МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Разработка компилятора SMV-2019»

Выполнил студент Шихов Максим Владиславович

(Ф.И.О.)

Руководитель проекта ст. преп. Наркевич Аделина Сергеевна

(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Пацей Наталья Владимировна

(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультант ст. преп. Наркевич Аделина Сергеевна

(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер ст. преп. Наркевич Аделина Сергеевна

(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2019

# Введение

Транслятор – это комплекс отдельных программ, позволяющих преобразовывать исходный код на одном языке программирования в исходный код на другом языке программирования.

Классический транслятор состоит из следующих частей:

1. Лексический анализатор;
2. Синтаксический анализатор;
3. Семантический анализатор:
4. Генератор кода.

Основной целью данной курсовой работы является разработка транслятора для языка программирования SMV-2019.

Исходя из цели курсового проекта, были определены следующие задачи:

– разбработка спецификации языка программирования;

– разбратка структуры транслятора;

– разработка лексического анализатора;

– разработка синтаксического анализатора;

– разработка семантического анализатора;

– обработка выражений;

– генерация кода в байт-код;

– трансляция байт-кода в язык ассемблера;

– тестирование транслятора.

Решения каждой из поставленных задач буду приведены в соответствующих главах курсового проекта, а именно :

1. спецификация языка программирования;
2. структура транслятора;
3. лексический анализатор;
4. синтаксический анализатор;
5. семантический анализатор;
6. преобразование выражений;
7. генерация байт-кода;
8. трансляция байт-кода в язык ассемблера;
9. тестирование транслятора.

# Спецификация языка программирования

## Характеристика языка программирования

Язык программирования SMV-2019 является универсальным языком высокого уровня, интерпретация которого происходит сначала в байт-код, а затем в код целевой платформы. Язык не является объектно-ориентированным в языке строгая типизация. В SMV-2019 используется три типа данных: целочисленные, строковые и логические.

## **Алфавит языка**

Алфавит языка SMV-2019 основан на кодировке Windows-1251[1], представленной на рисунке 1.1.

Рисунок 1.1. Алфавит входных символов

Исходный код SMV-2019 может содержать символы латинского алфавита большого и малого регистров, цифры десятичной системы счисления от 0 до 9, русские символы разрешены только в строковых литералах.

## **Символы-сепараторы**

Символы, которые являются сепараторами представлены в таблице 1.1.

Таблица 1.1. Сепараторы

|  |  |
| --- | --- |
| Сепаратор | Назначение |
| ;  ‘ ‘ (пробел)  , | Разделение инструкций |
| (  ) | Приоритетность операций(в выражениях) |
| [  ] | Параметры функций |

## Применяемые кодировки

Для написания исходного кода на языке программирования используется кодировка SMV-2019. При трансляции исходного кода из байт-кода в язык ассемблера символы перекодируются из Windows-1251 кодировки, которая представлена в пункте 1.2, в кодировку CP866. Отличия кодировки CP866 от Windows-1251 представлены на рисунке 1.2.

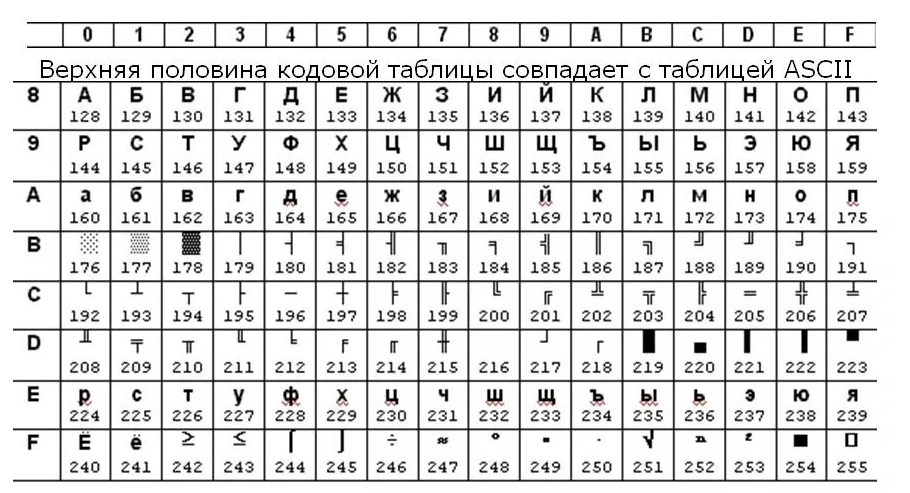


Рисунок 1.2. Отличия кодировки CP866 от Windows-1251

## Типы данных

В языке SMV-2019 есть 3 типа данных: знаковый целочисленный, строковый и логический. Описание типов данных, предусмотренных в данным языке представлено в таблице 1.2.

Таблица 1.2. Типы данных языка SMV-2019

|  |  |
| --- | --- |
| Тип данных | Описание типа данных |
| integer | Фундаментальный тип данных. Предусмотрен для объявления целочисленных данных (4 байта).  Автоматически инициализируется нулевым значением.  Возможные операции:  *арифметические*  + – бинарный, суммирование;  - – бинарный, вычитание;  \* – бинарный, умножение;  = – присваивание значения;  < - бинарный, сравнение;  > - бинарный, сравнение;  mod – бинарный, взятие остатка;  div – бинарный, взятие целой части от деления |
| str | Фундаментальный тип данных. Предусмотрен для объявления строк. (1 символ – 1 байт). Автоматическая инициализация строкой нулевой длины. Максимальное количество символов в строке – 254.  Возможные операции:  = – копирование одной строки в другую ; |

## Преобразование типов данных

В языке программирования SMV-2019 преобразование типов данных не поддерживается. Язык является строго типизируемым.

## Идентификаторы

В имени идентификатора допускаются символы латинского алфавита нижнего

и верхнего регистра ([a-z], [A-Z]), также имя идентификатора может содержать цифры ([0-9]). Имя идентификатора не должно совпадать с двоичным, восьмеричным или шестнадцатеричным литералом. Имя идентификатора не должно совпадать с именами ключевых слов. Максимальная длина идентификатора 20 символов. (в случае, если имя идентификатора превышает максимальную длину, применяется усечение, и происходит информирование пользователя). Максимальное число идентификаторов 4294967295 (232-1).

## Литералы

В языке существует 4 типа литералов целого типа: шестнадцатеричный, восьмеричный, двоичный, десятичный. В SMV-2019 представлены и строковые литералы. Все литералы являются rvalue. Описание литералов представлено в таблице 1.3.

Таблица 1.3. Описание литералов

|  |  |
| --- | --- |
| Тип литерала | Описание |
| Шестнадцатеричный | Может состоять из чисел [0-9], а также символов [a-f]. Оканчивается суффиксом  -h. Минимальное значение равно 0h, максимальное 7fffffffh. Не содержит дробную часть. При выходе за пределы будет произведен сдвиг бит, по принципу  С++. |
| Десятичный | Может состоять из чисел [0-9]. Минимальное значение равно 0, максимальное 2147483647. При выходе за пределы будет произведен сдвиг бит, по принципу С++. |
| Восьмеричный | Может состоять из чисел [0-7]. Оканчивается суффиксом –o. Минимальное значение равно 0o, максимальное 17777777777‬o. При выходе за пределы будет произведен сдвиг бит по принципу С++. |
| Строковый | Символы, заключённые в " " (двойные кавычки), инициализируются как строковые переменные. |

## Область видимости идентификаторов

Область видимости реализована сверху-вниз (по принципу С++). Все функции и процедуры имеют глобальную область видимости и могут быть объявлены только в глобальной области видимости. Язык поддерживает глобальную область видимости для переменных. Любые идентификаторы должны быть объявлены до их использования.

## Инициализация данных

Переменные объявляются при помощи конструкции:

value: <тип данных> <идентификатор>;

Для инициализации переменной каким-либо значением применяется конструкция:

<идентификатор> = <значение>;

Допустимой является следующая конструкция:

value: <тип данных> <идентификатор> = <значение>;

Когда объявляется переменная типа integer, она автоматически инициализируется нулевым значением, переменная типа str – строкой нулевой длинны соответственно ("").

## Инструкции языка

Все возможные инструкции языка программирования SMV-2019 представлены в общем виде в таблице 1.4.

Таблица 1.4. Инструкции языка программирования SMV-2019

|  |  |
| --- | --- |
| Инструкция | Запись на языке SMV-2019 |
| Объявление переменной | value: <тип данных> <идентификатор> ; |
| Присваивание | integer и str:  <идентификатор> = <литерал>;  <идентификатор> = <выражение>;  <идентификатор1> = <идентификатор2>; |
| Объявление внешней функции | function: <тип данных> <идентификатор> [<тип данных> <идентификатор>, ...] |
| Объявление процедуры | procedure: <идентификатор> |
| Блок инструкций | begin  …  end |
| Условная инструкция | if (<выражение>)  then  begin  …  end  else  begin  …  end |
| Вызов функции | <идентификатор функции>(<идентификатор / литерал>, …) |
| Вызов процедуры | <Идентификатор процедуры>; |
| Возврат из подпрограммы | return <выражение>; |
| Вывод данных | print (<идентификатор> / <литерал>); |

## Операции языка

Язык программирования SMV-2019 может выполнять арифметические операции, представленные в таблице 1.5.

Таблица 1.5. Приоритетности операций языка программирования SMV-2019

|  |  |  |
| --- | --- | --- |
| Операция | Арифметическое назначение | Приоритетность операции |
| () | Приоритетность операций | 1 |
| \* | Умножение | 2 |
| div | Деление | 2 |
| mod | Взятие остатка от деления | 2 |
| + | Сложение | 3 |
| - | Вычитание | 3 |
| > | Сравнение чисел | 3 |
| < | Сравнение чисел | 3 |

Максимальным значением приоритетности является “1”, минимальным “3” соответственно.

Также языком поддерживается условная операция сравнения знаковых операндов. Данная операция имеет самый низкий приоритет.

## Выражения и их вычисления

Выражения могут быть записаны в несколько строк. Круглые скобки в выражении используются для изменения приоритета операций. Также не допускается запись двух подряд идущих арифметических операций. В выражениях могут присутствовать вызовы функций. Реализация выражений происходит с помощью обратной польской записи.

## Программные конструкции языка

Ключевые программные конструкции языка программирования SMV-2019 представлены в таблице 1.6.

Таблица 1.6. Программные конструкции языка SMV-2019

|  |  |
| --- | --- |
| Главная функция(точка входа) | main  begin  <инструкции языка>  …  end |

Продолжение таблицы 1.6.

|  |  |
| --- | --- |
| Функция | function: <тип данных> <идентификатор>  [<тип данных> <идентификатор>, ...]  begin  <инструкции языка>  …  end |
| Процедура | procedure: <идентификатор>  begin  <инструкции языка>  …  end |

В языке SMV-2019 функции поддерживают, возврат значений. Процедуры и главная функция не поддерживают возврат значений

## Область видимости

Сверху-вниз по принципу С++. Поддерживается объявление глобальных переменных. Функции должны быть объявлены перед точкой входа.

## Семантические проверки

Семантические проверки невозможно проверить при помощи грамматик.

На этап лексического анализа код проверяется на наличие следующих ошибок:

– повторное объявление функции или процедуры

– повторное объявление переменной

– использование необъявленного идентификатора

На этапе семантического анализа код проверяется на наличие следующих ошибок:

– несоответствие параметров при вызове и при объявлении функции

– несоответствие типов левой и правой части выражения

– не найдено объявление функции

– не найдено объявление процедуры

– недопустимо использование математических операций над строками

– недопустимый вызов процедуры

– несоответствие типов возвращаемого значения и оператора return

– ошибка в условии оператора if

– ошибка в операторе print

Язык SMV-2019 предусматривает следующие виды предупреждений:

– неинициализированный идентификатор (применена инициализация по умолчанию)

– имя идентификатора, превышает 20 символов

– не найдено ни одного оператора return у функции

## Распределение оперативной памяти на этапе выполнения

Код, транслированный в язык ассемблера, использует две области памяти.

В сегмент констант заносятся все литералы. В сегмент данных заносятся переменные. Параметры функций объявляются при объявлении функции. Локальная область видимости реализуется за счёт генерирования уникальных имён идентификаторов.

## Стандартная библиотека и её состав

Функции стандартной библиотеки и их описание представлено в таблице 1.7.

Таблица 1.7. Состав стандартной библиотеки

|  |  |  |  |
| --- | --- | --- | --- |
| Имя функции | Возвращаемое значение | Принимаемые параметры | Описание |
| clength | integer | str parm1 - строка | Функция вычисляет длину строки parm1 |
| cmp | boolean | str parm1 – строка  str parm2 – строка | Функция выполняет побайтовое сравнение двух строк parm1 и parm2 |
| print | отсутствует | integer parm1 - число | Функция выводит на консоль число parm1 |
| print | отсутствует | str parm1 - строка | Функция выводит на консоль строку parm1 |

Стандартная библиотека реализована на языке ассемблера, добавляется в генерируемый код автоматически.

## Ввод и вывод данных

Ввод данных не поддерживается языком программирования SMV-2019. Вывод данных на консоль осуществляется при помощи функции print. Данная функция входит в состав стандартной библиотеки и описана в таблице 1.7.

## Точка входа

Точкой входа является функция main. Точка входа не может отсутствовать, также не может быть переопределена. В программе должна быть только одна точка входа.

## Препроцессор

Препроцессор в языке программирования SMV-2019 не предусмотрен.

## Соглашения о вызовах

В языке вызов функций происходит по соглашению о вызовах stdcall[3]. Особенности stdcall:

– все параметры функции передаются через стек;

– память высвобождает вызываемый код;

– занесение в стек параметров идёт справа налево.

## Объектный код

SMV-2019 транслируется в байт-код, затем в язык ассемблера, а далее - в объектный код.

## Классификация сообщений транслятора

В случае возникновения ошибки в коде программы на языке SMV-2019 и выявления её транслятором на экран консоли выводится сообщение, уникальный код ошибки, а трансляция останавливается. В случае возникновения предупреждения, трансляция продолжается, а сообщение выводится на экран консоли.

## Контрольный пример

Контрольный пример представлен в приложении А. На его примере будут показаны результаты работы всех этапов трансляции исходного кода на языке программирования SMV-2019 в язык ассемблера.

# 

# Структура транслятора

### Компоненты транслятора, их назначение и принципы взаимодействия

Все компоненты, входящие в состав транслятора SMV-2019 представлены на рисунке 2.1.



Рисунок 2.1. Структура транслятора языка программирования SMV-2019

Лексический анализатор – часть транслятора, выполняющая лексический анализ. На данном этапе распознаётся правильность составления лексем и идентификаторов. На вход лексического анализатора приходит файл с исходным кодом. По окончании данного этапа получаем таблицу лексем и таблицу идентификаторов.

Синтаксический анализатор – часть транслятора, выполняющая синтаксический анализ. Проверяется правильность расположения идентификаторов и ключевых слов в исходном коде. Для того, чтобы провести данную операцию используются таблица лексем и идентификаторов. На выходе получаем дерево разбора.

Семантический анализатор – часть транслятора, выполняющая семантический анализ, то есть исходный код проверяется на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Генератор кода – часть транслятора, выполняющая генерацию байт-кода на основе полученных данных на предыдущих этапах трансляции. На вход генератора подаются таблица лексем и таблица идентификаторов, на основе которых генерируется файл с ассемблерным кодом.

Транслятор байт-кода – часть транслятора, выполняющая трансляцию байт-кода в код целевой платформы, в данном случае это языке ассемблера.

### Перечень входных параметров транслятора

Входные параметры, которые можно указать транслятору представлены в таблице 2.1.

Табл. 2.1 Входные параметры транслятора

|  |  |
| --- | --- |
| Входной параметр | Описание параметра |
| -in:\*.txt | Файл с исходным кодом на языке SMV-2019, имеющий расширение \*.txt |
| -out:\*.txt | Параметр для вывода таблицы лексем и идентификаторов, полученных на этапе лексического анализа, а также трассировки синтаксического разбора и вывода дерева разбора |
| -log:\*.log | Параметр для вывода протокола работы транслятора |

Притом параметр –in:\*.txt должен быть указан обязательно. В случае, если он не будет задан, выполнение трансляции не начнётся. Параметры –out:\*.txt и

–log:\*.txt не являются обязательными. В случае, если они не будут заданы их имена сгенерируются автоматически, путём добавления соответствующего расширения к параметру –in.

### Перечень протоколов, формируемых транслятором и их содержимое

Протоколы работы транслятора приведены в таблице 2.2.

Табл. 2.2 Протоколы транслятора

|  |  |
| --- | --- |
| Формируемый протокол | Описание выходного протокола |
| “log.txt” | Файл с протоколом работы транслятора языка программирования SMV-2019. |
| “out.txt” | Файл, содержащий отладочную информацию языка программирования SMV-2019. |

### Перечень входных параметров транслятора байт-кода

Единственным входным параметром транслятора байт-кода является путь к файлу, с расширением .bin, в котором содержится правильный байт-код. Предполагается, что байт-код получаемый транслятором байт-кода будет верифицирован.

### Перечень файлов, получаемых в результате работы транслятора байт-кода

В результате работы транслятора байт-кода, получаем файл с расширением .asm, в котором содержится, код на языке целевой платформы, в данном случае код на языке ассемблера.

# Разработка лексического анализатора

#### Структура лексического анализатора

Структура лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1. Структура лексического анализатора

Лексический анализатор – часть транслятора, выполняющая лексический анализ. На данном этапе распознаётся правильность составления лексем и идентификаторов. На вход лексического анализатора приходит файл с исходным кодом. По окончании данного этапа получаем таблицу лексем и таблицу идентификаторов.

Обязательным входным параметром для лексического анализатора является файл с исходным кодом программы на языке программирования SMV-2019, имеющий расширение \*.txt.

Выходными параметрами являются таблицы лексем и идентификаторов, которые заполняются в процессе лексического анализа.

#### Контроль входных символов

Исходный код на языке программирования SMV-2019 прежде чем транслироваться проверяется на допустимость символов. То есть изначально из входного файла считывается по одному символу и проверяется является ли он разрешённым.

Для проверки символов была записана структура на языке программирования C++ аналогичная таблице кодировке Windows-1251. Данная структура представлена на рисунке 3.2.

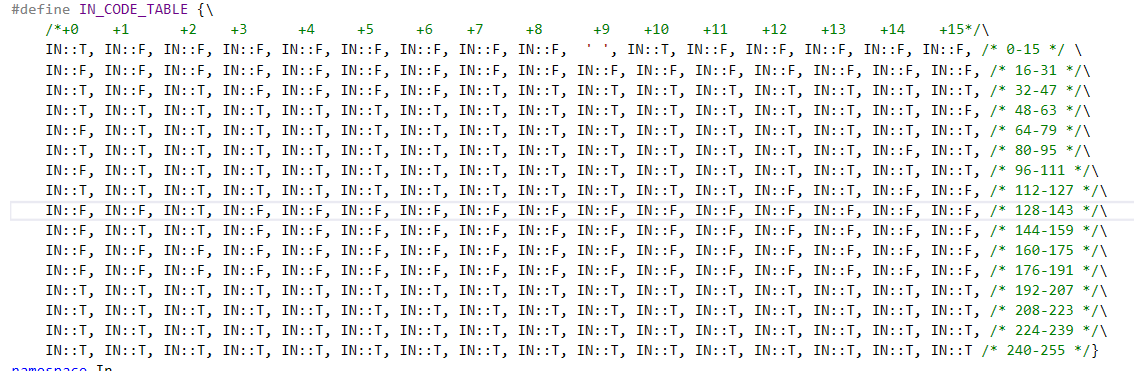


Рисунок 3.2. Таблица допустимых символов

Каждый считанный из файла с исходным кодом символ, проверяется на допустимость. В случае, если символ помечен значением T, то записывается в строку char\*, F – позиция данного символа записывается в структуру ошибок и в случае обнаружения хотя бы одного недопустимого символа сработает обработчик исключений и на консоль будут выведены позиции запрещённых символов в исходном файле. Если код символа, заменен в таблице другим символом, то происходит замена данного символа на указанный в исходном коде.

#### Удаление избыточных символов

Избыточными символами представлены: последовательность из двух и более пробелов, табуляций.

Алгоритм удаления избыточных символов:

1. Просматриваем текущий символ
2. Если флаг опущен и очередной символ - это пробел или табуляция записываем в обработанный текст, и поднимаем флаг.
3. Если флаг поднят и очередной символ пробел либо табуляция переходим к пункту 1.
4. Если это любой другой допустимый символ, записываем его в обработанный текст и опускаем флаг.

#### Перечень ключевых слов, сепараторов, символов операций и соответствующих им лексем и конечных автоматов

Каждому токену соответствует символ-лексема. Соответствие токенов и лексем приведено в таблице 3.1.

Табл. 3.1. Возможные токены

|  |  |  |
| --- | --- | --- |
| Токен | Лексема | Примечание |
| integer | t | Знаковый тип данных |
| str | t | Строковый тип данных |
| value: | o | Объявление переменной |
| procedure: | w | Объявление процедуры |
| function: | f | Объявление функции |
| return | r | Возврат значения из функции |
| main | m | Главная функция |
| print | p | Функция вывода в консоль |
| clength | z | Функции стандартной библиотеки |
| cmp | c |
| if | j | Условный оператор |
| then | q | Метка истины |
| else | n | Метка ложного условия |
| + | v | Знаки арифметических операций |
| - | v |
| \* | v |
| mod | v |
| div | v |
| < | v |
| > | v |
| = | = | Оператор присваивания |
| begin | b | Операторные скобки |
| end | e |
| [ | [ | Сепараторы |
| ] | ] |
| ( | ( |
| ) | ) |
| ; | ; |
| , | , |

Для каждой фразы также соответствует автомат, по которому происходит разбор выражения. Автомат является – детерминированным, то есть имеет конечное количество состояний.

Все конечные автоматы, записанные в виде структур на языке программирования С++, представлены в приложении Б.

Также, на рисунке 5.1 представлен пример графа перехода состояний конечного автомата.

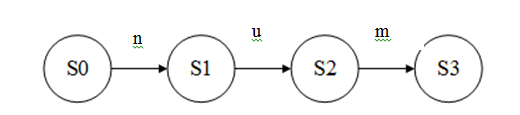


Рисунок 3.1. Граф переходов цепочки str

Аналогично можно построить графы переходов для конечного автомата для разбора всех ключевых слов, которые допустимы в языке SMV-2019.

#### Основные структуры данных

Основными структурами данных, используемыми на этапе лексического анализа являются записи в таблице лексем, записи в таблице идентификаторов, сами таблица лексем и таблица идентификаторов. Все они представлены в приложении Б.

#### Принцип обработки ошибок

В случае, если лексический анализатор обнаруживает ошибку в лексике в исходном коде он заполняет структуру, содержащую ошибку, произошедшую в процессе лексического анализа. Максимально допустимое число ошибок равно единице. По достижении данного количества ошибок транслятор прекращает свою работу и выводит на экран сообщение об ошибке, найденной в исходном коде. В случае обнаружения некритичной ошибки (предупреждения) транслятор выводит на экран соответствующее сообщение, а трансляция продолжается.

#### Структура и перечень сообщений лексического анализатора

Анализатор сигнализирует о следующих ошибках и предупреждениях. Представлены в таблице 3.2.

Таблица 3.2 - Сообщения лексического анализатора

|  |  |  |
| --- | --- | --- |
| Код | Тип | Сообщение |
| 6 | Ошибка | Найдено более одной функции main |
| 7 | Ошибка | Превышена длина строкового литерала |
| 8 | Ошибка | Отсутствует функция main |
| 9 | Ошибка | Нераспознанная лексема |
| 72 | Предупреждение | Имя идентификатора превышает 20 символов, применено усечение |

#### Входные параметры лексического анализатора

Входным параметром лексического анализа является исходный текст программы. Параметры не определяют режим работы лексического анализатора

#### Алгоритм лексического анализа

Изначально лексический анализатор получает на вход исходный текст программы в отформатированном виде, после каждой цепочки символов языка присутствует пробел. Затем каждая цепочка символов обрабатывается конечными автоматами. В случае, если автомат не был подобран, выводится диагностическое сообщение. Если токен разобран, то дальнейшие действия, которые будут с ним производиться, будут зависеть от того, чем он является.

В случае, если токен является идентификатором заносим в таблицу лексем соответствующую лексему. Затем проверяется таблица идентификаторов, если это объявление, то идентификатор заносится в таблицу идентификаторов, а в таблице лексем делается ссылка на него, однако, если идентичный идентификатор уже присутствует в таблице, то генерируется ошибка о повторном объявлении идентификатора. Если это не объявление идентификатора, а его использование, то он добавляется в таблицу лексем, а в таблице идентификаторов происходит поиск, если соответствующий идентификатор не найден, генерируется ошибка об использовании необъявленного идентификатора, в случае, если он найден, в таблицу лексем добавляется ссылка на уже существующий идентификатор. Также если данный идентификатор, является названием функции или процедуры, происходит изменение области видимости, путем добавления в стек.

В случае, если токен является литералом, то заносится в таблицу идентификаторов с именем “n”, где n является номером литерала.

В случае, если токен является ключевым словом “end”, то соответствующая лексема добавляется в таблицу лексем, а также происходит изменение области видимости, а именно из стека достается последний элемент.

В случае, если токен является ключевым словом “if”, ”then”, “else” происходит изменение области видимости, токен ложится в стек. И происходит добавление соответствующей записи в таблицу лексем.

Во всех остальных случаях происходит генерация соответствующей структуры и занесение её в таблицу лексем.

#### Контрольный пример

Полученные после выполнения лексического анализа таблицы лексем и идентификаторов после разбора контрольного примера представлены в приложении В.

# 

# Разработка синтаксического анализатора

##### Структура синтаксического анализатора

Структура синтаксического анализатора представлена на рисунке 4.1.



Рисунок 4.1. Структура синтаксического анализатора

Синтаксический анализатор – часть транслятора, выполняющая синтаксический анализ. Проверяется правильность расположения идентификаторов и ключевых слов в исходном коде. Для того, чтобы провести данную операцию используются таблица лексем и идентификаторов. На выходе получаем дерево разбора и протокол работы анализатора.

##### Контекстно свободная грамматика, описывающая синтаксис языка

Синтаксис языка SMV-2019 описывается при помощи контекстно свободной грамматики.

Контекстно-свободная грамматика – грамматика типа 2 по иерархии Хомского. Данная грамматика имеет вид , где

T – множество терминальных символов,

N – множество нетерминальных символов,

P – множество правил переходов,

S – стартовый символ.

В контекстно-свободной грамматике правила имеют вид: ,

где

,

,

 - словарь грамматики .

Грамматика языка SMV-2019 представлена в приложении Г в виде заполненной структуры на языке С++. В структуре терминальные символы помечены словом TS и все они являются сепараторами, либо строчными буквами. Нетерминальные же символы помечены словом NS и являются заглавными буквами латинского алфавита.

##### Построение конечного магазинного автомата

Принцип действия конечного магазинного автомата представлен на рисунке 4.2.



Рисунок 4.2 - МП-автомат

Формальное описание МП-автомата:



 - множество состояний;

 - алфавит входных символов;

 - специальный алфавит магазинных символов;

-функция переходов автомата , где  - множество подмножеств ;

 - начальное состояние автомата;

- начальное состояние магазина (маркер дна);

- множество конечных состояний.

Конфигурация (текущее состояние автомата) описывается тройкой , где - текущее состояние автомата, - остаток цепочки,  - цепочка-содержимое магазина.

Начальное состояние , - начальное состояние автомата,  - входная цепочка, - маркер дна магазина.

Цепочка  является допустимой (распознается) автоматом , если  и .

Работа автомата 

состояние автомата 

читает символ  находящийся под головкой (сдвигает ленту);

не читает ничего (читает , не сдвигает ленту);

из  определяет новое состояние , если  или .

читает верхний (в стеке) символ  и записывает цепочку  т.к. , при этом, если , то верхний символ магазина просто удаляется.

работа автомата заканчивается 

##### Описание алгоритма синтаксического разбора

В языке SMV-2019 синтаксический разбор выполняется с помощью автомата с магазинной памятью, алгоритм работы которого изложен в пункте 4.3

##### Структура и перечень сообщений синтаксического анализатора

Перечень сообщений синтаксического анализатора содержится в таблице 4.1

Таблица 4.1 Сообщения синтаксического анализатора

|  |  |
| --- | --- |
| Номер сообщения | Содержание сообщения |
| Ошибка 600 | Неверная структура программы |
| Ошибка 601 | Ошибочный оператор |
| Ошибка 602 | Ошибка в выражении |
| Ошибка 603 | Ошибка в параметрах функции |
| Ошибка 604 | Ошибка в параметрах вызываемой функции |
| Ошибка 605 | Ошибка мат. оператора |
| Ошибка 606 | Ошибка в параметрах функции cmp[str parm1, str parm2] |
| Ошибка 607 | Ошибка в параметрах функции clength[str parm] |
| Ошибка 608 | Ошибка в операторе if … then … else … |
| Ошибка 609 | Ошибка в объявлении идентификатора |
| Ошибка 610 | Ошибка в теле процедуры |
| Ошибка 611 | Ошибка в теле main |

##### Принцип обработки ошибок

Изначально, если анализатор разбирает часть исходного кода по какому-либо правилу, то запоминает его. При возникновении ошибки синтаксический анализатор откатиться назад до правила, при помощи которого разбор был успешным, если это возможно. После чего пытается применить последующие правила из грамматики. В случае, если правило невозможно подобрать выводится сообщение об ошибке.

##### Входные и выходные данные

Входными данными являются таблицы лексем идентификаторов. Выходными же данными синтаксического анализатора являются дерево разбора и трассировка разбора исходного кода.

##### Контрольный пример

Пример разбора исходного кода на языке программирования SMV-2019 синтаксическим анализатором представлен в виде дерева разбора, которое находится в приложении Д.

# 

# Разработка семантического анализатора

###### Структура семантического анализатора

Структура семантического анализатора представлена на рисунке 5.1.



Рисунок 5.1. структура семантического анализатора

Семантический анализатор – часть транслятора, выполняющая семантический анализ, то есть исходный код проверяется на наличие ошибок, которые невозможно отследить при помощи регулярной и контекстно-свободной грамматики. Входными данными являются таблица лексем и идентификаторов.

Семантический анализатор является отдельной частью транслятора SMV-2019. Однако часть семантических ошибок, проверяются на этапе лексического анализа и связано это с алгоритмом работы лексического анализатора, который изложен в пункте 3.9.

###### Функции и алгоритм работы семантического анализатора

Главной функцией семантического анализа является, функция check(), она выполняет проходку в один цикл по таблице лексем, в ходе которого, при встрече определенной лексемы запускает соответствующие проверки:

1. При встрече лексемы функции, запускается заполнение некоторых полей записи в таблице идентификаторов, таких как количество параметров функции, а также их тип. Эти данные необходимы далее как при генерации байт-кода, так и для проверки на соответствие параметров при вызове и при объявлении функции.
2. При встрече лексемы возврата из функции, выполняется проверка типа возвращаемого выражения и типа самой функции. А также выполняется диагностика на признак, есть ли у функции хотя бы одно ключевое слово возврата.
3. При встрече вызова функции, мы проверяем ранее заполненную информацию о параметрах функции и параметрах при вызове, и в случае чего генерируем соответствующее сообщение.
4. При встрече лексемы знака равенства, мы запускаем проверку типов левой и правой части выражения, в случае несоответствия типов, генерируем соответствующее сообщение.
5. При встрече идентификатора, проверяем присвоено ли ему значение, если нет генерируем соответствующее предупреждение.
6. При встрече лексемы условного оператора, проверяем тип условного выражения. В случае, если тип выражения не является boolean или integer, генерируем ошибку.
7. Если это лексема, какой-либо из функций стандартной библиотеки, мы проверяем тип параметров данной функции, т.к. их количество учтено ещё на этапе синтаксического анализа.

Корректную работу данного алгоритма обеспечивает предварительный синтаксический анализ, за счет которого мы гарантируем, что будет подряд идущий определенный набор лексем.

###### Структура и перечень сообщений семантического анализатора

Перечень сообщений семантического анализатора представлен в таблицах 5.1. и 5.2.

Таблица 5.1. Обрабатываемые семантическим анализатором ошибки

|  |  |
| --- | --- |
| Номер сообщения | Содержание сообщения |
| Ошибка 700 | Использование необъявленного идентификатора |
| Ошибка 701 | Несоответствие параметров при вызове и при объявлении функции |
| Ошибка 702 | Повторное объявление функции или процедуры |
| Ошибка 703 | Повторное объявление процедуры |
| Ошибка 704 | Повторное объявление переменной |
| Ошибка 705 | Несоответствие типов левой и правой части выражения |
| Ошибка 706 | Несоответствие типов (операторы >,< возвращают boolean значение) |

Продолжение таблицы 5.1.

|  |  |
| --- | --- |
| Номер сообщения | Содержание сообщения |
| Ошибка 707 | Не найдено объявление функции |
| Ошибка 708 | Не найдено объявление процедуры |
| Ошибка 709 | Тип левой части выражения не определен |
| Ошибка 710 | Недопустимо использование математических операций над строками |
| Ошибка 711 | Недопустимый вызов процедуры |
| Ошибка 712 | Несоответствие типа возвращаемого значения функции и выражения return |
| Ошибка 713 | Ошибка в условии оператора if |
| Ошибка 714 | Ошибка в операторе print |

Таблица 5.2. Сообщения о предупреждениях

|  |  |
| --- | --- |
| Предупреждение 70 | Неинициализированный идентификатор (Применена инициализация по умолчанию) |
| Предупреждение 71 | Не найдено ни одного оператора return у функции |

Все ошибки семантического анализатора имеют идентификатор 7xx.

###### Принцип обработки ошибок

При обнаружении хотя бы одной ошибки транслятор выводит диагностическое сообщение и останавливает свою работу. При возникновении предупреждения выводится соответствующее сообщение, а трансляция продолжается.

###### Контрольный пример

Результат работы семантического анализа описан в главе 9.

# 

# Преобразование выражений

* 1. Выражения, допускаемые языком

В языке программирования SMV-2019 допускаются вычисление выражений со стандартными типами данных: integer и boolean (если используются операторы сравнения). В выражениях могут присутствовать операции, которые также описаны в пункте 1.12., а также функции, которые имеют соответствующий возвращемый тип.

Каждая операция в выражении имеет свой приоритет. Значения приоритетов представлены в таблице 6.1. (по убыванию)

Таблица 6.1. Приоритеты операций

|  |  |  |
| --- | --- | --- |
| Операция | Арифметическое назначение | Приоритетность операции |
| (  ) | Увеличение приоритета операции | 1 |
| \* | Умножение | 2 |
| div | Деление | 2 |
| mod | Взятие остатка от деления | 2 |
| + | Сложение | 3 |
| - | Вычитание | 3 |
| > | Сравнение чисел | 3 |
| < | Сравнение чисел | 3 |

* 1. Польская запись

Все выражения преобразовываются к обратной польской записи.

Обратная польская запись[5] – форма записи выражений, в которой операнды расположены перед знаками операций.

Алгоритм преобразования выражений к польской записи:

– читаем очередной символ;

– если это идентификатор функции, то все последующие идентификаторы и литералы помещаем в стек, до встречи закрывающей скобки, которая обозначает конец параметров, затем в выходную строку помещаем все параметры из стека, а также после них добавляем спецсимвол «!», после которого будет идти сам идентификатор функции;

– если он является идентификатором или литералом, то добавляем его к выходной строке;

– если символ является открывающей скобкой, то она помещается в стек;

– если символ является закрывающей скобкой, то выталкиваем из стека в выходную строку все символы пока не встретим открывающую скобку. При этом обе скобки удаляются и не попадают в выходную строку;

– как только входная лента закончится все символы из стека выталкиваются в выходную строку;

–в случае если встречаются операции, то выталкиваем из стека в выходную строку все операции, которые имеют выше приоритетность чем последняя операция;

Если после преобразования, получилась строка меньшей длины, чем соответствующая запись в таблице лексем, то недостающее количество символов заполняем спецсимволом «@».

* 1. Программная реализация

Программная реализация преобразования выражений к польской записи, заключается в реализации алгоритма, изложенного выше.

* 1. Примеры преобразования выражений

Примеры преобразования выражений к польской записи представлены в таблице 6.2.

Таблица 6.2. Примеры выражений в польской записи

|  |  |
| --- | --- |
| Выражение | Выражение в польской записи |
| 5\*(3+7) | 537+\* |
| i[i,l] | il!i |

# 

# Генерация байт-кода

* 1. Структура генератора кода

Структура генератора байт-кода представлена на рисунке 7.1.



Рисунок 7.1. Структура генератора байт-кода

Генератор байт-кода – часть транслятора, выполняющая генерацию байт-кода на основе полученных данных на предыдущих этапах трансляции. На вход генератора таблица лексем и таблица идентификаторов, которая прошла все вышеперечисленные этапы трансляции. На основе данных таблиц генерируется файл с байт-кодом

* 1. Структура байт-кода

Байт-код языка SMV-2019 реализует стековую машину. Все команды и данные представляются в двоичном коде и записываются в файл с расширением .bin каждая программная конструкция языка имеет своё кодирование. Каждая команда должна начинаться с сегмента, к которому принадлежит. Ниже приведены таблицы, в которых записано двоичное представление сегментов, типов данных.

Таблица 7.1. Типы данных, интерпретированные в байт-код

|  |  |
| --- | --- |
| Тип данных | Запись в байт-коде |
| integer | 01 |
| boolean | 11 |
| str | 10 |

Таблица 7.2. Коды сегментов байт-кода SMV-2019

|  |  |
| --- | --- |
| Название сегмента | Запись в байт-коде |
| Константы | 0001 |
| Данные | 0010 |

Продолжение таблицы 7.2.

|  |  |
| --- | --- |
| Название сегмента | Запись в байт-коде |
| Функции | 0011 |
| Параметры | 0100 |
| main - сегмент | 0101 |
| Сегмент базовой библиотеки | 0110 |
| Сегмент команд | 0111 |
| Сегмент процедур | 1000 |

* 1. Представление констант в байт-коде

Константы в байт-коде языка SMV-2019, как и любые данные исходного кода байт-кода начинаются с принадлежности к соответствующему сегменту т.е. перед описанием каждой константы должен быть описан сегмент, к которому она принадлежит, код сегмента констант представлен в таблице 7.2. Потом записывается имя константы(32бит), для каждой константы генерируется уникальное имя, далее следует тип данных константы, типы данных констант представляются в соответствии с таблицей 7.1. Затем записывается значение константы. Значения констант описываются согласно таблице 7.3.

Таблица 7.3. Представление значений констант в байт-коде языка SMV-2019

|  |  |
| --- | --- |
| Тип данных константы | Представление данных в байт-коде |
| boolean | 32бит (значение в двоичном коде) |
| integer | 32бит (значение в двоичном коде) |
| str | 32бит (значение длины строки в двоичном коде, обозначим как n). Далее следует n\*8бит коды ASCII символов, которые содержит данная строка. |

* 1. Представление переменных в байт-коде

Переменные в байт-коде SMV-2019 представляются следующим образом: код сегмента данных (представлен в таблице 7.2.), далее следует имя переменной (генерируется автоматически 32 бит, уникальное в своем сегменте), затем следует тип данных переменной, описан в таблице 7.1.

* 1. Представление функций в байт-коде

Функции в байт-коде языка SMV-2019 кодируются следующим образом: код сегмента функции (представлен в таблице 7.2.), уникальное имя (в своём сегменте (32 бит)) функции, далее следует тип возвращаемых данных в соответствии с таблицей 7.1. Затем следует 32 бит число (n) в двоичной системе счисления, которое обозначает количество параметров. Далее n записей в соответствии с представлением, описанным в пункте 7.4. За исключением типа сегмента, здесь будет записываться код сегмента параметров, представленный в таблице 7.2

* 1. Представление процедур в байт-коде

Представление процедур в байт-коде языка SMV-2019 реализовано следующим образом. Код сегмента процедур, представлен в таблице 7.2. Затем следует уникальное название процедуры (32 бит уникальное в своем сегменте).

* 1. Команды байт-код машины SMV-2019

Байт-код машиной SMV-2019 используется стековая архитектура, которая поддерживает виды команд, представленные в таблице 7.4.

Таблица 7.4. Команды байт-код машины SMV-2019

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Описание | Кодирование | Код команды |
| OP\_PUSH | Кладет операнд в стек | Код сегмента  Код команды  Операнд | 000001 |
| OP\_POP | Достает значение из  стека в операнд | Код сегмента  Код команды  Операнд | 000010 |
| OP\_ADD | Суммирует два операнда с вершины стека, результат кладется в стек | Код сегмента  Код команды | 000011 |
| OP\_SUB | Вычитает два операнда с вершины стека, результат кладется в стек | Код сегмента  Код команды | 000100 |
| OP\_MUL | Умножает два операнда с вершины стека, результат кладется в стек | Код сегмента  Код команды | 000101 |
| OP\_DIV | Делит два операнда с вершины стека нацело, результат кладется в стек | Код сегмента  Код команды | 000110 |

Продолжение таблицы 7.4.

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Описание | Кодирование | Код команды |
| OP\_MOD | Остаток от деления двух операндов с вершины стека, результат кладется в стек | Код сегмента  Код команды | 000111 |
| OP\_CMP\_MORE | Сравнивает два операнда с вершины стека, результат кладется в стек. | Код сегмента  Код команды | 001000 |
| OP\_CMP\_LESS | Сравнивает два операнда с вершины стека, результат кладется в стек. | Код сегмента  Код команды | 001001 |
| OP\_WRITE | Вывод на экран целевой платформы значение операнда справа | Код сегмента  Код команды  Операнд | 010000 |
| OP\_IF | Создаётся метка условного оператора | Код сегмента  Код команды | 001010 |
| OP\_CREATE\_MARK\_FALSE | Создается метка ложного условия | Код сегмента  Код команды | 001011 |
| OP\_CREATE\_MARK\_AFTER\_FALSE | Создается метка после описания ложного условия | Код сегмента  Код команды | 001111 |
| OP\_CALL | Операция вызова функции | Код сегмента  Код команды  Операнд | 001100 |
| OP\_RET | Операция возврата из функции | Код сегмента  Код команды | 001101 |
| OP\_PUSH\_OFFSET | Положить в стек адрес операнда справа | Код сегмента  Код команды  Операнд | 001110 |
| OP\_PUSH\_LENGTH\_STRING | Положить в стек длину строки справа | Код сегмента  Код команды  Операнд | 010010 |

Продолжение таблицы 7.4.

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Описание | Кодирование | Код команды |
| OP\_CMP\_STR | Сравнить две строки с вершины стека. Результат кладется в стек | Код сегмента  Код команды | 010011 |
| OP\_CALL\_PROC | Вызвать процедуру | Код сегмента  Код команды  Операнд | 010100 |

В таблице под операндом понимается выражение, состоящее из 36 бит (4 бит сегмент операнда + 32 бит название операнда). Все математические операции достают из стека свои операнды, а кладут только результат.

* 1. Примеры транслированных программ в байт-код

В презентации, которая прилагается, вы можете ознакомится с некоторыми примерами программ и подробным разбором сгенерированного по ним байт-кода, также байт-код стартового примера будет представлен в приложении Е

# Транслятор байт-кода в язык ассемблера

* 1. Структура транслятора байт-кода

Структура транслятора байт-кода представлена на рисунке 8.1.



Рисунок 8.1. Структура генератора байт-кода

Транслятор байт-кода в язык ассемблера, является дополнительным приложением, позволяющим переводить команды байт-код машины языка SMV-2019 в ассемблерные команды, которые поддерживаются нашей целевой платформой. На вход подается файл с расширением .bin, в котором записаны команды виртуальной машины SMV-2019. Выходным файлом является файл с расширением .asm, в котором находится интерпретированный ассемблерный код.

* 1. Принцип работы транслятора байт-кода в язык ассемблера

На вход подается байт-код, считывается сегмент, далее следует интерпретация и шаги в зависимости от типа сегмента. Для сегмента констант далее следует считывание имени константы и его интерпретация, тип и сами данные в зависимости от типа. Для сегмента данных следует считывание имени переменной и типа и их последующая интерпретация. Для сегмента функций следует считывание имени функции, тип возвращаемого параметра, количество параметров, а так же сами параметры, которые далее интерпретируются в ассемблерный код. Для сегмента команд интерпретация происходит согласно таблице 8.1.

Таблица 8.1. Аналоги команд

|  |  |
| --- | --- |
| Название команды байт кода SMV-2019 | Ассемблерный эквивалент |
| OP\_PUSH | push <имя операнда> |
| OP\_POP | pop <имя операнда> |

Продолжение таблицы 8.1

|  |  |
| --- | --- |
| Название команды байт кода SMV-2019 | Ассемблерный эквивалент |
| OP\_ADD | pop ebx  pop eax  add eax, ebx  push eax |
| OP\_SUB | pop ebx  pop eax  sub eax, ebx  push eax |
| OP\_MUL | pop ebx  pop eax  imul ebx  push eax |
| OP\_DIV | pop ecx  pop eax  CDQ  cmp ecx, 0  jz ERRORZERODIVIZION  idiv ecx  push eax |
| OP\_MOD | pop ecx  pop eax  CDQ  cmp ecx, 0  jz ERRORZERODIVISION  idiv ecx  push edx |
| OP\_CMP\_MORE | call OPERATOR\_MORE |
| OP\_CMP\_LESS | call OPERATOR\_LESS |
| OP\_WRITE | INVOKE priintConsole, ADDR  <имя операнда>, sizeof <имя операнда> |
| OP\_CALL | call <имя операнда> |
| OP\_RET | ret |
| OP\_PUSH\_OFFSET | push offset <имя операнда> |
| OP\_PUSH\_LENGTH\_STRING | mov ah <имя операдна>  push ah |
| OP\_CMP\_STR | call B0 |
| OP\_CALL\_PROC | call <имя операнды> |

* 1. Представление типов данных в оперативной памяти

Соответствия между типами данных идентификаторов в языке SMV-2019 и на языке ассемблера приведены в таблице 8.1.

Таблица 8.1 Соответствия типов идентификаторов языка SMV-2019 и типов языка ассемблера

|  |  |  |
| --- | --- | --- |
| Тип идентификатора на языке SMV-2019 | Тип идентификатора на языке ассемблера | Пояснение |
| integer | SDWORD | Хранит целочисленный тип данных. |
| boolean | SDWORD | Хранит логический тип данных |
| str | BYTE | Хранит строку. Строка представляется в следующем виде:  1 байт длина строки (n) далее n кодов символов в кодировке CP866 |

* 1. Стандартная библиотека языка

В языке предусмотрена стандартная библиотека, написанная на языке ассемблера, включающая в себя следующие функции: вывод в консольное окно, операторы больше и меньше, функция побайтового сравнения строк. Их исходный код представлен в приложении Ж.

* 1. Контрольный пример

Результат генерации ассемблерного кода на основе контрольного примера из приложения А приведен в приложении З.

# Тестирование транслятора

В данной главе описаны возможные ошибки и предупреждение, возникающие на различных этапах работы транслятора, результат тестирования представлен в таблице 9.1.

Таблица 9.1 Тестирование транслятора

|  |  |
| --- | --- |
| Исходный код | Генерируемое сообщение |
| main  begin  value: integer c = 313:3;  end | Ошибка 9: Нераспознанная лексема, строка 3 позиция 26. |
| main main  begin  end | Ошибка 6: Найдено более одной функции main |
| value: integer poz = 25;  function: integer ErrorIf[integer iValue]  begin  if (iValue<26) begin  return poz=poz+1;  end      end  main  begin  end | 608: строка 4, Ошибка в операторе if… then… else  605: строка 4, Ошибка мат. Оператора  602: строка 4, ошибка в выражении |
| value: str string = "1234567890";  function: integer ErrorFunction[integer parm]  begin  return cmp[parm];  end  main  begin  ErrorFunction[string];  end | 605: строка 4, ошибка мат оператора  602: строка 4, ошибка в выражении  606: строка 4, ошибка в параметрах функции cm[str parm1, str parm2] |

Продолжение таблицы 9.1.

|  |  |
| --- | --- |
| Исходный код | Генерируемое сообщение |
| main  begin  value: integer val = ;  end | 602: строка 3, Ошибка в выражении  609: строка 3, Ошибка в объявлении идентификатора  611: строка 3, Ошибка в теле main |
| procedure: er  begin  return 0;  end  main  begin  end | 610: Cтрока 3, ошибка в теле процедуры  600: Строка 1, неверная структура программы |
| main  begin  return 2;  end | 611: строка 3, ошибка в теле main  600: строка 1, неверная структура программы |
| main  begin  value: integer a = clength[21];  end | 605: строка 3, ошибка мат. Оператора  602: строка 3, ошибка в выражении  607: строка 3, ошибка в параметрах функции clength[str parm] |
| function: integer a [integer e]  begin  return 0;  end  main  begin  a[2,3];  end | Ошибка 701: Несоответствие параметров при вызове и при объявлении функции a, строка 7 |
| procedure: proc  begin  value: integer a = 2;  end  main  begin  if (proc) then begin end  end | Ошибка 711: Недопустимый вызов процедуры: proc, строка 7 |

Продолжение таблицы 9.1.

|  |  |
| --- | --- |
| Исходный код | Генерируемая ошибка |
| function: integer a [integer e]  begin  return "233";  end  main  begin  a[2];  end | Ошибка 712: Несоответствие типа возвращаемого значения функции и выражения return, строка 3 |
| main  begin  value: integer a = 7>5;  end | Ошибка 706: Несоответствие типов( операторы >, < возвращают boolean значение) : a, строка 3 |
| main  begin  value: integer a = "2313";  end | Ошибка 705: Несоответствие типов левой и правой части выражения a, строка 3 |
| main  begin  value: str a = "31313" + "31313";  end | Ошибка 710: Недопустимо использование мат. Операций над строками: a, строка 3 |
| function: integer fi[integer a]  begin  end  procedure: fi  begin  end  main  begin  end | Ошибка 702: Повторное объявление функции или процедуры |
| main  begin  a = 10;  end | Ошибка 700: Использование необъявленного идентификатора a, строка 3 |
| function: integer ERROR[integer parm1]  begin  value: str a ;  if (a) then begin end  end  main  begin  end | Ошибка 713: Ошибка в условии оператора if, строка 4 |

Продолжение таблицы 9.1.

|  |  |
| --- | --- |
| Исходный код | Генерируемое сообщение |
| value:integer a;  main  begin  a;  end | Ошибка 708: Не найдено объявление процедуры: a, строка 4 |
| main  begin  value: integer a =4 ;  value: str a = "22";  end | Ошибка 704: Повторное объявление переменной: a, строка 4 |
| main  begin  value: integer a = a[2];  end | Ошибка 707: Не найдено объявление функции a, строка 3. |
| main  begin  print["313131",313];  end | Ошибка 714: Ошибка в операторе print, строка 3 |
| function: integer a [integer p1]  begin  value: integer a;  return a;  end  main  begin  end | Строка: 4. Предупреждение 70: Неинициализированный идентификатор: a (Применена инициализация по умолчанию)  Трансляция успешно завершена |
| main  begin  $  end | Ошибка 113: Недопустимый символ в исходном коде, строка 3 позиция 1 |
| function: integer f01234567890123456789 [integer p1]  begin  return 0 ;  end  main  begin  end | Строка: 1. Предупреждение 72: Имя идентификатора превышает 20 символов, применено усечение до:  f0123456789012345678  Трансляция успешно завершена |

Продолжение таблицы 9.1.

|  |  |
| --- | --- |
| Исходный код | Генерируемое сообщение |
| function: str string [str parm]  begin  end  main  begin  end | Строка: 1.Предупреждение 71: Не найдено ни одного оператора return у функции string, по умолчанию функции возвращают: integer – 0, string – “”, boolean – false  Трансляция успешно завершена |

# 

# Приложение А

value: integer globalInteger = 5;

function: integer factorial [integer parm]

begin

if (parm>1) then begin

parm = parm-1;

return (parm+1)\*factorial[parm];

end

else begin

return 1;

end

end

function: boolean printCMP [str string1, str string2]

begin

if (cmp[string1, string2]) then begin

print[string1]; print[" "]; print[string2]; print[" - равны\n"];

end

else begin

print[string1]; print[" "]; print[string2]; print[" - не равны\n"];

end

return false;

end

procedure: globalIntegerPlus5

begin

globalInteger=globalInteger+5;

end

main

begin

value: integer positive = 25;

value: integer negative = 0-positive\*2;

positive = positive + negative;

print["Результат выражения 25 + (-25\*2): "]; print[positive];

value: integer binary = 101b;

value: integer oct = 701o;

value: integer hex = ffh;

print["\nЗначение переменной binary, которое равно 101: "]; print[binary];

print["\nЗначение переменной oct, которое равно 701: "]; print[oct];

print["\nЗнчение переменной hex, которое равно ff: "]; print[hex];

value: integer expression = ((5\*25 + 10) div 3 \* 2) mod 100;

print["\nЗначение выражения ((5\*25 + 10) div 3 \* 2) mod 100: "]; print[expression];

value: str string = "123456789";

value: integer length = clength[string];

print["\nДлина строки: "]; print[string]; print[" = "]; print[length];

value: str string1 = "str1";

value: str string2 = "str2";

print["\nСравним побайтово несколько строк:\n"];

printCMP[string1, string2];

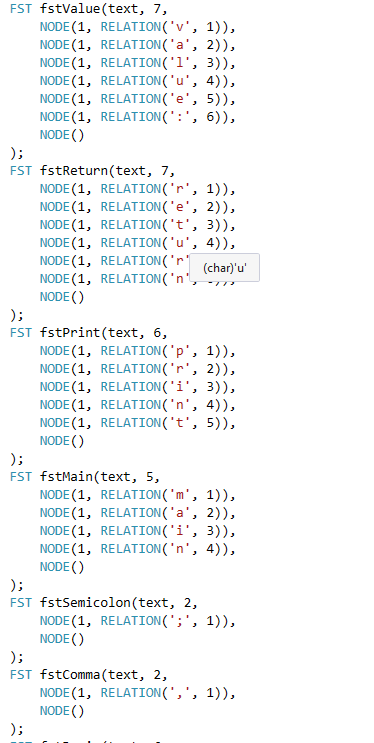
string2 = "str1";

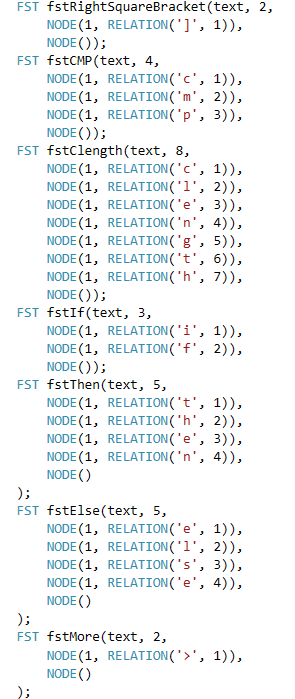
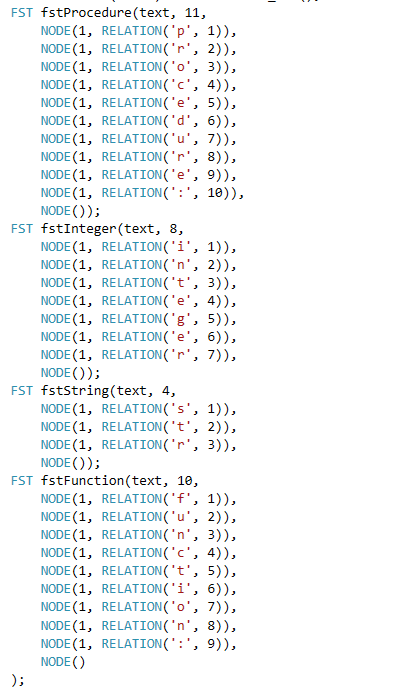
printCMP[string1, string2];

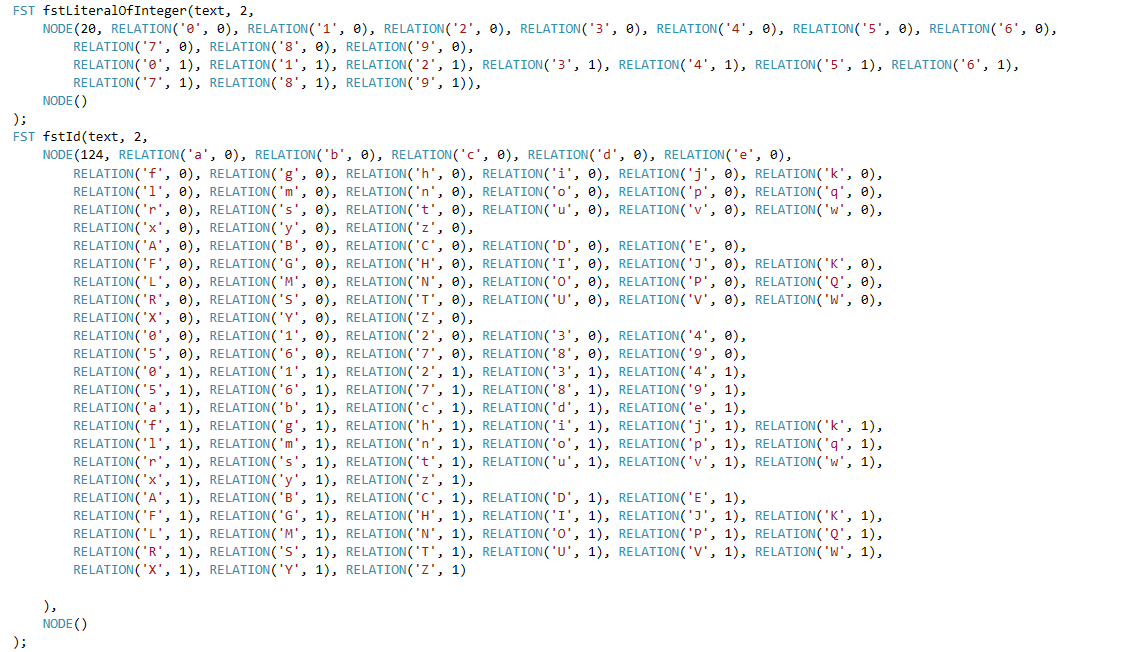
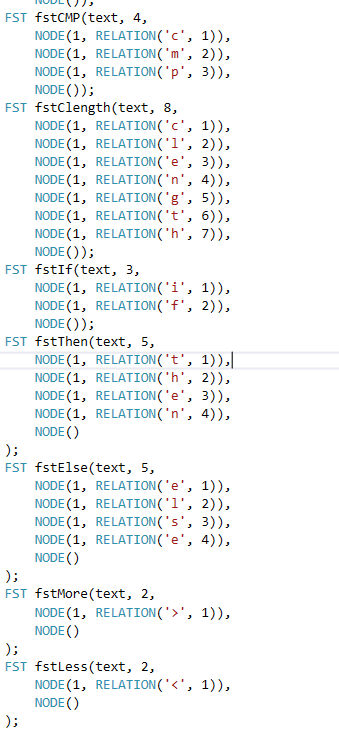
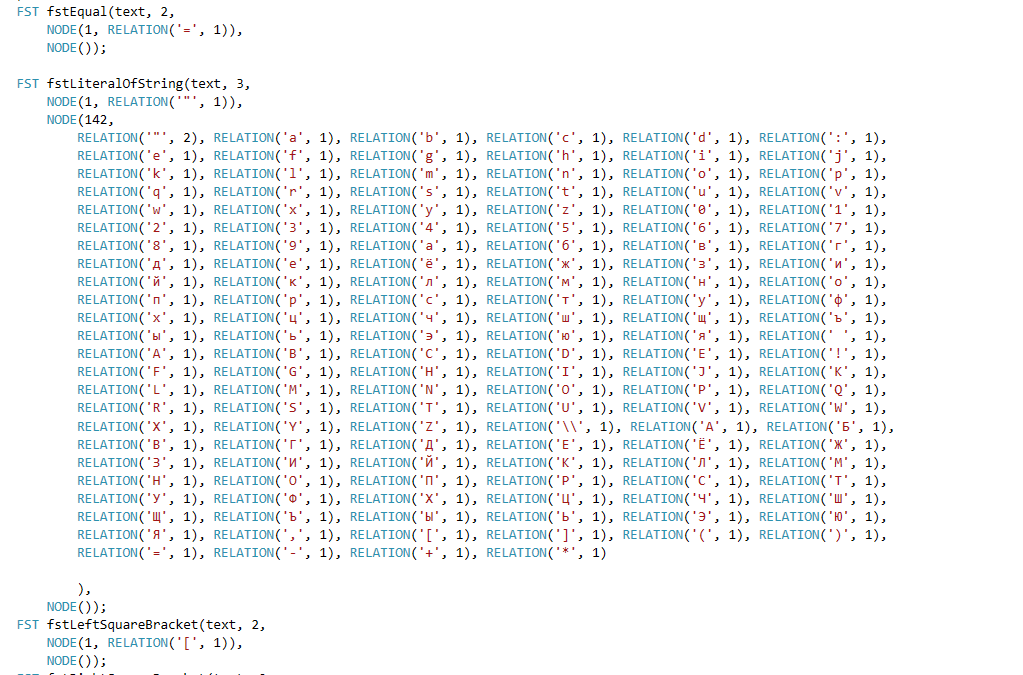
print["7 факториал : "]; value: integer result = factorial[7]; print[result];

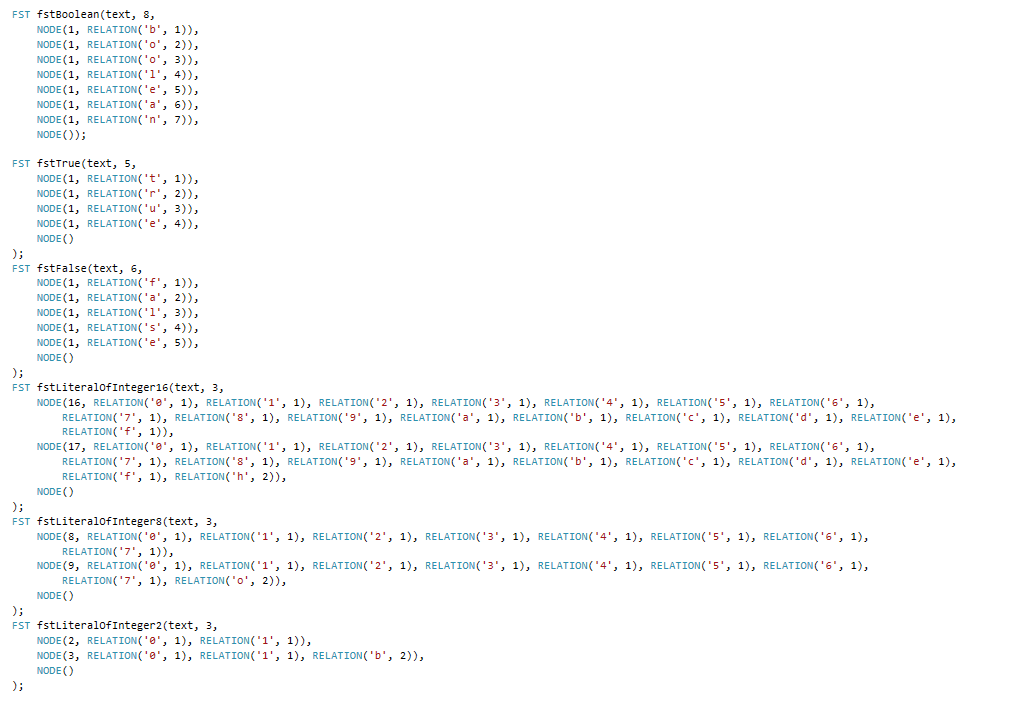
end

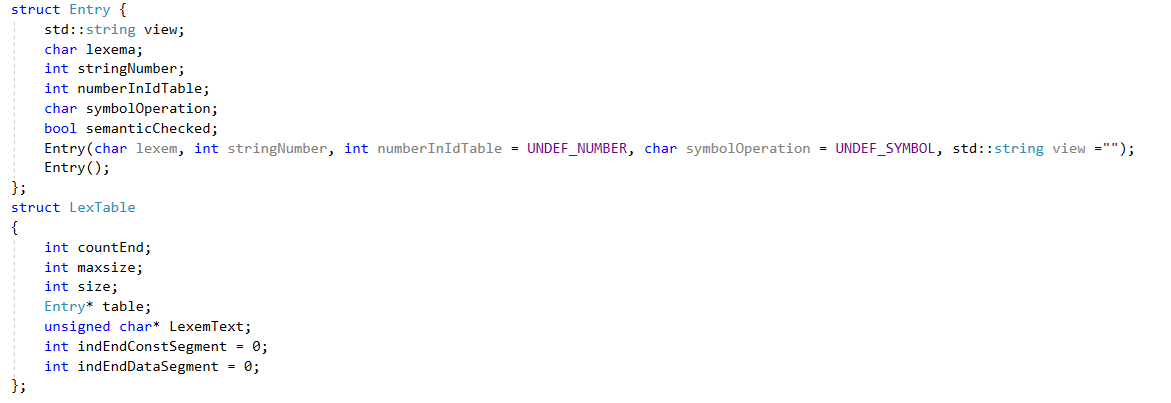
# Приложение Б

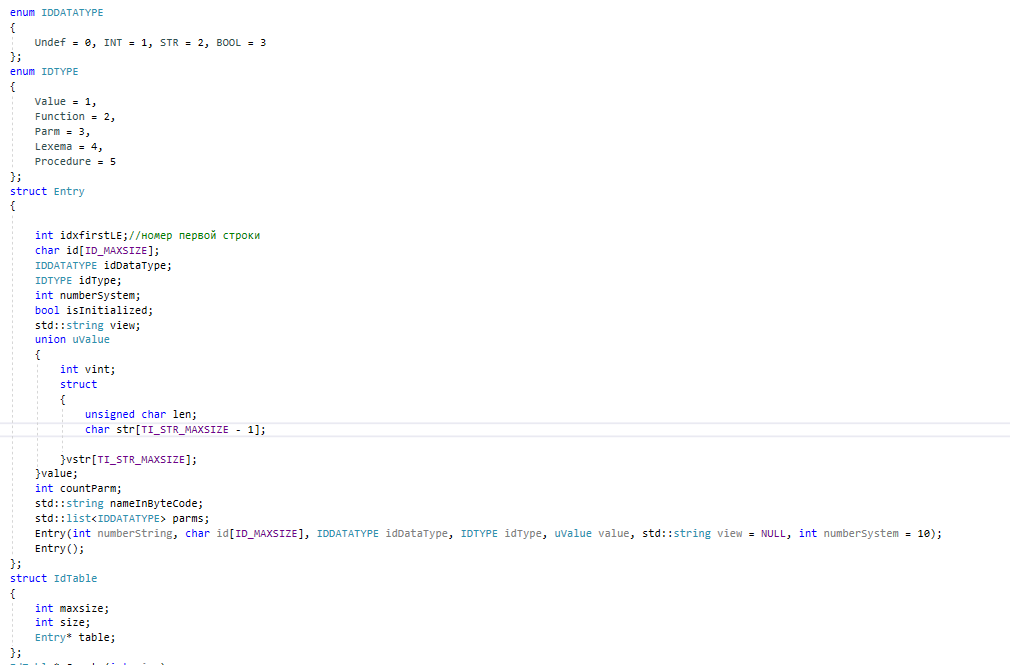




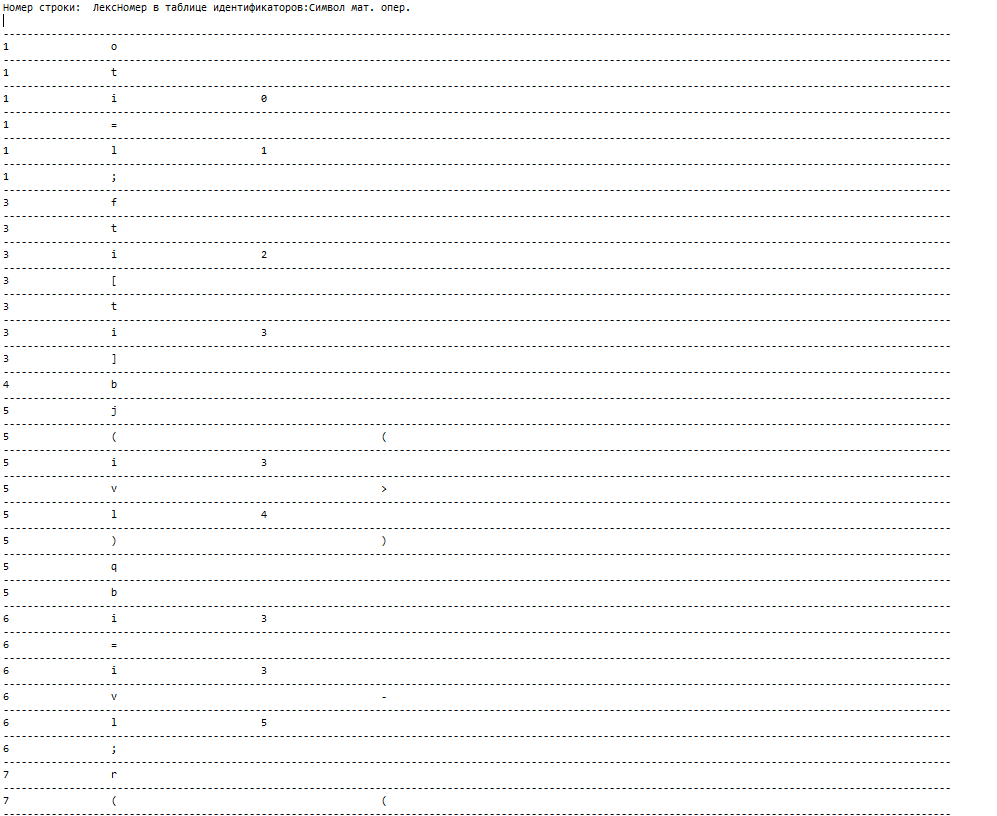
 

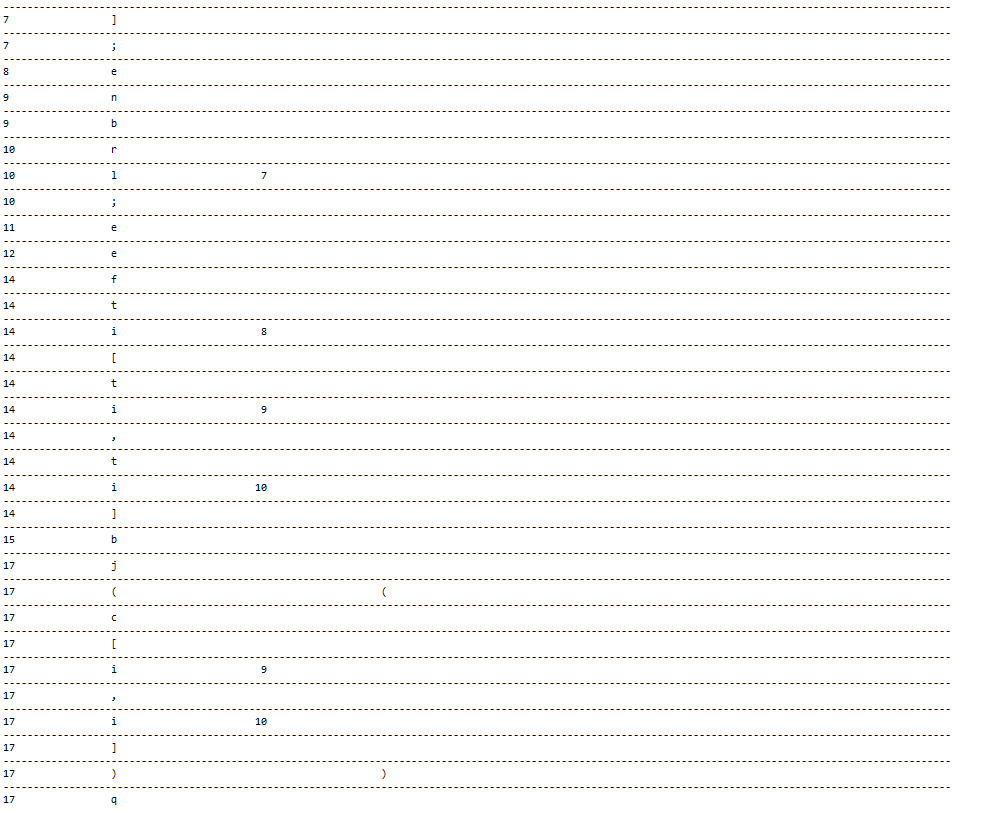




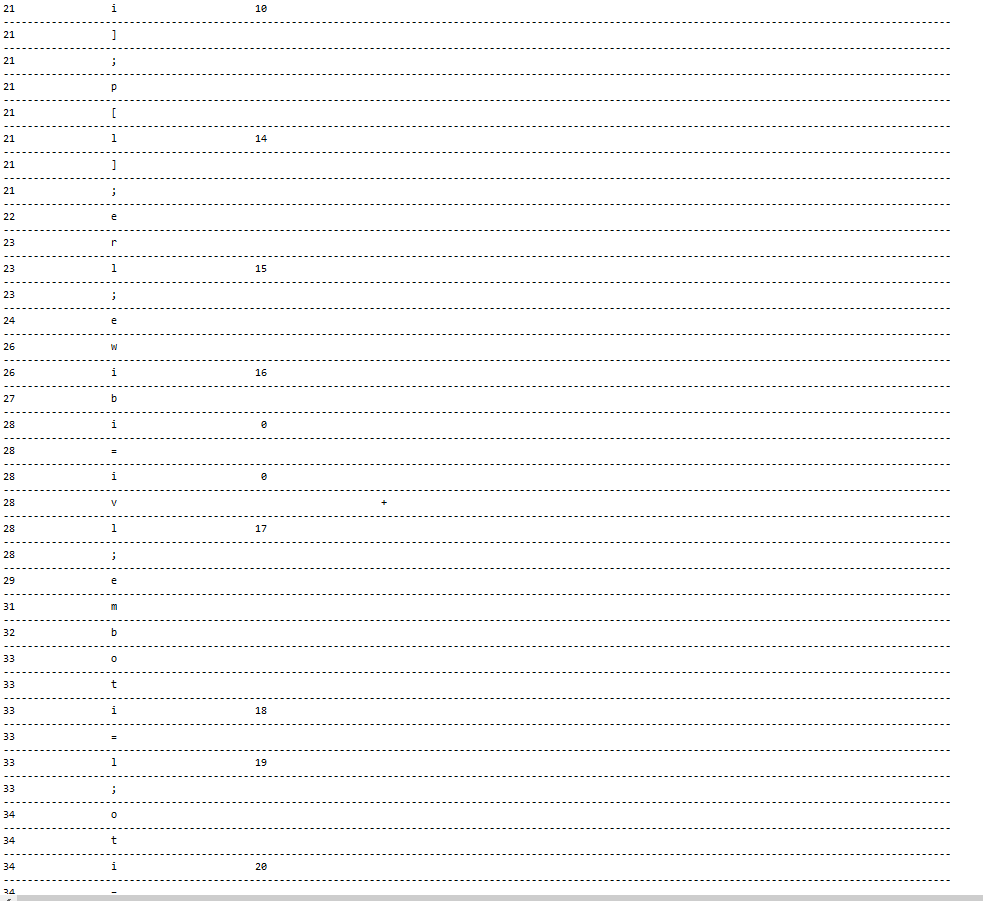
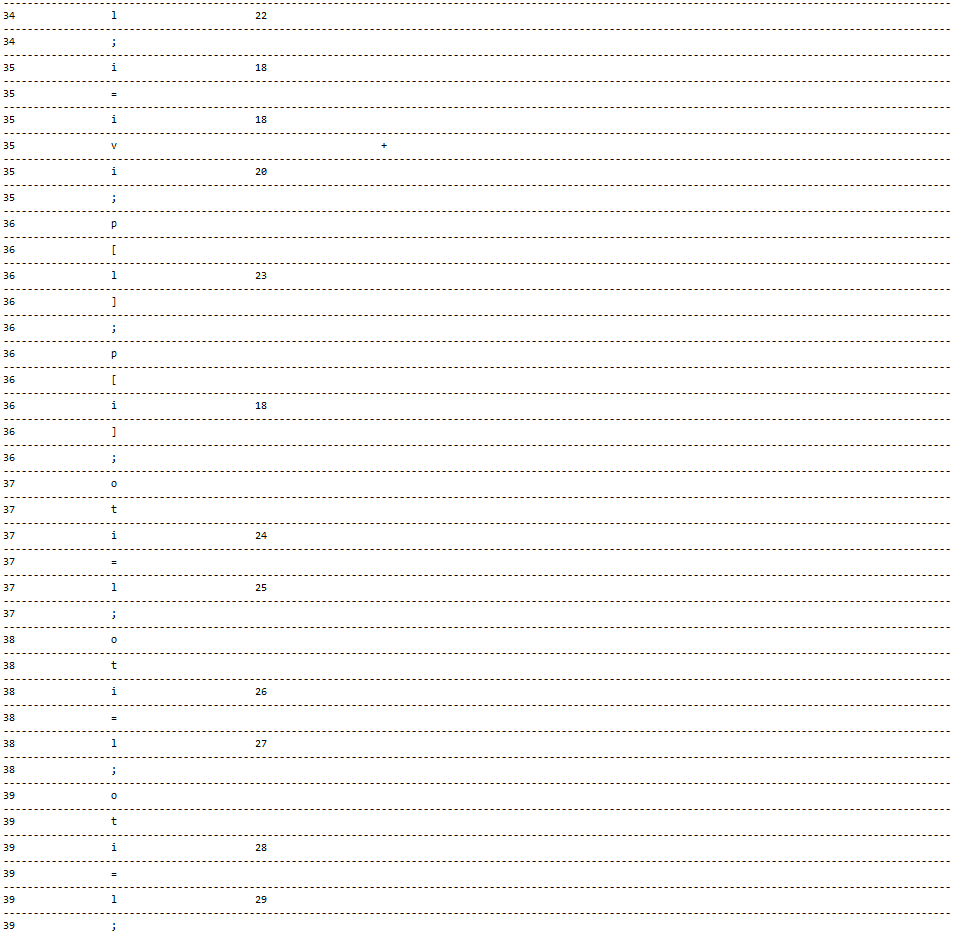


# Приложение В



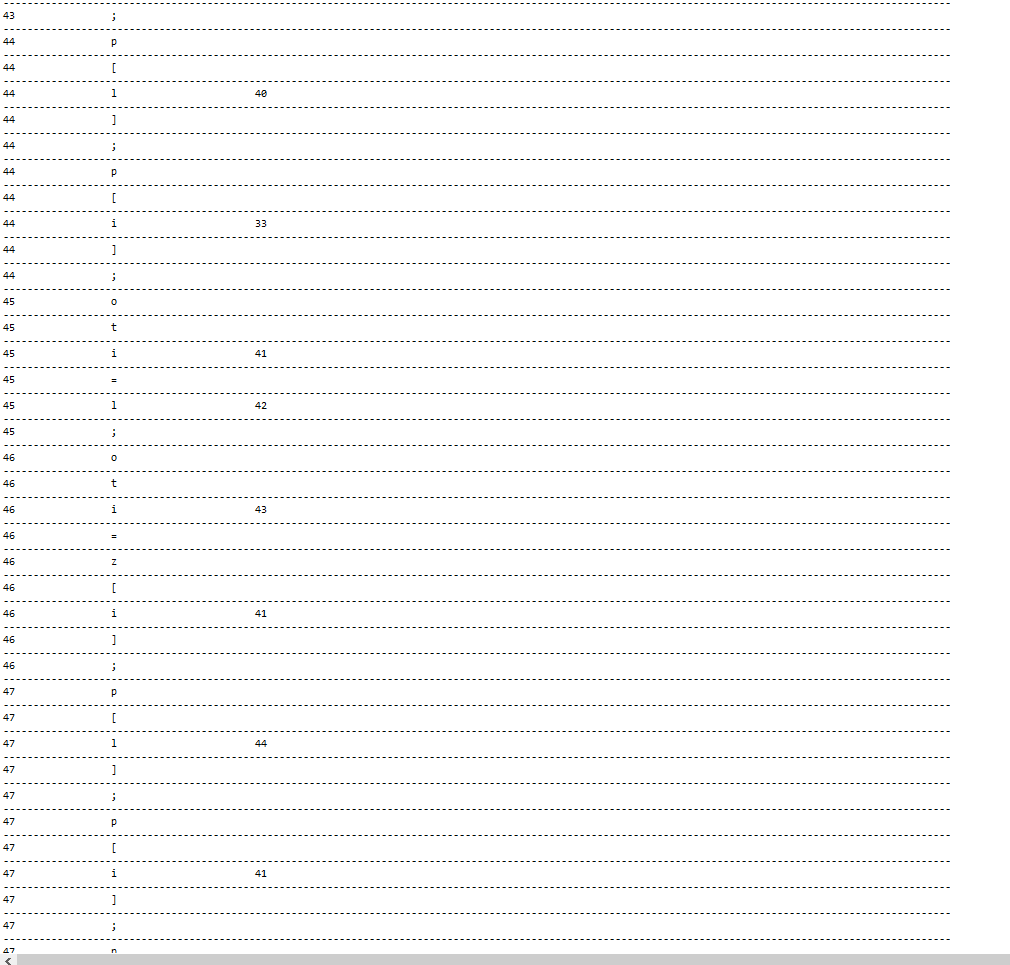




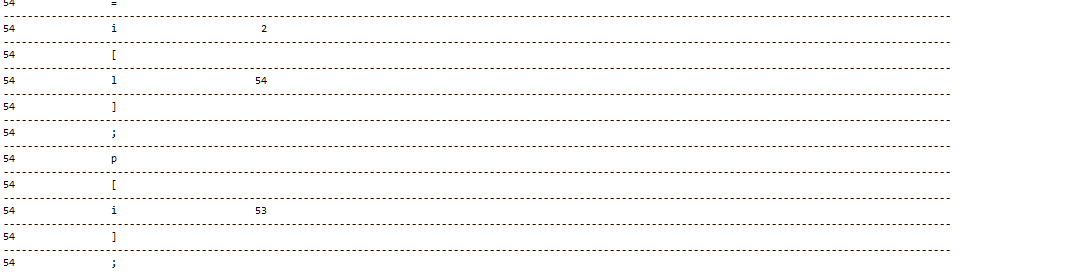
 

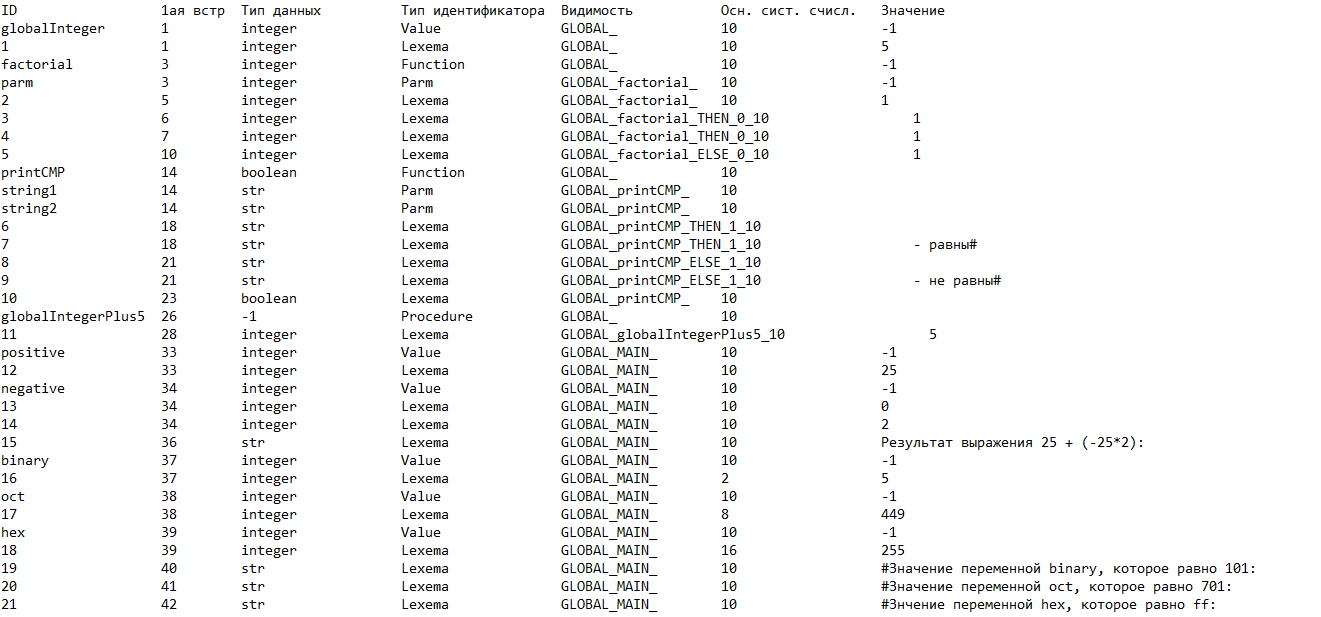


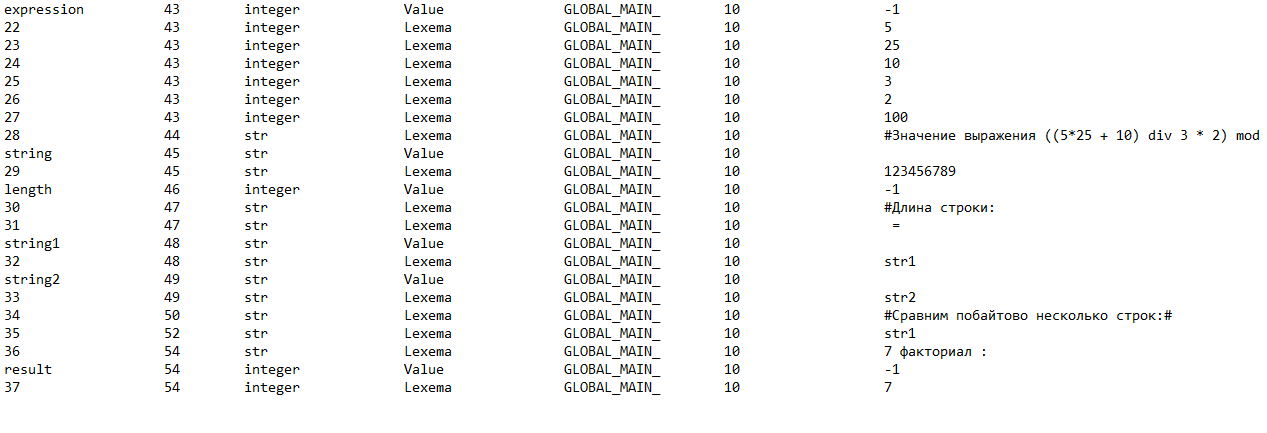




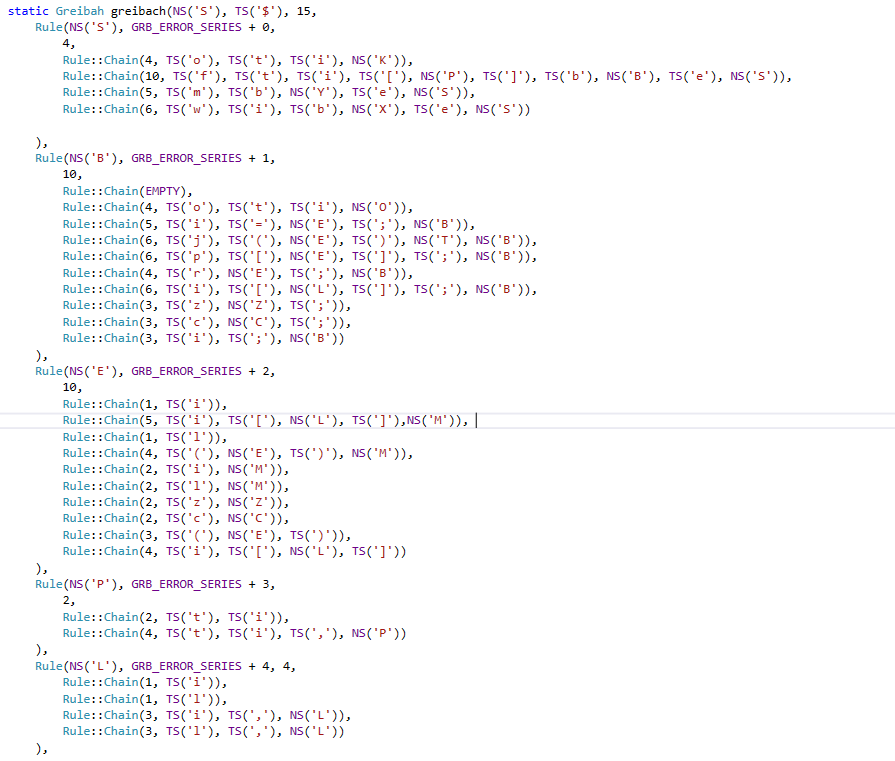


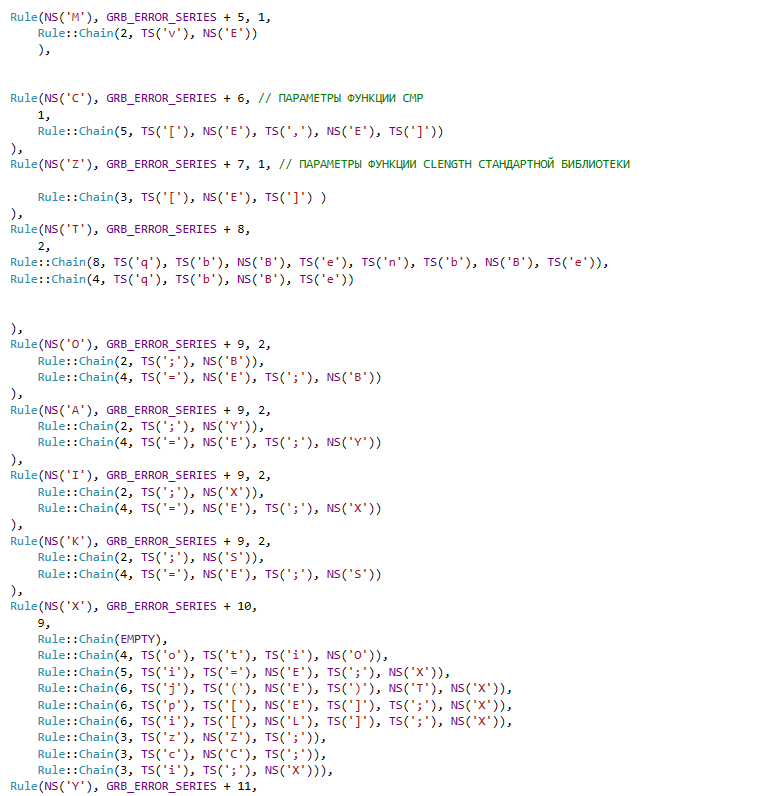


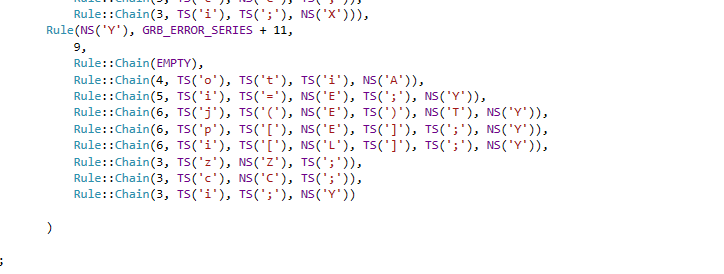




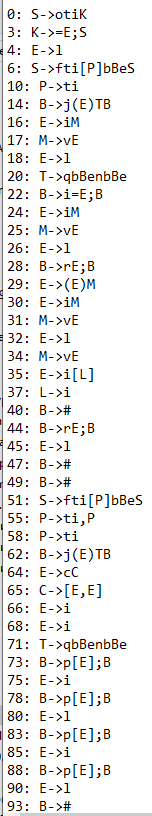
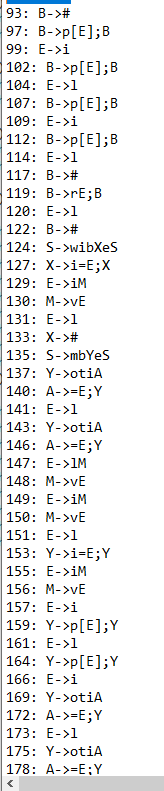
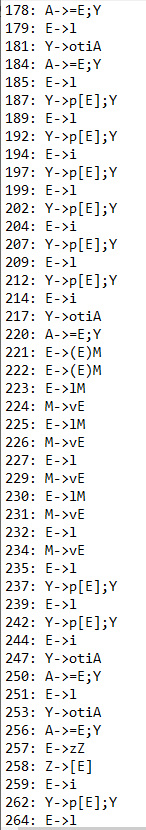
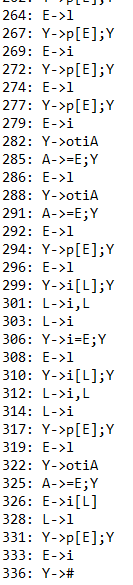
Приложение Г







# Приложение Д

# Приложение Е



# Приложение Ж

OPERATOR\_MORE PROC, PARM1: SDWORD, PARM2: SDWORD

mov eax, PARM1

mov ebx, PARM2

CMP eax, ebx

jg isGreater

mov eax, 0

jmp markAfterGreater

isGreater: mov eax, 1

markAfterGreater:

ret

OPERATOR\_MORE ENDP

OPERATOR\_LESS PROC, PARM3: SDWORD, PARM4: SDWORD

mov eax, PARM3

mov ebx, PARM4

CMP eax, ebx

jg isGreater2

mov eax, 1

jmp markAfterGreater2

isGreater2: mov eax, 0

markAfterGreater2:

ret

OPERATOR\_LESS ENDP

printConsole PROC pstr : dword, \_size : dword

push eax

INVOKE GetStdHandle, -11

INVOKE WriteConsoleA, eax, pstr, \_size, 0, 0

pop eax

ret

printConsole ENDP

intToString PROC number: sdword, pstr: dword

push eax

push ebx

push ecx

push edi

push esi

mov esi, offset strResult

mov ecx, sizeof strResult

mov al, ' '

CLEARSTR:

mov [esi], al

inc esi

LOOP CLEARSTR

mov edi, pstr

mov esi, 0

mov eax, number

cdq

mov ebx, 10

test eax, 80000000h

idiv ebx

jz plus

neg eax

neg edx

mov cl, '-'

mov[edi], cl

inc edi

plus :

push dx

inc esi

test eax, eax

jz fin

cdq

idiv ebx

jmp plus

fin :

mov ecx, esi

write :

pop bx

add bl, '0'

mov[edi], bl

inc edi

loop write

mov ecx, sizeof strResult

mov esi, offset strResult

COUNTER :

mov al, [esi]

cmp al, 32

jz IC

inc numberSize

IC :

inc esi

LOOP COUNTER

pop esi

pop edi

pop ecx

pop ebx

pop eax

ret

intToString ENDP

B1 PROC str1 : DWORD

mov edi, str1

xor eax, eax

mov al, [edi]

ret

B1 ENDP

B0 PROC str1 : DWORD, str2 : DWORD

mov edi, str1

mov esi, str2

xor eax, eax

xor ebx, ebx

mov al, [edi]

mov bl, [esi]

cmp al, bl

jne false

mov ecx, eax

inc ecx

CYCLE :

mov al, [edi]

mov bl, [esi]

cmp al, bl

jne false

inc edi

inc esi

LOOP CYCLE

jmp true

false:

xor eax, eax

ret

true :

mov eax, 1

ret

B0 ENDP

# Приложение З

F0 PROC P0 : SDWORD

PUSH P0

PUSH C2

CALL OPERATOR\_LESS

PUSH eax

pop eax

cmp eax, 0

jz ZI0

PUSH P0

PUSH C3

POP ebx

POP eax

SUB eax, ebx

PUSH eax

POP P0

PUSH P0

PUSH C4

POP ebx

POP eax

ADD eax, ebx

PUSH eax

PUSH P0

CALL F0

PUSH eax

POP ebx

POP eax

IMUL ebx

PUSH eax

POP eax

ret

jmp AE0

ZI0:

PUSH C5

POP eax

ret

AE0:

jmp MarkAfterError

ERRORZERODIVISION:

INVOKE printConsole, ADDR ERRMESS, sizeof ERRMESS

push 0

call ExitProcess

MarkAfterError:

xor eax, eax

ret

F0 ENDP

F1 PROC P1 : DWORD, P2 : DWORD

PUSH P2

PUSH P1

CALL B0

PUSH eax

pop eax

cmp eax, 0

jz ZI1

mov edi,P1

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

mov edi, offset C6

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

mov edi,P2

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

mov edi, offset C7

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

jmp AE1

ZI1:

mov edi,P1

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

mov edi, offset C8

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

mov edi,P2

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

mov edi, offset C9

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

AE1:

PUSH C10

POP eax

ret

jmp MarkAfterError

ERRORZERODIVISION:

INVOKE printConsole, ADDR ERRMESS, sizeof ERRMESS

push 0

call ExitProcess

MarkAfterError:

xor eax, eax

ret

F1 ENDP

FP0 PROC

PUSH D0

PUSH C11

POP ebx

POP eax

ADD eax, ebx

PUSH eax

POP D0

ret

FP0 ENDP

main PROC

INVOKE SetConsoleTitleA, ADDR consoletitle

PUSH C1

POP D0

PUSH C12

POP D1

PUSH C13

PUSH D1

PUSH C14

POP ebx

POP eax

IMUL ebx

PUSH eax

POP ebx

POP eax

SUB eax, ebx

PUSH eax

POP D2

PUSH D1

PUSH D2

POP ebx

POP eax

ADD eax, ebx

PUSH eax

POP D1

mov edi, offset C15

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

INVOKE intToString,D1, ADDR strResult

INVOKE printConsole, ADDR strResult, numberSize

mov numberSize, 0

PUSH C16

POP D3

PUSH C17

POP D4

PUSH C18

POP D5

mov edi, offset C19

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

INVOKE intToString,D3, ADDR strResult

INVOKE printConsole, ADDR strResult, numberSize

mov numberSize, 0

mov edi, offset C20

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

INVOKE intToString,D4, ADDR strResult

INVOKE printConsole, ADDR strResult, numberSize

mov numberSize, 0

mov edi, offset C21

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

INVOKE intToString,D5, ADDR strResult

INVOKE printConsole, ADDR strResult, numberSize

mov numberSize, 0

PUSH C22

PUSH C23

POP ebx

POP eax

IMUL ebx

PUSH eax

PUSH C24

POP ebx

POP eax

ADD eax, ebx

PUSH eax

PUSH C25

POP ecx

POP eax

CDQ

cmp ecx, 0

jz ERRORZERODIVISION

IDIV ecx

PUSH eax

PUSH C26

POP ebx

POP eax

IMUL ebx

PUSH eax

PUSH C27

POP ecx

POP eax

CDQ

cmp ecx, 0

jz ERRORZERODIVISION

IDIV ecx

PUSH edx

POP D6

mov edi, offset C28

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

INVOKE intToString,D6, ADDR strResult

INVOKE printConsole, ADDR strResult, numberSize

mov numberSize, 0

push offset C29

POP D7

PUSH D7

CALL B1

PUSH eax

POP D8

mov edi, offset C30

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

mov edi,D7

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

mov edi, offset C31

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

INVOKE intToString,D8, ADDR strResult

INVOKE printConsole, ADDR strResult, numberSize

mov numberSize, 0

push offset C32

POP D9

push offset C33

POP D10

mov edi, offset C34

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

PUSH D10

PUSH D9

CALL F1

push offset C35

POP D10

PUSH D10

PUSH D9

CALL F1

mov edi, offset C36

mov al, [edi]

inc edi

INVOKE printConsole, edi, al

PUSH C37

CALL F0

PUSH eax

POP D11

INVOKE intToString,D11, ADDR strResult

INVOKE printConsole, ADDR strResult, numberSize

mov numberSize, 0

jmp MarkAfterError

ERRORZERODIVISION:

INVOKE printConsole, ADDR ERRMESS, sizeof ERRMESS

MarkAfterError:

push 0

call ExitProcess

main ENDP

end main

# 

# Заключение

Таким образом, по окончании выполнения всех пунктов, изложенных ранее, получили рабочий транслятор языка программирования SMV-2019 в байт-код, а затем в язык ассемблера.

В результате получился исходный код транслятора, который состоит примерно из 4500+ строк кода на языке программирования С++.

Для разбора токенов было составлено 30+ конечных автоматов, а грамматика Грейбах для синтаксического разбора состоит из 80+ правил.

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

Процесс разработки интерпретатора был пройден в соответствии с полным классическим жизненным циклом разработки программного обеспечения, что позволило усовершенствовать навыки проектирования программной системы и её непосредственной реализации.

# 

# Литература

1. Язык ассемблера для процессоров Intel, 4-е издание.: Пер с англ. — М.: Издательский дом “Вильямс”, 2005 — 912c. : ил. — Парал. тит. англ.
2. Системной программное обеспечение: Учебник для вузов. 3-е изд. — СПБ.: Питер, 2010. — 400 с.:ил.
3. Построение компиляторов / Пер. с англ. Борисов Е. В., Чернышов Л.Н. — М.: ДМК Пресс, 2010. — 192с.: ил.
4. Интерпретаторы байт-кодов своими руками [электронный ресурс]. — Режим доступа: <https://habr.com/ru/company/badoo/blog/425325/>. — Дата доступа 18.11.2019.
5. Обратная польская запись [электронный ресурс]. — Режим доступа: <https://habr.com/ru/post/100869/>. — Дата доступа 02.11.2019.
6. Алгоритмы и методы: Обратная польская запись [электронный ресурс]. — Режим доступа: <http://www.interface.ru/home.asp?artid=1492>/. — Дата доступа 19.10.2019.
7. Лекции по дисциплине «Языки программирования» для специальности «Программное обеспечение информационных технологий».

# Содержание

[Введение 2](#_Toc27315231)

[Глава 1. Спецификация языка программирования 3](#_Toc27315232)

[1.1 Характеристика языка программирования 3](#_Toc27315233)

[1.2 Алфавит языка 3](#_Toc27315234)

[1.3 Символы-сепараторы 4](#_Toc27315235)

[1.4 Применяемые кодировки 4](#_Toc27315236)

[1.5 Типы данных 5](#_Toc27315237)

[1.6 Преобразование типов данных 5](#_Toc27315238)

[1.7 Идентификаторы 5](#_Toc27315239)

[1.8 Литералы 6](#_Toc27315240)

[1.9 Область видимости идентификаторов 6](#_Toc27315241)

[1.10 Инициализация данных 7](#_Toc27315242)

[1.11 Инструкции языка 7](#_Toc27315243)

[1.12 Операции языка 8](#_Toc27315244)

[1.13 Выражения и их вычисления 8](#_Toc27315245)

[1.14 Программные конструкции языка 8](#_Toc27315246)

[1.15 Область видимости 9](#_Toc27315247)

[1.16 Семантические проверки 9](#_Toc27315248)

[1.17 Распределение оперативной памяти на этапе выполнения 10](#_Toc27315249)

[1.18 Стандартная библиотека и её состав 10](#_Toc27315250)

[1.19 Ввод и вывод данных 10](#_Toc27315251)

[1.20 Точка входа 11](#_Toc27315252)

[1.21 Препроцессор 11](#_Toc27315253)

[1.22 Соглашения о вызовах 11](#_Toc27315254)

[1.23 Объектный код 11](#_Toc27315255)

[1.24 Классификация сообщений транслятора 11](#_Toc27315256)

[1.25 Контрольный пример 11](#_Toc27315257)

[Глава 2. Структура транслятора 12](#_Toc27315258)

[2.1 Компоненты транслятора, их назначение и принципы взаимодействия 12](#_Toc27315259)

[2.2 Перечень входных параметров транслятора 13](#_Toc27315260)

[2.3 Перечень протоколов, формируемых транслятором и их содержимое 13](#_Toc27315261)

[2.4 Перечень входных параметров транслятора байт-кода 13](#_Toc27315262)

[2.5 Перечень файлов, получаемых в результате работы транслятора байт-кода 14](#_Toc27315263)

[Глава 3. Разработка лексического анализатора 15](#_Toc27315264)

[Глава 4. Разработка синтаксического анализатора 20](#_Toc27315265)

[Глава 5. Разработка семантического анализатора 24](#_Toc27315266)

[Глава 6. Преобразование выражений 27](#_Toc27315267)

[Глава 7. Генерация байт-кода 29](#_Toc27315268)

[Глава 8. Транслятор байт-кода в язык ассемблера 34](#_Toc27315269)

[Глава 9. Тестирование транслятора 37](#_Toc27315270)

[Приложение А 42](#_Toc27315271)

[Приложение Б 44](#_Toc27315272)

[Приложение В 50](#_Toc27315273)

[Приложение Г 60](#_Toc27315274)

[Приложение Д 62](#_Toc27315275)

[Приложение Е 63](#_Toc27315276)

[Приложение Ж 64](#_Toc27315277)

[Приложение З 66](#_Toc27315278)

[Заключение 70](#_Toc27315279)

[Литература 71](#_Toc27315280)

[Содержание 72](#_Toc27315281)