МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ

ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра программной инженерии

Специальность 6-05-0612-01 Программная инженерия

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора GMS-2024»

Выполнил студент

(Ф.И.О.)

Руководитель проекта Наркевич Аделина Сергеевна

Заведующий кафедрой к.т.н., доц. Смелов Владимир Владиславович

Консультанты Наркевич Аделина Сергеевна

Нормоконтролер Наркевич Аделина Сергеевна

Курсовой проект защищен с оценкой

Минск 2024

Содержание

[Введение 6](#_Toc185386243)

[1. Спецификация языка программирования 7](#_Toc185386244)

[1.1 Характеристика языка программирования 7](#_Toc185386245)

[1.2 Определение алфавит языка программирования 7](#_Toc185386246)

[1.3 Применяемые сепараторы 7](#_Toc185386247)

[1.4 Применяемые кодировки 8](#_Toc185386248)

[1.5 Типы данных 8](#_Toc185386249)

[1.6 Преобразование типов данных 9](#_Toc185386250)

[1.7 Идентификаторы 9](#_Toc185386251)

[1.8 Литералы 10](#_Toc185386252)

[1.9 Объявление данных 10](#_Toc185386253)

[1.10 Инициализация данных 10](#_Toc185386254)

[1.11 Инструкции языка 11](#_Toc185386255)

[1.12 Операции языка 11](#_Toc185386256)

[1.13 Выражения и их вычисления 12](#_Toc185386257)

[1.14 Конструкции языка 12](#_Toc185386258)

[1.15 Область видимости идентификаторов 13](#_Toc185386259)

[1.16 Семантические проверки 13](#_Toc185386260)

[1.17 Распределение оперативной памяти на этапе выполнения 13](#_Toc185386261)

[1.18 Стандартная библиотека и её состав 14](#_Toc185386262)

[1.19 Ввод и вывод данных 14](#_Toc185386263)

[1.20 Точка входа 14](#_Toc185386264)

[1.21 Препроцессор 14](#_Toc185386265)

[1.22 Соглашения о вызовах 14](#_Toc185386266)

[1.23 Объектный код 15](#_Toc185386267)

[1.24 Классификация сообщений транслятора 15](#_Toc185386268)

[1.25 Контрольный пример 15](#_Toc185386269)

[2. Структура транслятора 16](#_Toc185386270)

[2.1 Компоненты транслятора их назначение и принципы взаимодействия 16](#_Toc185386271)

[2.2. Перечень входных параметров транслятора 16](#_Toc185386272)

[2.3 Протоколы, формируемые транслятором 17](#_Toc185386273)

[3. Разработка лексического анализатора 19](#_Toc185386274)

[3.1. Структура лексического анализатора 19](#_Toc185386275)

[3.2. Контроль входных символов 19](#_Toc185386276)

[3.3. Удаление избыточных символов 20](#_Toc185386277)

[3.4. Перечень ключевых слов 20](#_Toc185386278)

[3.5. Основные структуры данных 22](#_Toc185386279)

[3.6. Структура и перечень сообщений лексического анализатора 23](#_Toc185386280)

[3.7. Принцип обработки ошибок 23](#_Toc185386281)

[3.8. Параметры лексического анализатора 24](#_Toc185386282)

[3.9. Алгоритм лексического анализа 24](#_Toc185386283)

[3.10. Контрольный пример 25](#_Toc185386284)

[4. Разработка синтаксического анализатора 26](#_Toc185386285)

[4.1 Структура синтаксического анализатора 26](#_Toc185386286)

[4.2 Контекстно-свободная грамматика, описывающая синтаксис языка 26](#_Toc185386287)

[4.3. Построение конечного магазинного автомата 27](#_Toc185386288)

[4.4. Основные структуры данных 28](#_Toc185386289)

[4.5. Описание алгоритма синтаксического разбора 29](#_Toc185386290)

[4.6. Структура и перечень сообщений синтаксического анализатора 29](#_Toc185386291)

[4.7. Параметры синтаксического анализатора и режимы его работы 30](#_Toc185386292)

[4.8. Принцип обработки ошибок 30](#_Toc185386293)

[4.9. Контрольный пример 30](#_Toc185386294)

[5. Разработка семантического анализатора 31](#_Toc185386295)

[5.2. Функции семантического анализатора 31](#_Toc185386296)

[5.3. Структура и перечень сообщений семантического анализатора 32](#_Toc185386297)

[5.4 Принцип обработки ошибок 33](#_Toc185386298)

[5.5 Контрольный пример 33](#_Toc185386299)

[6.1 Вычисление выражений 34](#_Toc185386300)

[6.1. Выражения, допускаемые языком 34](#_Toc185386301)

[6.2 Польская запись и принцип её построения 34](#_Toc185386302)

[6.3 Программная реализация обработки выражений 35](#_Toc185386303)

[6.4 Контрольный пример 35](#_Toc185386304)

[7. Генерация кода 36](#_Toc185386305)

[7.1 Структура генератора кода 36](#_Toc185386306)

[7.2. Представление типов данных в оперативной памяти 36](#_Toc185386307)

[7.3. Статическая библиотека 36](#_Toc185386308)

[7.4. Особенности алгоритма генерации кода 37](#_Toc185386309)

[7.5. Входные параметры, управляющие генерацией кода 39](#_Toc185386310)

[7.6. Контрольный пример 39](#_Toc185386311)

[8. Тестирование транслятора 40](#_Toc185386312)

[8.1. Общие положения 40](#_Toc185386313)

[8.2 Результаты тестирования 40](#_Toc185386314)

[Заключение 42](#_Toc185386315)

[Список использованных источников 43](#_Toc185386316)

[Приложение А 44](#_Toc185386317)

[Приложение Б 45](#_Toc185386318)

[Приложение В 48](#_Toc185386319)

[Приложение Г 54](#_Toc185386320)

[Приложение Д 59](#_Toc185386321)

Введение

Целью курсового проекта является разработка транслятора для собственного языка программирования: GMS-2024.

Написание транслятора будет осуществляться на языке C++, при этом код на языке GMS-2024 будет транслироваться в язык ассемблера.

Исходя из цели курсового проекта, были определены следующие задачи:

– разбработка спецификации языка программирования;

– разбратка структуры транслятора;

– разработка лексического и семантического анализаторов;

– разработка синтаксического анализатора;

– преобразование выражений;

– генерация кода на язык ассемблера;

– тестирование транслятора.

Решения каждой из поставленных задач будут приведены в соответствующих главах курсового проекта.

1. Спецификация языка программирования

1.1 Характеристика языка программирования

Язык программирования GMS-2024 – это язык высокого уровня. Он является процедурным, компилируемым, строго типизируемый.

1.2 Определение алфавит языка программирования

В основе алфавита языка GMS-2024 лежит таблица символов windows-1251

Таблица 1.1 – Алфавит языка GMS-2024 в форме Бэкуса-Наура

|  |
| --- |
| <строчная буква латинского алфавита>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z |
| <прописная буква латинского алфавита>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z |
| <цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <символ- сепаратор>::= ' '|,|(|)|{|}|;| "" |=| |
| <арифметическая операция>::= + | - | \* | / |
| <операции сравнения>::= > | < | <= | >= | == | != |

1.3 Применяемые сепараторы

Символы-сепараторы служат в качестве разделителей операций языка. Сепараторы, используемые в языке программирования GMS-2024, приведены в таблице 1.2.

Таблица 1.2 – Сепараторы

|  |  |  |
| --- | --- | --- |
| Сепаратор | Название | Область применения |
| ‘ ‘ | Пробел | Допускается везде, кроме идентификаторов и ключевых слов |
| ; | Точка с запятой | Разделение конструкций |
| {…} | Фигурные скобки | Заключение программного блока |
| […] | Квадратные кавычки | Блок кода |
| (…) | Круглые скобки | Приоритет операций, параметры функции |
| “…” | Двойные кавычки | Строковый литерал |
| ‘…’ | Одинарные кавычки | Допускается везде, кроме идентификаторов и ключевых слов |
| = | Знак «равно» | Присваивание значения |
| , | Запятая | Разделение параметров |
| / + - \* : | Знаки «косая черта», «плюс», «минус», «астерикс», «двоеточие», | Выражения |

Окончание таблицы 1.2

|  |  |  |
| --- | --- | --- |
| Сепаратор | Название | Область применения |
| < > >= <= == != | Знаки “меньше,” “больше”, “больше либо равно”, “меньше либо равно”, “равно”, “не равно” | Операции сравнения |

1.4 Применяемые кодировки

Для написания исходного кода на языке программирования GMS-2024 используется кодировка Windows-1251, которая представлена на рисунке 1.1.



Рисунок 1.1 – таблица символов кодировки windows-1251

1.5 Типы данных

В языке GMS-2024 реализованы 2 типа данных: целочисленный (2 байта) и символьный. Описание типов данных, предусмотренных в данном языке представлено в таблице 1.3.

Таблица 1.3 – Типы данных языка GMS-2024

|  |  |
| --- | --- |
| Типы данных | Описание типа данных |
| Целочисленный int | Фундаментальный тип данных. Предусмотрен для объявления целочисленных данных (2 байта). Инициализация по умолчанию: значение 0. Максимально допустимое значение 215-1. Минимально допустимым является -215. |
| Символьный char | Фундаментальный тип данных. Используется для работы с символом, который в памяти занимает 1 байт. Инициализация по умолчанию: символ конца строки “\0”. |
| Строковой str | Строковой тип данных. Каждый символ занимает 1 байт и принадлежит кодировке windows1251. Значение по умолчанию отсутствует. Максимальная длина 255, хранится в стеке как массив. Является ссылочным типом. |

Пользовательские типы данных не поддерживаются.

1.6 Преобразование типов данных

В языке программирования GMS-2024 преобразование типов данных не поддерживается. Все типы данных определенны однозначно и не могут быть преобразованы в другие.

1.7 Идентификаторы

В имени идентификатора допускаются только символы латинского алфавита и знак «\_» и цифры. Имя идентификатора не может совпадать с ключевыми словами языка и именами функций стандартной библиотеки и не должно начинаться с цифры. Формальное описание идентификатора можно представить в следующем виде:

<идентификатор> ::= <буква> {<буква> | <цифра> | '\_'}, где:

<буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t

| u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

1.8 Литералы

В языке GMS-2024 предусмотрены 3 вида литералов: целочисленные, символьные и строковые. Описание литералов приведено в таблице 1.4.

Таблица 1.4 – Описание литералов

|  |  |
| --- | --- |
| Литерал | Пояснение |
| Целочисленный | Целочисленные литералы, по умолчанию инициализируются 0. Представление: шестнадцатеричное (0x05), восьмеричное (05), двоичное (0b101), десятичное (5). |
| Символьный | Используются одиночные символы кодировки windows-1251, заключенные в одинарные кавычки: ‘5’. |
| Строковой | Допускается использование только внутри вызова функции потокового вывода. Заключены в двойные кавычки: “строка” |

Числовые литералы являются константами времени компиляции и объявляются только один раз.

1.9 Объявление данных

В языке GMS-2024 требуется объявить идентификатор перед его использованием. Все переменные должны находится внутри программного блока. Допускается объявление переменных с одинаковыми именами в разных программных блоках. Область видимости реализована по принципу «сверху вниз».

1.10 Инициализация данных

При объявлении переменной возможна ее инициализация литералом. Однако при компиляции в язык ассемблера, все переменные в сегменте данных будут иметь значение по умолчанию.

Таблица 1.5 – Способы и присвоения значения переменной

|  |  |  |
| --- | --- | --- |
| Конструкция | Описание | Пример |
| <тип данных>  <идентификатор> =  <литерал>; | Объявление идентификатора и его инициализация. | int x = 10;  char c = ‘x’ |
| <тип данных>  <идентификатор>; | Объявление идентификатора и его автоматическая инициализация значением по умолчанию. | int x; |

Соответствие типов проверяется на этапе синтаксического анализа.

1.11 Инструкции языка

Все возможные инструкции языка программирования GMS-2024 представлены в общем виде в таблице 1.6.

Таблица 1.6 – Инструкции языка программирования GMS-2024

|  |  |
| --- | --- |
| Инструкция | Запись на языке GMS-2024 |
| Объявление переменной | <тип данных> <идентификатор>; |
| Объявление функции | <тип данных> <идентификатор> (<тип данных> <идентификатор>, …) {<блок кода>} |
| Вызов функции | <идентификатор функции> ( <параметры функции> | <пусто>); |
| Присваивание | <идентификатор> = <литерал> | <идентификатор> | <выражение>; |
| Блок инструкций | {  …  } |
| Возврат из подпрограммы | return <идентификатор> | <литерал> | <выражение>; |
| Условная инструкция | if(<условие>){<блок кода>}; |
| Вывод данных | print (<идентификатор> | <литерал> | <выражение>);  write <идентификатор> | <литерал> | <выражение>; |
| Ветвление | if (<условное выражение>) {<тело блока истины>} else [if (<условное выражение>)] {<тело блока лжи>} |

1.12 Операции языка

Язык программирования GMS-2024 может выполнять операции, представленные в таблице 1.7.

Таблица 1.7 – Операции языка программирования GMS-2024

|  |  |  |
| --- | --- | --- |
| Операция | Примечание | Пример |
| ( | Приоритет операций | (a + b) \* c; |
| ) |
| + | Суммирование | a + b; |
| - | Вычитание | a – b; |
| \* | Умножение | a \* b; |
| / | Деление | a / b; |
| == | Сравнение на равенство значений | a == b; |
| != | Сравнение на неравенство значений | a != b; |
| = | Присваивание | sum = 15;  chr = ‘T’; |
| <, > | Знаки «меньше», «больше» для условной инструкции | x < 3; |

1.13 Выражения и их вычисления

Круглые скобки в выражении используются для изменения приоритета операций. Два арифметических оператора не могут идти подряд. В выражении может быть вызов функции, если возвращаемый тип этой функции совместим с операциями, выполняемыми в данном выражении. Выражения не могут использоваться как аргументы функций, но могут быть использованы для присвоения значения переменной или в качестве логических операндов.

При генерации кода, для вычисления выражений используется польская запись.

1.14 Конструкции языка

Ключевые программные конструкции языка GMS-2024 представлены в таблице 1.8.

Таблица 1.8 – Программные конструкции языка GMS-2024

|  |  |
| --- | --- |
| Конструкция | Запись на языке GMS-2024 |
| Главная функция (точка входа) | main  {  …  [return <идентификатор> | <литерал>; ]  }; |
| Функция | <тип> <идентификатор> ([(<тип> <идентификатор>, …])  {  …  [return <идентификатор> | <литерал>; ]  }; |
| Условная конструкция | if(<условие>)  {  …  }  else [if…]  {  …  } |

В данной таблице представлены различные конструкции программирования и их реализация. Главная функция «main» является основной точкой входа в программу. Внешние функции определяются с помощью указания типа и названия, со списком параметров, который может отсутствовать, внутри круглых скобок. Условные конструкции позволяют выполнять различные кодовые блоки в зависимости от условий истинности или ложности.

1.15 Область видимости идентификаторов

В языке GMS-2024 все переменные, объявленные внутри блока кода, являются локальными, т.е. имеют функциональную область видимости. Они обязаны находится внутри программного блока функций. Объявление глобальных переменных не предусмотрено. Объявление пользовательских областей видимости не предусмотрено.

1.16 Семантические проверки

Таблица с перечнем семантических проверок, предусмотренных языком, приведена в таблице 1.8.

Таблица 1.8 – Семантические проверки

|  |  |
| --- | --- |
| № | Правило |
| 1 | Наличие точки входа main |
| 2 | Отсутствие переопределения идентификатора |
| 3 | Усекание длины имени идентификатора до 25 символов |
| 4 | Объявление идентификатора перед его использованием |
| 5 | Не допускается совпадения названий идентификаторов с ключевыми словами |
| 6 | Присвоение идентификатору значения, соответствующего его типу |
| 7 | Соответствие типов формальных и фактических параметров при вызове функций |
| 8 | Соответствие числа параметров при объявлении функции и ее вызове |
| 9 | Корректное использование операций сравнения и арифметических операций |
| 10 | Использование строго инициализированной переменной |
| 11 | Соответствие типов при различном их использовании в выражениях |
| 12 | Возвращаемое значение инструкцией return должно совпадать типу возвращаемого функцией значения, указанного при ее объявлении. |
| 13 | Деление на ноль |

Если семантическая проверка не проходит, то в лог журнал записывается соответствующая ошибка.

1.17 Распределение оперативной памяти на этапе выполнения

Все переменные размещаются в глобальной области памяти. Литералы заносятся в сегмент данных только для чтения (.const), а локальные переменные – в сегмент данных (.data). Эти секции используют статически выделенную память, доступную на протяжении всего времени выполнения программы. Локальная область видимости в исходном коде реализована с помощью правил префиксного именования переменных из разных блоков, что обуславливает их локальность в исходном коде.

1.18 Стандартная библиотека и её состав

Для использования функций стандартной библиотеки, не нужно явно подключать её. Все функции стандартной библиотеки доступны по умолчанию и не подлежат переопределению. Функции стандартной библиотеки с описанием представлены в таблице 1.9.

Таблица 1.9 – Состав стандартной библиотеки

|  |  |
| --- | --- |
| Функция | Назначение |
| int strcmp(str str1, str str2) | Принимает две строки и возвращает результат сравнения двух строк:  0, если 2 строки равны; 1, если первая строка лексикографически больше второй, иначе -1. |
| void print(int v) | Принимает значение любого типа и выводит его в консоль, автоматически добавляя символ новой строки в конце. |
| void print(char ch) |
| void print(str v) |

1.19 Ввод и вывод данных

В языке GMS-2024 не реализованы средства ввода данных. Для вывода данных в стандартный поток вывода предусмотрена функция print из стандартной библиотеки и оператор write.

1.20 Точка входа

В языке GMS-2024 каждая программа должна содержать главную функцию “main” - точку входа, с которой начнется последовательное выполнение программы. В программе может быть только одна точка входа.

1.21 Препроцессор

Препроцессор в языке программирования GMS-2024 не предусмотрен.

1.22 Соглашения о вызовах

В языке вызов функций происходит по соглашению о вызовах stdcall. Особенности stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

1.23 Объектный код

GMS-2024 транслируется в язык ассемблера MASM.

1.24 Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке GMS-2024 и выявления её транслятором в текущий файл протокола выводится сообщение. Их классификация сообщений приведена в таблице 1.10.

Таблица 1.10 – Классификация сообщений транслятора

|  |  |
| --- | --- |
| Интервал | Описание ошибок |
| 0-99 | Системные ошибки |
| 100-115 | Ошибки открытия и чтения файлов |
| 116-139 | Параметров |
| 140-200 | Ошибки лексического анализа |
| 130-139 | Ошибки таблиц лексем и таблиц идентификаторов |
| 600-699 | Ошибки синтаксического анализа |
| 700-900 | Ошибки семантического анализа |

Компилятор может обрабатывать до 1000 различных ошибок.

1.25 Контрольный пример

Контрольный пример представлен в Приложении А.

2. Структура транслятора

2.1 Компоненты транслятора их назначение и принципы взаимодействия

Язык GMS-2024 транслируется в язык ассемблера. Предварительно должен быть обработан исходный код, составлены таблицы идентификаторов и лексем, которые будут использованы для генерации кода.

Лексический анализатор разбивает исходный текст программы на лексемы, выделяя ключевые слова, идентификаторы, числа и другие элементы. Результатом его работы является последовательность лексем, которая также содержит ссылку на элементы таблицы идентификаторов.

Синтаксический анализатор проверяет последовательность лексем на соответствие грамматике языка GMS-2024. Он строит синтаксическое дерево, которое отображает структуру программы.

Семантический анализатор выполняет проверку логической корректности программы. Он контролирует типы данных, проверяет корректность операций и использования идентификаторов.

Генератор кода преобразует промежуточное представление исходного кода в текст программы на языке MASM. На этом этапе используются данные из таблицы идентификаторов для имен переменных и меток.

Принципы взаимодействия компонентов основываются на последовательной обработке данных. Исходный текст программы проходит через все этапы, начиная с лексического анализа и заканчивая генерацией кода. Каждый этап использует результаты предыдущего, обеспечивая корректность и точность трансляции.

2.2. Перечень входных параметров транслятора

Все доступные входные параметры транслятора, управляющие его работой, представлены в таблице 2.1

Таблица 2.1 – Входные параметры транслятора языка GMS-2024

|  |  |  |
| --- | --- | --- |
| Параметр | Описание параметра | Значение по умолчанию |
| -in:<путь к файлу> | Задает путь к файлу с исходным кодом на языке GMS-2024, имеющим расширение .txt | Не предусмотрено |
| -log:<путь к файлу> | Файл журнала для вывода протоколов работы программы. | <путь -in файла>.log |
| -out:<путь к файлу> | Выходной файл с результатом предварительной обработки исходного кода | <путь -in файла>.out |

Окончание таблицы 2.1

|  |  |  |
| --- | --- | --- |
| Параметр | Описание параметра | Значение по умолчанию |
| -lt:<путь к файлу> | Указывает выходной путь таблицы лексем. Все директории из пути должны существовать, а файл должен иметь расширение .html | <путь -in файла>.LT.html |
| -it:<путь к файлу> | Указывает выходной путь таблицы идентификаторов. Все директории из пути должны существовать, а файл должен иметь расширение .html | <путь -in файла>.IT.html |
| -fo:<путь к файлу> | Указывает выходной путь исполняемого файла. | <путь -in файла>.exe |
| -stack:<число> | Задает размер стека в байтах | 1024 |
| /cst | Включает вывод на консоль дерева разбора. | Вывод отключен |
| /lex | Включает сохранение таблиц идентификаторов и лексем | Сохранение отключено |

2.3 Протоколы, формируемые транслятором

Во время работы транслятор генерирует различные протоколы, представляющие результаты промежуточных этапов. Все файлы создаются в каталоге, в котором расположен исполняемый файл, если иное не было задано входными параметрами. В таблице 2.2 приведена информация про формируемые протоколы.

Таблица 2.2 – Протоколы, формируемые транслятором языка GMS-2024

|  |  |
| --- | --- |
| Протокол | Описание протокола |
| Файл журнала, задающийся параметром “-log” | Файл с журналом работы транслятора, содержащий основную информацию о результате работы транслятора: информацию про значения параметров, сведения об результатах работы синтаксического, семантического анализаторов, возникших ошибках. |
| Выходной файл предварительной обработки исходного кода, задающийся параметром “-out” | Файл содержит результат обработки исходного кода: текст, который поступит на лексический анализ. |

Окончание таблицы 2.2

|  |  |
| --- | --- |
| Протокол | Описание протокола |
| Выходной файл таблицы идентификаторов, задающийся параметром “-it” | Файл содержит представление таблицы идентификаторов, которая была получена в результате выполнения лексического анализа. |
| Выходной файл таблицы лексем, задающийся параметром “-lt” | Файл содержит представление таблицы лексем, которая была получена в результате выполнения лексического анализа. |
| Выходной файл с расширением “.asm” | Результат выполнения программы - исходный код на языке ассемблера. |

3. Разработка лексического анализатора

3.1. Структура лексического анализатора

Лексический анализатор — это программа, выполняющая разбор исходного текста и преобразование его в последовательность лексем. Структура лексического анализатора схематично представлена на рисунке 3.1.

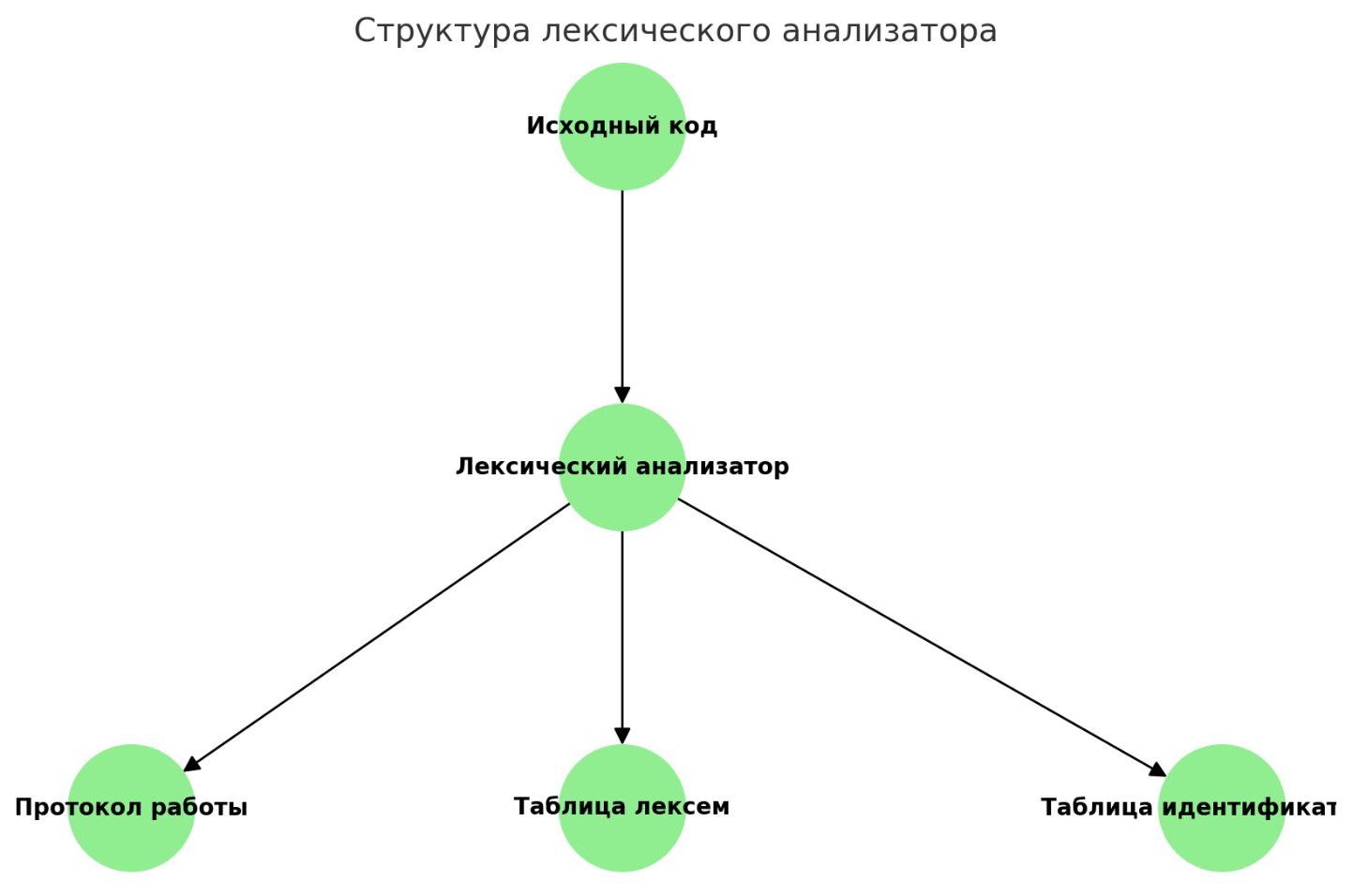


Рисунок 3.1 – структура лексического анализатора

Лексический анализатор принимает на вход исходный текст программы в текстовом формате. На выходе формируются таблицы распознанных лексем и идентификаторов.

3.2. Контроль входных символов

Для контроля правильности написания исходного текста используется таблица допустимых символов (рисунок 3.2). каждый элемент в ней соответствует коду символа в кодировке windows-1251 и представляет специальный “флаг” символа, определяющий, является ли символ допустимым и предопределяет обработку отдельных символов.

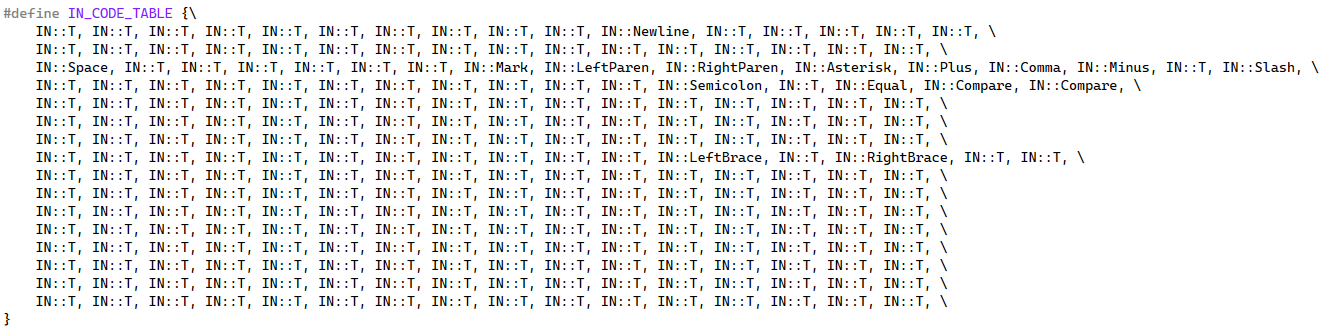


Рисунок 3.2 – таблица допустимых символов

3.3. Удаление избыточных символов

Для облегчения и шаблонизации лексического анализа используется предварительная обработка исходного кода, во время которой удаляются “избыточные” символы, а исходный код подготавливается к лексическому анализу.

“Избыточными” символами являются пробельные символы, стоящие там, где их присутствие не обязательно для разделения конструкций языка. Например, логические и арифметические операторы не обязаны быть разделены пробельными символами, а ключевые слова – да.

Алгоритм обработки исходного кода следующий: комментарии в исходном коде игнорируются, пробельные символы удаляются там, где их присутствие необязательно, а также там, где они дублируются, за исключением строковых литералов.

3.4. Перечень ключевых слов

Таблица 3.1 – Ключевые слова, операторы и сепараторы языка GMS-2024

|  |  |  |
| --- | --- | --- |
| Токен языка | Лексема | Назначение |
| main | m | Определяет точку входа в программе |
| if | ? | Оператор условного перехода |
| else | : | Оператор перехода при невыполнении условия логического оператора if |
| return | r | Оператор выхода из функции и возврата ею некоторого значения |
| int | t | Указывает тип данных идентификатора как целочисленный |
| str | t | Указывает тип данных идентификатора как строковой |
| char | t | Указывает тип данных идентификатора как один символ |
| ( ) | ( ) | Круглые скобки используются для изменения приоритета операций, для вызова функций, а также в них заключено условие логического оператора if. |

Окончание таблицы 3.1

|  |  |  |
| --- | --- | --- |
| Токен языка | Лексема | Назначение |
| + - \* / | v | Арифметическая операция между целочисленными типами |
| < > <= >= == != | c | Оператор логического сравнения |
| { } | { } | Указывают начало и конец некоторого блока кода |

Для распознания цепочек из таблицы 3.1 используются конечный автомат, который использует регулярные выражения. Пример регулярного выражения для идентификаторов: [a-zA-Z\_][a-zA-Z0-9\_]. Его реализация на языке c++ представлена в листинге 3.1.

Листинг 3.1 – Фрагмент кода на c++, реализующий конечный автомат

|  |
| --- |
| #define nums(state) \  R('0', state), R('1', state), R('2', state),\  R('3', state), R('4', state), R('5', state),\  R('6', state), R('7', state), R('8', state), R('9', state)  #define relationsForState(state) \  R('a', state), R('b', state), R('c', state), R('d', state), \  R('e', state), R('f', state), R('g', state), R('h', state), \  R('i', state), R('j', state), R('k', state), R('l', state), \  R('m', state), R('n', state), R('o', state), R('p', state), \  R('q', state), R('r', state), R('s', state), R('t', state), \  R('u', state), R('v', state), R('w', state), R('x', state), \  R('y', state), R('z', state), R('A', state), R('B', state), \  R('C', state), R('D', state), R('E', state), R('F', state), \  R('G', state), R('H', state), R('I', state), R('J', state), \  R('K', state), R('L', state), R('M', state), R('N', state), \  R('O', state), R('P', state), R('Q', state), R('R', state), \  R('S', state), R('T', state), R('U', state), R('V', state), \  R('W', state), R('X', state), R('Y', state), R('Z', state), R('\_', state)  FST::FST\* CreateIdentifierFST(const char\* str)  {  return new FST::FST(str,  2,  FST::NODE(NUM\_LETTERS \* 2 + 20,  relationsForState(0), nums(0),  relationsForState(1), nums(1)),  FST::NODE()  );  } |

3.5. Основные структуры данных

Для построения таблиц лексем и идентификаторов используются специальные структуры данных на языке c++.

Таблица идентификаторов представлена структурой ID\_Table (листинг 3.3), которая содержит объекты структуры Entry (листинг 3.2), представляющие собой запись про конкретный идентификатор. Она содержит ссылку на соответствующую ссылку в таблице лексем, символьное представление названия идентификатора, его тип и тип данных, значение идентификатора, а также область видимости – ссылку на другую запись в таблице идентификаторов, которая является функцией или блоком кода.

Листинг 3.2 – Структура Entry

|  |
| --- |
| enum IDDATATYPE { INT = 1, STR, CHAR};  enum IDTYPE { V = 1024, F, P, L };  struct Entry // строка таблицы идентификаторов  {  int idxfirstLE; // индекс первой строки в таблице лексем  char id[ID\_MAXSIZE]; // идентификатор (автоматически усекается до ID\_MAXSIZE)  IDDATATYPE iddatatype; // тип данных  IDTYPE idtype; // тип идентификатора (переменная, функция, параметр, локальная переменная)  Entry\* scope;  union  {  int vint; // значение integer  struct  {  int len; // кол-во символов в string  char str[TI\_STR\_MAXSIZE - 1]; // символы string  } vstr[TI\_STR\_MAXSIZE]; // значение string  } value; // значение идентификатора  }; |

Таблица идентификаторов содержит текущий и максимальный размер, а также саму таблицу, которая представлена массивом объектов Entry.

Листинг 3.3 – Структура ID\_Table

|  |
| --- |
| struct ID\_Table // экземпляр таблицы идентификаторов  {  int maxsize; // емкость таблицы идентификаторов < TI\_MAXSIZE  int size; // текущий размер таблицы идентификаторов < maxsize  Entry\* table; // массив строк таблицы идентификаторов  }; |

3.6. Структура и перечень сообщений лексического анализатора

После обнаружения ошибки лексический анализатор немедленно прекращает работу, выводит номер строки на консоль, опционально, дополнительную, поясняющую информацию, на консоль и записывает её в протокол.

Всевозможные ошибки перечислены в таблице 3.2

Таблица 3.2 – Номера ошибок при лексическом анализе и их пояснения

|  |  |
| --- | --- |
| Номер ошибки | Текст ошибки |
| 140 | Может быть только одна точка входа main |
| 141 | Строковые и символьные литералы не могут быть пустыми |
| 142 | Идентификатор использован, но не объявлен |
| 143 | Идентификатору целочисленного типа присваивается значение строкового |
| 144 | Идентификатору строкового типа присваивается значение целочисленного типа |
| 145 | Превышен максимальный размер входного текста |
| 146 | Переопределение функции |
| 147 | Переопределение параметра функции |
| 148 | Переопределение переменной |
| 149 | Некорректный литерал |
| 150 | Идентификатор не может начинаться с цифры |

3.7. Принцип обработки ошибок

Все ошибки являются критическим, при возникновении первой ошибки, выполнение лексического анализа останавливается, выводится соответствующее сообщение об ошибке на консоль, программа завершает свое выполнение.

3.8. Параметры лексического анализатора

Некоторые входные параметры транслятора управляют сохранением таблиц идентификаторов и лексем. Они представлены в таблице 3.3.

Таблица 3.3 – Параметры лексического анализатора

|  |  |  |
| --- | --- | --- |
| Параметр | Описание параметра | Значение по умолчанию |
| -lt:<путь к файлу> | Указывает выходной путь таблицы лексем. Все директории из пути должны существовать, а файл должен иметь расширение .html | <путь -in файла>.LT.html |
| -it:<путь к файлу> | Указывает выходной путь таблицы идентификаторов. Все директории из пути должны существовать, а файл должен иметь расширение .html | <путь -in файла>.IT.html |
| /lex | Включает сохранение таблиц идентификаторов и лексем | Сохранение отключено |

3.9. Алгоритм лексического анализа

Алгоритм лексического анализатора:

1. Входной текст считывается посимвольно. Символы сохраняются во временном буфере.
2. При встрече символа-сепаратора содержимое буфера разбирается конечным автоматам. При успешном распознании строки, конечный автомат возвращает лексему, соответствующую разобранному выражению. Если лексема была распознана, она сохраняется в таблицу лексем.
3. Если лексема равна лексеме типа, то в функции разбора строки устанавливается флаг, соответствующий типу данных. Иначе, если это лексема идентификатора – идет проверка из пункта 4.
4. Если перед лексемой идентификатора была сохранена лексема типа, то эта запись считается объявлением идентификатора, иначе выполняется пункт 5. В случае объявления, происходит проверка, нет ли идентификатора с таким именем и такой областью видимости для предотвращения переопределения. В случае успеха, идентификатор и лексема сохраняются в соответствующие таблицы и выполнение продолжается с пункта 1. В случае переопределения будет выброшено соответствующее исключение.
5. Если перед лексемой идентификатора не была сохранена лексема типа, то эта запись интерпретируется как использование ранее объявленного идентификатора. В таком случае выполняется его поиск в таблице идентификаторов и в случае отсутствия – выбрасывается соответствующее исключение. Иначе лексема сохраняется в таблице лексем.
6. Если лексема была распознана как литерал, будут осуществлены проверки на его корректность.
7. Если встречен символ ‘(‘, то будет осуществлена проверка предыдущих сохраненных лексем: если предыдущая лексема – это идентификатор, а предыдущая ей – тип, то предыдущий идентификатор будет расценен как функция или ее вызов. Это повлечет изменения в таблице идентификаторов для соответствующего идентификатора.
8. Если весь входной текст еще не разобран, то алгоритм возвращается к пункту 1, иначе завершается.

3.10. Контрольный пример

Результат лексического анализа в виде таблиц лексем и идентификатора представлены в приложении Б.

4. Разработка синтаксического анализатора

4.1 Структура синтаксического анализатора

Синтаксический анализ выполняется после лексического анализа и предназначен для распознавания заранее определенных синтаксических конструкций. Синтаксический анализатор принимает на вход таблицу лексем и генерирует дерево разбора или выявленные ошибки в синтаксисе.

4.2 Контекстно-свободная грамматика, описывающая синтаксис языка

В языке GMS-2024 синтаксический анализатор использует контекстно-свободную грамматику G = < T, N, P, S >, где:

– T – множество терминальных символов

– N – множество нетерминальных символов

– P – множество правил языка

– S – начальный символ грамматики (нетерминал)

Эта грамматика имеет нормальную форму Грейбах, так как она не содержит леворекурсивных правил.

Таблица 4.1 – Синтаксические правила языка GMS-2024

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочка (правило) | описание |
| S | ti(F){NrE;}S | Стартовый символ, порождающий всю структуру исходного кода |
| ti(F){rE;}S |
| ti(F){rE;}S |
| ti(F){rE;}S |
| m{NrE;} |
| m{N} |
| N | ti=E;N | Операторы программы |
| ti;N |
| i=E;N |
| rE;N |
| p(E);N |
| p(EcE);N |
| wE;N |
| wEcE;N |
| ?Y;N |
| i(W);N |
| i();N |
| ti=E; |
| ti; |
| i=E; |
| rE; |

Окончание таблицы 4.1

|  |  |  |
| --- | --- | --- |
| Нетерминал | Цепочка (правило) | описание |
| N | p(E); |  |
| p(EcE); |
| wE; |
| wEcE; |
| ?Y; |
| i(W); |
| i(); |
| E | l | Выражения |
| i |
| iM |
| lM |
| i(W)M |
| i(W) |
| i()M |
| i() |
| (E)M |
| (E) |
| A | l | Параметры стандартной функции strcmp |
| i |
| i(W) |
| i() |
| B | s(A,A) | Вызов стандартной функции strcmp |
| Y | (EcE){N}X | Условный оператор |
| (EcE){N} |
| (BcE){N}X |
| (BcE){N} |
| X | :{N} | Альтернативный блок |
| :?Y |
| W | i,W | Подвыражения |
| l,W |
| i |
| l |
| F | ti,F | Параметры функции |
| ti |
| M | vE M | Арифметические выражения |
| vE |

4.3. Построение конечного магазинного автомата

Формальное описание автомата с магазинной памятью можно представить в виде

– M = <Q, V, Z, δ, q0, z0, F>, где:

* Q – множество состояний управляющего устройства;

– V – алфавит входных символов;

– Z – специальный алфавит магазинных символов;

– δ – функция переходов автомата, представляющая собой множество правил, описанных в таблице 4.1;

– q0 ∈ Q – начальное состояние автомата;

– z0 ∈ Z – начальное состояние магазина (маркер дна);

– F⊆ Q – множество конечных состояний.

4.4. Основные структуры данных

Описание основных структур, описывающих контекстно-свободную грамматику, представлено в таблице 4.2.

Таблица 4.2 – Описание основных структур синтаксического анализатора

|  |  |
| --- | --- |
| Структура | Описание |
| Greibach | Представляет полную грамматику Грейбах, состоящую из множества правил.  Поля:  – size: Количество правил в грамматике (short).  – startN: Стартовый символ грамматики (нетерминал).  – stbottomT: Специальный символ, обозначающий дно стека.  – rules: Указатель на массив правил (Rule\*). Методы:  – getRule(GRBALPHABET pnn, Rule& prule): Возвращает правило по указанному нетерминалу и сохраняет в prule.  – getRule(short n): Возвращает правило по его индексу. |
| Rule | Представляет правило в грамматике Грейбах.  Поля:   * nn: Левый символ правила (нетерминал, < 0). * iderror: Идентификатор диагностического сообщения (int). * size: Количество правых частей правила (short). * chains: Массив цепочек правых частей правила (Chain\*).   Методы:   * getCChain(char\* b): Возвращает правую часть правила как строку. * T(char t): Преобразует символ в терминал. * N(char n): Преобразует символ в нетерминал. * isT(GRBALPHABET s): Проверяет, является ли символ терминалом. * isN(GRBALPHABET s): Проверяет, является ли символ нетерминалом. * alphabet\_to\_char(GRBALPHABET s): Преобразует GRBALPHABET в символ char. |

Окончание таблицы 4.2

|  |  |
| --- | --- |
| Структура | Описание |
| Rule::Chain | Описывает цепочку символов (правую часть правила). Поля:   * size: Длина цепочки (short). * nt: Массив символов цепочки (GRBALPHABET\*), где   терминалы > 0, а нетерминалы < 0. |

4.5. Описание алгоритма синтаксического разбора

1. В стек (магазин) помещается стартовый символ и символ дна стека “$”.
2. Из таблицы лексем формируется входная лента.
3. Выбирается цепочка, которая соответствует текущему нетерминальному символу. Эта цепочка записывается в магазин.
4. Если терминалы из вершины стека и ленты совпадают – данный терминал удаляется из ленты и с вершины стека. Иначе происходит возврат к последнему сохраненному состоянию, после чего происходит переход к пункту 3, выбирая следующую подходящую цепочку.
5. Если в магазине встречается нетерминал, то происходит сохранение текущего состояния и переход к пункту 3.
6. Если мы достигаем дна стека и лента пуста – синтаксический анализ выполнен успешно. Если достигнуто дно стека, но лента не пуста – генерируется исключение.

4.6. Структура и перечень сообщений синтаксического анализатора

Возможные ошибки при синтаксическом анализе представлены в таблице 4.3.

Таблица 4.3 – Перечень сообщений синтаксического анализатора

|  |  |
| --- | --- |
| Код ошибки | Текст ошибки |
| 601 | Ошибочный оператор |
| 602 | Ошибка в выражении |
| 603 | Ошибка в параметрах функции или ее использовании |
| 604 | Ошибка в параметрах вызываемой функции |
| 605 | Ошибка в условии |
| 606 | Ошибка в условном операторе |
| 607 | Ошибка в операторе else |
| 608 | Ошибка в арифметическом выражении |
| 609 | Ошибка в параметрах функции strcmp |
| 610 | Ошибка в функции strcmp |

4.7. Параметры синтаксического анализатора и режимы его работы

Синтаксический анализатор принимает на вход таблицы лексем и идентификаторов, а также единственный параметр, задающийся из командной строки, /cst который включает вывод дерева разбора в консоль.

4.8. Принцип обработки ошибок

Ниже представлен принцип обработки ошибок синтаксического анализа.

1. Синтаксический анализатор последовательно проверяет правила грамматики и их цепочки, чтобы найти соответствие для конструкции, заданной в таблице лексем.
2. Если подходящая цепочка отсутствует, генерируется ошибка, отражающая данную ситуацию с пояснениями.
3. Все выявленные ошибки сохраняются в общей структуре, предназначенной для их регистрации.
4. По завершении процесса анализа в протокол и на консоль выводятся диагностические сообщения о найденных ошибках.

Всего в структуре может быть сохранено до 3 ошибок. В случае, когда их больше, старые ошибки перезаписываются новыми. При возникновении ошибки анализ продолжается для выявления других возможных ошибок.

4.9. Контрольный пример

**Результат работы протокола синтаксического разбора и дерево разбора представлены в приложении В**

5. Разработка семантического анализатора

Семантический анализ — третья фаза работы транслятора, предназначенная для проверки исходного текста на соответствие семантическим правилам языка.

На вход семантический анализатор принимает информацию из таблиц лексем и идентификаторов. Результаты работы семантического анализатора записываются в файл протокола.

5.2. Функции семантического анализатора

Полный перечень семантических проверок приведен в таблице 5.1.

Таблица 5.1 – Перечень семантических проверок

|  |  |  |
| --- | --- | --- |
| № | Проверка | Фаза |
| 1 | Наличие точки входа main | Лексический |
| 2 | Отсутствие переопределения идентификатора | Лексический |
| 3 | Усекание длины имени идентификатора до 25 символов | Лексический |
| 4 | Объявление идентификатора перед его использованием | Лексический |
| 5 | Не допускается совпадения названий идентификаторов с ключевыми словами | Лексический |
| 6 | Присвоение идентификатору значения, соответствующего его типу | Лексический и семантический |
| 7 | Соответствие типов формальных и фактических параметров при вызове функций | Семантический |
| 8 | Соответствие числа параметров при объявлении функции и ее вызове | Семантический |
| 9 | Корректное использование операций сравнения и арифметических операций | Семантический |
| 10 | Использование строго инициализированной переменной | Семантический |
| 11 | Соответствие типов при различном их использовании в выражениях | Семантический |
| 12 | Возвращаемое значение инструкцией return должно совпадать типу возвращаемого функцией значения, указанного при ее объявлении. | Семантический |
| 13 | Деление на ноль | Время выполнения программы |

Семантический анализатор реализован множеством функций, которые осуществляют проверку определенных правил. Например, функция void detect\_cycles\_in\_references(const IT::ID\_Table& ID\_Table, const LT::LexTable& LEX\_Table) проверяет, есть ли «цикличные» вызовы функций: когда функции вызывают друг друга или сами себя.

5.3. Структура и перечень сообщений семантического анализатора

При работе семантического анализатора могут возникать различные ошибки. Например, при неправильном использовании идентификаторов и их значений. В таком случае семантический анализатор выводит в протокол и на консоль соответствующие данные.

Таблица 5.2 – Перечень кодов ошибок семантического анализатора и их сообщения

|  |  |
| --- | --- |
| Код ошибки | Текст ошибки |
| 700 | Ошибка 700. Строка {}: переменная {} не может быть использована в выражении, так как она не инициализирована. |
| 702 | ошибка 702. Строка {}: операции сравнения не могут быть применены к данным разных типов. |
| 703 | Ошибка 703. Строка {}: операции сравнения не могут быть применены данным типа char. |
| 704 | Ошибка 704. Строка {}: оператор write не может участвовать в выражениях. |
| 705 | Ошибка 705. Строка {}: функция print не может участвовать в выражениях. |
| 706 | Ошибка 706. Строка {}: результат сравнения используется вне контекста сравнения. |
| 707 | Ошибка 707. Строка {}: выражение содержит тип(ы), которые не поддерживают операцию {}. |
| 708 | Ошибка 708. Строка {}: тип {} не совместим с типом {}. |
| 709 | Ошибка 709. Строка {}: аргумент функции {} имеет тип {}, но ожидается {}. |
| 710 | Ошибка 710. Строка {}: Присваивание значения функции непосредственно переменной недопустимо. |
| 711 | Ошибка 711. Строка {}: арифметические операции не могут быть применены к данным разных типов или сравнениям. |
| 712 | Ошибка 712. Строка {}, число передаваемых аргументов в функцию {} не соответствует ее прототипу. |
| 713 | Ошибка 713. Строка {}: Функция {} ссылается на {} и образует цикл. |

Примечание – в таблице 5.2 символы “{}” заменяются на номер строки, имена переменных, функций, операторов и другие динамические значения.

****5.4 Принцип обработки ошибок****

Ошибки, возникающие в процессе семантического анализа, фиксируются в протокол и дублируются на консоль. При этом выполнение семантического анализатора продолжается. После завершения анализа, если возникли ошибки, работа транслятора будет завершена.

5.5 Контрольный пример

В таблице 3.3 представлен пример семантической проверки. Полный перечень контрольного примера приведен пункте 2 раздела 8.

Таблица 3.3 – Пример теста контрольного примера

|  |  |
| --- | --- |
| Исходный код | Сообщение |
| main  {  str s = "test";  int z = 12 + s;  } | Ошибка 707. Строка 4: выражение содержит тип(ы), которые не поддерживают операцию + |

6.1 Вычисление выражений

6.1. Выражения, допускаемые языком

В языке GMS-2024 допускаются вычисления выражений целочисленного типа данных с поддержкой вызова функций внутри выражений. Приоритет операций представлен на таблице 6.1.

Таблица 6.1. – Приоритеты операций

|  |  |
| --- | --- |
| Операция | Значение приоритета |
| ( ) | 3 |
| \* | 2 |
| / | 2 |
| + | 1 |
| - | 1 |

Сначала вычисляются операторы внутри скобок, затем умножение или деление, после сложение или вычитание.

6.2 Польская запись и принцип её построения

Польская запись является способом представления выражений, который заключается в помещении операндов и операций в стек, для облегчения их вычислений.

Алгоритм построения польской записи:

* Если встречается операнд, он добавляется в результирующую строку преобразования.
* Если встречается вызов функции, он помещается в стек.
* Если встречается операция, то:
  + Если стек не пуст, приоритет текущей операции сравнивается с приоритетом операции на вершине стека. Пока приоритет операции из вершины стека выше или равен приоритету текущей операции, операция извлекается из стека и переносится в результирующую строку.
  + Текущая операция помещается в стек. Если стек изначально был пуст, операция сразу добавляется в стек.
* Если встречается открывающая скобка, она помещается в стек.
* Если встречается закрывающая скобка, операции извлекаются из стека и добавляются в результирующую строку до тех пор, пока не встретится открывающая скобка. Открывающая скобка также извлекается из стека.
* Если выражение полностью обработано, но стек не пуст, все оставшиеся операции извлекаются из стека и добавляются в результирующую строку.

6.3 Программная реализация обработки выражений

Программная реализация преобразования выражений в польскую запись представлена в приложении Г.

6.4 Контрольный пример

Пример преобразования выражения приведен в таблице 6.2.

Таблица 6.2 – пример преобразования в польскую запись

|  |  |
| --- | --- |
| Выражение | Польская запись |
| 12 \* 3 - 1 | 12 3 \* 1 - |
| 12 \* (3 - 1) | 12 3 1 - \* |

7. Генерация кода

7.1 Структура генератора кода

Трансляция языка GMS-2024 происходит в язык ассемблера. Входными данными генератора кода является промежуточное представление исходного кода в виде таблиц лексем и идентификаторов.

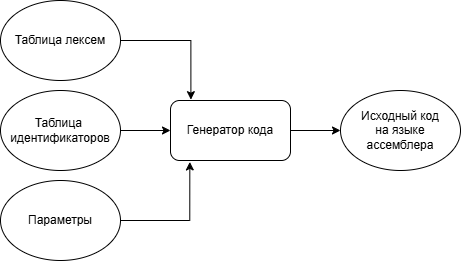


Рисунок 7.1 – структура генератора кода языка GMS-2024

7.2. Представление типов данных в оперативной памяти

Литералы и идентификаторы расположены в сегментах .const и .data соответственно. Соответствие типов данных в языке GMS-2024 типам в языке ассемблера приведено в таблице 7.1

Таблица 7.1 – Соответствие типов языка GMS-2024 типам языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип на языке GMS-2024 | Тип на языке ассемблера | Пояснение |
| int | sword | Хранит знаковые целые значения. |
| char | byte | Хранит 1 байт. |
| str | dword | Хранит адрес строкового литерала в сегменте .const, который представлен массивом типа byte. |

7.3. Статическая библиотека

Стандартная библиотека языка GMS-2024 является статической. Она содержит функции, разработанные на языке c++. Подключение библиотеки происходит с помощью директивы includelib.

Объявление прототипов ее функций происходит с использованием оператора PROTO.

Содержание стандартной библиотеки приведено в таблице 7.2.

Таблица 7.2 – Функции стандартной библиотеки

|  |  |
| --- | --- |
| Прототип | Описание |
| void StreamWriteStr(char\* arr) | Выводит строку в консоль, обрабатывая специальные символы, такие как n или t. |
| void NullDevisionException() | Сообщает об ошибке деления на ноль и завершает программу с кодом ошибки. |
| void StreamWriteChar(char\* arr) | Выводит первый символ строки в консоль. |
| void StreamWriteNumber(short n) | Выводит число в консоль. |
| void StreamWriteBool(bool b) | Выводит логическое значение true или false в консоль. |
| void PrintNumber(short n) | Выводит сообщение Number и значение числа в консоль. |
| void PrintBool(bool b) | Выводит сообщение Condition и логическое значение true или false в консоль. |
| void PrintStr(char\* arr) | Выводит сообщение Str и строку в консоль, обрабатывая специальные символы. |
| void PrintChar(char\* arr) | Выводит сообщение Char и первый символ строки в консоль. |
| int StrCmp(const char\* str1, const char\* str2) | Сравнивает две строки. Возвращает -1, если первая строка меньше второй, 1, если больше, и 0, если равны |

7.4. Особенности алгоритма генерации кода

В языке GMS-2024 генерация кода основана на таблицах лексем и идентификаторов.

Обобщенная схема работы генератора кода представлена на рисунке 7.2



Рисунок 7.2 – схема работы алгоритма генерации

Разработанные функции генерации кода представлены в таблице 7.3

Таблица 7.3 – Функции генерации кода

|  |  |
| --- | --- |
| Название | Описание |
| CD::CodeGeneration::generate\_math\_expressions | Генерирует код для математических выражений. |
| CD::CodeGeneration::\_\_s\_const | Генерирует код для сегмента констант. |
| CD::CodeGeneration::\_\_s\_data | Генерирует код для сегмента данных. |
| CD::CodeGeneration::parse\_function\_call | Парсит вызов функции. |
| CD::CodeGeneration::parse\_function | Парсит функцию (начало и конец). |
| CD::CodeGeneration::\_\_parse\_function\_body | Парсит тело функции. |
| CD::CodeGeneration::parse\_lexem | Парсит лексемы в результирующие инструкции. |

Окончание таблицы 7.3

|  |  |
| --- | --- |
| Название | Описание |
| CD::CodeGeneration::\_\_parse\_lexem\_equal | Парсит лексему для выражений присваивания. |
| CD::CodeGeneration::\_\_parse\_return\_lexem | Парсит возврат из функции. |
| CD::CodeGeneration::\_\_parse\_id\_lexem | Парсит идентификаторы. |
| CD::CodeGeneration::parse\_expression | Парсит выражение и возвращает параметры результата. |
| CD::CodeGeneration::parse\_print\_lexem\_\_ | Парсит лексемы печати. |
| CD::CodeGeneration::get\_string\_value | Возвращает текущее значение строки по индексу лексемы. |
| CD::CodeGeneration::generateCode | Основная функция. Генерирует финальный код и сохраняет его в файл. |
| CD::IfElseGeneration::cmp\_op\_to\_jmp | Конвертирует операцию сравнения в команду перехода. |
| CD::IfElseGeneration::compare\_ints | Генерирует инструкции для сравнения целых чисел. |
| CD::IfElseGeneration::compare\_strings | Генерирует инструкции для сравнения строк. |
| CD::IfElseGeneration::\_\_start\_if | Генерирует начало блока if. |
| CD::IfElseGeneration::\_\_start\_else | Генерирует начало блока else. |
| CD::IfElseGeneration::\_\_end\_if\_or\_else | Завершает блок if или else. |
| CD::IfElseGeneration::\_\_end\_expression | Завершает выражение в блоках if/else. |
| CD::IfElseGeneration::generate\_if\_statement | Генерирует полную конструкцию if-выражения. |
| CD::IfElseGeneration::generate\_label | Генерирует уникальные метки для блоков кода. |

7.5. Входные параметры, управляющие генерацией кода

На вход генератор кода принимает параметры, представленные в таблице 7.4.

Таблица 7.4 – Входные параметры генератора кода

|  |  |
| --- | --- |
| Название | Описание |
| -stack | Указывает размер стека выходного приложения. |
| -fo | Указывает имя выходного .exe файла. |

7.6. Контрольный пример

Результат работы генератора кода на основании контрольного примера из приложения А приведен в приложении Д.

8. Тестирование транслятора

8.1. Общие положения

Тестирование выполняется для выявления проблема и проверки транслятора на работоспособность.

8.2 Результаты тестирования

В таблице 8.1 приведены результаты тестирования всех фаз транслятора.

Таблица 8.1 – Результаты тестирования транслятора.

|  |  |
| --- | --- |
| Исходный код | Сообщение транслятора |
| int main{  write "error";  } | Ошибка 153: Нельзя явно указать тип для точки входа main - он всегда int. Строка: 1 |
| main{  write "error";  }  main{  write "error";  } | Ошибка 140: Может быть только одна точка входа main. Строка: 4 |
| main{  int x = 'a';  } | Ошибка 708. Строка 2: тип литерала не совместим с типом int |
| main{  int x = 0xffffff;  } | Ошибка в числовом литерале '0xffffff': Значение слишком большое для типа  Ошибка 149: Неккоректный литерал. Строка: 2 |
| main{  int x = 0xfff;  str a = x;  } | Ошибка 708. Строка 3: тип идентификатора не совместим с типом str |
| main{  str a = x;  } | Ошибка 142: Идентификатор использован, но не объявлен. Строка: 2 |
| main{  int x = 12 + x;  } | Ошибка 700. Строка 2: переменная x не может быть использована в выражении, так как она неинициализирована. |
| main{  str a = "x";  int x = 12 + a;  } | Ошибка 707. Строка 3: выражение содержит тип(ы), которые не поддерживают операцию + |

Окончание таблицы 8.1

|  |  |
| --- | --- |
| Исходный код | Сообщение транслятора |
| int sum(){return 12;}  main{  int x = 12 + sum(52);  } | Ошибка 712. Строка 3, число передаваемых аргументов в функцию sum не соответствует ее прототипу |
| int sum(int x){return 12;}  main{  int x = 12 + sum("52");  } | Ошибка 709. Строка 3: аргумент функции sum имеет тип str, но ожидается int |
| main{  if (12){  print("?");  }  } | Ошибка 608: строка 3, Ошибка в арифметическом выражении |
| main  {  if (12 < "52")  {  print("?" +12);  }  } | ошибка 702. Строка 3: операции сравнения не могут быть применены к данным разных типов.  Ошибка 707. Строка 5: выражение содержит тип(ы), которые не поддерживают операцию + |
| main  {  print(52 /);  } | Ошибка 602: строка 3, Ошибка в выражении |
| main  {  print(12 < "12");  } | ошибка 702. Строка 3: операции сравнения не могут быть применены к данным разных типов. |

Заключение

В процессе выполнения курсовой работы был разработан транслятор для языка программирования GMS-2024 с полной функциональной поддержкой. В результате были достигнуты следующие цели:

* Определена спецификация языка GMS-2024;
* Разработаны конечные автоматы и ключевые алгоритмы для эффективной работы лексического анализатора;
* Реализован лексический анализатор, распознающий допустимые цепочки конструкций языка;
* Разработана контекстно-свободная грамматика, приведённая к нормальной форме Грейбах, для описания синтаксически корректных конструкций;
* Выполнена программная реализация синтаксического анализатора;
* Создан семантический анализатор для проверки инструкций на соответствие логическим правилам;
* Разработан транслятор кода в язык ассемблера;
* Проведено тестирование всех компонентов системы.

Финальная версия языка GMS-2024 включает:

* 3 типа данных;
* Оператор потокового вывода;
* Возможность использования функций стандартной библиотеки;
* Наличие 4 арифметических операторов для вычисления выражений;
* Поддержка пользовательских функций и условного оператора;
* Структурированная и классифицированная система для обработки ошибок пользователя.

Проделанная работа позволила получить необходимое представление о структурах и процессах, использующихся при построении трансляторов, а также основные различия и преимущества тех или иных средств трансляции.

Список использованных источников

1. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. - 429 с.

2. Habr – Польская нотация или как легко распарсить алгебраическое выражение [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/search/>.

3. Герберт, Ш. Справочник программиста по C/C++ / Шилдт Герберт. - 3-е изд. – Москва : Вильямс, 2003. – 429 с.

4. Вирт, Н. Построение компиляторов / Никлаус Вирт. – Москва : 2010. – 192 с.

Приложение А

Контрольный пример

Листинг 1 – исходный код на языке GMS-2024

|  |
| --- |
| int sum(int x, int y) { return x + y; }  str get\_str(int x)  {  if (x < 10) { return "x меньше 10";}    else if (x == 10) { return "x равно 10";}  return "x больше 10";  }  int calc(int x, int y, int z) { return sum(x, y) \* z + (x - y);}  main  {  // проверка возврата строки функцией  str str\_1 = get\_str(0xff);  if (strcmp(str\_1, "x больше 10") == 0) {  print("Cool"); // должно быть выведено  }  if (sum(1, 2) == 3) {  write "aboba\n"; // должно быть выведено  }  char ch = '\n';  print(ch); // должно быть выведено  // проверка вычисления выражений  print(sum(12, 1) \* 2 - sum(-1, -5)); // 32  print(calc(12, 2, 3)); // 52  print(12 == 12); // True  // проверка условных оператор  int a = sum(12, 3); // 15  int b = sum(2, 13); // 15  int x = sum(a, b) + 100; // 130  if (x == 100) { // нет  print("неправда");  }  else if (x == sum(13, 7) + 10 \* 10 + 10) {// 130, истина  print("130, выходим...");  return 10;  }  } |

Приложение Б

Листинг 1 – таблица лексем, сгенерированная для контрольного примера

|  |  |  |  |
| --- | --- | --- | --- |
| Line | Lexem | Line | Lexem |
| 01 | ti(ti,ti){rivi;} | 32 | p(i); |
| 03 | ti(ti) | 35 | p(i(l,l)vlvi(l,l)); |
| 04 | { | 36 | p(i(l,l,l)); |
| 05 | ?(icl){ | 37 | p(lcl); |
| 06 | rl; | 40 | ti=i(l,l); |
| 07 | } | 41 | ti=i(l,l); |
| 08 | :?(icl){ | 42 | ti=i(i,i)vl; |
| 09 | rl; | 44 | ?(icl){ |
| 10 | } | 45 | p(l); |
| 11 | rl; | 46 | } |
| 12 | } | 47 | :?(ici(l,l)vlvlvl){ |
| 14 | ti(ti,ti,ti){ | 48 | p(l); |
| 15 | ri(i,i)viv(ivi); | 49 | rl; |
| 16 | } | 50 | } |
| 18 | m | 51 | } |
| 19 | { |  |  |
| 21 | ti=i(l); |  |  |
| 23 | ?(s(i,l)cl){ |  |  |
| 24 | p(l); |  |  |
| 25 | } |  |  |
| 27 | ?(i(l,l)cl){ |  |  |
| 28 | wl; |  |  |
| 29 | } |  |  |
| 31 | ti=l; |  |  |

Листинг 2 – таблица идентификаторов, сгенерированная для контрольного примера

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | Datatype | ID Type | Line | Value | Scope |
| sum | INT | F | 1 | 0 | - |
| x | INT | P | 1 | 0 | sum |
| y | INT | P | 1 | 0 | sum |
| get\_str | STR | F | 3 |  | - |
| x | INT | P | 3 | 0 | get\_str |
| L0 | INT | L | 5 | 10 | - |
| L1 | STR | L | 5 | x меньше 10 | - |
| L2 | STR | L | 5 | x равно 10 | - |
| L3 | STR | L | 5 | x больше 10 | - |
| calc | INT | F | 14 | 0 | - |
| x | INT | P | 14 | 0 | calc |
| y | INT | P | 14 | 0 | calc |
| z | INT | P | 14 | 0 | calc |
| main | INT | F | 18 | 0 | - |
| str\_1 | STR | V | 21 |  | main |
| L4 | INT | L | 21 | 255 | - |
| L6 | INT | L | 23 | 0 | - |
| L7 | STR | L | 23 | Cool | - |
| L8 | INT | L | 27 | 1 | - |
| L9 | INT | L | 27 | 2 | - |
| L10 | INT | L | 27 | 3 | - |
| L11 | STR | L | 27 | aboba\n | - |
| ch | CHAR | V | 31 |  | main |
| L12 | CHAR | L | 31 | \n | main |
| L13 | INT | L | 35 | 12 | - |
| L14 | INT | L | 35 | -1 | - |
| L15 | INT | L | 35 | -5 | - |
| a | INT | V | 40 | 0 | main |
| b | INT | V | 41 | 0 | main |
| ID | Datatype | ID Type | Line | Value | Scope |
| L16 | INT | L | 41 | 13 | - |
| x | INT | V | 42 | 0 | main |
| L17 | INT | L | 42 | 100 | - |
| L18 | STR | L | 42 | неправда | - |
| L19 | INT | L | 47 | 7 | - |
| L20 | STR | L | 5 | 130, выходим... | - |

Приложение В

Листинг 1 – Грамматика языка GMS-2024

|  |
| --- |
| Greibach greibach(NS('S'), TS('$'),  10,  Rule(NS('S'), GRB\_ERROR\_SERIES + 0,  6,  Rule::Chain(12, TS('t'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S')),  Rule::Chain(11, TS('t'), TS('i'), TS('('), NS('F'), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S')),  Rule::Chain(11, TS('t'), TS('i'), TS('('), TS(')'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S')),  Rule::Chain(10, TS('t'), TS('i'), TS('('), TS(')'), TS('{'), TS('r'), NS('E'), TS(';'), TS('}'), NS('S')),  Rule::Chain(7, TS('m'), TS('{'), NS('N'), TS('r'), NS('E'), TS(';'), TS('}')),  Rule::Chain(4, TS('m'), TS('{'), NS('N'), TS('}'))  ),  Rule(NS('N'), GRB\_ERROR\_SERIES + 1,  22,  Rule::Chain(6, TS('t'), TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(4, TS('t'), TS('i'), TS(';'), NS('N')),  Rule::Chain(5, TS('i'), TS('='), NS('E'), TS(';'), NS('N')),  Rule::Chain(4, TS('r'), NS('E'), TS(';'), NS('N')),  Rule::Chain(6, TS('p'), TS('('), NS('E'), TS(')'), TS(';'), NS('N')),  Rule::Chain(8, TS('p'), TS('('), NS('E'), TS('c'), NS('E'), TS(')'), TS(';'), NS('N')),  Rule::Chain(4, TS('w'), NS('E'), TS(';'), NS('N')),  Rule::Chain(6, TS('w'), NS('E'), TS('c'), NS('E'), TS(';'), NS('N')),  Rule::Chain(3, TS('?'), NS('Y'), NS('N')),  Rule::Chain(6, TS('i'), TS('('), NS('W'), TS(')'), TS(';'), NS('N')),  Rule::Chain(5, TS('i'), TS('('), TS(')'), TS(';'), NS('N')),  Rule::Chain(5, TS('t'), TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(3, TS('t'), TS('i'), TS(';')),  Rule::Chain(4, TS('i'), TS('='), NS('E'), TS(';')),  Rule::Chain(3, TS('r'), NS('E'), TS(';')),  Rule::Chain(5, TS('p'), TS('('), NS('E'), TS(')'), TS(';')),  Rule::Chain(7, TS('p'), TS('('), NS('E'), TS('c'), NS('E'), TS(')'), TS(';')),  Rule::Chain(3, TS('w'), NS('E'), TS(';')),  Rule::Chain(5, TS('w'), NS('E'), TS('c'), NS('E'), TS(';')),  Rule::Chain(2, TS('?'), NS('Y')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), TS(';')),  Rule::Chain(4, TS('i'), TS('('), TS(')'), TS(';'))  ),  Rule(NS('E'), GRB\_ERROR\_SERIES + 2,  10,  Rule::Chain(1, TS('l')),  Rule::Chain(1, TS('i')),  Rule::Chain(2, TS('i'), NS('M')),  Rule::Chain(2, TS('l'), NS('M')),  Rule::Chain(5, TS('i'), TS('('), NS('W'), TS(')'), NS('M')),  Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),  Rule::Chain(4, TS('i'), TS('('), TS(')'), NS('M')),  Rule::Chain(3, TS('i'), TS('('), TS(')')),  Rule::Chain(4, TS('('), NS('E'), TS(')'), NS('M')),  Rule::Chain(3, TS('('), NS('E'), TS(')'))  ),  Rule(NS('A'), GRB\_ERROR\_SERIES + 9,  4,  Rule::Chain(1, TS('l')),  Rule::Chain(1, TS('i')),  Rule::Chain(4, TS('i'), TS('('), NS('W'), TS(')')),  Rule::Chain(3, TS('i'), TS('('), TS(')'))  ),  Rule(NS('B'), GRB\_ERROR\_SERIES + 10,  1,  Rule::Chain(6, TS('s'), TS('('), NS('A'), TS(','), NS('A'), TS(')'))  ),  Rule(NS('Y'), GRB\_ERROR\_SERIES + 6,  4,  Rule::Chain(9, TS('('), NS('E'), TS('c'), NS('E'), TS(')'), TS('{'), NS('N'), TS('}'), NS('X')),  Rule::Chain(8, TS('('), NS('E'), TS('c'), NS('E'), TS(')'), TS('{'), NS('N'), TS('}')),  Rule::Chain(9, TS('('), NS('B'), TS('c'), NS('E'), TS(')'), TS('{'), NS('N'), TS('}'), NS('X')),  Rule::Chain(8, TS('('), NS('B'), TS('c'), NS('E'), TS(')'), TS('{'), NS('N'), TS('}'))  ),  Rule(NS('X'), GRB\_ERROR\_SERIES + 7,  2,  Rule::Chain(4, TS(':'), TS('{'), NS('N'), TS('}')),  Rule::Chain(3, TS(':'), TS('?'), NS('Y'))  ),  Rule(NS('W'), GRB\_ERROR\_SERIES + 3,  4,  Rule::Chain(3, TS('i'), TS(','), NS('W')),  Rule::Chain(3, TS('l'), TS(','), NS('W')),  Rule::Chain(1, TS('i')),  Rule::Chain(1, TS('l'))  ),  Rule(NS('F'), GRB\_ERROR\_SERIES + 4,  2,  Rule::Chain(4, TS('t'), TS('i'), TS(','), NS('F')),  Rule::Chain(2, TS('t'), TS('i'))  ),  Rule(NS('M'), GRB\_ERROR\_SERIES + 8,  2,  Rule::Chain(3, TS('v'), NS('E'), NS('M')),  Rule::Chain(2, TS('v'), NS('E'))  )  ); |

Листинг 2 – Структура магазинного автомата

|  |
| --- |
| typedef short GRBALPHABET;  #define GRB\_ERROR\_SERIES 600  #define NS(n) GRB::Rule::Chain::N(n)  #define TS(n) GRB::Rule::Chain::T(n)  #define ISNS(n) GRB::Rule::Chain::isN(n)  namespace GRB  {  struct Rule  {  GRBALPHABET nn;  int iderror;  short size;  struct Chain  {  short size;  GRBALPHABET\* nt;  Chain()  {  this->size = 0;  this->nt = 0;  };  Chain(short psize, GRBALPHABET s, ...);  char\* getCChain(char\* b);  static GRBALPHABET T(char t) { return GRBALPHABET(t); };  static GRBALPHABET N(char n) { return -GRBALPHABET(n); };  static bool isT(GRBALPHABET s) { return s > 0; };  static bool isN(GRBALPHABET s) { return !isT(s); };  static char alphabet\_to\_char(GRBALPHABET s)  {  return isT(s) ? char(s) : char(-s);  };  }\*chains;  Rule()  {  this->nn = 0x00;  this->size = 0;  }  Rule(GRBALPHABET pnn, int iderroe, short psize, Chain c, ...);  char\* getCRule(char\* b, short nchain);  short getNextChain(GRBALPHABET t, Rule::Chain& pchain, short j);  };  struct Greibach  {  short size;  GRBALPHABET startN;  GRBALPHABET stbottomT;  Rule\* rules;  Greibach() { this->size = 0; this->startN = 0; this->stbottomT = 0; this->rules = 0; };  Greibach(GRBALPHABET pstartN, GRBALPHABET pstbottomT, short psize, Rule r, ...);  short getRule(GRBALPHABET pnn, Rule& prule);  Rule getRule(short n);  };  Greibach getGreibach();  } |

Листинг 3 – Дерево разбора для контрольного примера

|  |
| --- |
| 0 : S->ti(F){rE;}S  3 : F->ti,F  6 : F->ti  11 : E->iM  12 : M->vE  13 : E->i  16 : S->ti(F){NrE;}S  19 : F->ti  23 : N->?Y  24 : Y->(EcE){N}X  25 : E->i  27 : E->l  30 : N->rE;  31 : E->l  34 : X->:?Y  36 : Y->(EcE){N}  37 : E->i  39 : E->l  42 : N->rE;  43 : E->l  47 : E->l  50 : S->ti(F){rE;}S  53 : F->ti,F  56 : F->ti,F  59 : F->ti  64 : E->i(W)M  66 : W->i,W  68 : W->i  70 : M->vEM  71 : E->i  72 : M->vE  73 : E->(E)  74 : E->iM  75 : M->vE  76 : E->i  80 : S->m{N}  82 : N->ti=E;N  85 : E->i(W)  87 : W->l  90 : N->?YN  91 : Y->(BcE){N}  92 : B->s(A,A)  94 : A->i  96 : A->l  99 : E->l  102 : N->p(E);  104 : E->l  108 : N->?YN  109 : Y->(EcE){N}  110 : E->i(W)  112 : W->l,W  114 : W->l  117 : E->l  120 : N->wE;  121 : E->l  124 : N->ti=E;N  127 : E->l  129 : N->p(E);N  131 : E->i  134 : N->p(E);N  136 : E->i(W)M  138 : W->l,W  140 : W->l  142 : M->vEM  143 : E->l  144 : M->vE  145 : E->i(W)  147 : W->l,W  149 : W->l  153 : N->p(E);N  155 : E->i(W)  157 : W->l,W  159 : W->l,W  161 : W->l  165 : N->p(EcE);N  167 : E->l  169 : E->l  172 : N->ti=E;N  175 : E->i(W)  177 : W->l,W  179 : W->l  182 : N->ti=E;N  185 : E->i(W)  187 : W->l,W  189 : W->l  192 : N->ti=E;N  195 : E->i(W)M  197 : W->i,W  199 : W->i  201 : M->vE  202 : E->l  204 : N->?Y  205 : Y->(EcE){N}X  206 : E->i  208 : E->l  211 : N->p(E);  213 : E->l  217 : X->:?Y  219 : Y->(EcE){N}  220 : E->i  222 : E->i(W)M  224 : W->l,W  226 : W->l  228 : M->vEM  229 : E->l  230 : M->vEM  231 : E->l  232 : M->vE  233 : E->l  236 : N->p(E);N  238 : E->l  241 : N->rE;  242 : E->l |

Приложение Г

Листинг 1 – Программная реализация механизма преобразования в ПОЛИЗ

|  |
| --- |
| unordered\_map <std::string, std::vector<std::string>> used\_functions;    int getPrecedence(char op) {  if (op == '+' || op == '-') return 1;  if (op == '\*' || op == '/') return 2;  if (op == '(') return 0;  return -1;  }  bool isOperator(char ch) {  return ch == '+' || ch == '-' || ch == '\*' || ch == '/';  }  vector<string> CD::CodeGeneration::\_\_parse\_math\_expression(const std::vector<int>& expression) {  stack<char> operators;  vector<string> output;  for (int id = expression.front(); id <= expression.back(); id++) {  if (LEX\_TABLE.table[id].lexema[0] == LEX\_ID && ID\_TABLE.table[LEX\_TABLE.table[id].idxTI].idtype == IT::IDTYPE::F)  {  std::string functionName = get\_id\_name\_in\_data\_segment(ID\_TABLE.table[LEX\_TABLE.table[id].idxTI]);  auto function = user\_functions[functionName];  stack<char> parenthesis;  parenthesis.push(LEX\_LEFTTHESIS);  std::vector<int> function\_call\_args;  id += 2;  while (id <= expression.back())  {  if (LEX\_TABLE.table[id].lexema[0] == LEX\_RIGHTTHESIS)  {  parenthesis.pop();  if (parenthesis.size() == 0)  {  break;  }  }  if (LEX\_TABLE.table[id].lexema[0] == LEX\_LEFTTHESIS)  {  parenthesis.push(id);  }  if (LEX\_TABLE.table[id].lexema[0] == LEX\_ID || LEX\_TABLE.table[id].lexema[0] == LEX\_LITERAL)  {  functionName += get\_id\_name\_in\_data\_segment(ID\_TABLE.table[LEX\_TABLE.table[id].idxTI]);  }  function\_call\_args.push\_back(id++);  }  if (used\_functions.find(functionName) == used\_functions.end())  {  used\_functions[functionName] = parse\_function\_call(function, function\_call\_args.front(), function\_call\_args.back());  }  output.push\_back(functionName);  }  else if (LEX\_TABLE.table[id].lexema[0] == LEX\_ID || LEX\_TABLE.table[id].lexema[0] == LEX\_LITERAL) {  output.push\_back(get\_id\_name\_in\_data\_segment(ID\_TABLE.table[LEX\_TABLE.table[id].idxTI]));  }  else if (LEX\_TABLE.table[id].lexema[0] == LEX\_MATH)  {  while (!operators.empty() &&  getPrecedence(operators.top()) >= getPrecedence(LEX\_TABLE.table[id].v)) {  output.push\_back(string(1, operators.top()));  operators.pop();  }  operators.push(LEX\_TABLE.table[id].v);  }  else if (LEX\_TABLE.table[id].lexema[0] == LEX\_LEFTTHESIS) {  operators.push('(');  }  else if (LEX\_TABLE.table[id].lexema[0] == LEX\_RIGHTTHESIS) {  while (!operators.empty() && operators.top() != LEX\_LEFTTHESIS) {  output.push\_back(string(1, operators.top()));  operators.pop();  }  if (!operators.empty()) operators.pop();  }  }  while (!operators.empty()) {  output.push\_back(string(1, operators.top()));  operators.pop();  }  return output;  }  void generateOperation(std::vector<std::string>& masmCode, const std::string& operation)  {  if (operation == "+") {  masmCode.push\_back("add ax, bx");  }  else if (operation == "-") {  masmCode.push\_back("sub ax, bx");  }  else if (operation == "\*") {  masmCode.push\_back("imul bx");  }  else if (operation == "/") {  masmCode.push\_back("cmp ebx, 0");  masmCode.push\_back("je division\_error");  masmCode.push\_back("cdq");  masmCode.push\_back("idiv bx");  }  else {  throw runtime\_error("Неизвестный оператор: " + operation);  }  }  void generateMASM(std::vector<std::string>& masmCode, const vector<string>& rpn) {  if (rpn.size() == 1)  {  if (used\_functions.find(rpn[0]) != used\_functions.end())  {  masmCode.push\_back("; function call");  masmCode.insert(masmCode.end(), used\_functions[rpn[0]].begin(), used\_functions[rpn[0]].end());  return;  }  masmCode.push\_back("push " + rpn[0]);  return;  }  if (rpn.size() == 3 &&  isOperator(rpn[2][0]))  {  if (used\_functions.find(rpn[0]) != used\_functions.end()  && used\_functions.find(rpn[1]) != used\_functions.end())  {  masmCode.push\_back("; function call");  masmCode.insert(masmCode.end(), used\_functions[rpn[1]].begin(), used\_functions[rpn[1]].end());  masmCode.push\_back("push ax");  masmCode.push\_back("; function call");  masmCode.insert(masmCode.end(), used\_functions[rpn[0]].begin(), used\_functions[rpn[0]].end());  masmCode.push\_back("pop bx");  }  else if (used\_functions.find(rpn[0]) != used\_functions.end())  {  masmCode.push\_back("; function call");  masmCode.insert(masmCode.end(), used\_functions[rpn[0]].begin(), used\_functions[rpn[0]].end());  masmCode.push\_back("mov bx, " + rpn[1]);  }  else if (used\_functions.find(rpn[1]) != used\_functions.end())  {  masmCode.push\_back("; function call");  masmCode.insert(masmCode.end(), used\_functions[rpn[1]].begin(), used\_functions[rpn[1]].end());  masmCode.push\_back("mov bx, ax");  masmCode.push\_back("mov ax, " + rpn[0]);  }  else  {  masmCode.push\_back("mov ax, " + rpn[0]);  masmCode.push\_back("mov bx, " + rpn[1]);  }  generateOperation(masmCode, rpn[2]);  return;  }  for (const string& token : rpn) {  if (token.size() > 1) {  if (used\_functions.find(token) != used\_functions.end())  {  masmCode.push\_back("; function call");  masmCode.insert(masmCode.end(), used\_functions[token].begin(), used\_functions[token].end());  masmCode.push\_back("push ax");  }  else  {  masmCode.push\_back("push " + token);  }  }  else if (isOperator(token[0])) {  masmCode.push\_back("pop bx");  masmCode.push\_back("pop ax");  generateOperation(masmCode, token);  masmCode.push\_back("push ax");  }  else {  throw runtime\_error("Неизвестный токен: " + token);  }  }  if (masmCode.back() == "push ax")  {  masmCode.pop\_back();  }  }  std::vector<std::string> CD::CodeGeneration::generate\_math\_expressions(const std::vector<int>& expr)  {  std::vector<std::string> masmCode;  vector<string> result = \_\_parse\_math\_expression(expr);  generateMASM(masmCode, result);  return masmCode;  } |

Приложение Д

Листинг 1 – Результат генерации кода для контрольного примера

|  |
| --- |
| .586  .model flat, stdcall  includelib gms2024stdlib.lib  ExitProcess PROTO : DWORD  \_\_PrintNumber PROTO :SDWORD  \_\_PrintBool PROTO :BYTE  \_\_PrintArray PROTO :DWORD, :DWORD, :DWORD  \_\_PrintStr PROTO :DWORD  \_\_PrintChar PROTO :DWORD  \_\_StreamWriteNumber PROTO :SDWORD  \_\_StreamWriteBool PROTO :BYTE  \_\_StreamWriteArray PROTO :DWORD, :DWORD, :DWORD  \_\_StreamWriteStr PROTO :DWORD  \_\_StreamWriteChar PROTO :DWORD  \_\_StrCmp PROTO :DWORD, :DWORD  \_\_NullDevisionException PROTO  .stack 8192  .const  \_\_INT\_L0 sword 10  \_\_STR\_L1 db "x меньше 10", 0  \_\_STR\_L2 db "x равно 10", 0  \_\_STR\_L3 db "x больше 10", 0  \_\_INT\_L4 sword 255  \_\_INT\_L6 sword 0  \_\_STR\_L7 db "Cool", 0  \_\_INT\_L8 sword 1  \_\_INT\_L9 sword 2  \_\_INT\_L10 sword 3  \_\_STR\_L11 db "aboba\n", 0  \_\_CHAR\_L12 db 10  \_\_INT\_L13 sword 12  \_\_INT\_L14 sword -1  \_\_INT\_L15 sword -5  \_\_INT\_L16 sword 13  \_\_INT\_L17 sword 100  \_\_STR\_L18 db "неправда", 0  \_\_INT\_L19 sword 7  \_\_STR\_L20 db "130, выходим...", 0  .data  @bool\_RESERVED byte ?  \_INT\_PARAM\_@sum\_\_x sword 0  \_INT\_PARAM\_@sum\_\_y sword 0  \_INT\_PARAM\_@get\_str\_\_x sword 0  \_INT\_PARAM\_@calc\_\_x sword 0  \_INT\_PARAM\_@calc\_\_y sword 0  \_INT\_PARAM\_@calc\_\_z sword 0  \_STR\_@main\_\_str\_1 dword ?  \_CHAR\_@main\_\_ch byte ?  \_INT\_@main\_\_a sword 0  \_INT\_@main\_\_b sword 0  \_INT\_@main\_\_x sword 0  .code  sum proc  start:  mov ax, [esp + 4]  mov \_INT\_PARAM\_@sum\_\_x, ax  mov ax, [esp + 6]  mov \_INT\_PARAM\_@sum\_\_y, ax  mov ax, \_INT\_PARAM\_@sum\_\_x  mov bx, \_INT\_PARAM\_@sum\_\_y  add ax, bx  sum\_END:  ret 4  division\_error:  call \_\_NullDevisionException  sum endp  get\_str proc  start:  mov ax, [esp + 4]  mov \_INT\_PARAM\_@get\_str\_\_x, ax  mov ax, \_INT\_PARAM\_@get\_str\_\_x ; lefthand операнд  mov bx, 10 ; righthand операнд    cmp ax, bx  mov @bool\_RESERVED, 0  jl @true0  jmp @false0  @true0:  mov @bool\_RESERVED, 1  @false0:  ; Начало if0 (x<10)  cmp @bool\_RESERVED, 1  je IF\_TRUE\_0  jmp ELSE\_0  IF\_TRUE\_0:  mov eax, offset \_\_STR\_L1  jmp get\_str\_END  jmp IF\_END\_0 ; Переход к выходу из 0 if-else  ELSE\_0:  mov ax, \_INT\_PARAM\_@get\_str\_\_x ; lefthand операнд  mov bx, 10 ; righthand операнд    cmp ax, bx  mov @bool\_RESERVED, 0  je @true1  jmp @false1  @true1:  mov @bool\_RESERVED, 1  @false1:  ; Начало if1 (x=10)  cmp @bool\_RESERVED, 1  je IF\_TRUE\_1  jmp IF\_END\_1  IF\_TRUE\_1:  mov eax, offset \_\_STR\_L2  jmp get\_str\_END  jmp IF\_END\_1 ; Переход к выходу из 1 if-else  IF\_END\_1:  jmp IF\_END\_0 ; Переход к выходу из 0 if-else  IF\_END\_0:  mov eax, offset \_\_STR\_L3  jmp get\_str\_END  get\_str\_END:  ret 2  division\_error:  call \_\_NullDevisionException  get\_str endp  calc proc  start:  mov ax, [esp + 4]  mov \_INT\_PARAM\_@calc\_\_x, ax  mov ax, [esp + 6]  mov \_INT\_PARAM\_@calc\_\_y, ax  mov ax, [esp + 8]  mov \_INT\_PARAM\_@calc\_\_z, ax  ; function call  push \_INT\_PARAM\_@calc\_\_y  push \_INT\_PARAM\_@calc\_\_x  call sum  push ax  push \_INT\_PARAM\_@calc\_\_z  pop bx  pop ax  imul bx  push ax  push \_INT\_PARAM\_@calc\_\_x  push \_INT\_PARAM\_@calc\_\_y  pop bx  pop ax  sub ax, bx  push ax  pop bx  pop ax  add ax, bx  calc\_END:  ret 6  division\_error:  call \_\_NullDevisionException  calc endp  main proc  start:  push \_\_INT\_L4  call get\_str  mov \_STR\_@main\_\_str\_1, eax  push offset \_\_STR\_L3  push \_STR\_@main\_\_str\_1  call \_\_StrCmp  push eax  movzx eax, \_\_INT\_L6  push eax  pop ebx  pop eax  cmp eax, ebx  mov @bool\_RESERVED, 0  je @true2  jmp @false2  @true2:  mov @bool\_RESERVED, 1  @false2:  ; Начало if2 (s(str\_1,x больше 10)=0)  cmp @bool\_RESERVED, 1  je IF\_TRUE\_2  jmp IF\_END\_2  IF\_TRUE\_2:  push offset \_\_STR\_L7  call \_\_PrintStr  jmp IF\_END\_2 ; Переход к выходу из 2 if-else  IF\_END\_2:  ; Вычисляем righthand операнд:  push \_\_INT\_L10  ; Вычисляем lefthand операнд:  ; function call  push \_\_INT\_L9  push \_\_INT\_L8  call sum  ; значение lefhand уже в eax  pop bx ; загружаем значение righthand операнда  cmp ax, bx  mov @bool\_RESERVED, 0  je @true3  jmp @false3  @true3:  mov @bool\_RESERVED, 1  @false3:  ; Начало if3 (sum(1,2)=3)  cmp @bool\_RESERVED, 1  je IF\_TRUE\_3  jmp IF\_END\_3  IF\_TRUE\_3:  push offset \_\_STR\_L11  call \_\_StreamWriteStr  jmp IF\_END\_3 ; Переход к выходу из 3 if-else  IF\_END\_3:  mov al, \_\_CHAR\_L12  mov \_CHAR\_@main\_\_ch, al  push offset \_CHAR\_@main\_\_ch  call \_\_PrintChar  ; function call  push \_\_INT\_L8  push \_\_INT\_L13  call sum  push ax  push \_\_INT\_L9  pop bx  pop ax  imul bx  push ax  ; function call  push \_\_INT\_L15  push \_\_INT\_L14  call sum  push ax  pop bx  pop ax  sub ax, bx  push eax  call \_\_PrintNumber  push \_\_INT\_L10  push \_\_INT\_L9  push \_\_INT\_L13  call calc  push eax  call \_\_PrintNumber  mov ax, 12 ; lefthand операнд  mov bx, 12 ; righthand операнд    cmp ax, bx  mov @bool\_RESERVED, 0  je @true4  jmp @false4  @true4:  mov @bool\_RESERVED, 1  @false4:  movzx ax, @bool\_RESERVED  push ax  call \_\_PrintBool  push \_\_INT\_L10  push \_\_INT\_L13  call sum  mov \_INT\_@main\_\_a, ax  push \_\_INT\_L16  push \_\_INT\_L9  call sum  mov \_INT\_@main\_\_b, ax  ; function call  push \_INT\_@main\_\_b  push \_INT\_@main\_\_a  call sum  mov bx, \_\_INT\_L17  add ax, bx  mov \_INT\_@main\_\_x, ax  mov ax, \_INT\_@main\_\_x ; lefthand операнд  mov bx, 100 ; righthand операнд    cmp ax, bx  mov @bool\_RESERVED, 0  je @true5  jmp @false5  @true5:  mov @bool\_RESERVED, 1  @false5:  ; Начало if4 (x=100)  cmp @bool\_RESERVED, 1  je IF\_TRUE\_4  jmp ELSE\_4  IF\_TRUE\_4:  push offset \_\_STR\_L18  call \_\_PrintStr  jmp IF\_END\_4 ; Переход к выходу из 4 if-else  ELSE\_4:  ; Вычисляем righthand операнд:  ; function call  push \_\_INT\_L19  push \_\_INT\_L16  call sum  push ax  push \_\_INT\_L0  push \_\_INT\_L0  pop bx  pop ax  imul bx  push ax  pop bx  pop ax  add ax, bx  push ax  push \_\_INT\_L0  pop bx  pop ax  add ax, bx  push ax    mov ax, \_INT\_@main\_\_x ; lefthand операнд  pop bx ; загружаем значение righthand операнда  cmp ax, bx  mov @bool\_RESERVED, 0  je @true6  jmp @false6  @true6:  mov @bool\_RESERVED, 1  @false6:  ; Начало if5 (x=sum(13,7)+10\*10+10)  cmp @bool\_RESERVED, 1  je IF\_TRUE\_5  jmp IF\_END\_5  IF\_TRUE\_5:  push offset \_\_STR\_L20  call \_\_PrintStr  movzx eax, \_\_INT\_L0  push eax  jmp main\_END  jmp IF\_END\_5 ; Переход к выходу из 5 if-else  IF\_END\_5:  jmp IF\_END\_4 ; Переход к выходу из 4 if-else  IF\_END\_4:  push 0  main\_END:  call ExitProcess  division\_error:  call \_\_NullDevisionException  main endp  END main |