Документацията е разделена на следните части:

1. Увод

2. Синтаксис на езика

3-4. Реализация на семантиката на езика

5. Обща архитектура

6-7. Имплементация

8. Заключение

**Увод**

Езикът за програмиране EXPR се характеризира прост и интуитивен синтаксис. Основната цел на този проект е да се построи интерпретатор език, който позволява изпълнението на програми. Също така проектът има за задача синтаксисът на базовия език да бъде разширен, за да даде възможност за използването на конструкции от по-високо ниво.

Основните цели, които биват реализирани в проекта, са следните:

* Лесен за използване интерфейс под формата на конзола.
* Прочитане на файлове с код и проверка за синтактична коректност на съдържанието им. Използване на стандартно разширение .EXPR за файловете с код на езика EXPR, въпреки че интерпретаторът може да чете текстови файлове с всякакво разширение.
* Изпълнение на кода в съответствие със семантичния смисъл на командите.
* Даване на информация за възникнали грешки преди и по време на изпълнение на програмата.
* Възможност за работа с произволно дълги цели неотрицателни числа и изпълнение на съответните аритметични операции.
* Реализиране на булеви изрази и условни оператори if.
* Реализиране на цикли while.
* Реализиране на рекурсивни функции и по-голяма свобода при дефинирането на функции като цяло.
* Реализиране на стекови рамки и локални дефиниции на променливи.

**Синтаксис на езика**

Синтаксисът, който бива разпознаван от компилатора, представлява разширение върху базовият синтаксис:

Sequence → Line **`\n`** Sequence | IfStatement **`\n`** Sequence | WhileStatement **`\n`** Sequence | RecDefinition **`\n`** Sequence |

IfStatement → if **`\n`** Cond **`\n`** then **`\n`** Sequence **`\n`** else **`\n`** Sequence **`\n`** endif

WhileStatement → while **`\n`** Cond **`\n`** Sequence **`\n`** endwhile

RecDefinition → recdef **`\n`** Fun**[**Var**]** **`\n`** Sequence **`\n`** endrecdef

Cond → **true** | **false** | **!(**Cond**)** | **(**Cond **&&** Cond**)** | **(**Cond **||** Cond**)** | **(**Expr **<** Expr**)** | **(**Expr **>** Expr**)** | **(**Expr **==** Expr**)**

Line → Var **=** Expr | Fun**[**Var**]** **=** Expr | **print** Expr | **read** Var | **return** Expr

Fun → **A** | **B** | … | **Z** | **A**Fun | **B**Fun | … | **Z**Fun

Var → **a** | **b** | … | **z** | **a**Var | **b**Var | … | **z**Var  
Num → **0** | … | **9** | **1**Num | … | **9**Num  
Expr → Expr **+** Term | Expr **-** Term | Term  
Term → Term **\*** Factor | Term **/** Factor | Term **%** Factor | Factor  
Factor → Var | Num | **(**Expr**)** | Fun**[**Expr**]**

Една програма е синтактично валидна и се изпълнява от компилатора точно когато може да се сведе до израз от тип Sequence.

Операторът **return** е особен с това, че е разрешено да се среща само след израз recdef и преди израз enrecdef. Всяка друга употреба се смята за синтактично невалидна от компилатора.

**Реализация на семантиката на езика**

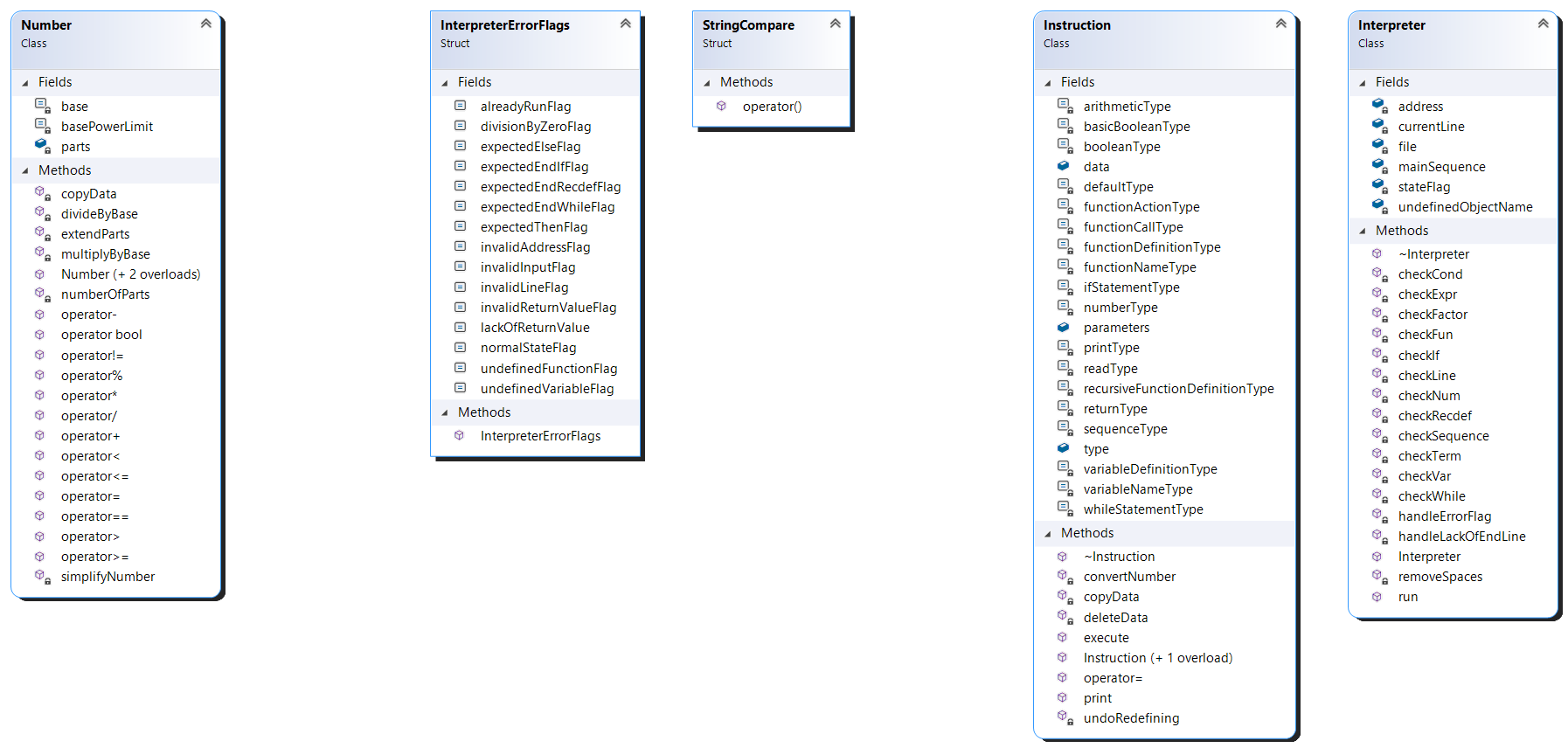
При започване на изпълнението на програмата по подразбиране се създава една основна стекова рамка.

Синтактичните обекти се интерпретират по следния начин:

* Sequence – Последователност от команди, които биват изпълнявани подред.
* IfStatement – Изразът Cond се оценява до истина или лъжа. Ако стойността е истина, то се изпълнява първият обект Sequence. В противен случай се изпълнява вторият обект Sequence.
* WhileStatement – Изразът Cond се оценява до истина или лъжа. Ако той е истина, то се изпълнява изразът Sequence и програмата отново се връща към същата проверката на израза Cond. Ако е лъжа, то инструкцията приключва.
* RecDefinition – Изразът Fun**[**Var**]** се интерпретира като дефиниция на функцията с име Fun, в която параметър е променливата с име Var. Изразът Sequence бива изпълняван като, ако по някое време се попадне на оператор от тип **return**, то изпълнението на Sequence бива спряно преждевременно и функцията връща стойността на оценения от оператора израз. При приключване на изпълнението на Sequence без да се попадне на оператор **return**, програмата извежда подходящо съобщение за грешка. Ако в съответната стекова рамка няма дефинирана функция с това име, то се създава нова такава, която да изпълнява зададената процедура. В противен случай се презаписва съществуващата локална дефиниция на функцията.
* Cond – Израз, който може да бъде оценен до истина или лъжа. Базовите стойности **true** и **false** съответно се оценяват до истина и лъжа. Операторите **!**, **&&** и **||** се интерпретират съответно като негация, конюнкция и дизюнкция, докато **<**, **>** и **==** се интерпретират съответно като строги неравенства и проверка за равенство. Ако операторът е **&&** или **||**, то оценява само първият аргумент. Ако той е лъжа в случая на **&&** или истина в случая на **||**, то директно се връща съответно стойност лъжа и истина. В останалите случай се оценяват всички аргументи преди да се приложи съответната булева операция.
* Line – Имаме следните случаи:
  + Изразът “Var **=** Expr” се интерпретира като първо се оценява изразът Expr и стойността му се дава на променливата Var. Ако в съответната стекова рамка няма дефинирана променлива с това име, то се създава нова такава, която приема тази стойност. В противен случай се презаписва стойността на съществуващата локална променлива.
  + Изразът “Fun**[**Var**]** **=** Expr” се интерпретира като дефиниция на функцията Fun, в която параметър е променливата с име Var. Резултатът от изпълнението на функцията е стойността от оценяването на израза Expr. Ако в съответната стекова рамка няма дефинирана функция с това име, то се създава нова такава, която да изпълнява зададената процедура. В противен случай се презаписва съществуващата локална дефиниция на функцията.
  + Изразът “**print** Expr” се интерпретира като първо се оценява изразът Expr и стойността бива принтирана на нов ред в конзолата.
  + Изразът “**read** Var” се интерпретира като се очаква потребителят да въведе цяло неотрицателно число, което да се даде като стойност на променливата Var. Ако се въведе низ, който не е число, то програмата дава подходящо съобщение за грешка. Ако в съответната стекова рамка няма дефинирана променлива с това име, то се създава нова такава, която приема тази стойност. В противен случай се презаписва стойността на съществуващата локална променлива.
  + Изразът “**return** Expr” се интерпретира като първо се оценява стойността на израза Expr и се използва като стойност на връщане за функцията, създала текущата стекова рамка.
* Var – Интерпретира се като име на променлива. Взима се стойността от “най-близката” стекова рамка, в която има дефинирана променлива с това име. При липсата на такава стекова рамка се дава подходящо съобщение за грешка.
* Fun – Интерпретира се като име на функция. Взима се процедурата от “най-близката” стекова рамка, в която има дефинирана функция с това име. При липсата на такава стекова рамка се дава подходящо съобщение за грешка.
* Num – Интерпретира се като цяло неотрицателно число.
* Expr – Оценяват се стойностите на аргументите и се прилага съответната операция (ако има такава). + и – се интерпретират съответно като събиране и изваждане. Поради работата на програмата само с неотрицателни числа, ако от по-малко число бъде извадено по-голямо, то резултатът ще бъде 0.
* Term – Оценяват се стойностите на аргументите и се прилага съответната операция (ако има такава). \*, / и % се интерпретират съответно като умножение, целочислено деление и остатък при деление. Ако в случай на / или % вторият аргумент има стойност 0, то програмата връща подходящо съобщение за грешка.
* Factor – Имаме следните случаи:
  + Изразите Var и Num се оценяват по вече описаните правила, а (Expr) се оценява като се оцени изразът Expr.
  + Изразът Fun**[**Expr**]** се интерпретира като първо се оценява изразът Expr. След това се взима процедурата, която съответства на функцията Fun по вече описаното правило. Създава се нова стекова рамка, в която за променливата Var (която е параметъра на съответната функция) се създава локална дефиниция. Като стойност на тази локална променлива се дава стойността от оценката на Expr.

Целият израз бива оценяван до върнатия от функцията резултат.

**Обща архитектура**



Архитектурата се състои от следните части:

* **class Number –** Реализация на цели неотрицателни числа с произволна дължина. Има вградени оператори за аритметични операции и за сравнение.
* **struct InterpreterErrorFlag** – Под формата на статични променливи от тип **char** са дефинирани стойности за грешки, които могат да възникнат при работата на интерпретатора.
* **struct StringCompare** – Структура за сравнение на елементи от тип **std::string**. Нужна е при дефинирането на някои обекти с шаблонен тип **std::map**.
* **class Instruction –** Реализира различните видове инструкции и променливи по време на изпълнение на дадена програма. Представлява вид полиморфизъм. Също така разполага с метод **execute** за изпълнение на исканата функционалност и няколко помощни метода. Съдържа динамичен масив от тип **vector<Instruction>**, пазещ параметрите на изпълняваната операция, а също така и указател от тип **void\***, пазещ данни от произволен тип.
* **class Interpreter –** Представя интерпретатора. Има парсер, който се реализира чрез редица помощни функции за разпознаване на синтактичните обекти, а също така разполага и с метод **handleErrorFlag** извеждане на подходящо съобщение при възникване на грешка при четенето или изпълнението на програмата. Съдържа член данни, които запаметяват информацията за настъпилата грешка. Самото изпълнение на интерпретатора се стартира чрез метода **run**.

**Имплементация**

Помощни преименувания на типове:

* DEFINITIONS = std::map<std::string, Instruction, StringCompare> - Използва се за пазене на най-скорошните дефиниции на променливите и функциите.
* DEFINED = std::map<std::string, bool, StringCompare> - Използва се за помнене кои локални дефиниции са налични в текущата стекова рамка.
* REDEFINED = std::stack<std::pair<std::string, Instruction>> - Използва се за помнене на старите стойностите на променливите и функциите в по-горните стекови рамки.

Реализация на основните структури и класове:

* **class Number** – Числата се представят като последователност от динамичен масив от **unsigned int** елементи. Всеки елемент е число от 0 до 109-1 включително като елементът с нулев индекс се умножава с 1, а при преминаване до индекс по-голям с единица, записаният елемент се умножава с 109 в повече спрямо предходния. Крайното число е сума от всички тези умножения.
* **struct InterpreterErrorFlag –** Класът няма конструктор и се ползва само за пазенето на статични данни.
* **struct StringCompare –** Използва се лексикографска наредба за сравнение.
* **class Instruction –**Методите имат следната функционалност:
  + **convertNumber –** Преобразува **std::string** в **Number**.
  + **undoRedefining –** Възстановява конкретен брой от старите стойности на променливите и функциите. Използва се при изтриване на стекова рамка.
  + **execute –** извършва вече описаната реализация на семантиката в зависимост от типа. Параметърът от тип **char&** се използва за указване състоянието на програмата, **std::string&** за името на проблематична променлива или функция, **DEFINITIONS&**, **DEFINED&** и **REDEFINED&** изпълняват гореописаната цел, **int&** за брой на предефинираните обекти в текущата стекова рамка, а **bool&** и **Number&** за наличие на върната стойност и съответно самата стойност.
  + **print –** Извежда инструкциите в четима форма (близка до синтаксиса).

Самият тип на инструкцията се определя от променливата **std::string type** като различните типове имат следната стандартна структура:

* + **defaultType** – стандартен тип за още недефиниран обект.
  + **sequenceType** – Динамичният масив съдържа произволен брой инструкции за изпълнение.
  + **ifStatementType** – Динамичният масив съдържа три елемента като първият е условието, вторият е редицата за изпълнение при истина, а третият редицата за изпълнение при лъжа.
  + **whileStatement** – Динамичният масив съдържа два елемента като първият е условието, а вторият е редицата за изпълнение.
  + **readType** – Динамичният съдържа един елемент, който е променливата, която трябва да се прочете.
  + **printType** – Динамичният масив съдържа един елемент, която е изразът, чиято стойност трябва да се изведе.
  + **returnType** – Динамичният масив съдържа една стойност, която е изразът, чиято стойност трябва да се върне.
  + **booleanType** – Динамичният масив съдържа една или две стойности, които са аргументи, а **data** сочи към тип **char**, който указва булевата операция за извършване.
  + **arithmeticType** – Динамичният масив съдържа две стойности, които са аргументи, а **data** сочи към тип **char**, който указва аритметичната операция за извършване.
  + **basicBooleanType** – Динамичният масив е празен, а **data** сочи към тип **bool**, който може да има стойност true или false.
  + **numberType** – Динамичният масив е празен, а **data** сочи към тип **Number**, който представлява стойността.
  + **variableNameType** – Динамичният масив е празен, а **data** сочи към тип **std::string**, който представлява името на променливата.
  + **functionNameType** – Динамичният масив е празен, а **data** сочи към тип **std::string**, който представлява името на функцията.
  + **variableDefinitionType** – Динамичният масив съдържа два елемента като първият е променливана, която трябва да получи стойност, а вторият е аритметичният израз.
  + **functionDefinitionType** – Динамичният масив съдържа три елемента като първият е името на функцията, която се дефинира, вторият е променливата параметър, а третият е аритметичният израз.
  + **recursiveFunctionDefinitionType** – Динамичният масив съдържа три елемента като първият е името на функцията, която се дефинира, вторият е променливата параметър, а третият е последователността от инструкции.
  + **functionCallType** – Динамичният масив съдържа два елемента като първият е името на извиканата функция, а вторият е аритметичният израз, който се аргумент на функцията.
  + **functionActionType –** Динамичният масив съдържа два елемента като първият е името на променливата параметър, а вторият е аритметичен израз или последователност от инструкции, които трябва да върнат стойност.
* **class Interpreter –** Основните методи имат следната функционалност:
  + **removeSpaces –** Увеличава левия коефициент и намалява десния, докато левият не надвиши десния или докато не сочат непразни символи в низа.
  + **handleLackOfEndLine –** обработва вида грешка при липса на затварящ израз **endif**, **endwhile** или **endrecdef**.
  + **handleErrorFlag –** Извежда подходящо съобщение в зависимост от грешката, която е записана в променливата **char stateFlag**.
  + **run –** стартира интерпретатора като очаква валиден адрес към файла, където е записана програмата. При невалиден адрес се преминава в подходяща грешка.

**Заключение**

Крайните резултати от проекта са следните:

* Успешно бяха реализирани всички задачи по проекта. Това включва:
  + Прочитане на файлове с програми, написани на езика EXPR, и коректно разпознаване на валиден синтаксис, а също така и засичане на основните грешки при наличие на невалиден синтаксис.
  + Коректно изпълнение на инструкциите подред в съответствие със семантиката на езика EXPR.
  + Извеждане на информативни съобщения при засичане на разнообразни проблеми както преди, така и по време на изпълнение на програмата.
* Също така бяха реализирани и следните допълнителни функционалности:
  + Възможност за работа с произволно дълги неотрицателни числа и поддържане на основните аритметични операции.
  + Възможност за работа с булеви изрази и булеви оператори, а също така и с основните оператори за сравнение на числа.
  + Въвеждане на стекови рамки и възможност за задълбочена работа с локални дефиниции на променливи и функции чрез **recdef** конструкция.
  + Възможност за работа с условни оператори чрез въвеждане на **if** конструкция.
  + Възможност за работа с цикли чрез въвеждане на **while** конструкция.
  + Възможност за дефиниране на рекурсивни функции чрез въвеждане на **recdef** конструкция, а също така и разширяване на възможностите за дефинирането и приложението на функции чрез въвеждане на **return** оператор.
* Други функционалности, който могат да бъдат реализирани в бъдеще, са следните:
  + Разширяване на синтаксиса, което да даде възможност за задълбочена работа със стекови рамки по начини, различни от използването на **recdef** конструкция.
  + Разширяване на синтаксиса чрез въвеждане на конструкция за **for** цикли.
  + Разширяване на синтаксиса, което да позволи дефинирането на функции на произволен брой променливи.
  + Опростяване на синтаксиса по отношение на булевите изрази и **if** конструкцията с цел по-интуитивно и по-лесно използване на езика EXPR.
  + Промяна на имплементацията чрез по-задълбочено използване на указатели и полиморфизъм с цел по-бърза работа на интерпретатора.
  + Разширяване на поддържаните типове данни.

Окончателно би могло да се заключи, че проектът е успешно реализиран.