;
5  }
6  function bubbleSortAlgo(arraaytest){
7    var len = arraaytest.length,
8    i, j, stop;
9    for (i=0; i < len; i++){
10     for (j=0, stop=len-i; j < stop; j++){
11       if (arraaytest[j] > arraaytest[j+1]){
12         swap(arraaytest, j, j+1);
13       }
14     }
15   }
16   return arraaytest;
17 }
18 console.log(bubbleSortAlgo([3, 6, 2, 5, -75, 4, 1]));
```

Algorytm 4.5: Algorytm sortowania bąbelkowego. Źródło: `https://www.educba.com/sorting-algorithms-in-javascript/`

Niestety funkcjonalność obsługi tablic nie została w pełni zaimplementowana, co powoduje, że powyższy kod dla stworzonego kompilatora nie jest poprawny. Problem występuje w dwóch miejscach. Pierwszy z nich znajduje się w wewnętrznej instrukcji `for` a dokładniej w warunku kończącym iterowanie pętli. W języku JavaScript przy odwołaniu się do elementu poza zakresem tablicy, nie zostanie wywołany błąd a jedynie zostanie zwrócona wartość typu `undefined`. W środowisku *.NET Framework* wyjście

poza zakres powoduje błąd i zatrzymanie działania programu. W celu uniknięcia tego, zmienna `stop` została zmniejszona o 1.

Drugim miejscem jest wywołanie funkcji, podając dla niej bezpośrednio tablicę wartości. Problem polega na braku obsłużenia przedstawionej sytuacji, w której przekazywane jest bezpośrednio nowo utworzona tablica elementów, co skutkuje wygenerowaniem błędnego kodu asemblera, którego po skompilowaniu do postaci binarnej, uruchomienie powoduje błąd środowiska maszyny wirtualnej. Aby uniknąć tego błędu, nowo tworzona tablica będzie najpierw przypisana do zmiennej, a dopiero następnie przekazana do funkcji.

```
function swap(arr, firstIndex, secondIndex){
  var temp = arr[firstIndex];
  arr[firstIndex] = arr[secondIndex];
  arr[secondIndex] = temp;
}
function bubbleSortAlgo(arraaytest){
  var len = arraaytest.length,
  i, j, stop;
  for (i=0; i < len; i++){
    for (j=0, stop=len - i - 1; j < stop; j++){
      if (arraaytest[j] > arraaytest[j+1]){
        swap(arraaytest, j, j+1);
      }
    }
  }
  return arraaytest;
}
var l = [3, 6, 2, 5, -75, 4, 1];
console.log(bubbleSortAlgo(l));
```

Algorytm 4.6: Zmodyfikowany kod programu sortowania bąbelkowego.

W celu skompilowania powyższego kodu algorytmu przy pomocy kompilatora JScript z pakietu *.NET Framework*, wystarczy jedynie dodać instrukcje importujące biblioteki standardowe `System` oraz `Microsoft.Win32`. Instrukcje należy umieścić na początku pliku. Kolejną konieczną rzeczą do skompilowania kodu, jest zmiana instrukcji standardowego wyjścia. Instrukcja `console.log(...)` powinna być zastąpiona instrukcją `Console.WriteLine(...)`. Tak przygotowany kod źródłowy można teraz skompilować kompilatorem JScript.

Kod JavaScript został odwzorowany również w kodzie C# i prezentuje się następująco:

```
using System;
using System.Collections.Generic;

namespace dotnet
{
  class Program
  {
    static void swap(List<int> arr, int firstIndex, int secondIndex)
    {
      int temp = arr[firstIndex];
      arr[firstIndex] = arr[secondIndex];
      arr[secondIndex] = temp;
    }

    static List<int> bubbleSortAlgo(List<int> arraaytest)
    {
      int len = arraaytest.Count, i, j, stop;
      for (i = 0; i < len; i++)
      {
        for (j = 0, stop = len - i - 1; j < stop; j++)
        {
```

```
22              if (arraaytest[j] > arraaytest[j + 1])
23              {
24                swap(arraaytest, j, j + 1);
25              }
26            }
27          }
28          return arraaytest;
29        }
30
31        static void Main(string[] args)
32        {
33          List<int> l = new List<int>() { 3, 6, 2, 5, -75, 4, 1 };
34          Console.WriteLine("[ "+string.Join(", ", bubbleSortAlgo(l))+" ]");
35        }
36      }
37    }
```

Algorytm 4.7: Odwzorowany kod JavaScript w języku C# algorytmu sortowania bąbelkowego.

Testy zostaną wykonane dla oryginalnego wektora wejściowego do przesortowania jak i dla losowo wygenerowanego wektora o wielkościach 1000, 5000 i 10000 składającego się z liczb całkowitych z przedziału od -10000 do 10000.

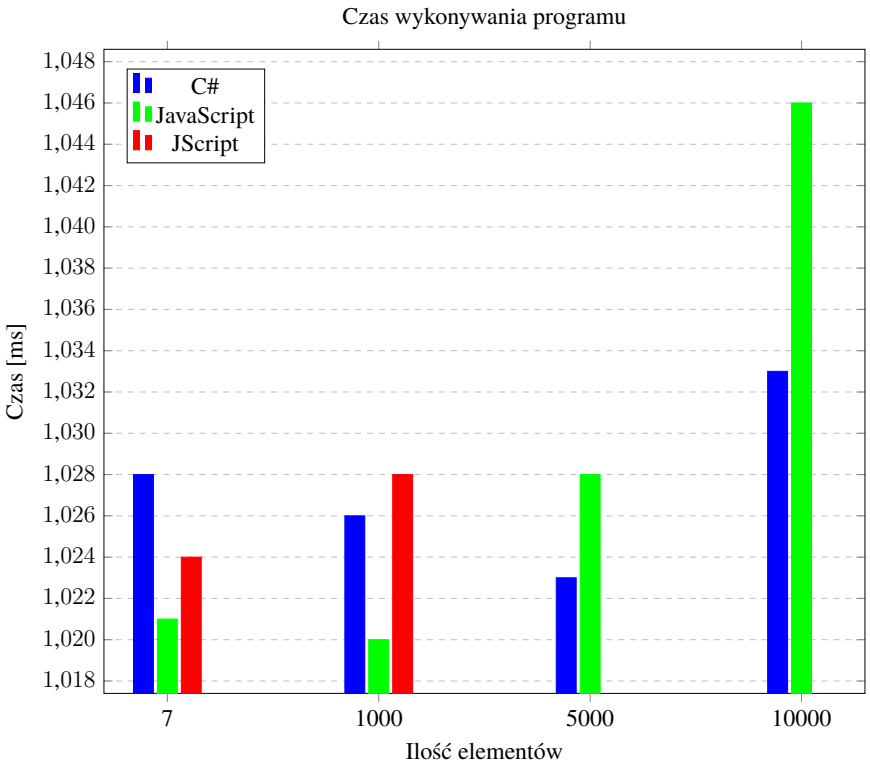
Pierwszym z kryteriów jest zmierzenie czasu wykonywania programu. Czas zmierzony będzie w w konsoli *PowerShell 7.1.1* przy pomocy komendy `Measure-Command` uruchamiającego proces skompilowanego pliku programu. Otrzymane wyniki prezentują się następująco:

Ilość elementów	C#	JavaScript	JScript
Oryginalne (7)	00:00:01.0280485	00:00:01.0211699	00:00:01.0249265
1000	00:00:01.0267120	00:00:01.0209711	00:00:01.0281652
5000	00:00:01.0231339	00:00:01.0286513	00:00:11.1320074
10000	00:00:01.0331534	00:00:01.0466795	00:00:47.3928596

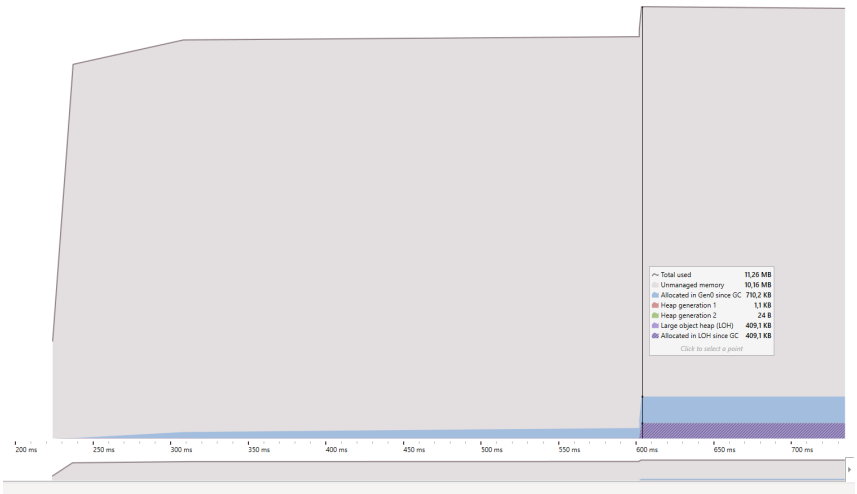
Obserwując wyniki pomiarów czasu wykonywania programu, można stwierdzić, że utworzony kompilator jest porównywalny do kompilatora C#. W przypadku kompilatora JScript różnicę można zauważyć przy większej ilości danych. W przypadku kiedy wektor testowy miał 5000 elementów, czas wykonywania programu zwiększył się średnio o 10 sekund, a w przypadku kiedy wektor testowy posiadał 10000 elementów, czas wykonania osiągną średnio 47 sekund.

Kolejnym z kryteriów jest sprawdzenie wykorzystania pamięci dla programów testowych. Zostanie do tego wykorzystane narzędzie *JetBrains dotMemory 2020.2.1*. Test zostanie wykonany dla programów z wektorem wejściowym posiadającym 10000.

Kompilator	Zużycie pamięci
C#	11,26 MB
JavaScript	11,25 MB
JScript	28,72 MB



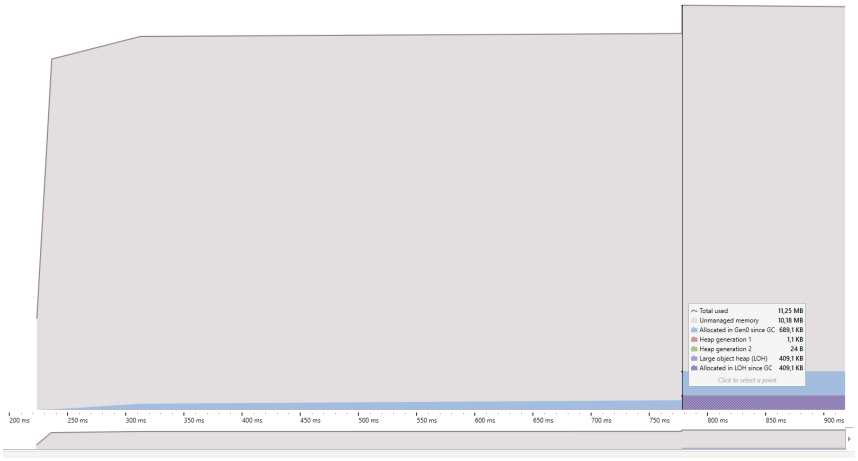
Rysunek 4.5: Pomiar czasu wykonywania programów dla algorytmu pierwszego.



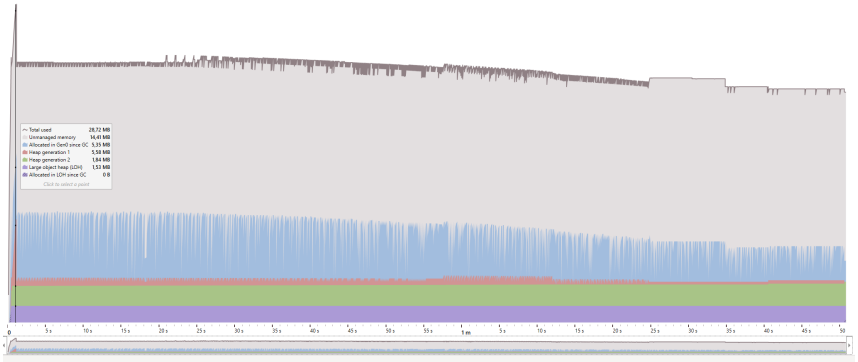
Rysunek 4.6: Pomiar zużycia pamięci dla algorytmu pierwszego dla kompilatora C#. Źródło: własne

Tak ja w poprzednim teście, wyniki pokazują, że pomiędzy programem C# oraz programem JavaScript nie występuje znaczna różnica. Inaczej jest z programem JScript który zużywa w piku ponad dwukrotnie więcej pamięci.

Ostatnim z kryteriów jest zmierzenie wielkości plików skompilowanych programów. Wyniki prezentują się następująco:



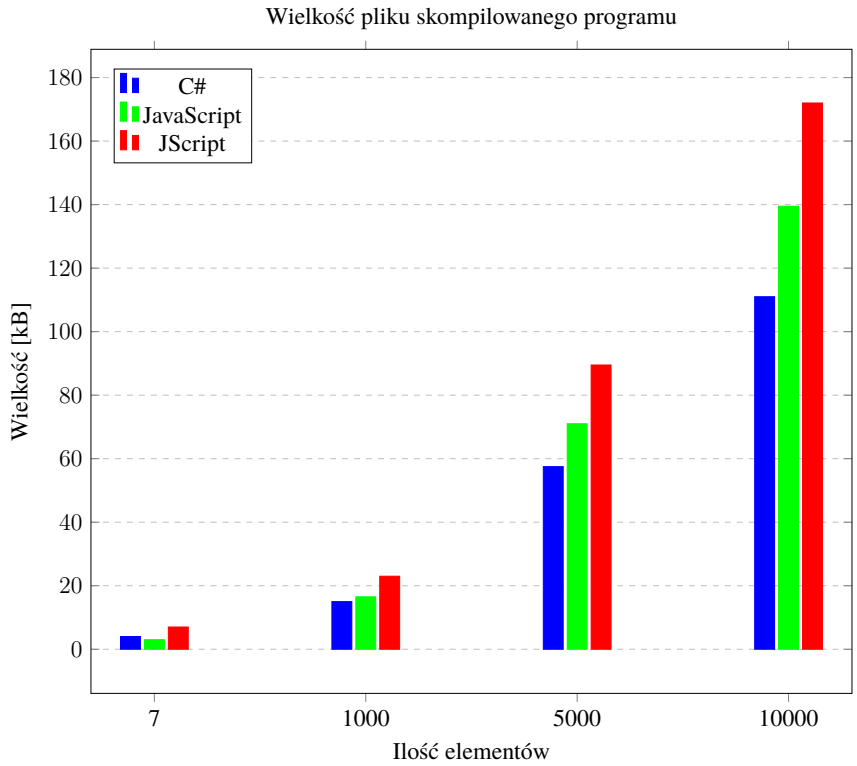
Rysunek 4.7: Pomiar zużycia pamięci dla algorytmu pierwszego dla kompilatora JavaScript. Źródło: własne



Rysunek 4.8: Pomiar zużycia pamięci dla algorytmu pierwszego dla kompilatora JScript. Źródło: własne

Ilość elementów	C#	JavaScript	JScript
Oryginalne (7)	4,00 kB	3,00 kB	7,00 kB
1000	15,00 kB	16,50 kB	23,00 kB
5000	57,50 kB	71,00 kB	89,50 kB
10000	111,00 kB	139,50 kB	172,00 kB

Pomiar wielkości plików wskazuje, że stworzony kompilator generuje więcej instrukcji dla programu w porównaniu do kompilatora C#. Spowodowane jest to słabą optymalizacją generowanego kodu asemblera dla instrukcji. Porównując wielkość programu tworzonego kompilatora do kompilatora JScript, można zauważyć, że wielkość tworzonego pliku wykonywalnego jest mniejsza.



Rysunek 4.9: Pomiar wielkości plików programów dla algorytmu pierwszego.

4.2.2 Algorytm przeszukiwania

Drugim odnalezionym kodem jest algorytm liniowego przeszukiwania tablicy elementów zamieszczonego pod adresem `https://www.scriptonitejs.com/javascript-searching-algorithms/`. Kod przedstawia się następująco:

```
var items = [2, 5, 3, 7, 8, 10, 15, 18, 24, 111, 12, 19, 87];

function itemSearch(item) {
    for(var i=0; i < items.length; i++) {
        if(items[i] == item) {
            console.log("Found item " + item + " at index " + i);
            return true;
        }
    }
    //item not found
    return false;
}

var item = itemSearch(15);
if(!item) {
    console.log('Item does not exist!');
}
```

Algorytm 4.8: Algorytm przeszukiwania. Źródło: `https://www.scriptonitejs.com/javascript-searching-algorithms/`

Niestety i ten program musiał zostać lekko zmodyfikowany. Modyfikacja polega na przekazaniu wektora testowego `items` jako argument w funkcji `itemSearch`. Spowodowane jest to nienajlepsza implementacją zakresu widoczności zmiennych kompilatora. W języku `JavaScript` jest możliwe odwołanie się do zmiennej zadeklarowanej poza

funkcją. Po modyfikacji program prezentuje się następująco:

```
var items = [2, 5, 3, 7, 8, 10, 15, 18, 24, 111, 12, 19, 87];

function itemSearch(items, item) {
    for(var i=0; i < items.length; i++) {
        if(items[i] === item) {
            console.log("Found item " + item + " at index " + i);
            return true;
        }
    }
    //item not found
    return false;
}

var item = itemSearch(items, 15);
if(!item) {
    console.log('Item does not exist!');
}
```

Algorytm 4.9: Zmodyfikowany kod programu przeszukiwania.

Dla powyższego kodu został stworzony odpowiednik dla kompilatora JScript oraz kod został odwzorowany w języku C#, który prezentuje się następująco:

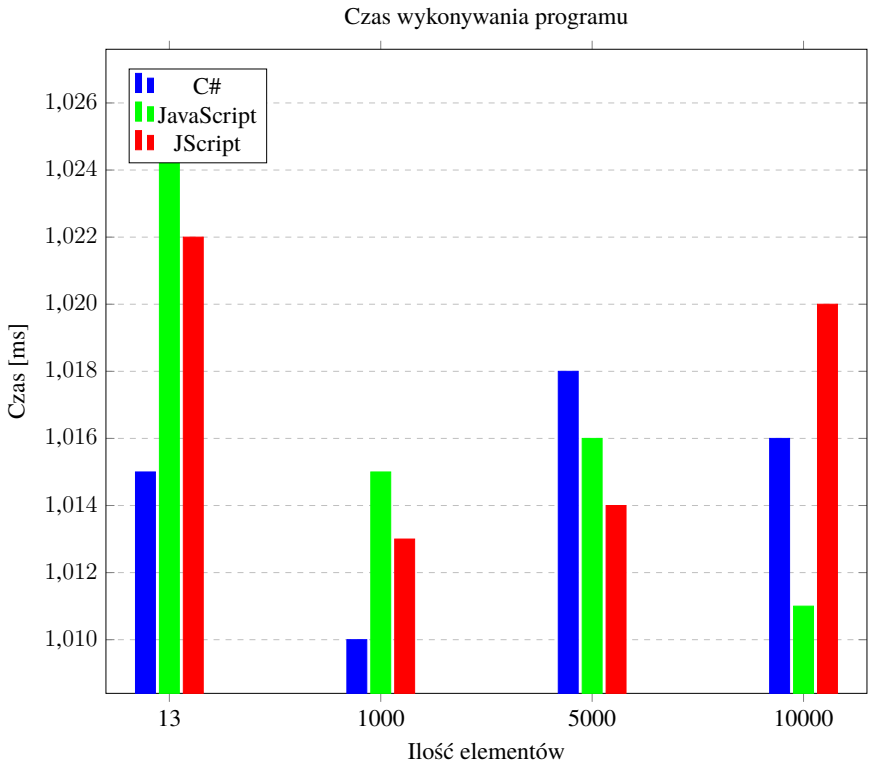
```
using System;
using System.Collections.Generic;

namespace dotnet
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> items = new List<int>() { 2, 5, 3, 7, 8, 10, 15, 18,
                24, 111, 12, 19, 87 };
            bool item = itemSearch(items, 15);
            if(!item) {
                Console.WriteLine("Item does not exist!");
            }
        }

        static bool itemSearch(List<int> items, int item){
            for (int i = 0; i < items.Count; i++)
            {
                if(items[i] == item){
                    Console.WriteLine("Found item " + item + " at index " +
                        i);
                    return true;
                }
            }
            return false;
        }
    }
}
```

Algorytm 4.10: Odwzorowany kod JavaScript w języku C# algorytmu przeszukiwania.

Tak jak dla algorytmu pierwszego, testy zostaną wykonane dla oryginalnego wektora wejściowego oraz dla losowo wygenerowanego wektora o wielkościach 1000, 5000 i 10000 składającego się z liczb całkowitych z przedziału od -10000 do 10000. Wyszukiwany element będzie znajdował się w połowie wektora testowego.

Algorytm został zbadany pod względem takich samych kryteriów jak algorytm pierwszy. Pierwszym z nich jest zmierzenie czasu wykonywania programu. Wyniki prezentują się następująco:

Ilość elementów	C#	JavaScript	JScript
Oryginalne(13)	00:00:01.0156735	00:00:01.0262087	00:00:01.0222548
1000	00:00:01.0101172	00:00:01.0150919	00:00:01.0134737
5000	00:00:01.0183013	00:00:01.0163112	00:00:01.0146355
10000	00:00:01.0167586	00:00:01.0116767	00:00:01.0203511



Rysunek 4.10: Pomiar czasu wykonywania programów dla algorytmu drugiego.

Wyniki pomiarów czasu wykonywania programu, wskazują nie wielkie różnice czasu pomiędzy kompilatorami jak i pomiędzy różnymi ilościami danych wejściowych.

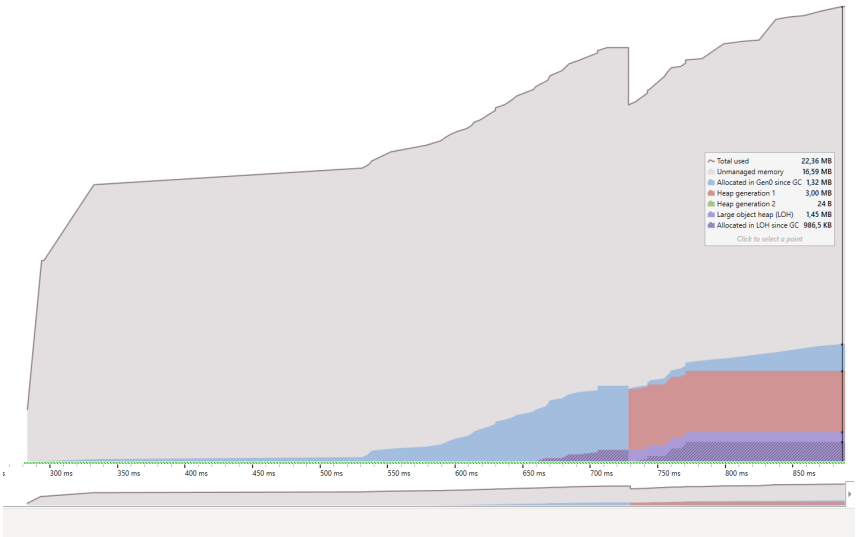
Drugim z kryteriów jest zbadanie wykorzystania pamięci przez programy. Test zostanie wykonany dla programów z wektorem wejściowym posiadającym 10000, oraz element poszukiwany będzie znajdować się w połowie zakresu.

Kompilator	Zużycie pamięci
C#	10,52 MB
JavaScript	10,53 MB
JScript	22,36 MB

Wyniki testu są podobne do wyników testu algorytmu pierwszego. Zużycie pamięci przez program C# jak i JavaScript jest prawie identyczne. Różnica występuje dla programu JScript, którego zużycie jest ponad dwukrotnie większe.

Rysunek 4.11: Pomiar zużycia pamięci dla algorytmu drugiego dla kompilatora C#. Źródło: własne

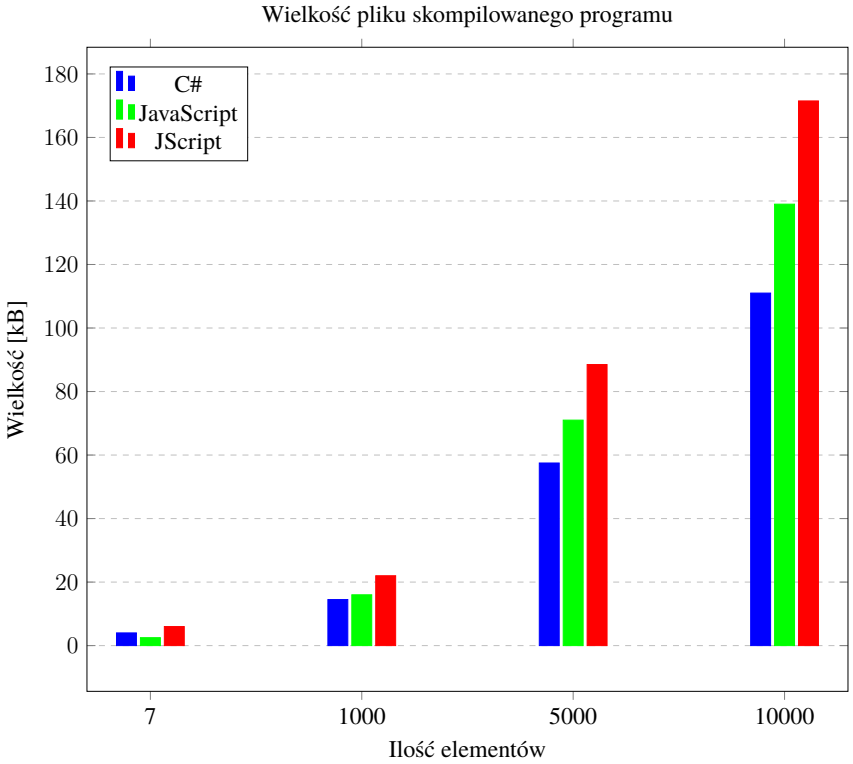
Rysunek 4.12: Pomiar zużycia pamięci dla algorytmu drugiego dla kompilatora JavaScript. Źródło: własne



Rysunek 4.13: Pomiar zużycia pamięci dla algorytmu drugiego dla kompilatora JScript. Źródło: własne

Ostatnim z testów jest porównanie zajmowanej przestrzeni dyskowej dla skompilowanych programów. Wyniki prezentują się w następujący sposób:

Ilość elementów	C#	JavaScript	JScript
Oryginalne(13)	4,00 kB	2,50 kB	6,00 kB
1000	14,50 kB	16,00 kB	22,00 kB
5000	57,50 kB	71,00 kB	88,50 kB
10000	111,00 kB	139,00 kB	171,50 kB



Rysunek 4.14: Pomiar wielkości plików programów dla algorytmu pierwszego.

Wyniki pomiarów są podobne jak w przypadku pomiarów algorytmu pierwszego. Wraz z zwiększeniem ilości elementów wektora wejściowego, pliki wykazują coraz większą różnicę zajmowanego miejsca na przestrzeni dyskowej.

Podsumowanie

Podsumowanie pracy powinno na maksymalnie dwóch stronach przedstawić główne wyniki pracy dyplomowej. Struktura zakończenia to:

1. Przypomnienie celu i hipotez
2. Co w pracy wykonano by cel osiągnąć (analiza, projekt, oprogramowanie, badania eksperymentalne)
3. Omówienie głównych wyników pracy
4. Jak wyniki wzbogacają dziedzinę
5. Zamknięcie np. poprzez wskazanie dalszych kierunków badań.

Spis literatury

Książki

[1] *Engineering a Compiler*. Elsevier, 2012. ISBN: 9780120884780. DOI: 10.1016/C2009-0-27982-7. URL: https://linkinghub.elsevier.com/retrieve/pii/C20090279827.

[2] Russ Ferguson. *Beginning JavaScript*. 2019. ISBN: 978-1-4842-4395-4. DOI: 10.1007/978-1-4842-4395-4.

[3] Marijn Haverbeke. *Eloquent JavaScript: A Modern Introduction to Programming*. 2011. ISBN: 1593272820. DOI: 10.1190/1.9781560801597.

Artykuły

[4] J.E. Smith i Ravi Nair. "The architecture of virtual machines". W: *Computer* 38.5 (maj 2005), s. 32–38. ISSN: 0018-9162. DOI: 10.1109/MC.2005.173. URL: http://ieeexplore.ieee.org/document/1430629/.

[5] Stefan Tilkov i Steve Vinoski. "Node.js: Using JavaScript to Build High-Performance Network Programs". W: *IEEE Internet Computing* 14.6 (list. 2010), s. 80–83. ISSN: 1089-7801. DOI: 10.1109/MIC.2010.145. URL: http://ieeexplore.ieee.org/document/5617064/.

Źródła internetowe i inne

[6] HOW TO ASP.NET. *.NET FCL (Framework Class Library)*. (dostępny Lipiec 6, 2020). URL: https://www.howtoasp.net/net-fcl-framework-class-library/.

[7] MDN contributors. *About JavaScript*. Maj 2020 (dostępny Maj 28, 2020). URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.

[8] MDN contributors. *Common Type System & Common Language Specification*. Czer. 2016 (dostępny Lipiec 6, 2020). URL: https://docs.microsoft.com/en-us/dotnet/standard/common-type-system.

[9] MDN contributors. *Common Language Runtime (CLR) overview*. Kw. 2019 (dostępny Lipiec 6, 2020). URL: https://docs.microsoft.com/pl-pl/dotnet/standard/clr.

[10] Stack Overflow contributors. *Lexer rules in v4*. (dostępny Maj 30, 2021). URL: https://sodocumentation.net/antlr/topic/3271/lexer-rules-in-v4.

[11] Sameers Javed. *Introduction to IL Assembly Language*. Kw. 2003 (dostępny Lipiec 8, 2020). URL: https://www.codeproject.com/Articles/3778/Introduction-to-IL-Assembly-Language.

[12] Paweł Łukasiewicz. *.NET Core vs .NET Framework*. (dostępny Lipiec 6, 2020). URL: https://www.plukasiewicz.net/Artykuly/NetFrameworkVsNetCore.

[13] Node.js. *About | Node.js*. 2017.