Содержание

[Введение 4](#_Toc532915217)

[1. Спецификация языка программирования 6](#_Toc532915218)

[1.1. Характеристика языка программирования 6](#_Toc532915219)

[1.2. Алфавит языка 6](#_Toc532915220)

[1.3. Применяемые сепараторы 7](#_Toc532915221)

[1.4. Типы данных. Инициализация по умолчанию 7](#_Toc532915222)

[1.5. Преобразование типов данных 8](#_Toc532915223)

[1.6. Идентификаторы 8](#_Toc532915224)

[1.7. Литералы 8](#_Toc532915225)

[1.8. Объявление данных и область видимости 8](#_Toc532915226)

[1.9. Инструкции языка 9](#_Toc532915227)

[1.10. Конструкции языка 9](#_Toc532915228)

[1.11. Операции языка 10](#_Toc532915229)

[1.12. Выражения и их вычисления 10](#_Toc532915230)

[1.13. Область видимости идентификаторов 11](#_Toc532915231)

[1.14. Семантические проверки 11](#_Toc532915232)

[1.15. Распределение областей оперативной памяти на этапе выполнения 11](#_Toc532915233)

[1.16. Стандартная библиотека и ее состав 12](#_Toc532915234)

[1.17. Ввод и вывод данных 12](#_Toc532915235)

[1.18. Точка входа 12](#_Toc532915236)

[1.19. Препроцессор 12](#_Toc532915237)

[1.20. Соглашение о вызовах 12](#_Toc532915238)

[1.21. Объектный код 13](#_Toc532915239)

[1.22. Классификация сообщений транслятора 13](#_Toc532915240)

[1.23. Контрольный пример 15](#_Toc532915241)

[2. Структура транслятора 16](#_Toc532915242)

[2.1. Компоненты транслятора их назначение и принципы взаимодействия 16](#_Toc532915243)

[2.2. Перечень входных параметров транслятора 17](#_Toc532915244)

[2.3. Перечень протоколов, формируемых транслятором и их содержимое 17](#_Toc532915245)

[3. Разработка лексического анализатора 18](#_Toc532915246)

[3.1. Структура лексического анализатора 18](#_Toc532915247)

[3.2. Контроль входных символов 18](#_Toc532915248)

[3.3. Удаление избыточных символов 19](#_Toc532915249)

[3.4. Перечень ключевых слов, сепараторов, символов операций соответствующих им лексем, регулярных выражений и конечных автоматов. 20](#_Toc532915250)

[3.5. Основные структуры данных 21](#_Toc532915251)

[3.6. Принцип обработки ошибок 22](#_Toc532915252)

[3.7. Структура и перечень сообщений лексического анализатора 22](#_Toc532915253)

[3.8. Алгоритм лексического анализа 22](#_Toc532915254)

[3.9. Контрольный пример 23](#_Toc532915255)

[4. Разработка синтаксического анализатора. 24](#_Toc532915256)

[4.1. Структура синтаксического анализатора 24](#_Toc532915257)

[4.2. Контекстно-свободная грамматика, описывающая синтаксис языка 24](#_Toc532915258)

[4.3. Принцип работы конечного автомата с магазинной памятью 25](#_Toc532915259)

[4.4. Принцип обработки ошибок 26](#_Toc532915260)

[4.5. Контрольный пример 26](#_Toc532915261)

[5. Разработка семантического анализатора 27](#_Toc532915262)

[5.1. Структура семантического анализатора 27](#_Toc532915263)

[5.2. Функции семантического анализатора 27](#_Toc532915264)

[5.3. Структура и перечень сообщений семантического анализатора 27](#_Toc532915265)

[5.4. Принцип обработки ошибок 27](#_Toc532915266)

[5.5. Контрольный пример 27](#_Toc532915267)

[6. Вычисление выражений 29](#_Toc532915268)

[6.1. Выражения, допускаемые языком 29](#_Toc532915269)

[6.2. Польская запись и принцип ее построения 29](#_Toc532915270)

[6.3. Программная реализация обработки выражений 29](#_Toc532915271)

[6.4. Контрольный пример 30](#_Toc532915272)

[7. Генерация кода 31](#_Toc532915273)

[8. Тестирование транслятора 33](#_Toc532915274)

[Заключение 34](#_Toc532915275)

[Список используемой литературы 35](#_Toc532915276)

[ПРИЛОЖЕНИЕ А 36](#_Toc532915277)

[ПРИЛОЖЕНИЕ Б 39](#_Toc532915278)

[ПРИЛОЖЕНИЕ В 40](#_Toc532915279)

[ПРИЛОЖЕНИЕ Г 44](#_Toc532915280)

[ПРИЛОЖЕНИЕ Д 45](#_Toc532915281)

[ПРИЛОЖЕНИЕ Е 47](#_Toc532915282)

[ПРИЛОЖЕНИЕ Ж 48](#_Toc532915283)

# Введение

В данном курсовом проекте была поставлена задача разработать собственный язык программирования PEA-2018, а также транслятор для него.

Для достижения цели курсового проекта были поставлены следующие задачи (описанные в главах):

* разработка спецификации для языка PEA-2018 (глава 1);
* разработка структуры транслятора (глава 2);
* разработка лексического анализатора (глава 3);
* разработка синтаксического анализатора (глава 4);
* разработка семантического анализатора (глава 5);
* разбор арифметических выражений (глава 6);
* разработка генератора кода (глава 7);
* тестирование транслятора (глава 8).

Данный язык программирования предназначен для выполнения арифметических действий и операций над строками.

Трансляция осуществляется в код на языке Assembler.

# 1. Спецификация языка программирования

## Характеристика языка программирования

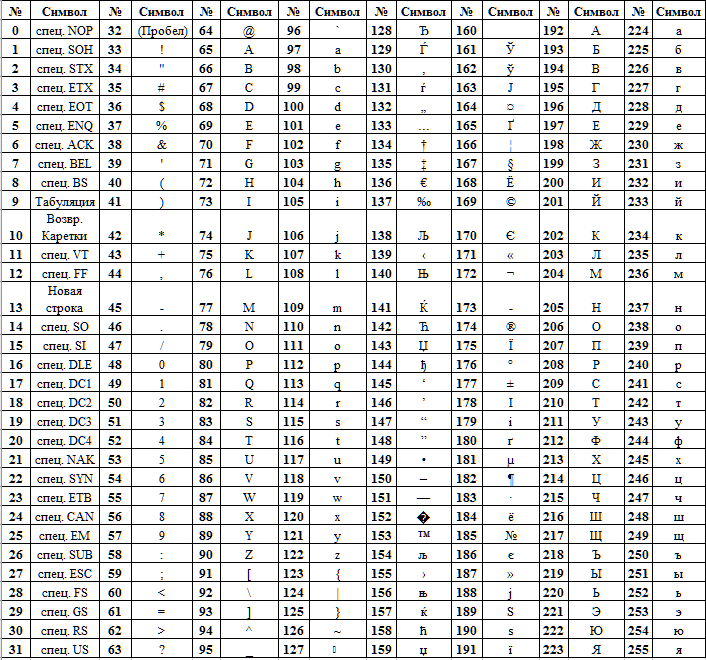
Разработанный язык программирования является универсальным функциональным. PEA-2018 строготипизированный, не объектно-ориентированный.

Язык допускает использование 2 типов данных: строковый и целочисленный. Все переменные локальные. Для работы с целочисленными данными разрешено использование деления, умножения, вычитания и сложения, а также скобок, обозначающих приоритет операций.

Максимальное количество параметров у пользовательской функции в данном языке ограничено десятью.

## Алфавит языка

Транслятор языка PEA-2018 использует кодировку Win-1251.



Для написания исходного кода допускаются символы латинского алфавита верхнего и нижнего регистров, цифры, знаки арифметических операций, а также специальные символы, такие как: пробел, запятая, точка с запятой, фигурные скобки, круглые скобки, символ табуляции, символ новой строки, двойные кавычки.

В транслированном коде используются символы латинского алфавита, цифры, специальные символы: пробел, запятая, символ новой строки, символ табуляции, одинарная кавычка, символ нижнего подчеркивания, а также символ “;” и русские символы для комментариев к строкам транслированного кода.

## Применяемые сепараторы

Таблица 1.1 – Сепараторы языка PEA-2018

|  |  |  |
| --- | --- | --- |
| Символ  (группа символов) | Код символа  (по таблице WIN-1251) | Назначение |
| ; | 59 | Разделитель инструкций |
| { | 123 | Начало тела функции |
| } | 125 | Конец тела функции |
| (  ) | 40  41 | Приоритет операций, параметры функции |
| = | 61 | Присвоение значений |
| +  -  \*  / | 43  45  42  47 | Арифметические операции |
| , | 44 | Разделитель параметров |
| Пробел | 32 | Разделитель слов |

Сепараторы данного языка совпадают с сепараторами языка C++. Для разделения слов используется символ ‘пробел’, программные блоки выделяются символами '{' и ‘}’.

## Типы данных. Инициализация по умолчанию

В данном языке можно использовать 2 типа данных.

Таблица 1.2 – Основные типы данных языка PEA-2018

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип данных | Краткое описание | Занимаемое место в памяти | Границы | Значение при инициализации: |
| int | Целое значение | 4 байта | От -2147483648 до 2147483647 | 0 |
| str | Строка | 201 байт | От 1 до 200  разрешенных символов | Пустая строка, длина строки 0 |

Строковое значения хранится следующим образом: разрешено присваивание переменной (до 200 символов), после чего добавляется символ нуль-терминала (‘\0’).

## Преобразование типов данных

В языке PEA-2018 преобразование данных запрещено, так как язык является строготипизированным. Попытка преобразования данных по типу приведет к ошибке на стадии семантического анализа.

## Идентификаторы

На этапе разбора идентификаторы укорачиваются до десяти символов (используется механизм обрезки символов, которые находятся в данном идентификаторе правее десятого). Идентификаторы не должны совпадать с ключевыми словами. Для написания идентификатора разрешается использовать только символы латинского алфавита.

Имена идентификаторов-функций не должны совпадать с именами ключевых слов языка Assembler, с именами функций стандартной библиотеки.

Ограничения по количеству идентификаторов:

Максимальное количество идентификаторов – литералов 1000.

Максимальное количество идентификаторов – переменных 1000.

Максимальное количество идентификаторов – функций 1000.

Максимальное количество идентификаторов – параметров 1000.

## Литералы

Допустимо использовать литералы двух типов:

Таблица 1.3 – Литералы в языке PEA-2018

|  |  |  |
| --- | --- | --- |
| Тип литерала | Допустимые символы | Допустимые значения: |
| int | Цифры от 0 до 9 | От 0 до 2147483647 |
| string | Разрешенные символы | От 1 до 200 символов |

Строковый тип – комбинация символов, которая может содержать символы латинского алфавита, а также символы пробела, цифр, спец символов (исключая символ двойных кавычек), ограниченных с помощью двойных кавычек с обеих сторон.

Целочисленный тип – комбинация из цифр от 0 до 9.

## Объявление данных и область видимости

Все идентификаторы имеют область видимости. Тип идентификатора может определяться 4 типами: функция, литерал, параметр, переменная.

Ключевое слово, обозначающее тип данных переменной, пишется с префиксом “var”. Ключевое слово, обозначающее тип данных, возвращаемого функцией, пишется с префиксом “fun”. Ключевое слово, обозначающее тип параметра, пишется с префиксом “par”.

Область видимости организовывается следующим образом:

Если переменная (параметр, литерал) объявляется внутри главной функции, то областью видимости является “begin”.

Остальные переменные (параметры, литералы) получают в качестве области видимости имя функции, в которой объявлены, с префиксом “fun”.

Во избежание конфликтов переменных с одинаковыми именами к имени приписывается справа символ нижнего подчеркивания вместе с номером данного идентификатора в таблице идентификаторов.

Полное имя переменной записывается следующим образом:

<имя идентификатора>\_<область видимости>.

Общее правило описания идентификатора и его типа:

<par|var|fun><integer|string> <имя идентификатора>

## Инструкции языка

Инструкции языка описаны в таблице 1.4.

Таблица 1.4 – Инструкции языка PEA-2018

|  |  |
| --- | --- |
| Объявление переменных | <var><тип данных> <идентификатор-переменная>; |
| Вызов функций | <идентификатор> (<идентификатор|литерал>, ...) |
| Присвоение значений | = |
| Вывод в стандартный поток | output <идентификатор>;  output <литерал>;  output; |
| Возвращение значения | recovery <идентификатор>; |

Для вывода используется 3 возможных типа операции: вывод идентификатора, вывод литерала, вывод символа новой строки(‘\n’).

В качестве возвращаемого идентификатора допустимо использовать идентификатор-параметр, либо идентификатор-переменную.

## Конструкции языка

Основными конструкциями языка PEA-2018 являются блоки главной и пользовательских функций. Описание данных конструкций указано в таблице 1.5.

Таблица 1.5 - Основные конструкции языка PEA-2018

|  |  |
| --- | --- |
| Название конструкции | Описание конструкции |
| Объявление функций | <fun><тип данных> <идентификатор> (<par><тип данных> <идентификатор-параметр> …)  {  …  recovery <идентификатор>;  } |
| Объявление главной функции begin | begin  {  …  } |

Блоки функций обозначаются внутри фигурных скобок {}.

Ограничением для функций является максимальное число параметров, которое равно десяти.

Вызов функции разрешен только после ее объявления.

Рекурсивные вызовы запрещены.

Из главной функции запрещен возврат значения.

В пользовательских функциях необходимо указывать в качестве возвращаемого значения идентификатор, находящийся в области видимости данной функции типа, указанного при объявлении функции.

## Операции языка

Допустимые операции языка представлены в таблице 1.6.

Таблица 1.6 – Операции языка PEA-2018

|  |  |  |
| --- | --- | --- |
| Операция | Описание | Приоритет |
| + | Сложение двух целочисленных значений | 1 |
| - | Разность двух целочисленных значений | 1 |
| \* | Умножение дух целочисленных значений | 2 |
| / | Целочисленное деление двух целых чисел | 2 |

Выражение в скобках имеет приоритет 3.

Вначале обрабатывается операция с приоритетом 3, затем операция с приоритетом 2. В последнюю очередь обрабатываются операции с приоритетом 1.

При равенстве приоритетов, первым выполняется операция, находящаяся левее остальных.

## Выражения и их вычисления

В языке PEA-2018 допустимы выражения вида:

<Идентификатор>= <выражение>, где под выражением понимается математическая запись, которая может содержать допустимые знаки арифметических действий, правильно расставленные скобки, а также вызовы функций в виде <имя функции>(<идентификатор-параметр>,…).

Выражения вида (<литерал>= <выражение>) приводят к ошибке на этапе синтаксического анализа и завершению работы программы.

Значения выражений вычисляются согласно приоритетам операций слева направо.

## Область видимости идентификаторов

Идентификаторы-литералы и идентификаторы-параметры видны только до обозначения конца блока функции (символ }).

Глобальных переменных нет. Все переменные являются локальными.

Параметры видны только внутри функции, в которой объявлены.

Функции видны только после их объявления.

Переменные внутри функции видны только после их объявления.

## Семантические проверки

В программе присутствуют следующие семантические проверки:

* наличие строго одной точки входа;
* проверка на использование необъявленной локальной переменной;
* проверка на использование необъявленной функции;
* проверка на количество параметров функции;
* проверка на передаваемые в функцию значения (тип данных);
* проверка на количество параметров, передаваемых в функцию;
* проверка на использование двух и более одинаковых имен переменных в одной области видимости;
* проверка на допустимое значение для литерала;
* проверка возвращаемого функцией идентификатора (по типу);
* проверка на дублирование имен функций;
* проверка на использование арифметических операций для строк;
* проверка на соответствие типов данных в выражении;
* проверка на попытку вызвать рекурсию;
* проверка на неверное использование зарезервированного слова.

При возникновении ошибок на этапе семантического анализа, генерация кода не выполняется, а также программа завершает работу с выводом соответствующего сообщения в файл for\_log.txt.

## Распределение областей оперативной памяти на этапе выполнения

Транслированный код использует 2 области памяти.

В область констант заносятся литералы.

В область глобальных переменных заносятся переменные и параметры, локальная область видимости в исходном коде определяется за счет использования правила именования данных идентификаторов.

## Стандартная библиотека и ее состав

Стандартная библиотека для языка PEA-2018 состоит из 3 функций:

int lenstr, (str) – данная функция возвращает длину строки;

str concat (str, str) – данная функция возвращает указатель на строку, которая состоит из суммы 2 поданных в функцию строк;

str strsub (str STR, int START, int END) – данная функция возвращает указатель на строку, которая содержит подстроку STR, начиная с элемента START и заканчивая элементом с номером END.

Объявление функций стандартной библиотеки происходит неявно для пользователя. Данная часть не влияет на нумерацию строк, но при выводе отформатированного исходного кода, а также далее в файле for\_log.txt пользователь видит данное объявление.

## Ввод и вывод данных

Ввод данных в языке PEA-2018 не предусмотрен.

Вывод данных осуществляется в стандартный поток вывода с помощью команды output.

Правило вызова операции вывода:

output <идентификатор|литерал>;

При использовании оператора вывода, выводится необходимое значение.

При использовании операции вывода ‘output;’ выводится символ новой строки.

## Точка входа

Точкой входа в программу является ключевое слово begin.

В программе должна быть только одна точка входа.

Имена идентификаторов не должны совпадать с именем точки входа.

## Препроцессор

При трансляции языка PEA-2018 использование препроцессора в данной программе не предусмотрено.

## Соглашение о вызовах

При генерации кода на Assembler, в транслированном коде используется соглашение о вызовах \_stdcall.

Аргументы функции передаются справа налево через стек.

## Объектный код

В результате выполнение программы, исходный код будет транслирован на язык Assembler. Транслирование осуществляется на основе таблицы лексем с использованием таблицы идентификаторов.

## Классификация сообщений транслятора

Таблица разделена на соответствующие разделы ошибок, которые определены на каждом из этапов работы программы.

Ошибки с их описанием представлены в таблице 1.7.

Таблица 1.7 – Перечень ошибок при работе транслятора

|  |  |
| --- | --- |
| Номер ошибки | Соответствующее сообщение |
| Ошибки, возникающие на этапе запуска программы, а также на этапе считывания исходного кода из файла | |
| 100 | Параметр -in должен быть задан |
| 105 | Превышена длина входного параметра |
| 110 | Ошибка при открытии файла с исходным кодом (-in) |
| 111 | Ошибка при создании файла кода на ASM (for\_out.txt) |
| 112 | Ошибка при создании файла протокола (for\_log.txt) |
| 115 | Слишком длинный строковый литерал |
| 116 | Недопустимые символы внутри строки-литерала! |
| 117 | Ошибка при обработке данных из файла : незакрытый символ начала строки |
| 118 | Ошибка при обработке данных из файла : встречена пустая строка-литерал |
| 119 | Ошибка при обработке данных из файла : недопустимый символ в исходном коде |
| Ошибки, которые возникают на этапе лексического анализа | |
| 300 | Невозможно распознать слово и преобразовать его в лексему |
| 310 | Неуспешная попытка записи лексемы в таблицу лексем - таблица лексем переполнена! |
| 311 | Использовано недопустимое количество идентификаторов типа: функция (больше ограничения, см. в документацию) |

Продолжение таблицы 1.7

|  |  |  |  |
| --- | --- | --- | --- |
| 312 | Использовано недопустимое количество идентификаторов типа: литерал (больше ограничения, см. в документацию) | | |
| 313 | Использовано недопустимое количество идентификаторов типа: переменная (больше ограничения, см. в документацию) | | |
| 314 | Использовано недопустимое количество идентификаторов типа: параметр (больше ограничения, см. в документацию) | | |
| Ошибки, которые возникают на этапе семантического анализа | | |
| 401 | Семантический анализ: Использована необъявленная переменная либо необъявленная функция! | |
| 402 | Семантический анализ: Превышено количество параметров в функции | |
| 403 | Семантический анализ: Несоответствие типов данных, передаваемых в функцию | |
| 404 | Семантический анализ: Несоответствие количества параметров, передаваемых в функцию | |
| 405 | Семантический анализ: Повторно объявленная переменная либо совпадение имени функции с именем переменной! | |
| 406 | Семантический анализ: Недопустимое значение для литерала! | |
| 407 | Семантический анализ: Тип возвращаемого значения функции не совпадает с объявленным! | |
| 409 | Семантический анализ: Дублирование имен функций! | |
| 410 | Семантический анализ: Попытка использование арифметической операции для строк! Используйте функции стандартной библиотеки | |
| 411 | Семантический анализ: Использование зарезервированного слова, обозначающего имя стандартной функции | |
| 413 | Семантический анализ: Несоответствие типов данных в выражении! | |
| 414 | Семантический анализ: Попытка использовать рекурсию! | |
| Ошибки, которые возникают на этапе синтаксического анализа | | |
| 600 | Синтаксический анализ: Неверная организация программы | |
| 601 | Синтаксический анализ: Ошибка в выражениях внутри тела функции | |
| 602 | Синтаксический анализ: Ошибка в параметрах функции | |
| 603 | | Синтаксический анализ: Ошибка в выражении (например, число\*число/литерал) | |
| 604 | | Синтаксический анализ: Ошибка в передаче аргументов в функцию, например, (, a) | |
| 605 | | Синтаксический анализ: Ошибка в возвращаемом значении | |
| 610 | | Синтаксический анализ: Ошибка на этапе работы синтаксического анализатора! | |

При возникновении ошибок на этап синтаксического анализа ошибки 600-605 являются некритическими и выводятся в файл for\_log.txt, но при наличии хотя бы одной из них, вызывается критическая ошибка 610, которая завершает работу программы с выводом соответствующего сообщения в for\_log.txt файл.

Таким образом, обработка ошибок с их выводом предусмотрена на всех подготовительных этапах перед генерацией кода: на этапе ввода исходного кода, на этапе лексического анализа, на этапе синтаксического анализа, а также на этапе семантического анализа.

## Контрольный пример

В качестве контрольного примера используется 2 исходных кода, написанных на языке PEA-2018. Первый контрольный пример демонстрирует все возможности языка, а также взаимодействие между различными операциями. Данный пример в первую очередь предназначен для тестирования конечного транслятора. Второй контрольный пример демонстрирует возможности языка и является основой для примеров в данной пояснительной записке. Оба примера представлены в приложении А.

# 2. Структура транслятора

2.1. Компоненты транслятора их назначение и принципы взаимодействия

Графическое представление транслятора показано на рисунке 2.1.



Рисунок 2.1 Схема работы транслятора языка PEA-2018

Исходный код - код пользователя из задаваемого пользователем файла.

Лексический анализатор - часть транслятора, составляющая таблицу лексем и таблицу идентификаторов. На вход подается исходных код.

Таблица лексем – промежуточный код.

Таблица идентификаторов – таблица, в которой содержится вся информация об идентификаторах (литералах, параметрах, переменных, функциях), использованных пользователем в своей программе.

Синтаксический анализатор – часть транслятора, выполняющая проверку порядка записи операций пользователем на основе таблицы лексем и с помощью соответствующей грамматики, заданной в форме грамматики Грейбах. На выход подается дерево разбора.

Семантический анализатор – часть кода, проверяющая соответствие исходного кода пользователя со спецификацией языка PEA-2018. На данном этапе используется таблица лексем и таблица идентификаторов.

## 2.2. Перечень входных параметров транслятора

В консоли пользователь указывает только один параметр:

-in:<имя файла исходного кода>.

При отсутствии данного параметра, программа выдает критическую ошибку и завершает работу.

Максимальная длина входного параметра 100.

Имена остальных файлов определены согласно спецификации.

## 2.3. Перечень протоколов, формируемых транслятором и их содержимое

В ходе выполнения трансляции выводится 2 файла: файл for\_log.txt и файл с транслированным кодом (for\_out.asm).

Файл for\_log.txt содержит:

* преобразованный исходный код;
* таблицу лексем и таблицу идентификаторов после лексического анализа;
* ход синтаксического анализа;
* дерево разбора (при разборе без ошибок);
* таблицу лексем после преобразования выражений к обратной польской нотации;
* ошибки и на каком этапе они произошли (если присутствуют в исходном коде).

# 3. Разработка лексического анализатора

## 3.1. Структура лексического анализатора

Структура работы лексического анализатора представлена на рисунке 3.1.



Рисунок 3.1 – Структура работы лексического анализатора

Входными данными для лексического анализатора является форматированный исходный код. После работы лексического анализатора, заполняются таблица идентификаторов и таблица лексем.

## 3.2. Контроль входных символов

Таблица для контроля входных символов представлена в таблице 3.1, составленной на основе кодировки Windows-1251.

Таблица 3.1 – Таблица контроля входных символов на основе таблицы Win-1251

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 0 | F | F | F | F | F | F | F | F | F | P | N | F | F | F | F | F |
| 1 | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| 2 | S | F | Q | F | F | F | F | F | D | D | A | A | D | A | F | A |
| 3 | L | L | L | L | L | L | L | L | L | L | F | D | F | G | F | F |
| 4 | F | E | E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| 5 | E | E | E | E | E | E | E | E | E | E | E | F | F | F | F | F |
| 6 | F | E | E | E | E | E | E | E | E | E | E | E | E | E | E | E |
| 7 | E | E | E | E | E | E | E | E | E | E | E | D | F | D | F | F |
| 8 | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| 9 | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| A | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| B | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| C | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| D | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| E | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |

Обозначения:

E - символ латинского алфавита

F - запрещенный символ

N - символ новой строки (‘\n’)

L - цифра

D - скобка (фигурная либо обычная) или точка с запятой или запятая

A - знаки арифметических действий + - \* /

S - символ (Пробел)

Q - двойная кавычка

G – символ '='

P - символ табуляции

Принцип действия:

если символу соответствует F, то выдаются ошибки о наличии запрещенного символа до конца ввода исходного текста, а после обработки всего исходного кода выдаётся критическая ошибка и программа завершает работу.

В остальных случаях обрабатывается (при необходимости) и дописывается в таблицу входных символов.

## 3.3. Удаление избыточных символов

Избыточные символы удаляются на этапе составления таблицы входных символов.

Алгоритм: пока встречаются символы из списка: (пробел), символ новой строки, точка с запятой, они игнорируются, но если символ ';' является значимым то он приписывается. Также оставляется символ (пробел) между идентификаторами и между идентификатором и литералом.

## 3.4. Перечень ключевых слов, сепараторов, символов операций соответствующих им лексем, регулярных выражений и конечных автоматов.

Перечень ключевых слов с соответствующим им лексемами представлен в таблице 3.1.

Таблица 3.1 – Перечень ключевых слов языка PEA-2018

|  |  |
| --- | --- |
| Ключевое слово | Соответствующая лексема |
| funinteger | f |
| funstring | f |
| parinteger | p |
| parstring | p |
| varinteger | x |
| varstring | x |
| begin | b |
| recovery | r |
| output | o |

Идентификатор может состоять из символов латинского алфавита. В качестве литерала может использоваться последовательность из цифр либо последовательность из разрешенных символов (кроме символа новой строки, табуляции), написанных между двойными кавычками с обеих сторон.

Для разбора идентификаторов и литералов используются, соответствующим образом настроенные, входные данные для недетерминированного конечного автомата.

Для проверки вспомогательных символов, таких как двоеточие, запятая, скобки, фигурные скобки, знаки арифметических операций, используются соответствующие входные данные для автомата.

В общем случае для каждого встреченного слова вызывается проверка на выполнения каждых входных данных для конечного автомата до тех пор, пока не будет найден сработавший вариант.

Автомат является недетерминированным, с конечным числом возможных состояний. Реализация структур конечного автомата описана в приложении Б.

## 3.5. Основные структуры данных

Основные структуры данных, в которых хранятся идентификаторы и лексемы представлены в таблице 3.2.

Таблица 3.2 – Основные структуры данных

|  |  |
| --- | --- |
| Структуры | Комментарий |
| struct LexTable  {  int maxsize;  int size;  Entry\* table;  }; | таблица лексем  максимальный размер  текущий размер  указатель на массив лексем |
| struct Entry  {  char lexema;  int sn;  int number\_in\_id;  int number\_in\_input;}; | одна лексема, элемент таблицы лексем  лексема  номер строки  номер в таблице идентификаторов (только для идентификаторов и литералов)  номер в таблице входных символов |
| struct IdTable {  int maxsize;  int size;  Entry\* table;  int kol\_literals = 0;  int kol\_variables = 0;  int kol\_functions = 0;  int kol\_parametrs = 0; }; | таблица идентификаторов  максимальный размер  текущий размер  указатель на массив идентификаторов  количество литералов в таблице  количество переменных в таблице  количество функций в таблице  количество параметров в таблице |
| struct Entry  {  int idxfirstLE;  int iddatatype;  int idtype;  string id;  union  {  int vint;  string vstr;  } value;  string visibility\_area;}; | идентификатор, один элемент таблицы идентификаторов  номер строки первого упоминания  тип данных  тип идентификатора  имя идентификатора  значение:  целочисленное  строка  область видимости |

Перечисления, используемые для заполнения таблицы идентификаторов:

enum IDDATATYPE

{ INT, STR };

Данное перечисление предназначено для обозначения типа данных литерала, переменной или параметра, а также для обозначения типа возвращаемого значения для функций, где INT-целочисленное значение, STR-строка.

enum IDTYPE {V, F, P, L };

Данное перечисление предназначено для обозначения типа идентификатора, где V-переменная, F-функция, P-параметр, L-литерал.

## 3.6. Принцип обработки ошибок

При возникновении ошибки на стадии выполнения лексического анализа, данная ошибка выводится в файл for\_log.txt. Максимальное количество таких ошибок равно пяти. При возникновении пятой ошибки работа программы принудительно завершается. Если найдена минимум 1 ошибка, то программа выдает соответствующее сообщение и прекращает работу после обработки всего кода лексическим анализатором.

Также является критической ошибкой переполнение таблицы лексем. Максимальный размер таблицы лексем 100 000.

## 3.7. Структура и перечень сообщений лексического анализатора

При возникновении ошибки на этапе лексического анализа, программа прекращает работу с выводом номера ошибки и соответствующим сообщением.

## 3.8. Алгоритм лексического анализа

Словесное объяснение реализации алгоритма на языке С++: из таблицы входных символов считывается строка до тех пор, пока не встретится один из следующих символов: пробел, запятая, точка с запятой, скобка, круглая скобка, знак арифметического действия.

При встрече символа из данного списка, слово передается в функцию, которая проверяет: при каких входных данных конечный автомат вернет истину на запрос.

Если найден автомат, который вернул истину на запрос, то добавляется лексема, соответствующая лексеме, которая хранится в структуре данного автомата. В основную функцию передается номер сработавшего автомата, согласно которому добавляется элемент в таблицу идентификаторов без идентификатора.

Если же номер сработавшего автомата соответствует номеру автомата, который определяет идентификатор, то происходит заполнение имени идентификатора последнего элемента в таблице идентификаторов.

Если не найден автомат, который разберет данное слово, то количество ошибок на данном этапе увеличивается на 1.

Пример разбора идентификатора SUBstr



Рис.

Конечным состоянием считается то, в котором номер состояния совпадает с конечным состоянием, указанным в автомате, а также в качестве строки содержащий пустую строку.

На каждом из этапов будет два варианта перехода: в нулевое состояние, переход в первое(конечное) состояние. Если строка непустая и состояние 1, то разбор неудачен.

## 3.9. Контрольный пример

При выводе таблицы лексем, вначале выводится таблица построчно без дополнительной информации. Ниже выводится каждая лексема с полной информацией о ней.

В приложении В приведена часть файла for\_log.txt с выведенной таблицей лексем и таблицей идентификаторов для второго контрольного примера.

# 4. Разработка синтаксического анализатора.

## 4.1. Структура синтаксического анализатора

Структура синтаксического анализатора указана на рисунке 4.1.



Рисунок 4.1 – Структура синтаксического анализатора

На вход синтаксического анализатора подается таблица лексем. На выходе выводятся ошибки, если программа неверно организована. В случае правильного разбора таблицы лексем выводится дерево разбора, с помощью которого можно проанализировать разбор исходного кода с помощью промежуточного кода.

Во время работы синтаксического анализа в for\_log.txt файл выводится процесс разбора таблицы лексем.

## 4.2. Контекстно-свободная грамматика, описывающая синтаксис языка

Контекстно-свободной называется грамматика, у которой левые части правил являются одиночными нетерминалами. Смысл контекстно-свободной грамматики в том, что правило перехода для нетерминала не зависит от контекста этого нетерминала.

Грамматика, описывающая синтаксис языка PEA-2018 описана с помощью грамматики Грейбах.

Стартовый символ S

Правила переходов, где большими буквами обозначены нетерминалы, указаны ниже.

Правило описания функций стандартной библиотеки:

S->si(pi, pi); si(pi, pi, pi); si(pi);S

Блок главной функции:

S->b{B}

Описание пользовательской функции:

S->fi(P){BG}S

Некоторая строка в блоке функции.

B->xi;

B->xi; B

B->i=V; B

B->i=V;

B->oi;

B->ol;

B->oi; B

B->ol; B

B->o; B

B->o;

Возврат из функции.

G->ri;

Параметры объявления функции.

P->pi

P->pi, P

Правила, описывающие возможные правые части выражений.

V->i

V->l

V->ivV

V->lvV

V->(V)

V->(V)vV

V->i(A)

V->i(A)vV

Варианты параметров, передаваемых в функцию.

A->i

A->l

A->i, A

A->l, A

Также грамматика языка в форме грамматики Грейбах, реализованной на С++, указана в приложении Г.

## 4.3. Принцип работы конечного автомата с магазинной памятью

Алгоритм работы конечного автомата с магазинной памятью следующий:

1. в магазин записывается стартовый символ;
2. на основе полученной раннее таблицы лексем формируется входная лента;
3. запускается автомат;
4. выбирается цепочка, соответствующая нетерминальному символу, записывается в магазин в обратном порядке;
5. если терминалы в стеке и на ленте совпадают, то данный терминал удаляется с ленты и стека, состояние сохраняется. Иначе идет возврат в предыдущее состояние, где происходит попытка применить следующее правило перехода для нетерминала;
6. если в магазине встретился нетерминал, переход к пункту 4;
7. если достигнуто дно стека, и лента в этот момент пуста, то синтаксический анализ выполнен успешно. Иначе вызывается ошибка.

## 4.4. Принцип обработки ошибок

При возникновении ошибки, идет проверка на дальность считывания лексем с магазина. Если текущая ошибка появилась на более дальнем этапе, чем одна из хранящихся в массиве ошибок, то идет перезапись.

В конце работы в массиве ошибок хранится до 5 ошибок, которые хранят строки, в которых и возникли ошибки, а также их описание.

При выводе ошибок в файл for\_log.txt, выводятся все ошибки, а далее вызывается критическая ошибка, завершающая работу программы.

## 4.5. Контрольный пример

Контрольными примерами для проверки работы синтаксического анализа являются контрольные примеры для всей программы. Они указаны в приложении А.

.

# 5. Разработка семантического анализатора

## 5.1. Структура семантического анализатора

Семантический анализатор расположен в 4 частях программы: ввод данных из файла, лексический анализатор, синтаксический анализатор, а также в вынесенной части семантического анализатора.

## 5.2. Функции семантического анализатора

Семантический анализатор предназначен для проверки соответствия исходного кода спецификации, например, соответствие исходного кода заданной грамматике (синтаксический анализ), длина идентификаторов.

В данной программе в отдельный блок семантического анализа вынесены те проверки, которые невозможно выполнить на других этапах, например, связанные с типами данных, а также те проверки, которые проще обработать в отдельном блоке, чем в связке с другими этапами.

## 5.3. Структура и перечень сообщений семантического анализатора

После успешной работы синтаксического анализатора, в файл for\_log.txt помимо хода разбора, выводится дерево разбора, а также сообщение об успешном прохождении синтаксического анализа.

Таблица ошибок, который могут возникнуть на этапе вынесенной части семантического анализатора описана в таблице 1.7 в пункте 1.22. Номера ошибок 401-405, 406,407,409-411, 413, 414.

## 5**.4. Принцип обработки ошибок**

На этапе семантического анализа ошибка использования необъявленной переменной выводится на все встреченные случаи в файл for\_log.txt, после чего выдается критическая ошибка и программа завершает работу.

На этапе лексического анализа при проверке соответствия слов входного кода заданным правилам, ошибка о неразобранной лексеме выводится не более 5ти раз, после чего программа завершает работу.

При возникновении других ошибок, выдается критическая ошибка и программа завершает работу.

## 5.5. Контрольный пример

Контрольные примеры, вызывающие ошибки на каждом из этапов, представлены в таблице 5.1.

Таблица 5.1.

|  |  |
| --- | --- |
| Исходный код | Вызываемая ошибка |
| begin  {  output "";  } | Ошибка на этапе вывода текста из файла  Номер ошибки: 121  Комментарий: пустая строка |
| begin  {  output 111111111111111 ;  } | Ошибка на этапе лексического анализа  Номер ошибки 406  Комментарий: недопустимое целочисленное значение |
| begin  {  recovery 3;  } | Ошибка на этапе синтаксического анализа  Номера ошибок: 600, 601,602, а также 610 (Критическая ошибка)  Комментарий: из главной функции возврат запрещен. |
| begin  {  output a;  } | Ошибка на этапе выполнения вынесенной части семантического анализатора.  Номер ошибки 401  Комментарий: использование необъявленной локальной переменной. |

# 6. Вычисление выражений

## 6.1. Выражения, допускаемые языком

В данном языке допускаются выражения с целочисленными значениями с обеих сторон, составленные с помощью формы записи

<идентификатор>=<выражение>;

Где под выражением понимается любая математически правильно составленная цепочка операций с использованием допустимых символов арифметических операций, а также круглых скобок и вызовов функций внутри выражения.

Для выражений со строками допускается 2 формы записи:

<идентификатор>=<идентификатор|литерал>;

Либо

<идентификатор>=<имя\_функции\_возвращающей\_строку> (<параметры\_передаваемые\_в\_функцию>);

## 6.2. Польская запись и принцип ее построения

Польская запись предназначена для упрощения обработки выражений, в том числе за счет избавления от скобок, обозначающих приоритет операций. В данной работе используется обратная польская запись. Отличительной особенностью данной записи является постфиксное расположение знаков операций.

Преобразование в польскую запись имеет смысл для вызовов функций и для выражений с целочисленными значениями.

Для упрощения дальнейшей обработки параметры функции вида (<параметр>, …) преобразуются к виду [<параметр>…]. Пояснение: убираются запятые и меняются скобки для определенности.

## 6.3. Программная реализация обработки выражений

При преобразовании выражения к обратной польской записи используется исходное выражение, стек для знаков, вектор для результата.

Запускается цикл для исходного выражения.

Если лексема литерал либо идентификатор, то номер данного элемента отправляется в вектор результата.

Если лексема - знак арифметического действия, то пока в вершине стека находится лексема, обозначающая арифметическое действие с приоритетом не меньше чем приоритет действия текущей лексемы, знаки из стека перемещаются в вектор результата, а исходный элемент добавляется в стек.

Если лексема-символ ‘(‘, то в стек добавляется число, которым обозначается данный символ.

Если лексема-символ ‘)’, то из стека перемещаются все знаки в вектор результата до первого числа, обозначающего символ ‘(‘, который удаляется из стека.

## 6.4. Контрольный пример

Ниже представлены примеры обработки цепочек лексем, полученных из некоторых выражений, а также более подробно представленный пример разбора выражения из первого контрольного примера.

Цепочка лексем:

i=i\*i\*i+i;

После преобразования к польской записи:

i=ii\*i\*i+;

Цепочка лексем:

i=i(i);

После преобразования к польской записи:

i=i[i];

Исходное выражение (из первого контрольного примера):

ggg=5+lenstr(testingSumming)\*5\*(lenstr(a)/lenstr(b)+c-d/(c+d-lenstr(a)\*100));

Цепочка лексем для исходного выражения:

i=lvi(i)vlv(i(i)vi(i)viviv(ivivi(i)vl));

Исходное выражения после преобразования к обратной польской записи:

ggg = 5 lenstr[testingSumming] 5\*lenstr[a] lenstr[b]/ c + d c d + lenstr[a] 100\*-/-\*+ ;

Цепочка лексем после преобразования выражения к обратной польской записи:

i=li[i]lvi[i]i[i]viviiivi[i]lvvvvvv;

# 7. Генерация кода

Генерация кода – заключительный этап работы транслятора. Результатом данного этапа будет код, сгенерированный для выполнения MASM, что является требуемым результатом работы программы.

Схема данного этапа изображена на рисунке 7.1.



Рисунок 7.1 – Схема работы генератора кода

На вход в генератор кода подаются таблица лексем и таблица идентификаторов.

Генерация происходит следующим образом: в зависимости от текущей (или при необходимости текущей, а также окружающих лексем) генерируется соответствующее выражение на ассемблере с использованием таблицы идентификаторов, ссылки на элементы которой хранятся в структуре каждой лексемы.

Такие участки конечного кода, как оформление верхней части кода до блока переменных (система команд, подключаемые библиотеки, прототипы функций), область переменных, область констант генерируются отдельно, блок кода. Область переменных и область констант генерируются с помощью таблицы идентификаторов.

Если встречена лексема вывода (‘o’), то генерируется один из вариантов вывода: если следующим словом от данной лексемы стоит идентификатор с целочисленным значением, то в стек добавляется значение данного идентификатора, вызывается внутренняя функция write\_int, которая выводит данное значение. Если следующим словом от данной лексемы является идентификатор со строковым значением, то в стек добавляется адрес данной строки, вызывается функция write\_str, которая выводит данную строку. Если следующим символом стоит ';', то вызывается функция, которая выводит символ новой строки.

Лексема возврата из функции обрабатывается аналогичным образом, однако возврат из функции строкового значения осуществляется через заданную буферную переменную, а возврат целочисленного значения осуществляется через регистр edx.

Выражения со строками обрабатываются отдельно: если идентификатору строкового типа присваивается значение функции, возвращающей строку, то вначале в стек добавляются параметры, передаваемые в функцию, а далее возвращенное значение через буферную строку присваивается исходному идентификатору, а если одному идентификатору строкового типа присваивается другой идентификатор строкового типа, то данное выражение обрабатывается как добавление адресов двух данных строк в стек, а затем копирование через указатели с помощью недоступной пользователю функции, размещенной в подключаемой библиотеке со стандартными функциями.

При обработке выражений с целыми значениями, используется обратная польская запись. Значения идентификаторов и литералов добавляются в стек, а при нахождении лексемы-знака, 2 последних значения достаются в регистры, выполняется необходимое действие и полученное значение добавляется в стек. Если необходимо выполнить деление, то перед выполнением происходит проверка значения регистра, так как деление на нуль недопустимо. Если делитель равен нулю, то идет переход к концу функции, где выводится строка о данной ошибке и завершается работа программы. Вызов функций аналогичен вызову функций при выражениях со строковым типом, однако значение функции возвращается через регистр edx.

Обработка остальных лексем выполняется аналогичным образом.

После генерации кода сгенерированная строка выводится в файл for\_out.txt.

Примеры участков сгенерированного кода для второго контрольного примера приведены в приложении Д.

# 8. Тестирование транслятора

Тестирование транслированного кода будет осуществляться в проекте Visual Studio, настроенном для компилятора MASM и с подключенной библиотекой со стандартными функциями.

Результат работы программы для первого контрольного примера представлен в приложении Е.

Результат работы программы для второго контрольного примера представлен в приложении Ж.

# Заключение

В ходе выполнения курсового проекта, были выполнены поставленные задачи: лексический анализатор был реализован на основе разбора слов с помощью недетерминированного конечного автомата, синтаксический анализатор реализован на конечном автомате с магазинной памятью, семантический анализатор был организован на основе циклических проверок выполнения заданных условий. Генератор кода был написан на основе прохода по таблице лексем. Тестирование проводилось параллельно написанию программы с помощью написания тестовых входных данных, вызывающих выполнение программы с последующей генерацией кода на assembler. Основным тестовым случаем был второй контрольный пример, в котором не только демонстрируются все возможности языка, но и взаимодействие данных возможностей. Результат первого контрольного примера был просчитан вручную, что в последствии дало возможность сравнить результат, полученный после выполнения транслированного кода с необходимым результатом. Также были разработаны тесты, в которых были допущены различные ошибки. Данные тесты были необходимы для проверки работоспособности и правильности обработки некорректных данных.

# Список используемой литературы

1. Наркевич А. С. Курс лекций по предмету языки программирования - 2018
2. Ахо А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – M.: Вильямс, 2003. – 768с.
3. Пирогов В.Ю. Ассемблер для Windows. -  БХВ-Петербург, 2012. -896с.
4. Молчанов А.Ю. Системное программное обеспечение. – СПб.: Питер, 2010. –  400с.
5. Ирвин К. Р. Язык ассемблера для процессоров Intel / К. Р. Ирвин. – M.: Вильямс, 2005. – 912с.
6. Калашников О. А. Ассемблер — это просто. Учимся программировать. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2011. — 336 с.

# ПРИЛОЖЕНИЕ А

Контрольный пример 1:

funinteger firstf(parinteger a)

{

varinteger g;

g=10;

g=g\*a\*a+a;

recovery g;

}

funstring secondf(parstring a)

{

varstring aaa;

aaa=" AND some text";

varstring bbb;

bbb=concat(a,aaa);

varinteger aabbcc;

aabbcc=firstf(1);

output "aabbcc = ";

output aabbcc;

output;

recovery bbb;

}

funstring thirdf(parstring a)

{

varstring b;

b="HELLOqwertyuWORLDiopasdfghjklzxcvbnm+= ";

varstring buffer;

buffer=strsub(b,1,5);

a=concat(a,buffer);

buffer=strsub(b,39,39);

a=concat(a,buffer);

buffer=strsub(b,13,17);

a=concat(a,buffer);

recovery a;

}

begin

{

output "STANDART TESTING";

varstring a;

varstring b;

varstring result;

varinteger c;

varinteger d;

a="first string";

b="second string";

c=2;

d=5;

output "a =";

output a;output;

output "b =";

output b;output;

output "c =";

output c;output;

output "d =";

output d;output;

varstring forccc;

result=concat(a,b);

output "result of SUM a and b = ";

output result;

output;

result=strsub(a,c,d);

output "result of SUB a from 2 to 5 = ";

output result;

c=lenstr(a);

output "LENGTH of a = ";

output c;

output;output;

forccc=secondf(b);

output forccc;

output ;output;

output "CHECK DIVISION";output;

c=100;

d=51;

output "c = ";

output c;

output;

output "d = ";

output d;

output;

c=c/d;

output "after division";output;

output "c = ";

output c;

output;

output "d = ";

output d;

output;

output;

output "TESTING SUMMING string to HELLO WORLD and get mystring with this answer";

varstring testingSumming;

testingSumming="start string ";

output;

testingSumming=thirdf(testingSumming);

output ;

output testingSumming;

output;output;

output "CHECK PROGRAM on using calling function in EXPRESSION";output;

output "BIG hard expression";output;

varinteger ggg;

output "5+lenstr(testingSumming)\*5\*(lenstr(a)/lenstr(b)+c-d/(c+d-lenstr(a)\*100))";

output;

ggg=5+lenstr(testingSumming)\*5\*(lenstr(a)/lenstr(b)+c-d/(c+d-lenstr(a)\*100));

output ggg;

output;

}

;; ; ; ;;

;

;

;;

Контрольный пример 2:

funstring thirdf(parstring a)

{

varinteger len;

len=lenstr(a);

a=strsub (a,1,len);

recovery a;

}

begin

{

varstring a;

a="HELLO";

a=thirdf(a);

output a;

}

# ПРИЛОЖЕНИЕ Б

struct RELATION

{

char symbol;

int nnode;

};

struct NODE

{

int n\_relation;

RELATION \*relations;

};

struct FST

{

std::string string;

enum\_FST number;

char lexem\_to\_return;

int position;

int nstates;

NODE \*nodes;

int \*rstates;

};

# ПРИЛОЖЕНИЕ В

Выведенная таблица лексем:

Всего лексем : 82

-3 si(pi,pi);

-2 si(pi,pi,pi);

-1 si(pi);

0 fi(pi)

1 {

2 xi;

3 i=i[i];

4 i=i[ili];\_\_

5 ri;

6 }

7 b{

8 xi;

9 i=l;

10 i=i[i];

11 oi;

12 }

LC №LT Lex №IT №IN

-3 0 s # 0

-3 1 i 0 10

-3 2 ( # 16

-3 3 p # 17

-3 4 i 1 27

-3 5 , # 28

-3 6 p # 29

-3 7 i 2 39

-3 8 ) # 40

-3 9 ; # 41

-2 10 s # 43

-2 11 i 3 53

-2 12 ( # 59

-2 13 p # 60

-2 14 i 4 70

-2 15 , # 71

-2 16 p # 72

-2 17 i 5 83

-2 18 , # 84

-2 19 p # 85

-2 20 i 6 96

-2 21 ) # 97

-2 22 ; # 98

-1 23 s # 100

-1 24 i 7 111

-1 25 ( # 117

-1 26 p # 118

-1 27 i 8 128

-1 28 ) # 129

-1 29 ; # 130

0 30 f # 132

0 31 i 9 142

0 32 ( # 148

0 33 p # 149

0 34 i 10 159

0 35 ) # 160

1 36 { # 162

2 37 x # 164

2 38 i 11 175

2 39 ; # 178

3 40 i 11 180

3 41 = # 183

3 42 i 7 184

3 43 [ # #

3 44 i 10 191

3 45 ] # #

3 46 ; # #

4 47 i 10 195

4 48 = # 196

4 49 i 3 197

4 50 [ # #

4 51 i 10 204

4 52 l 12 206

4 53 i 11 208

4 54 ] # #

4 55 ; # #

4 56 \_ # #

4 57 \_ # #

5 58 r # 214

5 59 i 10 223

5 60 ; # 224

6 61 } # 226

7 62 b 13 229

7 63 { # 234

8 64 x # 236

8 65 i 14 246

8 66 ; # 247

9 67 i 14 249

9 68 = # 250

9 69 l 15 252

9 70 ; # #

10 71 i 14 261

10 72 = # 262

10 73 i 9 263

10 74 [ # #

10 75 i 14 270

10 76 ] # #

10 77 ; # #

11 78 o # 274

11 79 i 14 281

11 80 ; # 282

12 81 } # 284

Количество литералов в таблице : 2

Количество функций в таблице : 5

Количество переменных в таблице : 2

Количество параметров функций в таблице : 7

Всего идентификаторов : 16

List of identifikators, literals

number name idtype linecode value

12 LIT1 integer 4 1

15 LIT2 string 9 "HELLO"

List of identifikators, variables

number name idtype linecode init\_value

11 len INT 2 0

visibility area: funthirdf

14 a STR 8 ""

visibility area: begin

List of identifikators, parametrs

number name type linecode

1 a string -3

visibility area: funconcat

2 b string -3

visibility area: funconcat

4 a string -2

visibility area: funstrsub

5 b integer -2

visibility area: funstrsub

6 c integer -2

visibility area: funstrsub

8 a string -1

visibility area: funlenstr

10 a string 0

visibility area: funthirdf

List of identifikators, functions

number name linecode type

0 concat -3 string

visibility area: STANDART

3 strsub -2 string

visibility area: STANDART

7 lenstr -1 integer

visibility area: STANDART

9 thirdf 0 string

visibility area:

13 begin 7 integer

visibility area: begin

# ПРИЛОЖЕНИЕ Г

// ГРАММАТИКА ГРЕЙБАХ

# Greibach greibach(

NS('S'), TS('$'), 6, // стартовый символ, дно стека, количество правил

Rule(

NS('S'), 600, 3, // неверная структура программы

Rule::Chain(10, TS('f'), TS('i'), TS('('), NS('P'), TS(')'), TS('{'), NS('B'), NS('G'), TS('}'), NS('S')),

Rule::Chain(4, TS('b'), TS('{'), NS('B'), TS('}')),

Rule::Chain(19, TS('s'), TS('i'), TS('('), NS('P'), TS(')'), TS(';'), TS('s'), TS('i'), TS('('), NS('P'), TS(')'), TS(';'), TS('s'), TS('i'), TS('('), NS('P'), TS(')'), TS(';'), NS('S'))

),

Rule(

NS('B'), 601, 10, // тело

Rule::Chain(3, TS('x'), TS('i'), TS(';')),

Rule::Chain(4, TS('x'), TS('i'), TS(';'), NS('B')),

Rule::Chain(5, TS('i'), TS('='), NS('V'), TS(';'), NS('B')),

Rule::Chain(4, TS('i'), TS('='), NS('V'), TS(';')),

Rule::Chain(4, TS('o'), TS('i'), TS(';'), NS('B')),

Rule::Chain(3, TS('o'), TS('i'), TS(';')),

Rule::Chain(3, TS('o'), TS('l'), TS(';')),

Rule::Chain(4, TS('o'), TS('l'), TS(';'), NS('B')),

Rule::Chain(2, TS('o'), TS(';')),

Rule::Chain(3, TS('o'), TS(';'), NS('B'))

),

Rule(

NS('G'), 605, 1, // возврат значения

Rule::Chain(3, TS('r'), TS('i'), TS(';'))

),

Rule(

NS('P'), 602, 2, // параметр

Rule::Chain(4, TS('p'), TS('i'), TS(','), NS('P')),

Rule::Chain(2, TS('p'), TS('i'))

),

Rule(

NS('V'), 603, 8,

Rule::Chain(1, TS('i')),

Rule::Chain(1, TS('l')),

Rule::Chain(3, TS('i'), TS('v'), NS('V')),

Rule::Chain(3, TS('l'), TS('v'), NS('V')),

Rule::Chain(3, TS('('), NS('V'), TS(')')),

Rule::Chain(5, TS('('), NS('V'), TS(')'), TS('v'), NS('V')),

Rule::Chain(4, TS('i'), TS('('), NS('A'), TS(')')),

Rule::Chain(6, TS('i'), TS('('), NS('A'), TS(')'), TS('v'), NS('V'))

),

Rule(

NS('A'), 604, 4, // аргумент

Rule::Chain(3, TS('i'), TS(','), NS('A')),

Rule::Chain(3, TS('l'), TS(','), NS('A')),

Rule::Chain(1, TS('i')),

Rule::Chain(1, TS('l'))

)

);

# ПРИЛОЖЕНИЕ Д

Пример объявления внешних функций:

strsub PROTO : DWORD, :SDWORD, :SDWORD, :DWORD

concat PROTO : DWORD, :DWORD, :DWORD

Пример объявления констант:

.const

consolename db 'PEA-2018',0 ; CONSOLE NAME

ZAP\_registers\_\_\_ byte 'ERROR on size of variable', 0

ERROR\_DIV\_NULL byte 'ERROR IN DIVISION(NULL)', 0

ERROR\_function\_strsub byte 'ERROR IN fucntion strsub', 0

ERROR\_function\_concat byte 'ERROR IN fucntion concat', 0

LIT1 SDWORD 1

LIT2 byte 'HELLO', 0

Пример объявления переменных:

.data

name\_buffer\_for\_NULL\_string byte 201 dup(0)

buffer\_string\_for\_return byte 201 dup(0)

len\_funthirdf\_11 SDWORD 0

a\_main\_14 byte 201 dup(0)

Пример функции:

thirdf PROC uses eax ebx ecx edi esi, a\_funthirdf\_10 : SDWORD

mov len\_funthirdf\_11, 0

mov eax, a\_funthirdf\_10

push a\_funthirdf\_10

call lenstr

push eax

pop len\_funthirdf\_11

push offset buffer\_string\_for\_return

push len\_funthirdf\_11

push LIT1

push a\_funthirdf\_10

call strsub

cmp eax, -1

je EXITstrsuberror

push a\_funthirdf\_10

push offset buffer\_string\_for\_return

call copystring\_

push offset buffer\_string\_for\_return

push a\_funthirdf\_10

call copystring\_

jmp EXIT

EXIT\_div\_on\_NULL:

push offset ERROR\_DIV\_NULL

call write\_str

push -1

call ExitProcess

EXITstrsuberror:

push offset ERROR\_function\_strsub

call write\_str

push -2

call ExitProcess

EXITconcaterror:

push offset ERROR\_function\_concat

call write\_str

push -3

call ExitProcess

ZAP\_registers\_\_:

push offset ZAP\_registers\_\_\_

call write\_str

push - 4

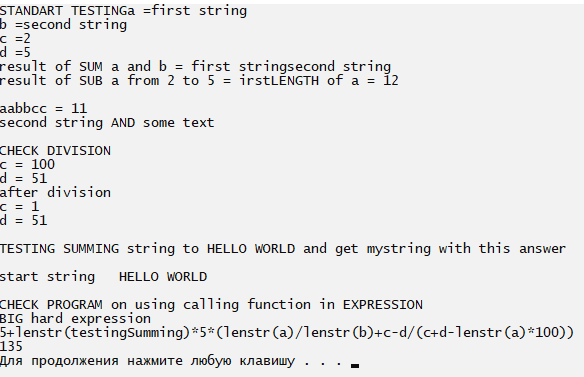
call ExitProcess

EXIT:

ret 0

thirdf ENDP

# ПРИЛОЖЕНИЕ Е



# ПРИЛОЖЕНИЕ Ж

