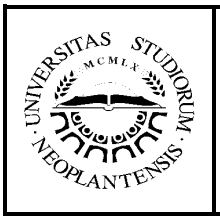


УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику
Одсек за рачунарску технику и рачунарске комуникације

ЗАВРШНИ (BACHELOR) РАД

Кандидат: **Алекса Чоловић**
Број индекса: **РА 38/2020**

Тема рада: **Преводилац из Структуираног Текста за ПЛЦ контролере у програмски језик Пајтон**

Ментор рада: **проф. др Мирослав Поповић**

Нови Сад, Јул 2024

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:			
Идентификациони број, **ИБР**:			
Тип документације, **ТД**:		Монографска документација	
Тип записа, **ТЗ**:		Текстуални штампани материјал	
Врста рада, **ВР**:		Завршни (Bachelor) рад	
Аутор, **АУ**:		Алекса Чоловић	
Ментор, **МН**:		проф. др Мирослав Поповић	
Наслов рада, **НР**:		Преводилац из Структуираног Текста за ПЛЦ контролере у програмски језик Пајтон	
Језик публикације, **ЈП**:		Српски / латиница	
Језик извода, **ЈИ**:		Српски	
Земља публиковања, **ЗП**:		Република Србија	
Уже географско подручје, **УГП**:		Војводина	
Година, **ГО**:		2024	
Издавач, **ИЗ**:		Ауторски репринт	
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6	
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		7/36/7/11/5/0/0	
Научна област, **НО**:		Електротехника и рачунарство	
Научна дисциплина, **НД**:		Рачунарска техника и рачунарске комуникације	
Предметна одредница/Кључне речи, **ПО**:		преводилац, ПЛЦ контролери, формална граматика, Структуирани Текст, Пајтон	
УДК			
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад	
Важна напомена, **ВН**:			
Извод, **ИЗ**:		Због модернизације у свету технологије, појавила се потреба за коришћењем савременијих програмских језика за програмирање ПЛЦ контролера. Због тога је за почетак, кључна ставка да се већ постојећи код у Структуираном Тексту преведе на програмски језик Пајтон, који нуди већу подршку за програмера. Генерисан је парсер за Структуирани Текст уз помоћ написане формалне граматике, где је након тога уз помоћ изграђеног стабла парсирања генерисан излазни Пајтон код. Резултат рада је успешно генерисан излазни код, који садржи преведене све главне функционалности Структуираног Текста.	
Датум прихватања теме, **ДП**:			
Датум одбране, **ДО**:			
Чланови комисије, **КО**:	Председник:	доц. др Миодраг Ђукић	
	Члан:	проф. др Иван Каштелан	Потпис ментора
	Члан, ментор:	проф. др Мирослав Поповић	

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Bachelor Thesis	
Author, **AU**:		Aleksa Čolović	
Mentor, **MN**:		Miroslav Popović, PhD	
Title, **TI**:		Compiler from Structured Text for PLC controllers to Python programming language	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		2024	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		7/36/7/11/5/0/0	
Scientific field, **SF**:		Electrical And Computer Engineering	
Scientific discipline, **SD**:		Computer Engineering and Computer Communications	
Subject/Key words, **S/KW**:		translator, PLC controllers, formal grammar, Structured Text, Python	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		Due to modernization in the world of technology, there arose a need for using more modern programming languages for programming PLC controllers. Therefore, as a starting point, the key task is to translate the existing code in Structured Text to the Python programming language, which offers greater support for the programmer. A parser for Structured Text was generated using a written formal grammar, and then, with the help of the built parse tree, the output Python code was generated. The result of the work is successfully generated output code, which includes all the primary functionalities of the original Structured Text.	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	Miodrag Đukić, PhD	
	Member:	Ivan Kaštelan, PhD	Menthor's sign
	Member, Mentor:	Miroslav Popović, PhD	

	УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА** 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број:
	ЗАДАТАК ЗА ЗАВРШНИ РАД	Датум:

(Податке уноси предметни наставник - ментор)

Студијски програм:	Рачунарство и аутоматика		
Студент:	Алекса Чоловић	Број индекса:	РА 38/2020
Степен и врста студија:	VII-1a Основне академске студије		
Област:	Електротехника и рачунарство		
Ментор:	проф. др Мирослав Поповић		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА ЗАВРШНИ РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;

НАСЛОВ ЗАВРШНОГ РАДА:

Преводилац из Структуираног Текста за ПЛЦ контролере у програмски језик Пајтон

ТЕКСТ ЗАДАТКА:

У оквиру овог дипломског рада потребно је урадити следеће:

1. Упознати се са Структуираним текстом за програмирање ПЛЦ контролера, као и са модерним алатима за имплементацију преводилаца
2. Израдити пројекат преводиоца из Структуираног текста у програмски језик Пајтон
3. Имплементирати преводилац према израђеном пројекту
4. Валидирати имплементацију на скупу примера програма у Структуираном тексту поређењем резултата оригиналних и преведених програма
5. Уочити предности и мане имплементираног преводиоца
6. Написати текст дипломског рада

Руководилац студијског програма:	Ментор рада:
др Милан Рапаић	проф. др Мирослав Поповић

Примерак за: o - Студента; o - Ментора

Захвалност

Захваљујем се свом ментору професору др Мирославу Поповићу, као и техничком ментору Милану Рељину на стручној помоћи, саветима и подршци током израде завршног рада.

Такође захваљујем се породици, пријатељима и колегама на неизмерној подршци.

За крај, овај рад бих желео свим срцем да посветим својој мајци, која не само да ми је пружила подршку и снагу за завршетак ових студија, већ и за све будуће амбиције и препреке које ме једнога дана дочекају.

Татјана Чоловић 1980 – 2022.

Садржај

Списак слика

Списак табела

Скраћенице

PLC	- Programmable Logic Controller - програмабилни логички контролери
IEC 61131	- International Electrotechnical Commission standard for programmable controllers - ИЕЦ стандард дефинисан за програмабилне контролере
LD	- Ladder Diagram - графики начин за опис рада контролера
ST	- Structured Text - текстуални начин за опис рада контролера
AST	- Abstract Syntax Tree - апстрактно синтаксно стабло
CFG	- Context - Free Grammar - контекстно независна граматика
BNF	- Backus - Naur Form - Бакус - Наурова нотација за опис формалне граматике
EBNF	- Extended Backus - Naur Form - проширена Бакус-Наурова нотација за опис формалне граматике
ANTLR4	- ANother Tool for Language Recognition - још један алат за препознавање језика
LL	- Left to right, Leftmost derivation - парсирање улаза слева надесно са крајњим левим изводом
LR	- Left to right, Rightmost derivation - парсирање улаза слева надесно са крајњим десним изводом
GUI	- Graphical User Interface - графичка корисничка спрега

1. Увод

Доступност и коришћење различитих сервиса, односно физичких производа у данашњем, модерном добу, можемо захвалити индустријској аутоматизацији. Историјски, концепт аутоматизације није новина. Још од давнина постоје записи неких једноставних начина аутоматизовања процеса као што је водени сат, а нама временски ближи и познатији рани изуми су воденица и ветрењача. Појавом електричних фабрика, почетком 20-ог века, уводи се релејска логика аутоматизовања са једноставним контролером. Како су се такви системи показали тешким за промену и проширивање самих процеса производње, касних 1960-их се изумом **PLC** контролера полако прелази на такво револуционаризовано управљање система. [1]

Како се **PLC** индустријске машине користе за ефективно управљање производног процеса, односно самих постројења, робота и других уређаја, велику намену имају у поузданом и брзом управљању тока, као и олакшање приликом дијагнозе проблема. [1] Програмирање ових контролера је дефинисано стандардом **IEC 61131** којим су се њихови произвођачи водили. Стандард је дефинисао графичке језике који су били погодни и за људе без претходног знања у програмирању (од којих је познатији **LD**), као и текстуалне језике (најчешће коришћен **ST**, који је такође и главна тема овог рада). [2]

Из тог разлога, узевши у обзир да је дефинисани стандард прилично застарео, све више младих инжењера тежи томе да избегне рад са оваквим програмским језицима, те су више наклоњени неким модернијим. Како би се омогућио наставак коришћења **PLC** контролера, али у модернијем облику, све се више ради на томе да се младима пружи могућност програмирања ових машина у језику по избору.

Први проблем би била већ постојећа инфраструктура контроле рада ових машина, те би инжењер морао ручно испочетка програмирати читаву логику контролера. Самим тим што је то изузетно дугачак процес, први корак како би се то убрзало јесте да се на неки начин изврши превођење постојећег *Legacy* кода, који је у приватној својини власника контролера који се програмирају. Пошто већина постојећих алата за програмирање **PLC** контролера нуди кориснику изузетно велики број уграђених функција, појављује се препрека из разлога јер су све имплементације истих сакривене од корисника.

Због тога се као почетни циљ овог пројекта поставља реализација преводиоца, који ће минимално омогућити већинско превођење постојећег кода и тиме убрзати процес ручног превођења, а да на кориснику остане одлука како ће реализовати недефинисане функције. Идеја је да се покуша превести што више механизама Структуираног Текста у Пајтон, који је изузетно коришћен језик са подршком велике заједнице корисника.

Саставни део рада представља писање функционалне формалне граматике за програмски језик Structуирани Текст, генерисање његовог парсера и генерисање кода на програмском језику Пајтон.

2. Теоријске основе

У овом поглављу, биће детаљније описане теоријске основе потребне за почетак израде оваквог рада, као и алати потребни за овакву имплементацију.

2.1 Преводилац

Преводилац представља програм који преводи један језик у неки други. На основу односа између та два језика, као и других особина, препознајемо различите врсте преводилаца. Иако класификације нису стриктне и зависе од контекста, неки од основних типова преводилаца представљају компајлери, асемблери, интерпретери и др. Док под традиционалним компајлером сматрамо преводилац који преводи програмски језик вишег нивоа у неки нижег, у нашем случају ради се о компајлеру који преводи из једног програмског језика вишег нивоа у други. Неке од честих имена за ову имплементацију су *Source-to-Source Compiler*, *Transcompiler* или једноставно *Transpiler*. Главна улога оваквог преводиоца је прелазак на други програмски језик ради већих погодности или једноставно звог застарелости језика, а да притом програмски код буде што сличнији оригиналном.

2.1.1 Процес превођења

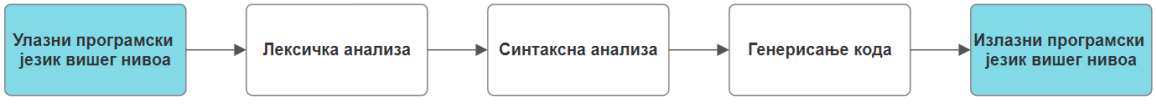
У зависности од крајњег циља, процес превођења се може састојати из променљивог броја корака. Најосновнији тип преводиоца има бар 3 корака (Слика 1):

1. Лексичка анализа
2. Синтаксна анализа
3. Генерисање кода

Лексичка анализа, или токенизација, представља процес састављања токена на основу секвенце узастопних карактера. [3] Токен представља градивну јединицу језика односно граматике. У једном програмском језику токен може бити нека кључна реч, идентификатор, литерал и сл. Улаз у овом процесу би био неки изворни код, док би излаз била секвенца направљених токена.

Синтаксна анализа, или парсирање, представља процес анализе секвенце токена како би се уочила структура језика на основу постојеће граматике. [3] Синтаксна анализа такође служи за валидирање начина на који су реченице оформљене. Улаз овог дела система представља сама секвенца токена, а излаз би био неки начин описа структуре грађења реченица. Тај опис структуре може бити **AST** или стабло парсирања (*Parse Tree*).

Генерисање кода представља један од завршних корака и заснива се на стварању кода за циљани језик на основу претходних излаза. [3] Улаз овог корака представља претходно формирано структурно стабло. То исто стабло може бити различито од оног које је потекло из синтаксне анализе уколико се програмер одлучио за неки сложенији преводилац. Овом кораку може на пример претходити поступак семантичке анализе, оптимизације и сл.



Слика 1 - Најједноставнији пример преводиоца

2.2 Формална граматика

Формална граматика представља веома важан концепт у рачунарству. Велики значај огледа се у дефинисању синтаксе за програмске језике, дефинишући саму структуру, као и правила која граде тај језик. Формална граматика је кључна за процес парсирања, тиме што омогућава генерисање стабла парсирања (*Parse Tree*) које је неопходно за компајлере како би генерисали излазни код.

2.2.1 Основе формалне граматике

Најчешће коришћен тип формалне граматике, као и онај који је употребљен у овом раду, представља контекстно независна граматика (CFG). Свака граматика се састоји из скупова нетерминалних симбола, терминалних симбола, правила продукције, као и од једног почетног симбола. За сам опис ове граматике, честе нотације су или Бакус-Наурова Форма (BNF) или проширена Бакус-Наурова Форма (EBNF). [4]

Нетерминални симбол - представља симбол који може бити замењен секвенцом других симбола

Терминални симбол - представља симбол који назначава крај проширивања и замене секвенцом симбола, односно представља саму реч анализираног језика.

Правило продукције - представља опис начина на који ће се вршити замена нетерминалних симбола, односно којим скупом симбола ће бити замењени.

Извод (*Derivation*) - представља секвенцу примењених правила продукције како би се добио неки скуп симбола. Крајњи леви (*Leftmost*) извод представља стратегију узастопног преписивања најлевљег нетерминалног симбола, а супротност представља крајњи десни (*Rightmost*)

Почетни симбол - представља нетерминални симбол који назначава почетак граматике из којег ће произаћи сва остала правила

Двосмисленост (*Ambiguity*) - постоји више извода како би се добила иста секвенца симбола

2.2.2 Контексно независна граматика

Овај тип формалне граматике карактерише могућност да се правило продукције примени на нетерминални симбол без обзира на његов тренутни контекст. [3]

$$A \rightarrow \alpha$$

Симбол A представља један нетерминални симбол, који се замењује α скупом симбола који такође може бити и празан.

Супротност овом типу граматике представља контекстно зависна граматика, код које да би се применило правило продукције на нетерминални симбол, потребно је сагледати контекст у коме се тај симбол налази.

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

2.2.3 BNF и EBNF

Ове нотације имају кључну улогу у дефинисању синтаксе неког језика, нудећи прецизну методологију за опис саме структуре тог језика. У зависности од језика који се описује и његове сложености, бира се једна од ове две постојеће нотације. [3]

BNF представља једноставан опис употребом нетерминалних и терминалних симбола, правила продукције и нуђењем алтернатива. Типично се правило састоји од једне линије, док то није случај код **EBNF**. Она представља проширење **BNF**, тиме што нуди различите додатне операторе. Ти оператори се односе на необавезне елементе граматике, понављања (нула пута, једном или више), груписање и друге погодности. Иако се од сваке граматике написане по **EBNF** нотацији може написати еквивалентна **BNF** нотација, она и даље има своје погодности, као што је лакше читање, већа могућност проширивања и већа прилагодљивост. Из тог разлога она је много више коришћена у пракси.

2.3 Парсери

Парсер, или синтаксни анализатор, представља граматички алат за разбијање сложених структура (типично текста) у скуп компоненти тј. речи на основу неке дефинисане улазне граматике. Парсирању типично претходи процес лексичке анализе, односно стварање токена на основу секвенце карактера. [5]
Главна подела парсера према врсти:

1. Анализа наниже (*Top - Down Parser*)
2. Анализа навише (*Bottom - Up Parser*)

Анализом наниже, стабло парсирања се гради од корена (почетног симбола) па надоле према листовима, док тежи за крајњим левим изводом. Често коришћен подтип је **LL** парсер.

Анализом навише се почиње од улазног текста и покушава градити стабло према почетном симболу, притом тежи крајњем десном изводу у обрнутом редоследу. Често коришћен подтип је **LR** парсер.

2.3.1 ANTLR4 – Алат за генерисање парсера

ANTLR4 представља једну моћну алатку која се употребљава при генерисању парсера, а користи се за читање, обраду, извршавање или превођење текста. Користи се у мноштву пројеката, а један од најпознатијих је социјална платформа Твитер (*Twitter*) где му је примена парсирање упита за претрагу. Овај алат је у могућности да на основу написане граматике, која је уједно и улаз овог процеса, генерише парсер за описан језик. Парсер се користи за аутоматско састављање стабла парсирања (*Parse Tree*), које чини једну структуру чији је циљ представљање начина на који су улазне речи састављене у реченице. [6]

ANTLR парсер користи адаптивну технику парсирања звану **ALL** која обавља динамичку анализу током извршавања, која представља унапређење у односу на претходне верзије које користе статичку анализу тј. **LL** парсери. Адаптивно парсирање уклања могућност постојања двосмислености тиме што је увек у могућности да разреши постојећу секвенцу. [6]

Један од проблема које **ANTLR4** успева да реши је проблем леве рекурзије, који може довести до бесконачне петље приликом парсирања. То аутоматски решава преписивањем правила у еквивалентан израз који није лево рекурзиван. [6]

2.4 Програмски језик Структуирани Текст (ST)

Структуирани Тест представља програмски језик вишег нивоа који се примењује приликом **PLC** програмирања. Језик је налик програмским језицима *Паскал* и *Ц*, а служи за писање сложених контролних алгоритама. [7]

2.4.1 Особине

Неке од предности овог језика предстаља то што је:

- Изузетно читљив и лак за разумевање
- Модуларан и омогућава лакше одржавање
- Усаглашен у функционисању са осталим **IEC 61131** програмским језицима
- Скалабилност која је погодна за све нивое сложености пројеката јер подржава све од једноставне логике, па све до сложених система аутоматизације
- Развојна могућност која подстиче структуиран и систематичан начин развоја система

Са друге стране, мане су то што је:

- Тежи за учење, посебно за особе без претходног знања програмирања
- Захтева много више ресурса за разлику од неког програмског језика нижег нивоа или придруженог графичког језика
- Платформска компатибилност где различити произвођачи **PLC** контролера нуде различите имплементације стандарда, иако већински прате **IEC 61131** стандард и даље доводи до проблема приликом миграције софтвера
- Мала заједница за подршку, а самим тим и недостатак ресурса, библиотека, као и практичне подршке

2.4.2 Алат за програмирање

Постоје разни алати за развој и симулацију аутоматизованих система, конкретно за PLC контролере. Неки од познатијих, који нуде подршку за специфичне моделе контролера различитих произвођача су:

- *B&R – Automation Studio*
- *Siemens – TIA Portal*
- *Beckoff – TwinCAT*
- *Codesys* и др.

Овај тип алата је најчешће комерцијализован и не постоји пуно опција за бесплатно коришћење. Једини начин представља пријава за привремену бесплатну лиценцу, која је истраживачког типа. Проверен софтвер са временски најдужом оваквом лиценцом је *Automation Studio* од компаније *B&R* где је могуће коришћење од максимално 90 дана.

3. Концепт решења

У овом поглављу, биће објашњени главни концепти тј. сегменти израде овог пројекта, као и коришћене методологије. Овај део ће се претежно заснивати на претходно објашњеним теоријским основама.

3.1 Главна структура пројекта

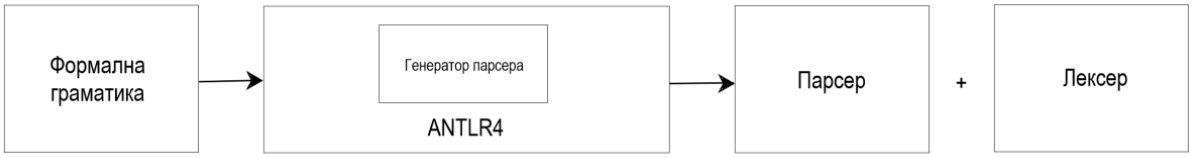
Слика 2 - Структура пројекта

Рад се састоји из неколико целина, које су оквирно представљене у склопу теорије о процесу превођења (Поглавље 2.1.1). Структуру чине (Слика 2):

- Лексичка анализа
- Синтаксна анализа
- Генерисање излазног кода

Иако се **ANTLR4** у овом пројекту претежно користи за генерисање функционалног парсера, он нуди и друге погодности. У склопу своје имплементације садржи лексички анализатор, као и механизме за генерисање излазног кода.

За успешност лексичке и синтаксне анализе, најважнији део представља постојање граматике програмског језика Структуирани Текст. Иако алат **ANTLR4** нуди велики број граматика за различите програмске језике, Структуирани Текст није једна од њих, те ће писање формалне граматике такође представљати срж овог пројекта (Слика 3).



Слика 3 - Генерисање парсера

3.1.1 Лексичка анализа

За успешно генерисање скупа токена, захтева се постојање дефинисаних лексичких правила у склопу граматике за **ANTLR4**. Формална граматика мора садржати све постојеће кључне речи које се користе у изворном језику. Такође треба садржати и правила за грађење токена који могу бити произвољне структуре, као што су идентификатори, литерали и сл.

3.1.2 Синтаксна анализа

Како би постојао функционалан парсер, потребно је да он буде генерисан уз помоћ **ANTLR4**. Да би било могуће то реализовати, претходно је потребно дефинисати формалну граматику односно парсерска правила те граматике.

3.1.3 Генерисање кода

ANTLR4, осим што нуди аутоматско прављење стабла парсирања, омогућава и механизам обиласка чворова ради извршавања потребних активности, у овом случају генерасања кода. То омогућава стварањем два посетиоца стабла који се називају *Listener* и *Visitor*.

Listener нуди два аутоматска начина обиласка чворова:

- Улазно посећивање у *Preorder* поретку (*Enter<Node>*)
- Излазно *Postorder* напуштање (*Exit<Node>*) тих чворова без могућности утицаја на његов редослед.

Уколико је потребно имати већу контролу над начином на који се пролази кроз чворове, користи се *Visitor*. Он нуди експлицитно одређивање редоследа којим се посећују чворови и нуди изузетну прилагодљивост. За наше потребе биће коришћена друга класа.

Алтернатива генерисању излазног кода помоћу *Visitor* класе може бити прављење неке врсте програмске структуре улазног језика помоћу исте класе, а затим коришћење процесора шаблона за генерисање излаза. У овом пројекту је одлучено коришћење прве методе ради брже израде решења због погодности коју класа нуди.

За успешно генерисање еквивалентног излазног кода, потребно је такође осмислити начин имплементације постојећих механизама у оквиру Структуираног Текста и прилагодити га за програмски језик Пајтон. Реализација ће бити детаљније објашњена у следећем поглављу.

4. Програмско решење

У овом поглављу, биће објашњена темељна реализација рада, у виду начина реализације, приступа, као и других спецификација. Више детаља ће бити речено о следећим имплементацијама:

- Дефинисању формалне граматике
- Генерисању парсера
- Генерисању излазног кода уз помоћ класе *Visitor*

4.1 Дефинисање формалне граматике за ST

Сама реализација се одвила у два дела:

- Дефинисање лексичких правила
- Дефинисање синтаксних правила

4.1.1 Лексичка правила

Приликом дефинисања правила грађења токена, темељно је проучен програмски језик **ST**, тако да садржи све постојеће кључне речи, идентификаторе, операторе, литерале, коментаре и остале специјалне знакове.

Писање лексичких правила се заснива на неколико принципа као што су:

- Назив лексичког правила почиње великим словом
- Приоритет грађења токена иде по редоследу дефинисања (правила изнад имају предност)
- Уколико не треба да се одради грађење токена за дефинисано правило, користи се кључна реч *fragment*
- Уколико треба прескочити неке токене који нису ни у једној постојећој секвенци, користи се кључна реч *skip*

Табела 1 - Пример дефинисања лексичких правила

```
Int: 'INT';
FloatValue: PositiveNumber '.' Digit+;
fragment PositiveNumber: Digit+;
fragment Digit: [0-9];
WS: [ \n\t\r] -> skip;
```

4.1.2 Синтаксна правила

Дефинисање синтаксних правила засновано је на приступу обраде од доле на горе (*Bottom - Up*), фокусирајући се прво на мање градивне целине, а затим постепено и на елементе чији су они саставни део. Писање синтаксних правила се заснива на следећим принципима:

- Назив синтаксног правила почиње малим словом
- Приоритет у оквиру једног правила, уколико постоји двосмисленост, има она алтернатива која је дефинисана раније
- Могуће је једно правило потпуно лабелирати, тако што се дода назив свакој алтернативи
- Постоји почетно правило из којег ће сва остала потећи

Као основна синтаксна правила можемо сматрати скупове лексичких правила (више врста литерала, типова и сл.) и специфичности као што је нпр. опсег вредности, приступ низу и сл.

Табела 2 - Пример основних лексичких правила

```
literal: BoolValue | StringValue | WstringValue | FloatValue | IntegerValue;
range: IntegerValue DotDot IntegerValue;
arrayAccessing: Identifier LeftBracket (IntegerValue | Identifier) RightBracket;
```

Следећа целина која представља мало сложенији градивни блок представљају изрази, од којих се гради већина исказа. Како на месту било ког операнда који је у саставу израза, може бити други израз, појављује се потреба рекурзивне реализације синтаксног правила. Код њих је такође потребно обратити пажњу и на приоритет извршавања оператора. У оквиру израза ће у даљем тексту бити објашњен концепт приоритетности алтернатива приликом прављења стабла парсирања. У следећој табели дат је пример неких делова правила за изразе.

Табела 3 - Исечак дефинисаног правила за изразе са лабелама

```
expression: functionCall expression* #FunctionCallExpr
    | LeftParenth expression RightParenth #ParenthesisExpr
                            ...
    | expression (Mul | Div | Mod) expression #HighPriorityOperationsExpr
    | expression (Plus | Minus) expression #LowPriorityOperationsExpr
                            ...
    | identifier #IdentifierExpr
    | literal #LiteralExpr
    ;
```

Као што је претходно споменуто, даље већу целину представљају искази, од којих су сачињени сви блокови у оквиру програмског језика. Пратећи даље дефинисање већине исказа, наилази се на рекурзију (*if* исказ се састоји од других исказа, итд.). У следећој табели приказано је дефинисано правило за исказе.

Табела 4 - Дефинисана правила за исказе

```
statement
    : Return Semi
    | ifStatement Semi?
    | caseStatement Semi?
    | forStatement Semi?
    | whileStatement Semi?
    | repeatStatement Semi?
    | assignStatement Semi;
```

Главну срж Структуираног Текста, чине различити блокови (главних програма, функција, променљивих, сложених типова). Већина њих се састоји од вишеструких исказа, док остали садрже декларације и/или дефиниције функција или променљивих. Блокови такође садрже и две ознаке (отварајући и затварајући *Tag* нпр. *Function* и *End_Function*). У следећој табели приказана је дефиниција блокова Структуираног Текста.

Табела 5 - Дефинисана правила за све типове блокова у оквиру ST

```
start: programBlock+ EOF;


programBlock
    : program
    | dataTypeBlock
    | functionDeclaration
    | function
    | functionBlock
    | functionBlockDeclaration
    | varConstant
    | variables
    ;
```

4.2 Генерисање парсера

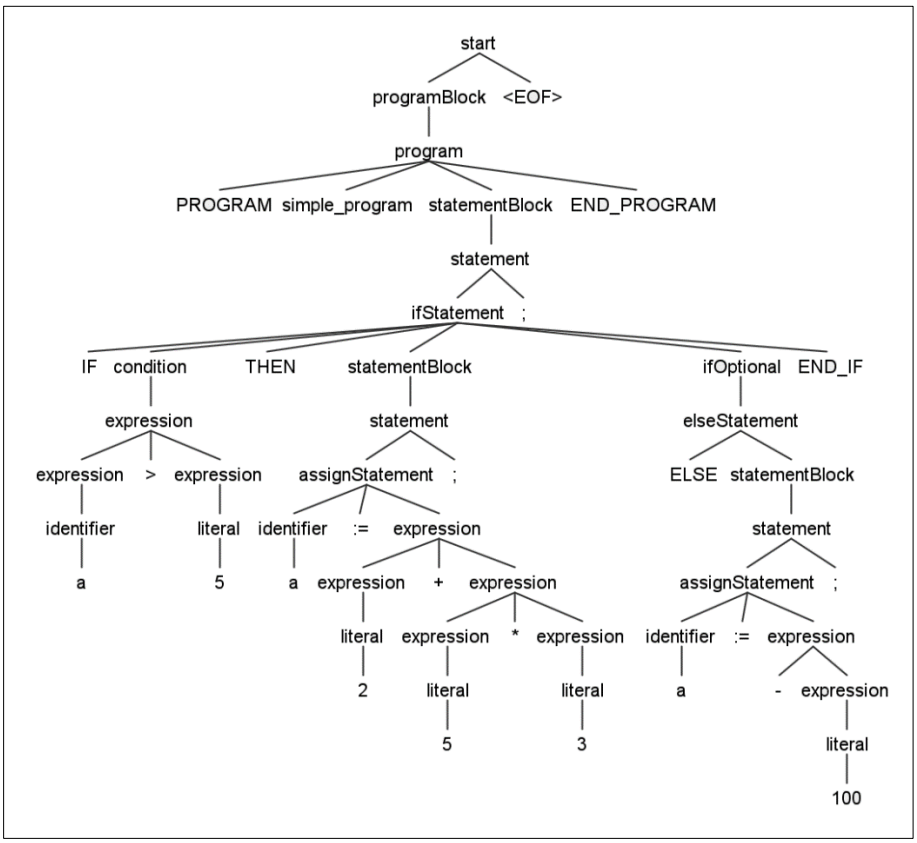
Након успешно написане формалне граматике језика Структуирани Текст, потребно је генерисати парсер у виду помоћних класа за функционисање пројекта. Приликом генерисања могуће је одабрати неколико опција. Како **ANTLR4** нуди подршку за генерисање класа за више различитих језика, у овом случају одлучено је да то буде Пајтон. Самим тим читав пројекат ће бити написан на програмском језику Пајтон. Друга опција је да **ANTLR4** уз стандардан *Listener*, генерише и класу *Visitor* ради потреба овог рада.

4.3 Генерисање излазног кода

Након што се успешно генерише парсер за Структуирани Текст, следећи корак је да се на основу стабла парсирања које представља излаз успешног парсирања, генерише излазни код. У овом случају је излазни код у програмском језику Пајтон.

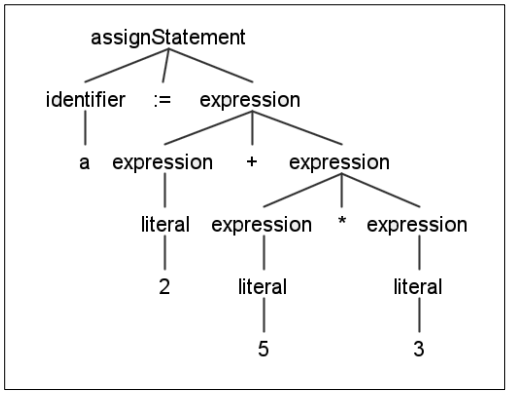
4.3.1 Стабло парсирања

Стабло парсирања садржи читаву синтаксну структуру (Слика 4), која ће бити обрађена помоћу постојеће класе *Visitor*.



Слика 4 - Пример једног ST стабла парсирања

Овде је такође приказан и механизам приоритетности алтернатива (Слика 5), где ће на пример за случај (2 + 5 * 3), због приоритета прво бити уклопљен шаблон за множење, а затим за сабирање. Сви крајњи листови представљају терминалне симболе, *start* представља почетни, док су сви остали нетерминални.



Слика 5 - Пример приоритетности алтернатива

4.3.2 Пресликавање функционалности

У овом одељку, биће објашњена реализација пресликавања функционалности и концепата Структуираног Текста у Пајтон. Језик поседује доста сличности са многим језицима, као и неке разлике.

Почевши од опште структуре програмског језика Структуирани Текст, састоји се из више блокова са ознакама (*Tags*). Најважнији, без којег програм не би имао смисла је програмски блок. У оквиру окружења *Automation Studio* постоје три основна програмска блока по једном функционалном програму. То су делови за иницијализацију (*INIT*), периодичну обраду (*CYCLE*) и део за излаз из периодичне обраде (*EXIT*).

Програм *Program_Example* је пресликан у Пајтон у виду класе, где су два програмска блока имплементирана преко конструктора и деструктора, док је трећи стандардна метода која се позива периодично у некој програмској примени.

Табела 6 - Пресликавање програмског блока

PROGRAM _INIT
    (* Code *)
END_PROGRAM
 	
PROGRAM _CYCLIC
    (* Code *)
END_PROGRAM
 	
PROGRAM _EXIT
    (* Code *)
END_PROGRAM
	class Program_Example:
	
	    def __init__(self):
	        print('_INIT CALLED')
	        return
	
	    def _CYCLIC(self):
	        return
	
	    def __del__(self):
	        print('_EXIT CALLED')
	        return

Структуирани Текст	Пајтон

Како би се могла описати функционалност програма, потребне су променљиве. У оквиру блока за променљиве могуће је поред декларације, дати и дефиницију променљивој.

Реализација различитих типова променљивих се своди на стандардни механизам променљивих у Пајтону и дефинишу се као поља класе програма. Самим тим што је у Структуираном Тексту могуће само декларисати променљиву, најсличније пресликавање у Пајтону представља коришћење кључне речи *None*. То решење се односи на све врсте типова. Како не постоји статичко заузимање одређене дужине низа у Пајтону, то је одрађено уз помоћ дефинисања листе са жељеним бројем елемената.

Због лакшег отклањања грешака, динамички се генерише додатна функција у оквиру програмске класе која служи за испис свих дефинисаних променљивих у оквиру тог програма. За њену имплементацију одрађена је редефиниција *__str__* методе класе.

Табела 7 - Пресликавање блока променљивих

VAR
    var_int: INT := 7;
    var_undef_int: INT;
    var_bool: BOOL := TRUE;
    var_real: REAL := 12.1;
    var_arr_int: ARRAY[0..2]
                OF INT := [3(5)];
    var_nodef_arr_int: ARRAY[0..4]
                OF INT;
END_VAR
	class Program_Example:
	    var_int = 7
	    var_undef_int = None
	    var_bool = True
	    var_real = 12.1
	    var_arr_int = [5, 5, 5]
	
	    var_nodef_arr_int = [None, None,
	                None, None, None]
	
	    # init, cycle and exit programs
	    # redefined __str__ method

Структуирани Текст	Пајтон

Реализација исказа нема никакве специфичности и слично је имплементирана, али користећи Пајтон синтаксу.

Поред простих типова, за декларисање променљивих постоје и сложени. Један вид сложених типова представљају структуре. Декларација променљивих у оквиру блока типова података је слична, као и у случају блока променљивих.

За реализацију ових блокова, коришћен је Пајтон модул *dataclasses* који служи за прављење класе али уз енкапсулацију тј. отклањање кода који није потребан за ову имплементацију. Модул нуди декларисање поља, заједно са типовима. Типови се могу дефинисати уз помоћ методе *field*, где није битно да ли је он простог или сложеног типа. Метода *field* поседује аргумент *default_factory* која омогућава дефинисање поља са променљивом (*mutable*) вредношћу. Како тај аргумент захтева објекат који може бити позван (*callable*) без аргумената, уколико се не ради о другој класи (нпр. листе), могуће је за дефинисање те листе користити *lambda* функцију без аргумената.

У склопу ових класа такође постоји динамички генерисана метода за испис која представља редефинисану методу *__str__*.

Табела 8 - Пресликавање блока сложених типова

Структуирани Текст	Пајтон

TYPE
Type_simple: STRUCT
x: INT := 0;
arr_nodef: ARRAY[0..1]OF INT;
 	
arr: ARRAY[0..5]OF UDINT
:= [2,0,4,2(0)];
END_STRUCT;
 	
Type_complex: STRUCT
var_string: STRING[80]
:= 'Hello';
var_complex: Type_simple;
END_STRUCT;
 	
END_TYPE
	@dataclass
	class Type_simple:
	x: int = 0
	arr_nodef: list[int] =
	field(default_factory =
	lambda: [None, None])
	arr: list[int] =
	field(default_factory =
	lambda: [2, 0, 4, 0, 0])
	# redefined __str__ method
	
	@dataclass
	class Type_complex:
	var_string: str = 'Hello'
	
	var_complex: Type_simple =
	field(default_factory =
	Type_simple)
	# redefined __str__ method

Функције су имплементиране слично као и у осталим језицима. Састоје се од декларације у оквиру библиотека, а одвојене су од своје дефиниције.

У оквиру декларације функције постоје улазне промењљиве (аргументи функције) и локалне промењљиве. Како се аргументи налазе у саставу декларације функције у Пајтону, тако се локалне промењљиве налазе на почетку тела дефиниције те функције. У зависности да ли су промењљиве дефинисане, зависи које вредности имају. Све функције имају повратну вредност, која се у Структуираном Тексту експлицитно дефинише тако што се имену функције додели жељена вредност. Исто је имплементирано и у преведеној функцији.

Табела 9 - Пресликавање дефиниције функција

Структуирани Текст	Пајтон

FUNCTION exponential_function
    (* Local vars: value, i *)
    (* Input vars: exponent, base *)
 	
    value := 1;
    IF base > 0 THEN
        FOR i := 1 TO exponent DO
            value := multiplication(
                        value, base);
 	
        END_FOR;
    END_IF
 	
    exponential_function := value;
 	
END_FUNCTION
	def exponential_function(exponent, base):
	    value = None
	    i = None
	
	    value = 1
	    if base > 0:
	        for i in range(1, exponent + 1):
	            value = multiplication(
	                        value, base)
	
	    exponential_function = value
	    return exponential_function

У оквиру Структуираног Текста, постоји специјалан тип функција, а то су функционални блокови и користе се као сложен тип променљиве. Они служе за статичко складиштење променљивих, тако да чува њихово стање тј. вредност кроз читав рад програма. Те специјалне променљиве се називају излазне (*Output*) променљиве. Њима је могуће приступити из било ког дела програма, без ограничења, али само за читање. Део функционалних блокова такође представља и сама дефиниција функције функционалног блока, попут обичних функција. Једина разлика је у томе што функционални блокови немају повратну вредност. Приступ пољима променљиве се обавља помоћу тачка оператора (нпр. променљива.излазна_променљива). Позив функције функционалног блока се обавља експлицитним позивом променљиве попут обичних функција (нпр. променљива(аргумент1 := вредност1,...))

Реализација је обављена уз помоћ Пајтон класа. Декларације променљивих постављене су и дефинисане као поља класе, док је сама функција функционалних блокова имплементирана у оквиру редефиниције методе *__call__*, како би објекат класе, односно променљива могла постати позивна (*callable*). У оквиру примера одрађен је позив дефинисане функције из претходног примера.

Табела 10 - Пресликавање дефиниције функционалног блока

Структуирани Текст	Пајтон

FUNCTION_BLOCK exponential_controller
(* Output var: exponential_value *)
 	
exponential_value :=
exponential_function(3, 2);
END_FUNCTION_BLOCK
	class exponential_controller:
	exponential_value = 0
	
	def __call__(self):
	self.exponential_value =
	exponential_function(3, 2);
	return

Библиотеке су реализоване појединачно, при чему скуп датотека Структуираног Текста једне библиотеке, представља један Пајтон модул. Једна библиотека се може састојати из више сегмената тј. датотека:

- Дефиниције сложених типова
- Дефиниције константних глобалних променљивих
- Декларације свих функција и функционалних блокова са њиховим променљивама
- Дефиниције свих функција и функционалних блокова

Реализација се своди на појединачно смештање свих претходно дефинисаних делова у оквиру једног Пајтон модула.

4.3.3 Ограничења реализације

Постоје неки делови који су реализовани на начин како би био што једноставнији за разумевање. Самим тим је било потребно изоставити неке функционалности, док је за друге остављен простор за будућа истраживања. Ти делови, иако постоји неки начин за имплементацију, због ауторове личне оцене сложености и корисности, изостављени су за ову тему рада.

Приликом дефинисања променљивих постоји различит избор особина за њих, као што су перзистентност, приступ меморији или чување стања и након отказивања машине. Пајтон не поседује механизам за дефинисање перзистентности променљиве. Приступ меморији није стандардна пракса због безбедности, коју Пајтон иначе обезбеђује аутоматски, али је могуће обавити приступ помоћу *ctypes* модула.

За сложене типове, типа структура, приликом појаве угњежденост типова (уколико је поље типа друга структура), где у оквиру Структуираног Текста, редослед дефинисања не представља проблем, док је у Пајтону супротно. Уколико је поље класе типа друга класа, која је истовремено дефинисана касније у коду, појавиће се извршна (*Runtime*) Пајтон грешка. Овај проблем настаје због начина на који је имплементиран преводилац, где се кроз један пролаз редослед кода, а самим тим и дефиниција одржава.

Код реализације функционалних блокова, ограничење је то што у Пајтону не постоји потпуна забрана приступа пољу класе ради уписа. Постоје механизми који то могу да ограниче до неке мере, али због практичности и једноставности реализације, они су изостављени.

Неки други проблем представљају прости типови података, као што су *TIME* и *DATE*, где њихова реализација није решена због аспекта физичке архитектуре. Време и датум се мењају периодично, тек након једног извршења програма који ради са најдужом периодом извршавања. Овај проблем ствара ограничење приликом тестирања брзине извршавања програма написаног у Структуираном Тексту.

Реализација интегрисаних функција је остављена за корисника преводиоца да је доврши. Сваки нефункционални сегмент преведеног кода је динамички назначен са коментаром за поправку (*#FIXME*).

4.4 Програмска реализација

У овом поглављу, биће објашњена програмска реализација преводиоца, не укључујући делове генерисања парсера и пре тога. Састоји се од три датотеке:

- *MyParser.py*
- *MyVisitor.py*
- *Translator.py*

4.4.1 MyParser.py

Овај модул садржи имплементацију самог преводиоца, као и друге помоћне функције за превођење различитих типова датотека. Постоје различити типови ST датотека у оквиру пројекта, као што су:

- *.st* - садржи дефиниције програма и функција
- *.var* - садржи дефиниције и декларације променљивих
- *.typ* - садржи неке сложене типове
- *.fun* - садржи декларације функција и функционалних блокова

Главна функција је функција *transpile,* која се позива приликом сваког превођења одређене датотеке, а остале су помоћне функције које се користе за обраду специфичних случајева улазне датотеке. Примена:

- *def* ***transpile*** *(input_file, function_arguments = [], is_library = False, function_decl = {}, is_library_const_file = False)* – служи за сам процес превођења једне датотеке тиме што иницира лексер, парсер и класа *Visitor*. Повратна вредност представља генерисан излазни код.
- *def* ***transpile_as_lib*** *(function_declarations, input_dir)* – служи за генерисање излазног кода библиотеке на основу декларација из *.fun* датотеке.
- *def* ***generate_function_code*** *(function_name, function_variables, definition, is_function_block = False)* – служи за генерисање излазног кода дефинисаних функција и функционалних блокова приликом превођења библиотека.

4.4.2 MyVisitor.py

Овај модул обавља пролаз кроз стабло парсирања, и на основу посете одређеног чвора врши генерисање кода по потреби. У оквиру класе овог модула, постоји око 80 редефинисаних метода које се позивају приликом посете сваког синтаксног правила у оквиру стабла које је такође дефинисано у формалној граматици. Због обимности, нису наведене њихове декларације, већ је објашњен уопштен начин њихових имплементација.

MyVisitor класа наслеђује *STVisitor* класу и ту се налазе редефиниције свих функција посете чворова. Класа садржи поље *target_code* које садржи генерисан код, који се допуњава приликом позива функција чворова током посете. У оквиру једне функције чвора, постоји аргумент који представља тренутни контекст. У оквиру контекста налазе се различита поља и методе везане за тај чвор попут информација који је предак чвора, који чворови су потомци, текстуални садржај тренутног чвора, итд. Контекст нуди методе за експлицитни позив посете чворова потомака у жељеном редоследу или приступ њиховим контекстима, као и разне друге.

def ***visitStart*** *(self, ctx:STParser.StartContext)* - По имплементацији, почетни чвор *start* је онај који се први позива приликом иницирања процеса превођења и који има повратну вредност генерисаног излазног кода.

4.4.3 Translator.py

Овај Пајтон модул служи за генерисање излазне структуре преводиоца, као и обраду различитих формата пројеката написаних на програмском језику Структуирани Текст. Тренутно имплементиране подршке су за:

- Пројекат где су све **ST** датотеке у оквиру једног директоријума без било какве хијерархије
- *Automation Studio* тип пројекта, где су сви програми одвојени у различите целине, док су библиотеке у оквиру свог посебног директоријума.

Постојеће функције у оквиру модула:

- *def translate (proj_type, project_dir, output_dir)* – служи за иницирање превођења једног пројекта
- *def translate_single_dir ()* – служи за превођење првог типа пројекта, где су све датотеке у оквиру истог директоријума тј. хијерархијски су једнаке
- *def translate_automation_studio_project (output_dir)* – служи за превођење другог типа пројекта, у *Automation Studio* формату
- *def translate_as_programs (packages, output_dir, lib_includes = "")* – служи за превођење **ST** програмских блокова
- *def load_as_program_list_root (dir, packages, program_list)* – служи за учитавање програма за превођење који нису садржани у оквиру неког пакета
- *def translate_as_libs (packages, output_dir)* – служи за превођење *Automation Studio* формата библиотека
- *def translate_as_functions (dir, lib_type)* – служи за превођење **ST** функција
- *def translate_as_types (dir)* – служи за превођење **ST** сложених типова
- *def translate_as_consts (dir)* – служи за превођење ST константних глобалних променљивих
- *def load_as_libs (dir, libraries, library_types)* – служи за учитавање постојећих библиотека ради превођења
- *def load_as_packages (dir, packages)* – служи за учитавање постојећих пакета програма ради превођења

5. Резултати

Као резултат пројекта, добијен је успешно преведен програм из Структуираног Текста у Пајтон са кључним функционалностима. Валидност програма тестирана је одређеним методологијама и резултати тог тестирања биће поменути у оквиру овог поглавља. Постоје делови Структуираног Текста који нису имплементирани у пројекту из разлога јер су ретко коришћене функционалности или због тежине еквивалентног превођења у програмски језик Пајтон. Преведен програм се успешно покреће и даје задовољавајуће резултате.

5.1 Тестирања

Због сложености тестирања преводиоца, поступак се морао поделити на више целина односно сегмената. Након валидирања сваког сегмента понаособ, уколико је тај део изолован од осталих може се са одређеном сигурношћу рећи да задовољава наше услове, а уколико два или више сегмента имају заједнички механизам рада, потребне су додатне методологије тестирања.

Временски, за овакав тип пројекта потребно је дугорочно тестирање, као и додатни број људи који ће своје вештине искористити за темељно валидирање и покушати да пронађе недостатке у постојећем програму. Особе са већим знањем свих могућности Структуираног Текста свакако имају предност у томе.

У случају овог конкретног пројекта, тестирање је одрађено темељно у складу са могућностима, јер узимајући у обзир пристрасност програмера тешко је са сигурношћу рећи у којој мери је имплементација валидна. Потешкоће такође ствара енкапсулација и ограниченост развојног окружења за Структуирани Текст. Самим тим, једна од могућих опција је ручно тестирање и визуелно поређење резултата. За аутоматизовано тестирање, био би потребан неки вид рада са датотекама и бележење излаза тестних случајева, али то у овој ситуацији није могуће због самих могућности алата *Automation Studio*.

Тестирање смо поделили у више сегмената које укључују:

1. Променљиве
2. Сложени типови података (структуре)
3. Библиотеке заједно са функцијама/функционалним блоковима, типовима и константним променљивама
4. Програми

У сваком сегменту је тестиран скуп могућности и варијација за сваки од концепата којима се тестира начин превођења и пресликавања неких специфичних функционалности из једног језика у други. Такође у склопу примера постоје и реални тестни случајеви које би програмер написао, којим се тестира валидност саме логике тј. какве резултате очекује.

5.1.1 Тестирање променљива

Први сегмент тестирања је обављен у оквиру 5 тестних примера, где су тестиране све могуће комбинације које графичка корисничка спрега (**GUI**) омогућава. Тестирање се показало успешно на свим тестовима, дајући очекиване резултате у складу са имплементацијом. Тестирана је валидност у следећим кључним случајевима и њиховим комбинацијама:

- Уколико нема ниједне променљиве
- Декларисање свих простих типова променљивих за које постоји подршка
- Декларисање променљивих са и без њене дефиниције
- Коришћење низова променљивих

Валидација исправности је обављена на основу визуелног поређења очекиваних резултата, као и ручним покретањем оба програма ради потврде функционалности. Везано за сегмент променљивих омогућено је и валидирање уз помоћ исписа стања

дефинисаних променљивих који је такође и доступан механизам у склопу главног програма. Стања променљивих забележена су у посебну датотеку (*Log File*).

5.1.2 Тестирање сложених типова података

Други сегмент тестирања је обављен у оквиру 6 тестних примера, где су тестиране све могуће комбинације коју графичка спрега омогућава. Кључне ставке тестирања и њихове комбинације:

- Уколико структура нема поља
- Декларисање структура чија су поља простог типа
- Декларисање структура, као и дефиниције њених поља
- Декларисање структура чија су поља типа низ
- Декларисање структура чија су поља типа друга структура
- Декларисање променљивих типа структура

Након тестирања свих 6 тестних случајева, показало се визуелном провером да резултати одговарају жељеној имплементацији. Ручно тестирање (покретање програма) се обавило над последњим тестним случајем који је у саставу имао све претходне примере, које се показало такође успешно. Овим тестирањем се такође приказао и рад променљивих чији тип је сложена структура. За овај сегмент је такође употребљен испис дефинисаних променљивих.

5.1.3 Тестирање библиотека и њених компоненти

Трећи и четврти сегмент су спојени у један из разлога што је за функционисање и употребу функција или функционалних блокова пожељно да оне буду имплементиране у склопу једне библиотеке. Кључне ставке овог тестирања, као и њихове комбинације, представљају следеће:

- Уколико функција/функционални блок нема променљиве, ни аргументе
- Декларисање функције која има аргументе и то произвољан број
- Дефинисање функције / функционалног блока
- Декларисање и дефинисање функција са и без променљивих
- Тестирање празних библиотека
- Тестирање константних променљива у склопу библиотека
- Тестирање типова у склопу библиотека
- Тестирање функција и функционалних блокова у склопу библиотека

Тестови су се показали успешним над свим примерима у складу са очекиваним исходом. Овај сегмент је ручно тестиран на једноставан начин уз помоћ понаособног позива свих елемената у оквиру једне библиотеке и поређењем исписа (функије, функционални блокови, типови, променљиве). Резултати су поређени са очекиваним и показали су се успешним. Тестирање је забележено у посебну датотеку.

5.1.4 Тестирање програмских блокова

Последњи, као и најважнији сегмент представља главни програм. Он је тестиран по неколико главних ставки, као и њиховим комбинацијама:

- Празна структура програма
- Тестирање свих постојећих исказа као и израза
- Тестирање петљи, условних исказа, као и њихово угњеждавање
- Коришћење библиотека
- Коришћење декларисаних и дефинисаних променљивих

Тестови са визуелном провером су се показали успешним над свим примерима, односно преводи се како је и очекивано. Проблем се појавио приликом тестирања програма који користе функције интегрисане у оквиру радног окружења *Automation Studio*. Ове програме није могуће тестирати покретањем, те ћемо у овом случају да их сматрамо да су без закључака (*inconclusive*).

5.1.5 Поређење брзине извршавања

Покушај мерења брзине извршавања програма написаног на програмском језику Структуирани Текст се показао неуспешним, самим тим је немогуће извршити поређење програма пре и након превођења. Ограничења и проблеми коришћења временских типова података за мерење времена су наведени у оквиру поглавља програмског решења. Самим тим што не постоји никакав механизам мерења протеклог времена у оквиру једне периоде програма, онемогућава било какво поређење брзине са преведеним Пајтон програмом. Покушај прављења посебног временског бројача се показао тешким за реализовати због недостатка доступне документације коришћеног алата. Потешкоћа је такође стварана због недостатка знања о могућностима физичких уређаја који су симулирани.

5.2 Резиме тестирања

У овом делу сумирани су резултати претходних тестирања, као и неки њихови описи. Резулати су се показали задовољавајућим и поузданим тј. даје одговарајуће резултате у различитим случајевима. Иако су сви тестови прошли визуелну проверу и дају задовољавајућ излаз, као тест успешности је стављен стриктнији захтев да програм мора бити успешно и покренут. Процентуално је тешко дати меру валидности рада, јер је број тестних примера за овакав рад потребно да буде знатно већи него тренутни. То је за сада неизводљиво због недостатка јавно доступних библиотека, што писање и извођење сложенијих тестова чини немогућим.

Табела 11 - Резиме тестирања

Циљани сегмент	**Број тестних случајева**	**Број успешних тестних случајева**	**Начин тестирања**	**Напомена о специфичности тестирања**
Променљиве у оквиру главног програма	5	5	Визуелно поређење и ручно тестирање (покретање)	Тестирање уз помоћ исписа променљивих
Структуре у оквиру главног програма	6	6	Визуелно поређење и ручно тестирање (покретање)	Тестирање уз помоћ исписа променљивих
Функције, функционални блокови, као и механизми библиотека	13	13	Визуелно поређење и ручно тестирање (покретање)	Тестирање ручним позивањем свих елемената библиотека
Главни програм са свим функционалностима	14	10	Визуелно поређење и ручно тестирање (покретање)	Тестирање уз помоћ покретања читавог програма

6. Закључак

Овим радом, приказано је једно решење оваквог типа програмског преводиоца. Корист овог преводиоца, осим што је добар почетни корак у модернизацији писања програма за **PLC** контролере, такође представља и користан алат за убрзавање процеса превођења већ постојећих пројеката.

Валидација решења је обављена у складу са могућностима и ограничењима коришћеног алата и потврђује исправност у одређеној мери. Иако нема могућности тестирања саме логике сложенијих тестних примера који се користе у реалном свету и даље омогућава превођење већег дела пројекта, а за делове који нису функционални или реализовани оставља се на програмеру да одлучи о даљим корацима. Такође постоје и ограничења приликом тестирања брзине улазног програма, што доводи до недостатка поређења брзине извршавања између два језика.

Делови пројекта који нису успешно реализовани, као нпр. питање интегрисаних функција, отварају пут за даљи истраживачки рад и реализацију. Иако са тренутно постојећом технологијом није могуће икада добити 100% еквивалентан код, без да се зна имплементација одређених библиотека, напредовање технологија односно машинског учења ће потенцијално омогућити приближавање том броју у некој мери

7. Литература

[1] "Programmable_logic_controller", 25 June 2024. [Online]. Available: https://en.wikipedia.org/wiki/Programmable_logic_controller.

[2] "IEC_61131-3", 10 April 2024. [Online]. Available: https://en.wikipedia.org/wiki/IEC_61131-3.

[3] Vladimir Kovačević, Miroslav Popović, Sistemska programska podrška u realnom vremenu 1, Novi Sad: FTN Izdavaštvo, 2011.

[4] Maggie Johnson, Julie Zelenski, „Formal Grammars“, Stanford University, California, 2012.

[5] "Parsing", 10 June 2024. [Online]. Available: https://en.wikipedia.org/wiki/Parsing.

[6] T. Parr, The Definitive ANTLR 4 Reference 2nd Edition, Pragmatic Bookshelf, 2013.

[7] „Structured Text Programming“, PDHonline Course E334 (3 PDH), 2020.