Szegedi Tudományegyetem
Informatikai Intézet

Ipari JavaScript elemző kiegészítése TypeScript támogatással

Industrial JavaScript analyzer enhancement with TypeScript support

Szakdolgozat

Készítette:

Pozsgai Alex

programtervező informatikus BSc szakos hallgató

Témavezető:

Dr.Antal Gábor

egyetemi docens

Szeged
2023

Feladatkiírás

Manapság kezd a piacon egyre több kódelemező megjelenni, többek között JavaScript programozási nyelvre is. Jelenlegi projekt is egy JavaScript Analyzer. A cél az, hogy a projekt több legyen, mint a piacon a többi kódelemező, ezért bővíteni kell TypeScript támogatással.

A hallgató feladata ennek a programnak(JSAN) a fejlesztése úgy, hogy tudjon TypeScript fileokat és projekteket is kellő pontossággal elemezni. Ezután pedig a meglévő programot optimalizálni, hogy kevesebb erőforrást vegyen igénybe a futtatása, és hogy gyorsabban fusson le.

A megoldási módszerek a hallgató kreativitására vannak bízva.

Tartalmi összefoglaló

A téma megnevezése:

JavaScript Analyzer kiegészítése TypeScript supporttal, JSAN optimalizálása.

A megadott feladat megfogalmazása:

A feladat során el kell érni azt, hogy a JavaScript Analyzer Tool tudjon TypeScript fileokra lefutni, és azokat nagy pontossággal elemezni. Emellett a JSAN működését optimalizálni.

A megoldási mód:

A megoldás során kell változtatni a JavaScriptSchémán, és a JSAN-ban az AstTransformer.js fileban.

Alkalmazott eszközök, módszerek:

A megoldáshoz Visual Studio Code IDE-t, a JavaScriptSchema szerkesztéséhez Visual paradigm-t használta. A projekt lebuildeléséhez és ellenőrzéséhez Visual Studio 2017 programot használtam.

Elért eredmények:

JSAN képes TypeScript fileokat nagy pontossággal elemezni, nagyobb projektre lényegesen gyorsabban fut le, kevesebb erőforrást igényel, mint előtte.

Kulcsszavak:

JavaScriptAnalyzer, TypeScriptAnalyzer, JavaScript, TypeScript, Optimalizálás, Visual Studio Code, Visual Paradigm, Vpp, C++

Tartalomjegyzék

Bevezetés

Szakdolgozatomban a JavaScript Analyzer Tool továbbfejlesztése a cél. A továbbfejlesztés arról szól, hogy a javascript mellett előtérbe jön a typescript is. Ezáltal a typescript fileokat és projekteket is elemezni kell. Ez azt eredményezi, hogy a Javascript Analyzer Toolt gyökerestül át kell írni, annak érdekében, hogy JavaScriptet és TypeScriptet is tudjon egyaránt elemezni.

A JavaScript Analyzer Tool az egy nagyobb projektnek az egyik alprojektje. Ezt a nagyobb projektet Analyzer-JavaScriptnek hívják. Az Analyzer-Javascriptnek több alprojektje is van, amiben sok minden egymásra épül. A szakdolgozatomban a következő alprojektekben fogok változtatni: JSAN, és az erre épülő JSAN2Lim. Emellett még megtalálhatóak a következő alprojektek is: ESLintRunner, ESLint2Graph, LIM2Metrics, LIM2Patterns, DuplicatedCodeFinder és a ChangeTracker. Azért kell a JSAN2Limben is változtatni, mivel ha gyökerestül megváltoztatom a JSAN-t, akkor a JSAN2Lim rossz eredményeket fog visszaadni a typescript fileok elemzése közben.

A projekten amin fejlesztettem, azon többen is fejlesztettek egyszerre, ezáltal voltak átfedések, oszthatatlan részek. A projekt legtöbb része egymásra épül, ezért kiemelten fontos volt a csapatmunka egyes részeknél, hogy a projekt eredményesen és hatékonyan haladjon. Az eredményes csapatmunka magában foglalja az eredmények és az előrehaladás folyamatos kommunikálását, ami azt jelenti, hogy mindketten megértjük, hogy milyen célkitűzések vannak a projektben, és rendszeresen jelentést adunk egymásnak az elvégzett munkáról és az elért eredményekről.

Fejlesztés során úgynevezett JavaScriptSchema.vpp file szerkesztése is csoportos munka volt, hiszen minden erre épült, ez volt az alapja mindennek. Sok időt vett igénybe a szerkesztés. Nulláról kellett kellett újraírni az egész vpp filet. Sajnos dokumentáció nem készült az előző filehoz, ezért nekünk kellett kitalálni, hogy mi mit csinált, hiszen akik ezt

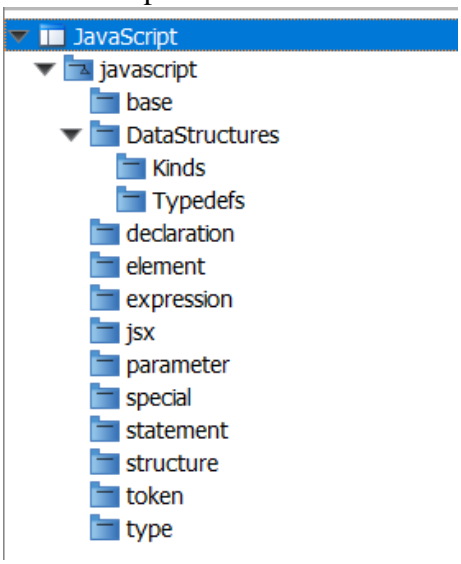
írták, ők már nem foglalkoztak ezzel. A szerkesztéshez szükséges volt a Visual Paradigm alkalmazást használni. Ezzel egyikőnk sem találkozott még, szóval először ezt kellett tanulmányozni, megérteni. Ezután értelmezni kellett a meglévő schemát, hiszen eddig az jól működött, csak nem lehetett könnyen bővíteni. Mérlegeltük a két opciót, ahol vagy megpróbáljuk bővíteni a jelenlegi schémát, vagy nulláról elkezdjük újraírni. Végül az újraírás mellett döntöttünk, hiszen ezt láttuk gyorsabb és könnyebb megoldásnak. Egy könnyen bővíthető, dokumentál schema volt az elképzelés. Ketten fejlesztettük le végül ezt a schemát.

JavaScriptSchema

1.1. A JavaScriptSchemáról

A JavaScriptSchéma egy UML Diagramhoz hasonlító schéma. Jelen esetben a Visual Paradigm alkalmazással szerkeszthető. A felépítése a következőképpen alakul:

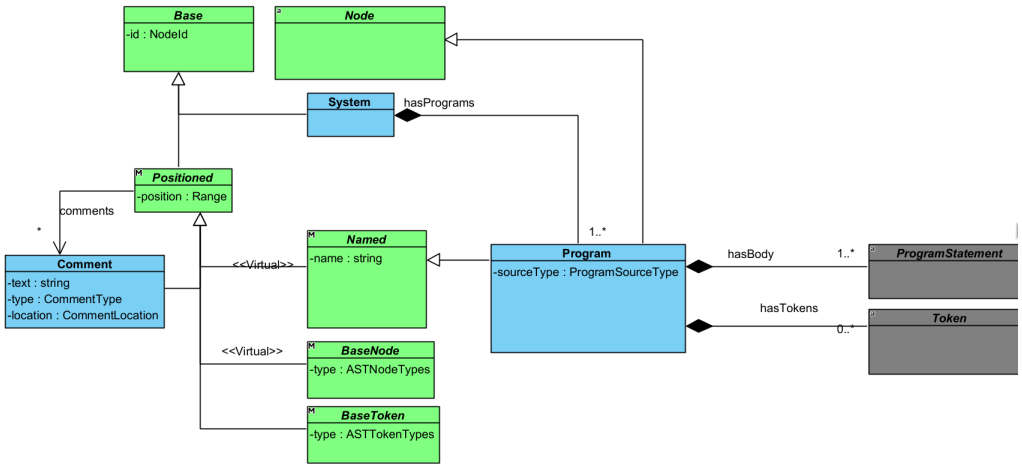
1.1. ábra. JavaScriptSchema struktúrális felépítése



Az 1.1 ábrán a következő struktúra figyelhető meg: ProjektNév-Model-Packagek. A projekt neve a JavaScript, ezen belül található egy model, amit javascript-nek hívnak. A modellen belül találhatóak meg a packagek. A Packagek nem véletlenül így lettek elnevezve. Az alábbi hivatkozáson található meg, hogy milyen logika alapján neveztük el a packageket: [1] Nem feltétlenül muszáj több packaget létrehozni, ez csupán az átláthatóság és a könnyen bővíthetőség céljából lett így megvalósítva. Követtük a typescript-eslint githubon lévő projekt struktúrális logikáját, több helyen is eltértünk tőle, mivel a mi projektünk másabb, illetve speciális megoldásokat igényelt egyes helyeken. A következő

oldalakon bemutatom a packagek felépítését néhány egyszerűbb példán szemléltetve. A packagek a következőképpen épülnek fel:

1.2. ábra. A base package felépítése



Az 1.2 ábrán különböző osztályok láthatóak. A Base osztály mindennek az alapja, ebből öröklődik minden más. Látható, hogy egy id attribútummal rendelkezik, aminek a típusa NodeId. A zöld háttérszínű osztályok az absztrakciót jelentik. Működésben nincs jelentősége, csak a schéma szerkeszthetősége és olvashatósága miatt van így jelezve. A szürke háttérszínű osztályok pedig csupán annyit jelentenek, hogy más packageben lettek definiálva. A kék a default, normális osztályt jelölik. A Positioned és a System származik le a Baseből, értelemszerűen a system az maga a program lesz. A Positioned az azért absztrakt, mivel majd ebből fognak leszármazni a kisebb osztályok, mint például az expression, statement és a többi. A Positioned osztálynak van egy attribútuma, a poisition, ami egy Range típus. Emellett még tartozhatnak hozzá kommentek is. A komment egyaránt leszármazik a positioned-ből, ezzel garantálva azt, hogy a kommentnek van pozíciója. Látható, hogy a kommentnek van egy text, type és location attribútuma. A CommentType és a CommentLocation a DataStructures-ben van definiálva. Emellett a Named, BaseNode és a BaseToken származik le a Positionedből. A Named osztály azt jelenti, hogy egy nodenak van-e name attribútuma vagy sem. A BaseNodeból és a BaseTokenből fog nagyon sok minden leszármazni aminek van typeja. Végül, megtalálható a Program osztály, aminek van name attribútuma, mivel Named-ből származik le, és van Systeme. Az 1..* jelenti azt, hogy legalább 1 Systeme van, de lehet több is. A hasPrograms-nak majd máshol lesz jelentősége, a mi esetünkben majd a javascriptben. Tetszőlegesen el lehet nevezni,

de konzisztencia miatt, minden attribútum ami osztály (és nem a DataStructuresben azon belül is a Kindsban van definiálva, hanem van neki egy osztály, mint pl Comment) azt a hasOsztály névvel fogjuk ellátni. A ProgramSourceType is a Kinds packageben van definiálva, ami csupán azt mondja meg, hogy az adott program vagy script vagy module típusú. A base package nem követi a typescript-eslint githubon lévő projekt base mappáját, ez teljesen egyedi, átememeltük az egészet apróbb módosítással az előző projekt verzióból.

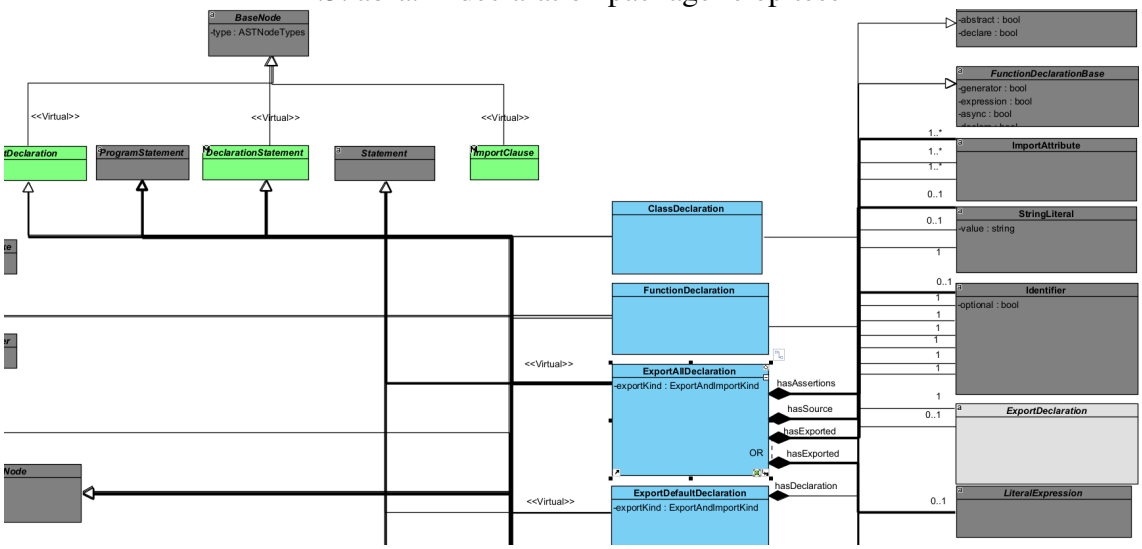
A base packagen kívül bemutatom a declaration packaget, hogy lássuk milyen logikát követtünk a megírás során. Ezen belül is az ExportAllDeclaration bemutatása:

```
export interface ExportAllDeclaration extends BaseNode {
      type: AST_NODE_TYPES.ExportAllDeclaration;
      assertions: ImportAttribute[];
      exported: Identifier | null;
      exportKind: ExportKind;
      source: StringLiteral;
}
```

Kódrészlet 1.1. ExportAllDeclaration typescriptes megvalósítása

Az 1.1 kódrészlet így lett megvalósítva a JavaScriptSchémában:

1.3. ábra. A declaration package felépítése

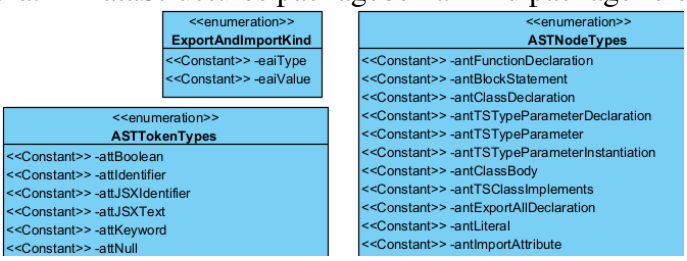


Az 1.3 ábrán látható az, hogy minden a BaseNodeból származik. Az 1.1 kódrészletben ez az extends BaseNode. Itt maga a declaration osztály az a DeclarationState-

ment névre hallgat, csupán azért, mert követtük a typescriptes kódot. A főbb osztályok a BaseNode alatt találhatóak. Jobb oldalt a sötétszürke és a világosszürke osztályok azok az attribútumok. Ebben az esetben az ExportAllDeclaration osztály leimplementálását mutatom be a JavaScriptSchemában. Az ExportAllDeclaration öröklődik a Statement, DeclarationStatement, Node és a ProgramStatementből. Ezeket az öröklődéseket ugyanúgy a typescript-eslint githubról néztük, unions mappában érhetőek el. Ezáltal megkapja az összes szülőnek a tulajdonságait. A DeclarationStatement öröklődik a BaseNodeból, ami azt jelenti, hogy a BaseNode tulajdonságait is megkapja az ExportAllDeclaration. A BaseNode ugye meg származik a Positionedből, ez látható az 1.2 ábrán. Emiatt az ExportAllDeclaration-nek lesz pozíciója, kommentje és NodeId-ja is. Assertions attribútum típusa az ImportAttribute, látható, hogy egy tömböt vár, ezért 1..* a multiplicityje. Exported attribútumnál egy Identifier típusú attribútumot vár, itt mi kiegészítettük még egy LiteralExpressiönnel is, ami maga a Literal. A vagyolást egy OR-al jeleztük a JavaScriptSchemában. Mivel lehet null-is, ezért 0..1 a multiplicity, szóval vagy 0 vagy 1. Az ExportKind az a Kindsban található meg, így szimplán csak arra hivatkozunk, mint ExportAndImportKind. Végül a source attribútuma egy StringLiteral, nálunk is így szerepel. Természetesen ami sötétszürkével van jelölve, az máshol létre van hozva és vannak neki attribútumai. A világosszürke annyit jelöl, hogy ebben a packageben lett létrehozva az osztály, de láthatóság szempontból többször szerepel, mivel lehet attribútum is. Így lett minden egyes osztály felépítve, természetesen Az 1.3 ábra csak egy részlete a declaration packagenek, ennél jóval nagyobb. Vannak úgynevezett gyűjtő osztályok is, mint pl a Statement, ezekre azért van szükség, mert majd később ha le lett generálva minden, akkor tudjunk majd vizsgálni különböző nodeokra, mint például isStatement().

Sokat említettem a DataStructurest. Hadd mutassam be egy példán keresztül, hogy ez hogy néz ki:

1.4. ábra. A DataStructures packageben a Kind package felépítése



A DataStructures packageben található 2 package, ez az 1.1 ábrán látható. Ebből a Kinds packaget mutatom be. Az 1.4 ábrán látható egy kis szelet a Kinds packageből. Enumok találhatóak ebben a packageben. Például ha az ExportAndImportKind-ot adjuk meg típusnak az egyik attribútumnak, akkor az lehet vagy eaiType vagy eaiValue. Minden constant előtt van 3 karakter, ez azért szükséges, mert később ezekkel még foglalkozni fogunk.

A JavaScriptSchema magában még semmit sem csinál. Először ki kell exportálni az egész projectet xml formátumba. Ezután az xml filet átkonvertáljuk egy asg filera. Ez úgy történik, hogy a JavaScript Analyzer projektnek van egy kisebb alprojektje, amit UmlToAsg-nek hívnak. Ez a java kód átkonvertálja az xml fileban lévő adatot egy asg fileba. Jobban nem térnék ki az UmlToAsg projektre, mivel nem szerkesztettük. Az UmlToAsg projektet a SchemaGenerator generálta le. A SchemaGenerator c++ nyelven megírt program. Több mindent is generál c, c++, vagy java nyelven. Ez a projekt is a JavaScript Analyzer alprojektei közé tartozik. A legenerált asg file a következőképp néz ki: A file elején a következő található meg:

```
1 NAME = javascript;
2
3 APIVERSION = 0.3.1;
4 BINARYVERSION = 0.3.1;
5 CSIHEADERTEXT = JavaScriptLanguage;
```

Kódrészlet 1.2. Asg file első sorai

A verziókat kézzel tudjuk átírni abban a fileban ami generálja ezt, ez egy c++ file, a SchemaGeneratorban található meg. Ezután a Kinds mappa tartalmát írja bele a következőképpen:

```
1 KIND ASTNodeTypes (ant) {
2       FunctionDeclaration;
3       BlockStatement;
4       ClassDeclaration;
5       TSTypeParameterDeclaration;
6       TSTypeParameter;
7 }
```

Kódrészlet 1.3. Asg file kind

Az a 3 karakter amit minden constant elé tettünk azt kitette paraméterbe és csak az utáni stringet írta át. Ugyanígy van a többi kindnál is. Ha az összes kindot beleírta, akkor kezdi írni sorban a többi packaget. Az 1.1 alapján megy sorba. Példának a declaration packaget mutatom be, azon belül is az ExportAllDeclaration-t.

```
1  SCOPE declaration {
2
3        NODE DeclarationStatement : virtual base::BaseNode [ABSTRACT] {
4        }
5
6        NODE ExportAllDeclaration : DeclarationStatement, statement::
             Statement, virtual statement::ProgramStatement, special::Node
              {
7              ATTR ExportAndImportKind exportKind;
8              EDGE TREE 1 hasExported (expression::Identifier |
                   expression::LiteralExpression);
9              EDGE TREE 1 hasSource (structure::StringLiteral);
10             EDGE TREE * hasAssertions (special::ImportAttribute);
11       }
12 }
```

Kódrészlet 1.4. Asg file ExportAllDeclaration

Látható, hogy a Packaget SCOPE-nak értelmezi, és ezen belül NODE-ok találhatóak. A Nodeoknak ATTR és EDGE TREE van. A vagyolás is látható. Az öröklődik egy kettőspont után, felsorolás szerűen írta át, ha valami másból származik, akkor packageNev::osztalyNev szerint. Az kapott ATTR jelölést ami a Kinds-ban megtalálható vagy egy szimpla típus (mint pl string, int). Minden mást EDGE TREE-nek nevezett el. Az Edge tree utáni szám vagy csillag az a multiplicitást jelenti. Ha 1es, akkor a multiplicitás 0..1, ha *, akkor vagy 0..* vagy 1..* a multiplicitás. Az 1.3as ábrán látható, hogy mit hogyan írt át.

1.2. JavaScriptSchema átírása

A JavaScript Analyzer a JavaScriptSchemára épül. Ha valamit meg akarunk változtatni gyökerestül a JavaScript Analyzerbe, akkor a schémát is változtatni kell. Azt elérni,

hogy javascript mellett még typescriptes kódokat is elemezzen a JavaScript Analyzer, ahhoz gyökerestül meg kellett változtatni a JavaScript Analyzert. Az előző JavaScriptSchema (ami csak javascriptet elemzett) az a javascript hivatalos oldala alapján készült. Ami ábrák fentebb megtalálhatóak, azok már az átírt schémából vannak. Több opció is volt, hogy most vagy legyen átírva a schéma, vagy legyen egy külön typescriptre is írva. Végül rájöttünk, hogy ha van egy typescriptes schémánk, az képes javascriptet ugyanúgy elemezni ha megfelelően van lefejlesztve. Így egy schémát fejlesztettünk le. Az előző schéma átláthatatlan volt, ezért úgy döntöttünk, hogy majdnem a nulláról újraírjuk. Egyedül a base packaget emeltük át a régiből, BaseNode-al és BaseTokennel kibővítve(Az 1.2 ábrán látható). Emellett el kellett dönteni, hogy milyen struktúrát kövessünk, ami jól átlatható és később könnyebben bővíthető. Végül a typescript-eslint official github alapján haladtunk. Annyi változtatással, hogy ami nekik a base mappában volt, mi arra létrehoztunk egy külön structure packaget. Mindenhez készítettünk dokumentációt, jól érthetően leírtuk, hogy mit miért kellett csinálni. Mivel minden is egymásra épül a typescriptes schémában, ezért nem tudtuk tesztelni minden egyes package után, hogy működik-e vagy sem. Mint például, látható az 1.4 kódrészleten, hogy az ExportAllDeclaration mennyi mindenből származik le vagy mennyire sok attribútuma van. Az ExportAllDeclaration egyik attribútuma, az assertions típusa az ImportAttribute. Ahhoz, hogy jól észrevegye az ExportAllDeclarationt az Analyzer, ahhoz jól le kellett fejleszteni az ImportAttributet.

```
1 export interface ImportAttribute extends BaseNode {
2       type: AST_NODE_TYPES.ImportAttribute;
3       key: Identifier | Literal;
4       value: Literal;
5 }
```

Kódrészlet 1.5. ImportAttribute

Ahhoz, hogy az ImportAttributet letudjuk fejleszteni a schémában, ahhoz le kell fejleszteni az Identifiert, és a Literalt. Kicsit összetett, hogy mi minden épül egymásra. Ebből adódik a következő alfejezet mondandója is.

1.3. Nehézségek, problémák

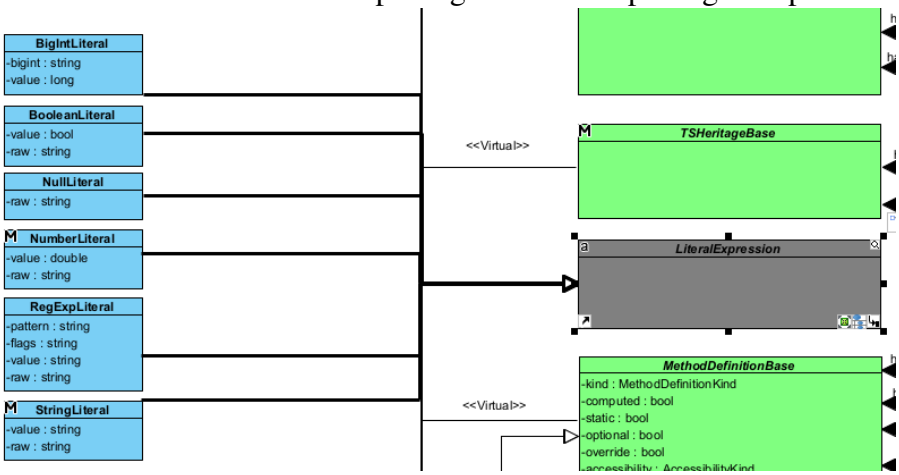
A schéma átírása során több nehézségbe is ütköztünk. A legelső nehézség az a Visual Paradigm program korrekt használata volt. Egy kisebb időbe tellett rájönni arra, hogy egy adott classnak hogyan kell megváltoztatni a háttérszínét, packagek közötti mozgást, hogyan kell egy adott classra hivatkozni ami másik packageben volt. Ezután ami szerintem a legnagyobb nehézség volt, az az, hogy nem volt dokumentáció az előző schémához. Mindent nekünk kellett kitalálni, hogy mit miért csináltak az előző fejlesztők. Akik ezt a schémát fejlesztették le, ők már nem foglalkoztak ezzel, és más nem nagyon mélyedt bele ebbe az egészbe azóta. A következő nagyobb nehézség, az az volt, hogy összehozzunk egy olyan schémát ami működőképes, és az analyzer ezt tudja is használni. Előző alfejezetben említettem, hogy egy adott classt (mint például ExportAllDeclaration) lefejlesszünk, ahhoz sok minden mást is le kell fejleszteni, amihez meg még több minden kellett. Ebből adódóan tesztelni nagyon nem tudtunk, mivel az egésznek működnie kellett ahhoz, hogy az analyzer egyáltalán lefusson. Mivel mi majdnem nulláról írtuk újra, ez volt a hátrány. Amikor elérkeztünk ahhoz az állapothoz, hogy mindenre rámondjuk azt, hogy működik, akkor jött egy nagyobb probléma. Valahol Segmentation Fault-ot kapott a program, és nem nagyon tudtuk ezt debugolni. Több sejtésünk is volt, hogy mi lehet a baj. Emiatt az egész schémát át kellett nézni alaposan, hogy hol vétettünk hibákat. Sok helyen voltak pontatlanságok, rossz osztályból származtattunk le, rossz típus volt megadva attribútumnak. Ezeket mind kijavítottuk, de most már más hibát kaptunk. A JavaScript Analyzerben kiderítettük, hogy pontosan hol szállt el a program. A Literal volt a hiba, mivel ezt nem a github alapján írtuk meg, hanem egyedi ötlettel. Előző schémában is máshogy volt megoldva.

Az 1.5 ábrán látható, hogy a LiteralExpressionből(Ami a Literal) több literal is öröklődik.

```
1 export interface LiteralBase extends BaseNode {
2   type: AST_NODE_TYPES.Literal;
3   raw: string;
4   value: RegExp | bigint | boolean | number | string | null;
5 }
```

Kódrészlet 1.6. Literal

1.5. ábra. A DataStructures packageben a Kind package felépítése



Próbáltuk úgy megoldani, hogy 4 vagyolással 4 különböző literál tartalmazza, de az analyzerben máshogy kezeltük le a literalt, mint nodeot. Ezért a tartalmazás helyett inkább örököltettünk a literalból, így megoldva ezt a problémát. Még az is probléma volt, hogy volt egy LiteralBaseünk, amiből származott ez az 5 literal. A LiteralBase származott a LiteralExpressionből, de valamiért Segmentation fault lett a vége ha így próbáltuk megoldani. Ezért a LiteralBase-t kivettük, és LiteralExpressionből származik minden literal. Végül még az volt a nehézség, hogy kódrészletet keressünk az analyzernek, amin le tudjuk tesztelni, hogy az analyzer jó outputot ad-e.

Mivel nem nagyon volt typescriptes háttértudásunk, először a typescriptet kellett átnézni, hogy mit hogyan tudunk megvalósítani. Emellett még a javascriptes outputokat is át kellett nézni, mivel a mi célunk a fejlesztés volt. A jelenlegi javascriptes projektekre ugyanolyan eredményt adott, sőt néhány helyen jobbat is, mert az előző schémában is voltak pontatlanságok. A typescriptes projektek nagy részét tudja elemezni az analyzer, de közel sem tökéletes, mivel egyfolytában kellene fejleszteni a schémát, mivel a typescript nem annyira régi.

JavaScriptAddon szerkesztése

-mit ad factory, mi ez a külön file -Miben változott

2.1. JavaScriptAddonról

2.2. Miben változott

JSAN2Lim átírása

-halstead, stb -miben kellett bőviteni, mi hiányzott -mit nem detektált

3.1. JSAN2Limről

3.2. Bővítések

3.3. Miket nem detektált

AST Binder Optimalizálás

-node addon mi -miért lassú -javítás, ha lehet, vagy ha nem, miért - nehézségek, eredmények

4.1. Binderről

4.2. Lassúság okai

4.3. Optimalizálás

4.4. Eredmény

Regteszt frissítés

jobb lett, mükszi hogy van tesztelve, output és performance vizsgálat TS tesztek hozzáadása / eredmények átnézése Specifikusabb tesztek hozzáadása (kapcsolókra tesztelünk –> JSAN / ESlintrunner 1 mappa / 1 kapcsoló) Python beli nehézségek tesztátírás során

5.1. Röviden a regtesztről

5.2. Tesztekről

5.3. Tesztek kibővítése

Összefoglaló

- javult a program - VPP up do datere hozása - JSAN további optimalizálása

6.1. Program javulása

6.2. Tervek

6.2.1. JavaScriptSchema naprakészre hozása

6.2.2. JSAN további optimalizálása

Nyilatkozat

Alulírott Pozsgai Alex programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2023. május 1. .....

aláírás

Köszönetnyílvánítás

Irodalomjegyzék

[1] Typescript-eslint github repository. https://github.com/typescript-eslint/typescript-eslint/tree/main/packages/ast-spec/src.