ГУАП

КАФЕДРА № 44

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

канд. техн. наук, доцент		Н.В. Кучин
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №2

ПОСТРОЕНИЕ РАСПОЗНАВАТЕЛЯ ДЛЯ АНАЛИЗА ОПИСАНИЙ ТИПОВ ДАННЫХ И ПЕРЕМЕННЫХ И ОПРЕДЕЛЕНИЕ ОБЪЁМА ПАМЯТИ ДЛЯ СТРУКТУР ДАННЫХ

по курсу: Системное программное обеспечение

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. № 4143		Е. Д. Тегай
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

Цель работы

Построение распознавателя исходного текста программы, содержащего описания типов данных, структур данных и переменных.

Изучение основных принципов распределения памяти, ознакомление с алгоритмами расчета объема памяти, занимаемой простыми и составными структурами данных, получение практических навыков создания простейшего анализатора для расчета объема памяти, занимаемого заданной структурой данных.

Задание по лабораторной работе

Для выполнения лабораторной работы требуется написать программу, которая анализирует текст входной программы, содержащий описания типов данных и переменных. Результатом работы программы является проверка корректности исходного описания и подготовка внутренней структуры данных для вычисления объёма памяти, необходимой для размещения переменных, определенных во входном тексте (результат работы программы будет далее использован).

Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, если структура входной программы не соответствует заданию. Если в исходном тексте встречаются типы данных или структуры данных, не предусмотренные заданием, программа должна сигнализировать об ошибке.

Длину идентификаторов и строковых констант можно считать ограниченной 32 символами. Программа должна допускать наличие комментариев неограниченной длины во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

Входная программа представляет собой последовательность двух блоков: первый блок – описание типов, начинающееся с ключевого слова type, второй блок – описание переменных, начинающееся с ключевого слова var. Описание типов и описание переменных выполняется в стиле языка Pascal.

Необходимо разобрать описания всех типов, рассчитать объём для каждого типа данных на основе известных алгоритмов и размеров скалярных типов данных, указанных в задании. Расчёт надо выполнить в двух случаях: с учётом кратности распределения памяти и без него. Затем на основе рассчитанных размеров типов данных необходимо рассчитать объём памяти, занимаемой всеми описанными переменными (расчёт также должен выполняться в двух вариантах: с учетом кратности распределения памяти и без него). Результатом выполнения программы должны быть две величины: размер памяти, требуемой для всех описанных переменных, с учётом кратности распределения и тот же самый размер без учёта кратности распределения.

Во всех вариантах символ S является начальным символом грамматики; L, T, R, V, K, D, F и E обозначают нетерминальные символы. Терминальные символы выделены жирным шрифтом. Терминальный символ c соответствует одному из двух скалярных типов, указанных в задании. Терминальный символ t соответствует одному из типов данных, который может быть описан в секции type, а терминальный символ a соответствует переменным, которые могут быть описаны в секции var. Терминальный символ d соответствует любой целочисленной константе. Грамматики в заданиях отличаются только правилами для терминальных символов D, F и E, которые описывают вариант допустимых типов данных – возможны три варианта: структура данных, союз (запись с вариантами) или массив.

Кроме расчёта объёма памяти программа, построенная на основе задания, должна выполнять синтаксическую проверку и элементарный семантический контроль входной программы – в том случае, если встречается тип данных, не описанный в секции type, должно выдаваться сообщение об ошибке.

Входная грамматика описаний типов для программы описывается следующими правилами:

$$S \rightarrow \text{type L var R}$$

$$L \rightarrow \text{T;} \mid \text{T; L}$$

$$T \rightarrow \text{t} = \text{c} \mid \text{t} = \text{D}$$

$$R \rightarrow \text{V;} \mid \text{V; R}$$

$$V \rightarrow \text{K: t} \mid \text{K: c} \mid K: D$$

$$K \rightarrow \text{a} \mid \text{K, a}$$

$$D \rightarrow \text{record F end}$$

$$F \rightarrow E; \mid E; F$$

$$E \rightarrow \text{K: c} \mid \text{K: t}$$

Также вариант №19 дополняется следующим содержанием:

Скалярные типы (размер в байтах) – byte (1 байт), word (2 байта)

Кратность распределения памяти – 2

Кратность элементов структур - Да

Запись заданной грамматики входного языка в форме Бэкуса-Наура

Исходная запись продемонстрирована ниже:

$$G(\{S, L, T, R, V, K, D, F\ E\}, \{type, var, record, end, t, c, a, =, ,, :, ;\}, P, S)$$

Описание алгоритма расчёта объёма памяти для структуры данных, указанной в задании

Шаг 1. Для каждого типа данных определяется его размер в байтах. В нашем случае для byte – 1 байт, для real – 6 байт. Причём byte с учётом выравнивания и особенностей системы, фактически может занимать 2 байта.

Шаг 2. Определяем структуры данных. После того, как размеры примитивных типов определены, переходим к вычислению размера сложных типов данных - структур (как например EmployeeRecord, myDepartment в примере в дальнейшем). Структура состоит из различных полей, каждое из которых может быть либо примитивным типом, либо другим типом данных (также структурой). Для каждого поля структуры нужно выполнить следующие шаги:

- Если поле является примитивным типом (byte, real), то для него используется заранее известный размер.
- Если поле является другой структурой, то для вычисления его размера нужно рекурсивно рассчитать размер каждого поля вложенной структуры.

Шаг 3. Рассчитаем размер структуры. После того, как был вычислен размер каждого поля структуры, суммируем эти размеры. Однако здесь стоит помнить о выравнивании данных в памяти.

Шаг 4. Рассчитаем объём памяти для переменных. Для каждой переменной, объявленной в программе, определяется её тип. На основе типа переменной (byte, real или сложная структура) рассчитывается размер памяти, который она занимает.

- Если переменная является примитивом, то её размер будет равен размеру этого типа.
- Если переменная является структурой, то используется её размер, рассчитанный для соответствующего типа структуры.

Этот процесс повторяется для каждой переменной, чтобы найти в дальнейшем общий объём памяти, необходимый для хранения всех переменных в программе.

Шаг 5. Итоговый расчёт. Суммируем объём памяти для всех переменных, чтобы вычислить общий объём памяти, который потребуется для работы программы.

Разработка программы

В качестве языка программирования был выбран JavaScript, так как он использовался и в предыдущих лабораторных работах. Сам проект является многофайловым, который состоит из 3 файлов: 2 вспомогательных (предварительные этапе распознавания) и 1 основного, внутри которого и производятся расчёты для вычисления памяти. Разберём подробнее каждый из них.

Первым вспомогательным файлом является LEXER.js. В нём описана реализация лексера для последующего парсинга исходного кода с использованием библиотеки Chevrotain.

Это такая библиотека для создания парсеров и лексеров на выбранном мной языке. Она используется для обработки текста в структуры данных, подходящие для дальнейшего анализа, компиляции или интерпретации. Основными терминами данной библиотеки являются лексер и парсер. Лексер - компонент, который разбивает исходный текст на лексемы (токены). Лексемы представляют собой минимальные смысловые единицы, такие как ключевые слова, операторы, идентификаторы и символы. Парсер - компонент, который принимает лексемы, созданные лексером, и анализирует их в соответствии с правилами синтаксиса языка. Он строит синтаксическое дерево, которое представляет структуру программы или текста. Токенами же называются минимальные элементы, которые лексер извлекает из текста.

В начале файла импортируются необходимые компоненты из этой библиотеки, а если быть точнее, то: lexer – класс создания лексера и createToken – функция для создания токенов, которые будут использованы лексером для обработки входных данных.

Далее определяются токены:

Space – пробельные символы (пробелы, табуляции, новые строки и так далее), которые затем будут пропускаться лексером

TypeToken – токен, соответствующий ключевому слову type

VarToken – токен, соответствующий ключевому слову var

ByteToken – токен, соответствующий ключевому слову byte

RealToken – токен, соответствующий ключевому слову real

RecordToken – токен, соответствующий ключевому слову record

EndToken – токен, соответствующий ключевому слову end

IdentifierToken – токен для идентификаторов, состоящих из букв и цифр, которые начинаются с буквы или подчеркивания (например, имена переменных или типов)

SemicolonToken – токен для символа ‘;’

ColonToken – токен для символа ‘:’

CommaToken – токен для символа ‘,’

ColonToken – токен для символа ‘:’

EqualsToken – токен для символа ‘=’

Каждому токену соответствует регулярное выражение, которое определяет, какие строки будут соответствовать данному токену.

После этого объявляется массив Tokens, в котором перечисляются все токены, которые будут использованы лексером. Этот массив, собственно, содержит все ранее описанные токены. Затем создаётся лексер с использованием соответствующего класса Lexer. Он принимает массив токенов в качестве аргумента. Лексер будет использовать эти токены для парсинга входного текста и выделения соответствующих лексем.

Завершающим шагом является экспорт лексера для дальнейшего использования в основном файле.

Вторым вспомогательным файлом является файл PARSER.js, в котором был реализован парсер для грамматики с использованием той же библиотеки Chevrotain. Этот парсер использует лексемы, определённые ранее в файле LEXER.js. Сначала в этом файле определяется класс Parser, который наследуется от CstParser, что позволяет работать с синтаксическими деревьями. В конструкторе этого класса определяются правила для каждой из частей грамматики. Внутри этих определений использовались следующие методы:

1. This.CONSUME(token) – метод, который «поглощает» токен из входной строки, соответствующий определённому типу лексемы
2. This.SUBRULE(rule) – метод для вызова другого правила грамматики в рамках текущего
3. This.MANY(callback) – метод для указания, что правило может повторяться несколько раз. В данном случае, он используется для того, что

позволить множественные элементы (например, несколько идентификаторов или несколько полей).

4. This.OPTION(callback) – метод для указания необязательности правила. Например, если одно правило может быть либо пустым, либо повторяться.

5. This.OR([…]) – метод для описания альтернативных правил, то есть для выбора одного из нескольких вариантов.

В конце конструктора вызывается метод this.performSelfAnalysis(), который выполняет анализ грамматики на основе определённых правил и проверяет, что все правила корректны и не противоречат друг другу.

Разберём подробнее основной файл SPO4.js. Код в нём представляет собой основную программу для анализа данных из файла. Он использует ранее созданные лексер и парсер для обработки текста, описывающего структуры данных, типы переменных и их размеры, а затем вычисляет объём памяти, необходимый для их хранения.

Сначала идёт импорт и инициализация. То есть, подключается модуль fs для работы с файловой системой, а затем импортируются LEXER и PARSER. Затем идёт глобальная основная функция main, содержимое которой можно разделить на несколько этапов:

1. Чтение входного файла. Если возникает ошибка, программа выводит соответствующее сообщение об ошибке и завершает работу.

2. Вывод содержимого файла.

3. Лексический и синтаксический анализ. Лексический анализ преобразует текст из файла в список токенов с помощью лексера. Синтаксический анализ строит синтаксическое дерево с помощью парсера, используя токены в качестве входных данных. Если при синтаксическом анализе возникают ошибки, они выводятся в консоль.

4. Вычисление размеров типов. Из синтаксического дерева извлекаются описания типов из правила. Для каждого типа определяется его размер.

5. Расчёт размеров с учётом кратности. Для каждого типа рассчитывается общий размер с учётом кратности.

6. Расчёт размеров переменных. Переменные извлекаются из правила и для каждой переменной определяется её тип, и с учётом размеров типов вычисляется объём памяти, необходимый для хранения переменной.

7. Вывод результатов. Здесь выводятся размеры всех типов данных и переменных с учётом и без учёта кратности.

Пример выполнения расчёта объёма памяти для простейшей входной программы

Рассмотрим пример, продемонстрированный на рисунке 1.

```
type
  EmployeeRecord = record
    id, age: byte;
    salary: real;
  end;

  myDepartment =  record
    employeeId, employeeAge: real;
    supervisor: EmployeeRecord;
  end;

  PositionType = byte;

var
  john, emily: EmployeeRecord;
  managerPosition: PositionType;
  salaryManager, hoursWorked: byte;
  alice, bob: myDepartment;
```

Рисунок 1 - Пример

Данный пример содержит в себе несколько типов данных, включая записи (структуры) и примитивные типы. Эти типы затем используются для создания переменных, каждая из которых имеет определённый размер в памяти. Попробуем провести все расчёты вручную.

Первым делом можно увидеть, что EmployeeRecord представляет собой структуру record. Это значит, что её размер сразу не известен и его придётся высчитывать путём суммирования объёмов каждого из полей. Заметим, что внутри этой структуры видно три поля: два из них имеет тип byte и одно – real. Из входных данных мы видим, что первый тип весит 1 байт, а второй – 6. Но

при этом не забываем, что именно два поля весят по 1 байту и одно – 6 байт. То есть итого получаем размерность, равную 8 байтам без учёта кратности.

Ниже представлена ещё одна структура record, состоящая также из 3 полей. Два из них имеют тип real, а последнее поле ссылается на структуру, рассмотренную ранее. То есть мы уже можем сказать, что последнее поле будет весить 8 байт. Первые же два будут весить всего 12 байт. И итого получаем, что вторая структура весит 20 байт без учёта кратности.

Далее идёт строка PositionType = byte. Это означает, что тип PositionType будет весить 1 байт. Затем идёт блок с переменными. Видим, что переменные john, emily имеют тип структуры EmployeeRecord, то есть эти переменные будут весить по 8 байт. Затем идёт переменная managerPosition, которая будет весить 1 байт. Далее - salaryManager, hoursWorked, которые будут весить по 1 байту. Alice, bob будут весить по 20 байт каждая.

Значит, итого имеем:

$$8 \text{ байт} + 8 \text{ байт} + 1 \text{ байт} + 1 \text{ байт} + 1 \text{ байт} + 20 \text{ байт} + 20 \text{ байт} = \\ = 59 \text{ байт (без учёта кратности)}$$

Попробуем провести расчёты теперь уже с учётом кратности. Так как кратность равна 2, то тип byte будет уже по факту занимать не 1 байт, а ближайшее число, которое кратно двум – то есть 2 байта. Итого получаем, что структура EmployeeRecord будет весить:

$$2 \text{ байта}(id) + 2 \text{ байта } (age) + 6 \text{ байт } (salary) = 10 \text{ байт}$$

А структура myDepartment будет весить:

$$6 \text{ байта}(idemployeeId) + 6 \text{ байта } (employeeAge) + 10 \text{ байт } (supervisor) \\ = 22 \text{ байта}$$

Тип PositionType будет весить 2 байта. А значит, что переменные будут весить:

john, emily – по 10 байт каждая

managerPosition – 2 байта

salaryManager, hoursWorked – по 2 байта каждая

alice, bob – по 22 байта каждая

А значит итого получаем объём:

$$10 \text{ байт} + 10 \text{ байт} + 2 \text{ байта} + 2 \text{ байта} + 2 \text{ байта} + 22 \text{ байта} + 22 \text{ байта} =$$
$$= 70 \text{ байт (с учётом кратности)}$$

Текст программы

Ниже продемонстрированы листинги программы, разделённой на три файла: LEXER.js, PARSER.js и SPO4.js.

Вспомогательный файл LEXER.js

```
import { Lexer, createToken } from "chevrotain"; // Импортируем необходимые
компоненты из библиотеки chevrotain.

const SpaceBar = createToken({ // Создаем токен для пробельных символов.
  name: "Space", // Название токена.
  pattern: /\s+/, // Регулярное выражение для пробельных символов.
  group: Lexer.SKIPPED, // Указываем, что этот токен будет пропускаться лексером.
});

export const TypeToken = createToken({ // Создаем токен для "type".
  name: "TypeToken", // Название токена.
  pattern: /type/ // Регулярное выражение для совпадения со словом "type".
});
export const VarToken = createToken({ // Создаем токен для "var".
  name: "VariableToken", // Название токена.
  pattern: /var/ // Регулярное выражение для совпадения со словом "var".
});
export const ByteToken = createToken({ // Создаем токен для "byte".
  name: "ByteToken", // Название токена.
  pattern: /byte/ // Регулярное выражение для совпадения со словом "byte".
});
export const RealToken = createToken({ // Создаем токен для "real".
  name: "RealToken", // Название токена.
  pattern: /real/ // Регулярное выражение для совпадения со словом "real".
});
export const RecordToken = createToken({ // Создаем токен для "record".
  name: "RecordToken", // Название токена.
  pattern: /record/ // Регулярное выражение для совпадения со словом "record".
});
export const EndToken = createToken({ // Создаем токен для "end".
  name: "EndToken", // Название токена.
  pattern: /end/ // Регулярное выражение для совпадения со словом "end".
});
export const IdentifierToken = createToken({ // Создаем токен для
идентификаторов.
  name: "IdentifierToken", // Название токена.
  pattern: /[a-zA-Z_]\w*/ // Регулярное выражение.
});
export const SemicolonToken = createToken({ // Создаем токен для символа ";".
```

```
  name: "SemicolonToken", // Название токена.
  pattern: /;/ // Регулярное выражение.
});
export const ColonToken = createToken({ // Создаем токен для символа ":".
  name: "ColonToken", // Название токена.
  pattern: /:/ // Регулярное выражение.
});
export const CommaToken = createToken({ // Создаем токен для символа ",".
  name: "CommaToken", // Название токена.
  pattern: /,/ // Регулярное выражение для символа ",".
});
export const EqualsToken = createToken({ // Создаем токен для символа "=".
  name: "EqualsToken", // Название токена.
  pattern: /=/ // Регулярное выражение.
});

export const Tokens = [ // Создаем массив токенов, упорядоченных по приоритету.
  SpaceBar, TypeToken, VarToken, ByteToken, // Сначала пробельные символы и
ключевые слова.
  RealToken, RecordToken, EndToken, SemicolonToken, // Другие ключевые слова и
символы.
  ColonToken, CommaToken, EqualsToken, IdentifierToken, // Символы и
идентификаторы.
];

const LEXER = new Lexer(Tokens); // Создаем лексер, используя массив токенов.
export default LEXER; // Экспортируем лексер для использования в других модулях.
```

Вспомогательный файл PARSER.js

```
import { CstParser } from "chevrotain"; // Импортируем класс CstParser из
библиотеки chevrotain.
import { // Импортируем токены из файла LEXER.js.
  Tokens, TypeToken, VarToken, SemicolonToken,
  IdentifierToken, EqualsToken, CommaToken, ColonToken,
  ByteToken, RealToken, RecordToken, EndToken
} from "./LEXER.js";

class Parser extends CstParser { // Создаем класс парсера, наследующего от
CstParser.
  constructor() { // Конструктор класса.
    super(Tokens); // Инициализируем CstParser с массивом токенов.

    // S -> type L var R
    this.RULE("S", () => { // Определяем правило "S".
      this.CONSUME(TypeToken); // Сопоставляем токен "type".
      this.SUBRULE(this.L); // Вызов правила "L".
      this.CONSUME(VarToken); // Сопоставляем токен "var".
      this.SUBRULE(this.R); // Вызов правила "R".
```

```
    });

    // L -> T; | T;L
    this.RULE("L", () => { // Определяем правило "L".
      this.MANY(() => { // Правило "L" может содержать несколько элементов "T;".
        this.SUBRULE(this.T); // Вызов правила "T".
        this.CONSUME(SemicolonToken); // Сопоставляем токен ";".
      });
    });

    // T -> t=c | t=D
    this.RULE("T", () => { // Определяем правило "T".
      this.CONSUME(IdentifierToken); // Сопоставляем идентификатор.
      this.CONSUME(EqualsToken); // Сопоставляем токен "=".
      this.OR([ // Правило "T" имеет несколько альтернатив.
        { ALT: () => this.CONSUME(ByteToken) }, // Альтернатива: "byte".
        { ALT: () => this.CONSUME(RealToken) }, // Альтернатива: "real".
        { ALT: () => this.SUBRULE(this.D) } // Альтернатива: правило "D".
      ]);
    });

    // R -> V; | V;R
    this.RULE("R", () => { // Определяем правило "R".
      this.MANY(() => { // Правило "R" может содержать несколько элементов "V;".
        this.SUBRULE(this.V); // Вызов правила "V".
        this.CONSUME(SemicolonToken); // Сопоставляем токен ";".
      });
    });

    // V -> K:t | K:c | K:D
    this.RULE("V", () => { // Определяем правило "V".
      this.SUBRULE(this.K); // Вызов правила "K".
      this.CONSUME(ColonToken); // Сопоставляем токен ":".
      this.OR([ // Правило "V" имеет несколько альтернатив.
        { ALT: () => this.CONSUME(ByteToken) }, // Альтернатива: "byte".
        { ALT: () => this.CONSUME(RealToken) }, // Альтернатива: "real".
        { ALT: () => this.CONSUME(IdentifierToken) }, // Альтернатива:
идентификатор.
        { ALT: () => this.SUBRULE(this.D) } // Альтернатива: правило "D".
      ]);
    });

    // K -> a | K,a
    this.RULE("K", () => { // Определяем правило "K".
      this.OPTION(() => { // Необязательная часть "K,a".
        this.MANY(() => { // Повторяющийся список идентификаторов, разделенных
запятыми.
          this.CONSUME(IdentifierToken); // Сопоставляем идентификатор.
          this.CONSUME(CommaToken); // Сопоставляем токен ",".
        });
      });
```

```
      this.CONSUME2(IdentifierToken); // Завершающий идентификатор
(обязательный).
    });

    // D -> record F end
    this.RULE("D", () => { // Определяем правило "D".
      this.CONSUME(RecordToken); // Сопоставляем токен "record".
      this.SUBRULE(this.F); // Вызов правила "F".
      this.CONSUME(EndToken); // Сопоставляем токен "end".
    });

    // F -> E; | E;F
    this.RULE("F", () => { // Определяем правило "F".
      this.MANY(() => { // Правило "F" может содержать несколько элементов "E;".
        this.SUBRULE(this.E); // Вызов правила "E".
        this.CONSUME(SemicolonToken); // Сопоставляем токен ";".
      });
    });

    // E -> K:c | K:t
    this.RULE("E", () => { // Определяем правило "E".
      this.SUBRULE(this.K); // Вызов правила "K".
      this.CONSUME(ColonToken); // Сопоставляем токен ":".
      this.OR([ // Правило "E" имеет несколько альтернатив.
        { ALT: () => this.CONSUME(ByteToken) }, // Альтернатива: "byte".
        { ALT: () => this.CONSUME(RealToken) }, // Альтернатива: "real".
        { ALT: () => this.CONSUME(IdentifierToken) }, // Альтернатива:
идентификатор.
        { ALT: () => this.SUBRULE(this.D) } // Альтернатива: правило "D".
      ]);
    });

    this.performSelfAnalysis(); // Завершаем анализ правил и создаем необходимые
структуры.
  }
}

const PARSER = new Parser(); // Создаем экземпляр парсера.
export default PARSER; // Экспортируем парсер для использования в основном файле.
```

Основной файл SPO4.js

```
import fs from 'fs'; // Импортируем модуль для работы с файловой системой.
import LEXER from './LEXER.js'; // Импортируем модуль с лексером.
import PARSER from './PARSER.js'; // Импортируем модуль с парсером.

async function main() { // Главная асинхронная функция.
  const inputText = await new Promise((resolve, reject) => { // Промис для
асинхронного чтения файла.
```

```
    fs.readFile(process.argv.slice(2)[0], 'utf8', (err, data) => { // Читаем файл
из первого аргумента командной строки.
      if (err) { // Если возникает ошибка,
        console.error('\x1b[31mК сожалению, возникла ошибка при чтении исходного
файла:\x1b[0m', err); // то выводим сообщение об ошибке.
        reject(err); // Отклоняем промис.
        return; // Завершаем выполнение.
      }
      resolve(data); // Передаем содержимое файла.
    });
  });

  console.log('\x1b[35m--------------------------------СОДЕРЖИМОЕ ПРИМЕРА-------
--------------------------\x1b[0m'); // Заголовок вывода содержимого примера.
  console.log(inputText); // Выводим содержимое файла.
  console.log('\x1b[35m-----------------------------------------------------------
--------------------------\x1b[0m'); // Разделитель для более понятного
восприятия.

  const tokensList = LEXER.tokenize(inputText); // Токенизируем входной текст с
помощью лексера.
  PARSER.input = tokensList.tokens; // Передаем токены в парсер.
  const parsingTree = PARSER.S(); // Строим дерево разбора, начиная с правила
"S".

  if (PARSER.errors.length > 0) { // Если есть ошибки парсинга,
      console.error('\x1b[31mК сожалению, возникла ошибка при совершении
парсинга:\x1b[0m', PARSER.errors); // то выводим соответствующие ошибки.
  }

  const typeDefiitionsMas = parsingTree.children.L[0].children.T; // Извлекаем
определения типов.
  const typesMas = {}; // Инициализируем объект для хранения типов.

  typeDefiitionsMas.forEach(typeDef => { // Обрабатываем каждое определение типа.
    if (typeDef.children.ByteToken) { // Если тип - "byte",
      typesMas[typeDef.children.IdentifierToken[0].image] = ['byte']; // то
записываем тип "byte".
    }
    if (typeDef.children.RealToken) { // Если тип - "real",
      typesMas[typeDef.children.IdentifierToken[0].image] = ['real']; // то
записываем тип "real".
    }
    if (typeDef.children.D) { // Если тип - это запись (record),
      const recordFields  = typeDef.children.D[0].children.F[0].children.E; // то
извлекаем поля записи.
      const sizesMas = []; // Инициализируем массив для хранения размеров полей.

      recordFields.forEach(field => { // Обрабатываем каждое поле записи.
        if (field.children.ByteToken) { // Если поле - "byte",
          sizesMas.push(
```

```
            ...Array(field.children.K[0].children.IdentifierToken.length).fill('b
yte') // то добавляем "byte" для каждого идентификатора.
          );
        }
        if (field.children.RealToken) { // Если поле - "real",
          sizesMas.push(
            ...Array(field.children.K[0].children.IdentifierToken.length).fill('r
eal') // то добавляем "real" для каждого идентификатора.
          );
        }
        if (field.children.IdentifierToken) { // Если поле ссылается на другой
тип,
          sizesMas.push(
            ...Array(field.children.K[0].children.IdentifierToken.length).fill(
              field.children.IdentifierToken[0].image // то добавляем ссылочный
тип.
            )
          );
        }
      });
      typesMas[typeDef.children.IdentifierToken[0].image] = sizesMas; //
Сохраняем информацию о полях записи.
    }
  });

  const TypeSizesMult = { // Инициализируем объект для хранения размеров типов с
кратностью.
    "byte": 2, // Размер "byte" с кратностью.
    "real": 6, // Размер "real" с кратностью.
  };

  Object.keys(typesMas).forEach(it => { // Для каждого пользовательского типа
    TypeSizesMult[it] = typesMas[it].reduce((acc, rec) => { // вычисляем размер
типа
      const currentSize = TypeSizesMult[rec] || 0; // и получаем размер текущего
типа.
      return acc + currentSize; // Суммируем размеры.
    }, 0);
  });

  console.log('\x1b[35m--------------------ПОЛУЧЕННЫЙ РАЗМЕР ТИПОВ ДАННЫХ(С
КРАТНОСТЬЮ)--------------------\x1b[0m'); // Заголовок вывода размеров типов.
  console.log(
    Object.keys(TypeSizesMult) // Для каждого типа
      .map(
        typeName =>
          `${typeName}:  \x1b[36m${TypeSizesMult[typeName]} (байт)\x1b[0m` //
форматируем и выводим размер.
      )
      .join('\n') // Объединяем строки.
  );
```

```
  console.log('\x1b[35m------------------------------------------------------------
-------------------------\x1b[0m'); // Разделитель для удобства восприятия.

  const varDefMul = parsingTree.children.R[0].children.V; // Извлекаем
определения переменных.
  const varSizesMul = {}; // Инициализируем объект для размеров переменных.

  varDefMul.forEach(def => { // Обрабатываем каждое определение переменной.
    let dataType = "byte"; // По умолчанию тип "byte".
    if (def.children.IdentifierToken) { // Если задан пользовательский тип,
      dataType = def.children.IdentifierToken[0].image; // то присваиваем этот
тип.
    }
    if (def.children.RealToken) { // Если тип "real",
      dataType = 'real'; // то присваиваем "real".
    }
    const variableSize = TypeSizesMult[dataType]; // Получаем размер переменной.
    const variables = def.children.K[0].children.IdentifierToken; // Получаем
список имен переменных.

    variables.forEach(variable => { // Для каждой переменной
      varSizesMul[variable.image] = variableSize; // сохраняем ее размер.
    });
  });

  console.log('\x1b[35m--------------------ПОЛУЧЕННЫЙ РАЗМЕР ПЕРЕМЕННЫХ(С
КРАТНОСТЬЮ)--------------------\x1b[0m'); // Заголовок вывода.
  console.log(
    Object.keys(varSizesMul) // Для каждой переменной
      .map(
        varName =>
          `${varName}:\x1b[32m${varSizesMul[varName]} байт\x1b[0m` // форматируем
и выводим размер.
      )
      .join('\n') // Объединяем строки.
  );
  console.log('\x1b[35m------------------------------------------------------------
-------------------------\x1b[0m'); // Разделитель для удобства восприятия.

  const memoryMul = Object.values(varSizesMul).reduce( // Вычисляем общий объем
памяти.
    (total, size) => total + size, // Суммируем размеры переменных.
    0
  );

  console.log(
    '\n\x1b[35mИТОГО ОБЪЁМ ПАМЯТИ СОСТАВЛЯЕТ:\x1b[0m' + // Заголовок вывода
итогового объема памяти.
      `\x1b[33m${memoryMul} байт\x1b[0m` // Выводим объем памяти.
  );
```

```
  console.log('\x1b[35m------------------------------------------------------------
-------------------------\x1b[0m'); // Разделитель для удобства восприятия.

  const TypeSizes = { // Инициализируем объект размеров типов без кратности.
    "byte": 1, // Размер "byte".
    "real": 6, // Размер "real".
  };

  Object.keys(typesMas).forEach(it => { // Для каждого пользовательского типа
    TypeSizes[it] = typesMas[it].reduce((acc, rec) => { // Вычисляем размер типа,
      const currentSize = TypeSizes[rec] || 0; // получаем размер текущего типа,
      return acc + currentSize; // и суммируем размеры.
    }, 0);
  });

  console.log('\x1b[35m--------------------ПОЛУЧЕННЫЙ РАЗМЕР ТИПОВ ДАННЫХ(БЕЗ
КРАТНОСТИ)-------------------\x1b[0m'); // Заголовок вывода размеров.
  console.log(
    Object.keys(TypeSizes) // Для каждого типа
      .map(
        type =>
          `${type}:\x1b[36m${TypeSizes[type]} байт\x1b[0m` // форматируем и
выводим размер.
      )
      .join('\n') // Объединяем строки.
  );
  console.log('\x1b[35m------------------------------------------------------------
-------------------------\x1b[0m'); // Разделитель для удобства восприятия.

  const varDef = parsingTree.children.R[0].children.V; // Извлекаем определения
переменных.
  const varSizes = {}; // Инициализируем объект для размеров переменных.

  varDef.forEach(def => { // Обрабатываем каждое определение переменной.
    let type = "byte"; // По умолчанию тип "byte".
    if (def.children.IdentifierToken) { // Если задан пользовательский тип,
      type = def.children.IdentifierToken[0].image; // то присваиваем этот тип.
    }
    if (def.children.RealToken) { // Если тип "real",
      type = 'real'; // то присваиваем "real".
    }
    const varSize_1 = TypeSizes[type]; // Получаем размер переменной.
    const var_s = def.children.K[0].children.IdentifierToken; // Получаем список
имен переменных.

    var_s.forEach(variable => { // Для каждой переменной
      varSizes[variable.image] = varSize_1; // сохраняем ее размер.
    });
  });
```

```
  console.log('\x1b[35m--------------------ПОЛУЧЕННЫЙ РАЗМЕР ПЕРЕМЕННЫХ(БЕЗ
КРАТНОСТИ)-------------------\x1b[0m'); // Заголовок вывода.
  console.log(
    Object.keys(varSizes) // Для каждой переменной
      .map(
        varName =>
          `${varName}:\x1b[32m${varSizes[varName]} байт\x1b[0m` // форматируем и
выводим размер.
      )
      .join('\n') // Объединяем строки.
  );
  console.log('\x1b[35m-------------------------------------------------------------
-------------------------\x1b[0m'); // Разделитель для удобства восприятия.

  const memory = Object.values(varSizes).reduce( // Вычисляем общий объем памяти.
    (total, size) => total + size, // Суммируем размеры переменных.
    0
  );

  console.log(
    '\n\x1b[35mИТОГО ОБЪЁМ ПАМЯТИ СОСТАВЛЯЕТ:\x1b[0m' + // Заголовок вывода
итогового объема памяти.
      `\x1b[33m${memory} байт\x1b[0m` // Выводим объем памяти.
  );
  console.log('\x1b[35m-------------------------------------------------------------
-------------------------\x1b[0m'); // Разделитель для удобства восприятия.
}

// Запуск основной функции
main();
```

Результат выполнения программы

В качестве примера в текстовый файл был добавлен тот же пример, который и был рассмотрен ранее при ручном вычислении памяти. Искомые результаты продемонстрированы на рисунках 2 – 3.

```
PS C:\Users\kater\OneDrive\Рабочий стол\Системное программное обеспечение\laba3> node SP04.js тест.txt
---------------------------------СОДЕРЖИМОЕ ПРИМЕРА---------------------------------
type
  EmployeeRecord = record
    id, age: byte;
    salary: real;
  end;

  myDepartment =  record
    employeeId, employeeAge: real;
    supervisor: EmployeeRecord;
  end;

  PositionType = byte;

var
  john, emily: EmployeeRecord;
  managerPosition: PositionType;
  salaryManager, hoursWorked: byte;
  alice, bob: myDepartment;

-------------------------------------------------------------------------------------
--------------------ПОЛУЧЕННЫЙ РАЗМЕР ТИПОВ ДАННЫХ(С КРАТНОСТЬЮ)--------------------
byte:  2 (байт)
real:  6 (байт)
EmployeeRecord:  10 (байт)
myDepartment:  22 (байт)
PositionType:  2 (байт)
-------------------------------------------------------------------------------------
--------------------ПОЛУЧЕННЫЙ РАЗМЕР ПЕРЕМЕННЫХ(С КРАТНОСТЬЮ)--------------------
john:10 байт
emily:10 байт
managerPosition:2 байт
salaryManager:2 байт
hoursWorked:2 байт
alice:22 байт
bob:22 байт
-------------------------------------------------------------------------------------

ИТОГО ОБЪЁМ ПАМЯТИ СОСТАВЛЯЕТ:70 байт
-------------------------------------------------------------------------------------
```

Рисунок 2 – Результат выполнения программы (начало)

```
--------------------ПОЛУЧЕННЫЙ РАЗМЕР ТИПОВ ДАННЫХ(БЕЗ КРАТНОСТИ)--------------------
byte:1 байт
real:6 байт
EmployeeRecord:8 байт
myDepartment:20 байт
PositionType:1 байт
-------------------------------------------------------------------------------------
--------------------ПОЛУЧЕННЫЙ РАЗМЕР ПЕРЕМЕННЫХ(БЕЗ КРАТНОСТИ)--------------------
john:8 байт
emily:8 байт
managerPosition:1 байт
salaryManager:1 байт
hoursWorked:1 байт
alice:20 байт
bob:20 байт
-------------------------------------------------------------------------------------

ИТОГО ОБЪЁМ ПАМЯТИ СОСТАВЛЯЕТ:59 байт
-------------------------------------------------------------------------------------
```

Рисунок 3 – Результат выполнения программы (окончание)

Выводы

В данной лабораторной работе был построен распознаватель исходного текста программы, содержащего описания типов данных, структур данных и переменных.

Также изучены основные принципы распределения памяти, прошло удачное ознакомление с алгоритмами расчёта объёма памяти, занимаемой простыми и составными структурами данных, получены практические навыки создания простейшего анализатора для расчёта объёма памяти, занимаемого заданной структурой данных.