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 9

1 Pojęcia i technologie 11

1.1 Zakres projektu 11

1.1.1 Kompilator 11

1.1.2 Maszyna wirtualna 12

1.1.3 JavaScript 12

1.1.4 Node.js 13

1.1.5 .NET Core 13

1.1.6 IL Assembler 14

1.2 Technologie pokrewne 14

1.2.1 ActionScript 14

1.2.2 JScript 15

1.2.3 TypeScript 15

1.2.4 ?CoffeeScript 15

1.2.5 Babel 15

1.2.6 Deno 15

1.2.7 asm.js 15

1.2.8 Emscripten 15

1.2.9 WebAssembly 15

1.2.10 C# 15

1.2.11 .NET Framework 15

1.2.12 Mono 15

1.2.13 DotGNU 15

1.2.14 ?Roslyn 16

2 Projekt kompilatora 17

2.1 Środowisko i narzędzia 17

2.2 Analiza języka JavaScript i określenie zakresu implementacji 17

2.2.1 Wyrażenia 17

2.2.2 Komentarze 18

2.2.3 Deklaracje zmiennych i stałych 18

2.2.4 Typy danych . . . 18
2.2.5 Operacje arytmetyczne . . . 19
2.2.6 Operacje porównania . . . 19
2.2.7 Instrukcje warunkowe . . . 19
2.2.8 Pętle . . . 20
2.2.9 Tablice . . . 22
2.2.10 Funkcje . . . 23
2.2.11 Zakres implementacji projektu . . . 23
2.3 Parser . . . 24
2.4 Struktura projektu . . . 24

3 Implementacja aplikacji kompilatora . . . 27
3.1 Parser . . . 27
3.1.1 Analiza leksykalna . . . 27
3.1.2 Gramatyka . . . 29
3.2 Kompilator . . . 30
3.2.1 Wywołania parsera . . . 31
3.2.2 Moduły . . . 31

4 Testy . . . 33
4.1 Proste operacje matematyczne . . . 33
4.2 Kolejność wykonywania działań . . . 33
4.3 Wyrażenia warunkowe . . . 33
4.4 Tablice . . . 33
4.5 Obiekty . . . 33
4.6 Klasy . . . 33
4.7 Funkcje . . . 33
4.8 Algorytm 1 . . . 34
4.8.1 Opracowanie pseudokodu algorytmu 1 . . . 34
4.8.2 Implementacja algorytmu 1 . . . 34
4.8.3 Testy algorytmu 1 . . . 34
4.9 Algorytm 2 . . . 34
4.9.1 Opracowanie pseudokodu algorytmu 2 . . . 34
4.9.2 Implementacja algorytmu 2 . . . 34
4.9.3 Testy algorytmu 2 . . . 34

Podsumowanie . . . 35

Spis literatury . . . 37
Książki . . . 37
Artykuły . . . 37

Źródła internetowe i inne 37

A Dodatek 39

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 "/"

```
var v1 = 10 + 15
var v2 = 100 + 100 / 2 * 3 + 10
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.

```
var v1 = 2 == 2
var v2 = 1 + 1 != 2
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
```

```
5  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.

```
1  var tablica = []
2  tablica[0] = 'element1'
3  tablica[1] = 2
4  tablica[3] = true
5  var len = tablica.length
6  // tablica zawiera 4 elementy
7  // 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.

```
1  function square(number) {
2    return number * number;
3  }
```

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.

```
1  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 obej-

mować 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`

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

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;

options {
  tokenVocab=JavaScriptLexer;
}
```

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.

```
booleanValue
: BOOLEAN
;

constantValue
: booleanValue
| stringValue
| numberValue
;
```

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.

```
/// <summary>
/// Enter a parse tree produced by <see
    cref="JavaScriptParser.constantValue"/>.
/// <para>The default implementation does
    nothing.</para>
/// </summary>
/// <param name="context">The parse tree.</param>
public virtual void EnterConstantValue([NotNull]
    JavaScriptParser.ConstantValueContext context) { }
/// <summary>
/// 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 nazwie 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.

```
1   static void Main(string[] args) {
2     if (args.Length < 1) {
3       throw new ArgumentException("Needs file name as
            argument");
4     }
5
6     using (AsmGenerator.Create(args[0])) {
7       using (FileStream fs = File.OpenRead(args[0])) {
8         AntlrInputStream imputStream = new
              AntlrInputStream(fs);
9         JavaScriptLexer lexer = new
              JavaScriptLexer(imputStream);
10        CommonTokenStream commonTokenStream = new
              CommonTokenStream(lexer);
11        JavaScriptParser parser = new
              JavaScriptParser(commonTokenStream);
12        parser.BuildParseTree = true;
13        JavaScriptParser.ParseContext context =
              parser.parse();
14
15        JavaScriptListner listner = new
              JavaScriptListner();
```

```
16            ParseTreeWalker walker = new ParseTreeWalker();
17            walker.Walk(listner, context);
18          }
19        }
20        Console.WriteLine("Done.");
21    }
```

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.

```
1   public override void
        ExitIfStatement(JavaScriptParser.IfStatementContext
        context)
2   {
3     conditionModule.EndIfStatement();
4   }
```

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.

```
1   public override void
        ExitIdentifierValue(JavaScriptParser.IdentifierValueContext
        context) {
2     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 Moduły

4. Testy

Opis zakresu testów i jak będą przebiegać. Każdy z poniższych testów będzie sprawdzany pod kątem poprawności wykonywania działań, czasu wykonywania w porównaniu do wykonania kodu na Node.js (dla bardziej złożonych testów z czasem będzie więcej), zajętość pamięci (również tylko przy tych których to ma sens) oraz porównaniu kodu z asemblerowego wygenerowanego za pośrednictwem języka C#.

4.1 Proste operacje matematyczne

Test dodawania, odejmowania, mnożenia, dzielenia, przypisywania.

4.2 Kolejność wykonywania działań

Test na bardziej złożonych wyrażeniach. Sprawdzenie poprawności działania nawiasów oraz kolejności wykonywania działań.

4.3 Wyrażenia warunkowe

Test wyrażeń warunkowych. Kolejność wykonywania operacji and i or.

4.4 Tablice

Test obsługi tablic jedno i wielowymiarowych.

4.5 Obiekty

Test obsługi obiektów.

4.6 Klasy

Jeśli będzie implementacja. Test działania obiektów klas.

4.7 Funkcje

Test działania funkcji. 1. Funkcja "void"bez parametrów. 2. Funkcja "voidź parametrami. 3. Funkcja zwracająca różne typy (proste, tablice, obiekty) bez parametrów. 4. Funkcje zwracająca różne typy z parametrami. 5. inne

4.8 Algorytm 1

4.8.1 Opracowanie pseudokodu algorytmu 1

4.8.2 Implementacja algorytmu 1

4.8.3 Testy algorytmu 1

4.9 Algorytm 2

4.9.1 Opracowanie pseudokodu algorytmu 2

4.9.2 Implementacja algorytmu 2

4.9.3 Testy algorytmu 2

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.

A. Dodatek

W dodatkach mogą znaleźć się większe fragmenty kodów źródłowych, instrukcje działania programów, specyfikacje opcji, z którymi wywołuje się program, większe dane tabelaryczne, specyfikacje oprogramowania, dłuższe opisy teoretyczne, itp.

What is Lorem Ipsum? Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry'es standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Why do we use it? It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using Lorem Ipsum is that it has a more-or-less normal distribution of letters, as opposed to using 'Content here, content here', making it look like readable English. Many desktop publishing packages and web page editors now use Lorem Ipsum as their default model text, and a search for 'lorem ipsum' will uncover many web sites still in their infancy. Various versions have evolved over the years, sometimes by accident, sometimes on purpose (injected humour and the like).

Where does it come from? Contrary to popular belief, Lorem Ipsum is not simply random text. It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum"(The Extremes of Good and Evil) by Cicero, written in 45 BC. This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, Łorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

The standard chunk of Lorem Ipsum used since the 1500s is reproduced below for those interested. Sections 1.10.32 an

What is Lorem Ipsum? Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic

typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Why do we use it? It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using Lorem Ipsum is that it has a more-or-less normal distribution of letters, as opposed to using 'Content here, content here', making it look like readable English. Many desktop publishing packages and web page editors now use Lorem Ipsum as their default model text, and a search for 'lorem ipsum' will uncover many web sites still in their infancy. Various versions have evolved over the years, sometimes by accident, sometimes on purpose (injected humour and the like).

Where does it come from? Contrary to popular belief, Lorem Ipsum is not simply random text. It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum"(The Extremes of Good and Evil) by Cicero, written in 45 BC. This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, Łorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

The standard chunk of Lorem Ipsum used since the 1500s is reproduced below for those interested. Sections 1.10.32 anWhat is Lorem Ipsum? Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Why do we use it? It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using Lorem Ipsum is that it has a more-or-less normal distribution of letters, as opposed to using 'Content here, content here', making it look like readable English. Many desktop publishing packages and web page editors now use Lorem Ipsum as their default model text, and a search for 'lorem ipsum' will uncover many web sites still in their infancy. Various versions have evolved over the years, sometimes by accident, sometimes on purpose (injected humour and the like).

Where does it come from? Contrary to popular belief, Lorem Ipsum is not simply random text. It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum"(The Extremes of Good and Evil) by Cicero, written in 45

BC. This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, Łorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

The standard chunk of Lorem Ipsum used since the 1500s is reproduced below for those interested. Sections 1.10.32 an