Техническая документация для языка MAK++

Разработчики: Радаева Карина, Трубников Максим

20 апреля 2021 г.

Содержание

1 Введение

Данная документация предназначена для разработчиков языка МАК++ и содержит основную информацию о работе компилятора. Язык предназначен для обучения и является "урезанной"версией языка C++ с некоторыми добавлениями.

2 Формальная грамматика

- <**Программа**>::= <**Функция**> { <**Функция**> } | <**Процедура**> { <**Процедура**> } | <**переменная**> ';' | main '(' ')' <**составной оператор**>
- <**Функция**>::= <**тип**> <**имя**> <**список параметров**> <**составной оператор**>
- <**Процедура**>::='def' <**имя**> <**список параметров**> <**составной оператор**>
- <**список параметров**>::= '(' <**переменная**> { ',' <**переменная**> } ')'
- <**Переменная**>::= <**тип**> <**имя**>
- <**имя**>::=<**буква**> | <**имя**> <**буква**> | <**имя**> <**цифра**>
- <**тип**>::= 'int' | 'boolean' | 'double' | 'string'
- <**константа**>::= <**число**> | <**логическое значение**> | <**строка**>
- <**число**>::= <**знак**> <**цифра**> {<**цифра**>} | <**цифра**> {<**цифра**>} '.' <**цифра**> {<**цифра**>}
- <**логическое значение**>::= 'true' | 'false'
- <**строка**> ::= "{<**символ**>}"
- <**составной оператор**>::= '{' { <**оператор**> {<**оператор**>} } '}'
- <**оператор**>::= <**оператор ввода**> | <**оператор вывода**> | <**оператор выражение**> ';' | <**вызов процедуры**> | <**спец. оператор**> | <**вызов функции**>
- <**оператор ввода**>::= 'read' '(' <**имя**> ')' ';'
- <**оператор вывода**>::='write' '(' [<**имя**> | <**строка**>] { ',' [<**имя**> | <**строка**>] }')' ';'
- <**оператор выражения**>::= <**имя**><**приоритет_1**><**приоритет_2**> | <**переменная**> '=' <**приоритет_2**> | <**приоритет_2**>
- <**приоритет_1**>::= '=' | '+=' | '-=' | '*=' | '/=' | '%='
- <**приоритет_2**>::= <**приоритет_3**> {<**знак_2**> <**приоритет_3**>}
- <**знак_2**>::= '||'
- <**приоритет_3**>::= <**приоритет_4**> {<**знак_3**><**приоритет_4**>}

- <**знак_3**>::= ‘&&’
- <**приоритет_4**>::= <**приоритет_5**> {<**знак_4**><**приоритет_5**>}
- <**знак_4**>::= ‘|’
- <**приорите_5**>::= <**приоритет_6**> {<**знак_5**><**приоритет_6**>}
- <**знак_5**>::= ‘^’
- <**приорите_6**>::= <**приоритет_7**> {<**знак_6**><**приоритет_7**>}
- <**знак_6**>::= ‘&’
- <**приоритет_7**>::= <**приоритет_8**> {<**знак_7**><**приоритет_8**>}
- <**знак_7**>::= ‘==’ | ‘<>’
- <**приоритет_8**>::= <**приоритет_9**> {<**знак_8**><**приоритет_9**>}
- <**знак_8**>::= ‘<’ | ‘>’ | ‘<=’ | ‘>=’
- <**приоритет_9**>::= <**приоритет_10**> {<**знак_9**><**приоритет_10**>}
- <**знак_9**>::= ‘«’ | ‘»’
- <**приоритет_10**>::= <**приоритет_11**> {<**знак_10**><**приоритет_11**>}
- <**знак_10**>::= ‘+’ | ‘-‘
- <**приоритет_11**>::= <**приоритет_12**> {<**знак_11**><**приоритет_12**>}
- <**знак_11**>::= ‘*’ | ‘/’ | ‘%’
- <**приоритет_12**>::= <**знак_12**><**приоритет_13**> | <**приоритет_13**>
- <**знак_12**>::= ‘!’
- <**приоритет_13**>::= <**знак_13**><**приоритет_14**> | <**приоритет_14**>
- <**знак_13**>::= ‘++’ | ‘–’
- <**приоритет_14**> ::= <**приоритет_15**> {<**знак_14**><**приоритет_15**>}
- <**знак_14**> ::= ‘-‘ | ‘+’
- <**приоритет_15**> ::= <**константа**> | ‘(‘ <**приоритет_2**> ‘)’ | <**вызов функции**> | <**имя**>
- <**вызов функции**>::= <**имя**> ‘(‘<**передаваемые параметры**> { ‘,’<**передаваемые параметры**> }‘)’

- **<передаваемые параметры>**::= **<имя>** | **<константа>**
- **<вызов процедуры>**::=**<имя>** ‘(‘<передаваемые параметры> { **<передаваемые параметры>** }‘)”;’
- **<спец. оператор>**::= **<оператор цикла>** | **<условный оператор>** | **<goto и дети>**
- **<оператор цикла>**::= **<for>** | **<elfor>** | **<while>** | **<dowhile>**
- **<for>**::= ‘for”(‘**<переменная>**’;’**<оператор выражения>**’;’ **<оператор выражения>**’)’**<составной оператор>** | ‘for”(‘**<имя>**’;’**<оператор выражения>**’;’ **<оператор выражения>**’)’**<составной оператор>**
- **<elfor>**::=‘elfor’ ’(‘ **<переменная>**’;’**<оператор выражения>**’;’ **<оператор выражения>**’)’**<составной оператор>** **<else>** | ‘elfor’ ’(‘**<имя>**’;’**<оператор выражения>**’;’ **<оператор выражения>**’)’**<составной оператор>** **<else>**
- **<else>**::= ‘else’ **<составной оператор>**
- **<while>**::=’while”(‘**<оператор выражения>**’)’**<составной оператор>**
- **<dowhile>**::=’do’**<составной оператор>**’while’(‘**<оператор выражения>**’)’ ’;’
- **<условный оператор>**::=’if”(‘<оператор выражения>’)’ <составной оператор> <ветвление условного оператора> <ветвление условного оператора>
- **<ветвление условного оператора>**::= ‘else’ **<составной оператор>** | ‘elseif’ ‘(‘**<оператор выражения>**’)’**<составной оператор>**
- **<goto и дети>** ::= **<return>** | **<break>**
- **<return>**::= ‘return’ **<оператор выражения>**’;’
- **<break>**::= ‘break’ ’;’

3 Файлы проекта

3.1 Используемые библиотеки

- **SQL** — предназначен для работы с базой данных SQLITE версии 3. Состоит из файлов: **sqlite3.c** и **sqlite3.h**

3.2 Файлы ресурсов

- **errors_list.txt** — файл содержащий расшифровку ошибок. Подробнее об ошибках смотрите в пункте **??**

3.3 Заголовочные файлы (*.p)

- **Constant.h** — содержит програмные константы и коды ошибок для использование в коде
- **MainCompler.h** — заголовочный файл класса *MainCompiler* (описание в пункте 5.7)
- **Types.h** — заголовочный файл класса *SemanticAnalyzer* (описание в пункте 5.7)
- **LexicalAnalyzer.h** — заголовочный файл класса *LexicalAnalyzer* (описание в пункте 5.7)
- **SyntaxAnalyzer.h** — заголовочный файл класса *SyntaxAnalyzer* (описание в пункте 5.7)
- **SemanticAnalyzer.h** — заголовочный файл класса *SemanticAnalyzer* (описание в пункте 5.7)
- **PolizItem.h** — заголовочный файл класса *PolizItem* (описание в пункте 5.7)
- **GenCode.h** — заголовочный файл класса *GenCode* (описание в пункте 5.7)

3.4 Исходные файлы (*.cpp)

- **MainCompiler.cpp** — исходный файл класса *MainCompiler* (описание в пункте 5.7)
- **Types.cpp** — исходный файл класса *Types* (описание в пункте 5.7)
- **LexicalAnalyzer.cpp** — исходный файл класса *LexicalAnalyzer* (описание в пункте 5.7)
- **SyntaxAnalyzer.cpp** — исходный файл класса *SyntaxAnalyzer* (описание в пункте 5.7)

- **SemanticAnalyzer.cpp** — исходный файл класса *SemanticAnalyzer* (описание в пункте 5.7)
- **PolizItem.cpp** — исходный файл класса *PolizItem* (описание в пункте 5.7)
- **GenCode.cpp** — исходный файл класса *GenCode* (описание в пункте 5.7)

4 Существующие классы

4.1 Описание задач классов

- **MainCompiler** — отвечает за: последовательный вызов функций анализа текста программы; компиляцию; создание и обновление базы данных зарезервированных слов и символов; создание и обновление базы данных ошибок.
- **Types** — все структуры данных компилятора. От него наследуются классы: SemanticAnalyzer, PollizItem, GenCode.
- **LexicalAnalyzer** — разбор текста прграммы на токены.
- **SyntaxAnalyzer** — отвечает за синтаксический анализ токенов, полученных от лексического анализатора. Внутри в процессе анализа также происходит семантический анализ в определенных местах. Рекурсивный спуск.
- **SemanticAnalyzer** — отвечает за семантический анализ; генераций POLIZ'а выражений для транслятора.
- **GenCode** — отвечает за генерацию и исполнение программы.
- **PolizItem** — тип, из которого состоит POLIZ программы.

5 Методы/функции классов

5.1 Методы/функции класса MainCompiler

- **Public:**
  - **void StartCompilation(int)** — вызов выбраного файла для компиляции или обновление БД. Вход: код файла для обработки. Выход: void
- **Private:**
  - **void CreateReserverWordsTable_()** — создание таблицы БД зарезервированных слов. Вход: -. Выход: void.

- **void UpdateTableReservedWords_()** — Обновление таблицы зарезервированных слов в БД. Вход: -. Выход: void.
- **void CreateErrorsTable_()** — создание таблицы ошибок в БД. Вход: -. Выход: void.
- **void UpdateErrorsTable_()** — обновление таблицы ошибок в БД. Вход: -. Выход: void.
- **std::string getTextErrorByCode(short int)** — получение текста ошибки из БД по коду. Вход: код ошибки. Выход: текст ошибки.
- **std::vector<std::pair<int, std::string» StartLexicalAnalyze_(std::string)** — открытие нужного файла, чтение и запуск лексического анализатора. Вход: текст программы считанный из файла. Выход: вектор пар типа: код типа лексемы, лексема.
- **void StartSyntaxAnalyzer_(std::vector<std::pair<int, std::string»)** — запуск синтаксического анализатора по полученым токенам. Вход: вектор пар типа: код типа лексемы, лексема. Выход: -.

5.2 Методы/функции класса LexicalAnalyzer

- **Public:**
  - **std::vector<std::pair<int, std::string» analyze(std::string&)** — разбор текста программы с помощью конечного автомата(КА) на токены, которым разделяются на определенные группы(см. пункт группы лексем). Вход: текст программы. Выход: вектор пар типа: код типа лексемы, лексема. Алгоритм КА.
- **Private:**
  - **int getTypeCategory(std::string token)** — получение категории токена. Вход: токен. Выход: код группы токена.
  - **void LoadDataBase()** — загрузка таблицы из БД зарезервированных слов. Вход: -. Выход: void.

5.3 Методы/функции класса SyntaxAnalyzer

- **Public:**
  - **void StartSyntaxAnalyzer()** — запуск синтаксического анализатора + try...catch() ошибок + подготовка токенов к анализу(подсчет слов в строке для определения строки с возможной ошибкой). Вход: -. Выход: void.
  - **SyntaxAnalyzer(std::vector<std::pair<int, std::string», GenCode&)** — конструктор. Вход: вектор пар типа: код типа лексемы, лексема; ссылка на генератор POLIZ'а.

- **SyntaxAnalyzer()** — деструктор.

- **Private:**
  - **std::string getLineOfError_()** — поиск номера строки, на которой ошибка. Вход: -. Выход: номер строки, содержащей ошибку.
  - **void getReadyToken_()** — подготовка вектора токенов к анализу(убираем токены перевода строки и сохраняем количество слов в строке для получения номера строки где ошибка). Вход: -. Выход: void.
  - **void AnalyzeProgramm_()** — рекурсивнй разбор понятия <Программа> (см. Грамматику). Вход: -. Выход: void.
  - **void AnalyzeGlobalIdentifierOrFunc_()** — разбор глобальной переменной или идентификатора. Вход: -. Выход: void.
  - **void AnalyzeParamsOfFuncProcedure_(bool, bool)** — анализ параметров объявления функции(метода). Вход: был ли до этого идентификтаор; была ли до этого открывающаяся скобка. Выход: void.
  - **void AnalyzeParamsMain_()** — разбор и анализ парметров main. Вход: -. Выход: void.
  - **void AnalyzeIdentificator_()** — разбор и анализ идентификатор(тип + имя). Вход: -. Выход: void.
  - **void AnalyzeName_()** — разбор и анализ имени. Вход: -. Выход: void.
  - **void AnalyzeServiceType_()** — разбор и анализ типа. Вход: -. Выход: void.
  - **void AnalyzeCompoundOperator_(bool)** — рекурсивнй разбор понятия <составной оператор> (см. Грамматику). Вход: была ли открывающаяся фигурная скобка. Выход: -.
  - **void AnalyzeOperators_()** — рекурсивнй разбор понятия <оператор> (см. Грамматику). Вход: -. Выход: void.
  - **void AnalyzeOperatorInput_()** — рекурсивнй разбор понятия <оператор ввода> (см. Грамматику). Вход: -. Выход: void.
  - **void AnalyzeOperatorOutput_()** — рекурсивнй разбор понятия <оператора вывода> (см. Грамматику). Вход: -. Выход: void.
  - **void AnalyzeParamsOperatorOutput_(bool)** — разбор и анализ параметров оператора вывода. Вход: ожидается ли запятая. Выход: void.
  - **short int AnalyzeOperatorExpression_()** — рекурсивнй разбор понятия <оператор выражения> (см. Грамматику). Вход: -. Выход: код типа результата выражения.

- **void AnalyzePriorityAssignable_()** — разбор и анализ операций присвоения. Вход: -. Выход: void.
- **void AnalyzePriority_1_()** — рекурсивнй разбор понятия <**приоритет_1**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_2_()** — рекурсивнй разбор понятия <**приоритет_2**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_3_()** — рекурсивнй разбор понятия <**приоритет_3**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_4_()** — рекурсивнй разбор понятия <**приоритет_4**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_5_()** — рекурсивнй разбор понятия <**приоритет_5**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_6_()** — рекурсивнй разбор понятия <**приоритет_6**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_7_()** — рекурсивнй разбор понятия <**приоритет_7**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_8_()** — рекурсивнй разбор понятия <**приоритет_8**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_9_()** — рекурсивнй разбор понятия <**приоритет_9**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_10_()** — рекурсивнй разбор понятия <**приоритет_10**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_11_()** — рекурсивнй разбор понятия <**приоритет_11**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_12_()** — рекурсивнй разбор понятия <**приоритет_12**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_13_()** — рекурсивнй разбор понятия <**приоритет_13**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_14_()** — рекурсивнй разбор понятия <**приоритет_14**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzePriority_15_()** — рекурсивнй разбор понятия <**приоритет_15**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzeCallFuncOrIdentifier_()** — анализ вызова функции(метода). Вход: -. Выход: void.
- **void AnalyzeCallFuncParams_(bool, std::string, int)** — анализ параметров вызова функции(метода). Вход: ожидается ли запятая; имя вызываемой функции; кол-во параметров. Выход: void.
- **void AnalyzeOperatorWhile_()** — рекурсивнй разбор понятия <**while**> (см. Грамматику). Вход: -. Выход: void.

- **void AnalyzeOperatorIf_()** — рекурсивнй разбор понятия <**условный оператор**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzeOperatorDoWhile_()** — рекурсивнй разбор понятия <**dowhile**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzeOperatorFor_()** — рекурсивнй разбор понятия <**for**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzeParamsFor_(short int)** — анализ параметров for/elfor. Вход: номер параметра для анализа. Выход: -.
- **void AnalyzeOperatorElFor_()** — рекурсивнй разбор понятия <**elfor**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzeOperatorElseElFor_()** — рекурсивнй разбор понятия <**else**> (см. Грамматику). Вход: -. Выход: void.
- **void AnalyzeOperatorReturn_()** — рекурсивнй разбор понятия <**return**> (см. Грамматику). Вход: -. Выход: void.
- **void ChechCorrectName_(std::string)** — анализ имени на корректность(см. Правила именования переменных).
- **void CheckCorrectConstant_(short int)** — анализ корректности констант. Вход: тип константы. Выход: void.
- **bool CheckCorrectNumberConstant_()** — анализ корректности числовой констнаты. Вход: -. Выход: true, если корректно, false, если нет.
- **bool CheckCorrectLogicalConstant_()** — анализ корректности логических констант. Вход: -. Выход: true, если корректно, false, если нет.

5.4 Методы/функции класса SemanticAnalyzer

- **Public:**
  - **SemanticAnalyzer(GenCode&)** — констурктор. Вход: ссылка на генератор POLIZ'a. Выход: -.
  - **SemanticAnalyzer()** — деструктор
  - **void pushID(std::string, std::string)** — запись идентификатора в TID, если до этого отсутствовал. В случае, если он был обьявлен до этого бросает ошибку. Вход: тип(буквенное обозначение); имя. Выход: void.
  - **bool deepSearchID(std::string)** — поиск имени переменной в TID на текущем и более высоких уровнях. Вход: имя переменной. Выход: true, если нашел; false, если не нашел.
  - **void IsIDDeclarring(std::string)** — проверка объявление переменной. Вход: имя. Выход: void.

- **void pushFUNC(std::string, std::string)** — запись объявления функции во временную переменную. Далее в нее будут добавлены параметры выхова и только после этого она будет добавлена в TID. Вход: тип(буквенное обозначение), имя. Выход: void.
- **void pushParamsNewFUNC(std::string, std::string)** — добавления параметров к описанию функции. Вход: тип(буквенное обозначение), имя. Выход: void.
- **void finishPushFunc()** — добавление новой функции в TID
- **bool deepSearchFUNC(FUNC_)** — поиск переопределения функции. Вход: новая функция. Выход: true, если переопределение, false, если новая.
- **void backLVLTID()** — возврат на родителя TID. Вход: -. Выход: void.
- **void goNextLVLTID(short int)** — создание нового уровня TID. Вход: тип TID(цикл, функция, условный оператор и т.д.). Выход: void.
- **void CheckParamsOperatorOutput(std::string)** — проверка существования переменных для оператора вывода. Вход: имя переменной. Выход: void.
- **void CheckParamsOperatorInput(std::string)** — проверка существования переменных для оператора ввода. Вход: имя переменной. Выход: void.
- **void setNameCallFunc(std::string)** — запоминание имени вызываемой функции. Вход: имя функции. Выход: void.
- **void addCallParamsFunc(std::string, short int, std::string)** — присвоение параметров для вызываемой функции. Вход: имя переменной, тип, значение в случае константного параметра. Выход: void.
- **int stopAnalyzeCallFunc()** — поиск вызываймой функции. Вход: -. Выход: -.
- **void addOperand(std::string, short int, int, std::string)** — добавление операнда выражения. Вход: имя переменной; тип(константа или переменная); id функции, если это вызов функции; значение(если константа). Выход: void.
- **void addOperation(std::string)** — добавление операции в выражение. Вход:операция. Выход: void.
- **short int calc_expr()** — подсчет совместимости типов при выполении операций выражения. Вход: -. Выход: код итогового типа.
- **bool FindTypeTID(short int)** — поиск необходимого типа TID. Вход: код типа. Выход: true, если найдено и false, если нет.
- **void setGeneratorTid()** — передача TID генератору POLIZ'а.

- **Private:**
  - **std::string check_compatible_op_and_st_(std::string, std::string, std::string, bool)** — проверка совместимости операндов и операции. Вход: левый операнд; правый операнд; операция; унарная операций или нет. Выход: тип результата операции.
  - **std::string getTypeByCode_(short int)** — получения текстового обозначения типа. Вход: код типа. Выход: текстовое обозначение типа.
  - **short int getCodeType(std::string, short int type)** — получения кода типа из текстового обозначения. Вход: текстовое обозначение; функция или переменная. Выход: код типа.

5.5 Методы/функции класса GenCode

- **Public:**
  - **void insert_start_block()** — вставка в POLIZ начала составного оператора. Вход: -. Выход: void.
  - **void insert_end_block()** — вставка в POLIZ конец составного оператора. Вход: -. Выход: void.
  - **void insert_write_func()** — вставка в POLIZ оператора вывода. Вход: -. Выход: void.
  - **void insert_read_func()** — вставка в POLIZ оператора ввода. Вход: -. Выход: void.
  - **void insert_semicolon()** — вставка в POLIZ точки с запятой. Вход: -. Выход: void.
  - **void insert_return()** — вставка в POLIZ оператора возврата. Вход: -. Выход: void.
  - **void insert_declaration_func(FUNC_)** — вставка в POLIZ описания функции. Вход: функция. Выход: void.
  - **void insert_expr_poliz(short int type, IDENT_& id)** — вставка переменной в выражение для генерации POLIZ'а. Вход: тип элемента POLIZ'а; переменная. Выход: void.
  - **void insert_expr_poliz(short int type, IDENT_& id, bool)** — вставка параметров вызова функции в выражение для генерации POLIZ'а. Вход: тип элемента POLIZ'а; переменная; объявление или использование. Выход: void.
  - **void insert_expr_poliz(short int type, std::string)** — вставка операции в выражение для генерации POLIZ'а. Вход: тип элемента POLIZ'а; операция. Выход: void.
  - **unsigned int insert_conditional_jump()** — вставка в POLIZ условного перехода по лжи. Вход: -. Выход: индекс вставки.

- **unsigned int insert_conditional_jump_dowhile(unsigned int)** — вставка в POLIZ условного перехода по истине. Вход: позиция для прыжка. Выход: индекс вставки.
- **unsigned int insert_unconditional_jump()** — вставка в POLIZ безусловного перехода. Вход: -. Выход: индекс вставки.
- **unsigned int insert_unconditional_jump(unsigned int);** — вставка в POLIZ условного перехода по лжи. Вход: индекс для прыжка. Выход: индекс вставки.
- **void push_pos_break(unsigned int)** — вставка в POLIZ оператора прерывания. Вход: позиция для прыжка. Выход: void.
- **unsigned int top_pos_break()** — получения позиции перехода оператора прерывания. Вход: -. Выход: индекс вставки.
- **unsigned int getSizePosBreak()** — кол-во обработанных операторов прерывания. Вход: -. Выход: индекс вставки.
- **void pop_pos_break()** — удаление оператора прерывания. Вход: -. Выход: void.
- **void insert_index_jump(unsigned int)** — установка текущей позиции полиза в переход, хранящийся по переданному индексу. Вход: позицмя для прыжка. Выход: void.
- **void genPolizExpr()** — генерация из выражения записи POLIZ. Вход: -. Выход: void.

- **Private:**
  - **void InterputProgrammMain_(int, TID_*)** — запуск выполнения программы по POLIZ. Вход: позиция старта; TID. Выход: void.
  - **std::pair<int, std::string> InterputFunc_(int, std::vector<IDENT_>, bool, unsigned int)** — выполнение функции программы. Вход: позиция начала выполнения; массив параметров; является ли функция main; глубина рекурсии. Выход: пара(тип возвращаймого значения; значение).
  - **PolizItem input_poliz_item(std::string)** — определение типа, введенной строки. Вход: введенное значение. Выход: элемент полиза.
  - **PolizItem make_operation(PolizItem, PolizItem, PolizItem)** — выполнение операций. Вход: левый операнд; правый операнд; операция. Выход: результат операции(элемент полиза).
  - **void make_operation_assignm(PolizItem, PolizItem, PolizItem)** — выполнение операций присвоения. Вход: левый операнд; правый операнд; операция. Выход: результат операции(элемент полиза).

- **PolizItem make_operation(PolizItem, PolizItem)** — выполнение унарных операций Вход: операнд; операция. Выход: результат операции(элемент полиза).
- **unsigned int FindPosFunc_(std::string, std::vector<IDENT_>)** — поиск индекса вызываймой функции. Вход: имя функции; список прааметров. Выход: позиция начала
- **void GetReadyFunc_(int, std::vector<IDENT_>)** — подготовка функции к вызову(присвоение передаваемых параметров). Вход: позиция начала функции; массив значений передаваемых параметров. Выход: void.
- **void GetBackFuncAfterCall_(int)** — возврат функции к исходному состоянию после вызова(обнуление объявленных переменных). Вход: позиция начала функции. Выход: void.

5.6 Методы/функции класса PolizItem

- **Public:**
  - **PolizItem(short int type)** — конструктор для блоков. Вход: тип элемента POLIZ'а.Выход:-.
  - **PolizItem(short int type, IDENT_&)** — конструктор для переменных, констант, вызова функций. Вход: тип элемента POLIZ'а, идентификатор. Выход:-.
  - **PolizItem(short int type, IDENT_&, bool)** — конструткор для параметров вызова функций(метода). . Вход: тип элемента POLIZ'а; идентификатор; является ли параметром вызова функции(метода). Выход:-.
  - **PolizItem(short int type, long long)** — конструктор для функций. . Вход: тип элемента POLIZ'а; позиция в массиве функций(методов). Выход:-.
  - **PolizItem(short int type, std::string)** — конструктор для операций. . Вход: тип элемента POLIZ'а; операция. Выход:-.
  - **bool getUnary()** — получение унарности операции. Вход: -. Выход: унарность операции.
  - **void setUnary(bool unar)** — установка унарности операции. Вход: унарность. Выход: void.
  - **short int getTypeItem()** — получения типа элемента POLIZ'а. Вход: -. Выход: код типа.
  - **std::string getOper()** — получение операции. Вход: -. Выход: операция.
  - **IDENT_ getIdent()** — получение указателя на переменную. Вход: -. Выход: указатель на переменную.

- **void setPos(unsigned int pos)** — установка позиции перехода. Вход: индекс позиции: Выход: void.
- **void setVal(std::string val)** — установка значения. Вход: значение. Выход: void.
- **void setType(unsigned short int type)** — установка типа элемента POLIZ'а. Вход: код типа элемента POLIZ'а. Выход: void.
- **unsigned int getPos()** — получение позиции перехода. Вход: -. Выход: индекс перехода.
- **PolizItem& operator=(const PolizItem& poliz_item)** — переопределение оператора присваивания.
- **PolizItem operator+(const PolizItem& poliz_item) const** — переопределение оператора сложения
- **PolizItem operator-(const PolizItem& poliz_item) const** — переопределение оператора вычитания
- **PolizItem operator*(const PolizItem& poliz_item) const** — переопределение оператора умножения
- **PolizItem operator/(const PolizItem& poliz_item) const** — переопределение оператора деления
- **PolizItem operator%(const PolizItem& poliz_item) const** — переопределение оператора остатка от деления
- **const bool operator==(const PolizItem& poliz_item) const** — переопределение оператора равенства
- **const bool operator==(const bool& boolean) const** — переопределение оператора равенства
- **const bool operator!=(const PolizItem& poliz_item) const** — переопределение оператора неравенства
- **const bool operator<(const PolizItem& poliz_item) const** — переопределение оператора меньше
- **const bool operator>(const PolizItem& poliz_item) const** — переопределение оператора больше
- **const bool operator<=(const PolizItem& poliz_item) const** — переопределение оператора меньше или равно
- **const bool operator>=(const PolizItem& poliz_item) const** — переопределение оператора больше или равно
- **PolizItem& operator++()** — переопределение оператора префиксного инкремента
- **PolizItem& operator- -()** — переопределение оператора префиксного декримента
- **const bool operator||(const PolizItem& poliz_item) const** — переопределение логического оператора ИЛИ

- **const bool operator&&(const PolizItem& poliz_item) const** — переопределение логического оператора И.
- **PolizItem operator&(const PolizItem& poliz_item) const** — переопределение побитового оператора И.
- **PolizItem operator|(const PolizItem& poliz_item) const** — переопределение побитового оператора ИЛИ.
- **PolizItem operator«(const PolizItem& poliz_item) const** — переопределение побитового сдвига ВЛЕВО.
- **PolizItem operator»(const PolizItem& poliz_item) const** — переопределение побитового сдвига ВПРАВО.
- **PolizItem& operator~()** — переопределение побитового XOR
- **PolizItem operator^(const PolizItem& poliz_item) const** — переопределение побитовой операции исключающего ИЛИ.

5.7 Методы/функции класса Types

- **Protected:**
  - **структура IDENT_** — структура для хранения перемененых.
  - **структура FUNC_** — структура для хранения описания функции(метода).
  - **структура IDENTIFIER_** — структура для хранения переменных и функций(методов) одного блока программы.
  - **структура TID_** — структура таблицы TID.
  - **int getPriority(std::string, bool)** — получение приоритета операций. Вход: операция; унарность. Выход: приоритет.

6 Наследование классов

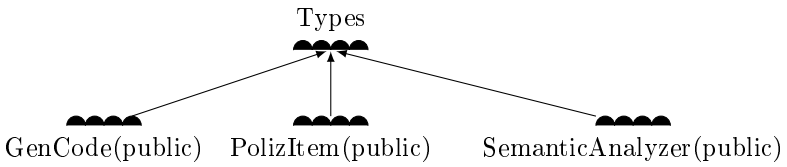

Рис. 1: Наследование классов

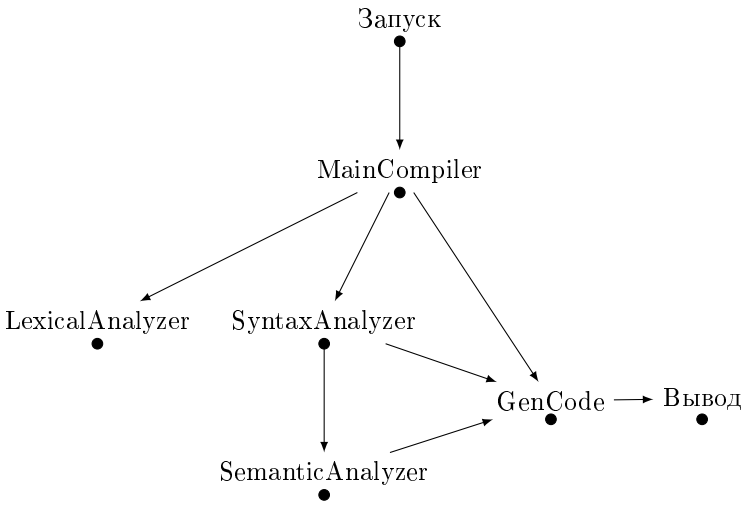

Рис. 2: Вызовы классов

7 Вызовы классов

8 Совместимость типов

Тип	int	boolean	double	string
int	+	+	+	-
boolean	+	+	+	-
double	+	+	+	-
string	-	-	-	+

9 Заметки

Будущим разработчикам рекомендуется улучшить распределение типов, т.к. в данной реализации наблюдаются большие неудобства, требующие большого количества дополнительных строк. С Наилучшими пожеланиями, разработчики!