Практические задания по дисциплине «Системное программное обеспечение»

Общие требования к программной реализации работ

Для реализации необходимо использовать язык программирования Си, если в описании к лабораторной работе не сказано иное.

Исходный код лабораторных работ размещать в репозитории Gitlab факультета ПИиКТ.

Также необходимо следовать данным пунктам:

1. **«Нет» статичности.** Все структуры данных должны допускать создание множества их экземпляров.
2. **«Нет» «магическим» константам.** Все значения должны либо вычисляться из обрабатываемых программой данных, либо задаваться с помощью аргументов командной строки или конфигурационных файлов.
3. **«Нет» бесконечным циклам.** Все циклы должны иметь понятные условия выхода: не допускается использовать, например, while (true), for (; ;) и т.д.
4. **«Нет» утечке ресурсов.** Все ресурсы, которые были использованы в программе и требуют освобождения (закрытия), должны корректно освобождаться (закрываться) независимо от возникновения ошибочных ситуаций или исключений. Например, открытый файл должен быть закрыт после того, как он перестал использоваться в программе; аллоцированная вручную память обязательно должна освобождаться.
5. **«Нет» неожиданным завершениям программы.** Все процессы, нити (threads) должны корректно завершаться в результате выполнения работы, а не прерываться функциями вида Abort/Exit.
6. **«Нет» побайтовому вводу-выводу.** Все данные должны обрабатываться частями (блоками) известного размера, с учетом целесообразного размера буфера.

Настоятельно рекомендуется:

- В начале работы подготовить окружение разработчика, включающее отладчик, поддерживающий визуализацию структур данных и возможность отладки программ, выполняющихся под управлением ОС семейств Windows и *NIX.
- Сборку проекта осуществлять с помощью кроссплатформенных средств автоматизации, таких как мэйкфайлы.
- Следовать общим принципам грамотной разработки ПО, таким как SOLID, DRY, и др., грамотно использовать непрозрачные типы данных (opaque data types), разделять публичную и приватную функциональность модулей.

Оформление отчетов

По каждому из заданий должен быть представлен отчет, содержащий следующие части:

1. Титульный лист установленной формы, включающей следующие сведения: отчёт к практическому заданию №X по дисциплине У студента Z, группа, наименование факультета и вуза, год обучения.
2. Цели – описание цели задания (см. текст задания)
3. Задачи – путь достижения цели, что именно нужно было сделать для выполнения задания (план хода вашей работы)
4. Описание работы – внешнее описание созданной программы, состава модулей, способов её использования, примеры входной и выходной информации (модули, интерфейсы, тесты)
5. Аспекты реализации – внутреннее описание созданной программы, особенности алгоритмов, примеры кода
6. Результаты – что было сделано для выполнения задач кратко по пунктам (созданные артефакты, результаты тестов, количественные оценки)
7. Выводы – что было достигнуто в отношении цели задания. (что показали тесты, почему, как это было достигнуто, чему научились, качественные оценки)

Описание заданий

Задание 1

Использовать средство синтаксического анализа по выбору, реализовать модуль для разбора текста в соответствии с языком по варианту. Реализовать построение по исходному файлу с текстом синтаксического дерева с узлами, соответствующими элементам синтаксической модели языка. Вывести полученное дерево в файл в формате, поддерживающем просмотр графического представления.

Порядок выполнения:

1. Изучить выбранное средство синтаксического анализа
    a. Средство должно поддерживать программный интерфейс, совместимый с языком Си
    b. Средство должно параметризоваться спецификацией, описывающей синтаксическую структуру разбираемого языка
    c. Средство может функционировать посредством кодогенерации и/или подключения необходимых для его работы дополнительных библиотек
    d. Средство может быть реализовано с нуля, **в этом случае оно должно использовать обобщённый алгоритм, управляемый спецификацией**
2. Изучить синтаксис разбираемого по варианту языка и записать спецификацию для средства синтаксического анализа, включающую следующие конструкции:
    a. Подпрограммы со списком аргументов и возвращаемым значением
    b. Операции контроля потока управления – простые ветвления if-else и циклы или аналоги
    c. В зависимости от варианта – определения переменных
    d. Целочисленные, строковые и односимвольные литералы
    e. Выражения численной, битовой и логической арифметики
    f. Выражения над одномерными массивами
    g. Выражения вызова функции
3. Реализовать модуль, использующий средство синтаксического анализа для разбора языка по варианту
    a. Программный интерфейс модуля должен принимать строку с текстом и возвращать структуру, описывающую соответствующее дерево разбора и коллекцию сообщений ошибке
    b. Результат работы модуля – дерево разбора – должно содержать иерархическое представление для всех синтаксических конструкций, включая выражения, логически представляющие собой иерархически организованные данные, даже если на уровне средства синтаксического анализа для их разбора было использовано линейное представление
4. Реализовать тестовую программу для демонстрации работоспособности созданного модуля
    a. Через аргументы командной строки программа должна принимать имя входного файла для чтения и анализа, имя выходного файла записи для дерева, описывающего синтаксическую структуру разобранного текста
    b. Сообщения об ошибке должны выводиться тестовой программной (не модулем, отвечающим за анализ!) в стандартный поток вывода ошибок
5. Результаты тестирования представить в виде отчета, в который включить:
    a. В части 3 привести описание структур данных, представляющих результат разбора текста (3а)
    b. В части 4 описать, какая дополнительная обработка потребовалась для результата разбора, предоставляемого средством синтаксического анализа, чтобы сформировать результат работы созданного модуля
    c. В части 5 привести примеры исходных анализируемых текстов для всех синтаксических конструкций разбираемого языка и соответствующие результаты разбора

Задание 2

Реализовать построение графа потока управления посредством анализа дерева разбора для набора входных файлов. Выполнить анализ собранной информации и сформировать набор файлов с графическим представлением для результатов анализа.

Порядок выполнения:

1. Описать структуры данных, необходимые для представления информации о наборе файлов, наборе подпрограмм и графе потока управления, где:
    a. Для каждой подпрограммы: имя и информация о сигнатуре, граф потока управления, имя исходного файла с текстом подпрограммы.
    b. Для каждого узла в графе потока управления, представляющего собой базовый блок алгоритма подпрограммы: целевые узлы для безусловного и условного перехода (по мере необходимости), дерево операций, ассоциированных с данным местом в алгоритме, представленном в исходном тексте подпрограммы
2. Реализовать модуль, формирующий граф потока управления на основе синтаксической структуры текста подпрограмм для входных файлов
    a. Программный интерфейс модуля принимает на вход коллекцию, описывающую набор анализируемых файлов, для каждого файла – имя и соответствующее дерево разбора в виде структуры данных, являющейся результатом работы модуля, созданного по заданию 1 (п. 3.b).
    b. Результатом работы модуля является структура данных, разработанная в п. 1, содержащая информацию о проанализированных подпрограммах и коллекция с информацией об ошибках
    c. Посредством обхода дерева разбора подпрограммы, сформировать для неё граф потока управления, порождая его узлы и формируя между ними дуги в зависимости от синтаксической конструкции, представленной данным узлом дерева разбора: выражение, ветвление, цикл, прерывание цикла, выход из подпрограммы – для всех синтаксических конструкций по варианту (п. 2.b)
    d. С каждым узлом графа потока управления связать дерево операций, в котором каждая операция в составе текста программы представлена как совокупность вида операции и соответствующих операндов (см задание 1, пп. 2.d-g)
    e. При возникновении логической ошибки в синтаксической структуре при обходе дерева разбора, сохранить в коллекции информацию об ошибке и её положении в исходном тексте
3. Реализовать тестовую программу для демонстрации работоспособности созданного модуля
    a. Через аргументы командной строки программа должна принимать набор имён входных файлов, имя выходной директории
    b. Использовать модуль, разработанный в задании 1 для синтаксического анализа каждого входного файла и формирования набора деревьев разбора
    c. Использовать модуль, разработанный в п. 2 для формирования графов потока управления каждой подпрограммы, выявленной в синтаксической структуре текстов, содержащихся во входных файлах
    d. Для каждой обнаруженной подпрограммы вывести представление графа потока управления в отдельный файл с именем “sourceName.functionName.ext” в выходной директории, по-умолчанию размещать выходной файлы в той же директории, что соответствующий входной
    e. Для деревьев операций в графах потока управления всей совокупности подпрограмм сформировать граф вызовов, описывающий отношения между ними в плане обращения их друг к другу по именам и вывести его представление в дополнительный файл, по-умолчанию размещаемый рядом с файлом, содержащим подпрограмму main.
    f. Сообщения об ошибке должны выводиться тестовой программной (не модулем, отвечающим за анализ!) в стандартный поток вывода ошибок
4. Результаты тестирования представить в виде отчета, в который включить:
    a. В части 3 привести описание разработанных структур данных
    b. В части 4 описать программный интерфейс и особенности реализации разработанного модуля
    c. В части 5 привести примеры исходных анализируемых текстов для всех синтаксических конструкций разбираемого языка и соответствующие результаты разбора

Задание 3

Реализовать формирование линейного кода в терминах некоторого набора инструкций посредством анализа графа потока управления для набора подпрограмм. Полученный линейный код вывести в мнемонической форме в выходной текстовый файл.

Подготовка к выполнению по одному из двух сценариев:

1. Составить описание виртуальной машины с набором инструкций и моделью памяти по варианту
    a. Изучить нотацию для записи определений целевых архитектур
    b. Составить описание ВМ в соответствии с вариантом
        i. Описание набор регистров и банков памяти
        ii. Описать набор инструкций: для каждой инструкции задать структуру операционного кода, содержащего описание операндов и набор операций, изменяющих состояние ВМ
            1. Описать инструкции перемещения данных и загрузки констант
            2. Описать инструкции арифметических и логических операций
            3. Описать инструкции условной и безусловной передачи управления
            4. Описать инструкции ввода-вывода с использованием скрытого регистра в качестве порта ввода-вывода
        iii. Описать набор мнемоник, соответствующих инструкциям ВМ
    c. Подготовить скрипт для запуска ассемблированного листинга с использованием описания ВМ:
        i. Написать тестовый листинг с использованием подготовленных мнемоник инструкций
        ii. Задействовать транслятор листинга в бинарный модуль по описанию ВМ
        iii. Запустить полученный бинарный модуль на исполнение и получить результат работы
        iv. Убедиться в корректности функционирования всех инструкций ВМ
2. Выбрать и изучить прикладную архитектуру системы команд существующей ВМ
    a. Для выбранной ВМ:
        i. Должен существовать готовый эмулятор (например qemu)
        ii. Должен существовать готовый тулчейн (набор инструментов разработчика): компилятор Си, ассемблер и дизассемблер, линковщик, желательно отладчик
    b. Согласовать выбор ВМ с преподавателем
    c. Изучить модель памяти и набор инструкций ВМ
    d. Научиться использовать тулчейн (собирать и запускать программы из листинга)
    e. Подготовить скрипт для запуска ассемблированного листинга с использованием эмулятора
        i. Написать тестовый листинг с использованием инструкций ВМ
        ii. Задействовать ассемблер и компоновщик из тулчейна
        iii. Запустить бинарный модуль на исполнение и получить результат его работы

Порядок выполнения:

1. Описать структуры данных, необходимые для представления информации об элементах образа программы (последовательностях инструкций и данных), расположенных в памяти
    a. Для каждой инструкции – имя мнемоники и набор операндов в терминах данной ВМ
    b. Для элемента данных – соответствующее литеральное значение или размер экземпляра типа данных в байтах
2. Реализовать модуль, формирующий образ программы в линейном коде для данного набора подпрограмм
    a. Программный интерфейс модуля принимает на вход структуру данных, содержащую графы потока управления и информацию о локальных переменных и сигнатурах для набора подпрограмм, разработанную в задании 2 (п. 1.a, п. 2.b)
    b. В результате работы порождается структура данных, разработанная в п. 1, содержащая описание образа программы в памяти: набор именованных элементов данных и набор именованных фрагментов линейного кода, представляющих собой алгоритмы подпрограмм

    c. Для каждой подпрограммы посредством обхода узлов графа потока управления в порядке топологической сортировки (начиная с узла, являющегося первым базовым блоком алгоритма подпрограммы), сформировать набор именованных групп инструкций, включая пролог и эпилог подпрограммы (формирующие и разрушающие локальное состояние подпрограммы)
    d. Для каждого базового блока в составе графа потока управления сформировать группу инструкций, соответствующих операциям в составе дерева операций
    e. Использовать имена групп инструкций для формирования инструкций перехода между блоками инструкций, соответствующих узлам графа потока управления в соответствии с дугами в нём
3. Доработать тестовую программу, разработанную в задании 2 для демонстрации работоспособности созданного модуля
    a. Добавить поддержку аргумента командной строки для имени выходного файла, вывод информации о графах потока управления сделать опциональных
    b. Использовать модуль, разработанный в п. 2 для формирования образа программы на основе информации, собранной в результате работы модуля, созданного в задании 2 (п. 2.b)
    c. Для сформированного образа программы в линейном коде вывести в выходной файл ассемблерный листинг, содержащий мнемоническое представление инструкций и данных, как они описаны в структурах данных (п. 1), построенных разработанным модулем (пп. 2.c-e)
    d. Проверить корректность решения посредством сборки сгенерированного листинга и запуска полученного бинарного модуля на эмуляторе ВМ (см. подготовка п. 1.c или п. 2.e)
4. Результаты тестирования представить в виде отчета, в который включить:
    a. В части 3 привести описание разработанных структур данных
    b. В части 4 описать программный интерфейс и особенности реализации разработанного модуля
    c. В части 5 привести примеры исходных текстов, соответствующие ассемблерные листинги и примера вывода запущенных тестовых программ

Задание 4

Изменить разработанную в третьем задании программу так, чтобы она формировала дополнительную секцию с информацией о соответствии частей модели программы в исходном тексте частям модели программы в целевой машине во время выполнения. Использовать это информацию из бинарного исполняемого модуля посредством существующих программных интерфейсов для инспекции состояния программы.

Дополнить разработанный в предшествующих заданиях программный комплекс поддержкой пользовательских типов данных с возможностью прямого наследования членов и дополнительной функциональностью по варианту.

Подготовка к выполнению по одному из двух сценариев, в зависимости от типа ВМ в задании 3:

1. Для учебной ВМ – Изучить GDB/MI API
    a. Способ организации команд и результатов их выполнения
    b. Команды управления ходом выполнения программы и точками останова, такие как:
       exec-run, exec-continue, exec-next-instruction, gdb-exit,
       break-list, break-insert, break-delete, break-enable, break-disable
    c. Команды инспекции состояния программы, такие как:
       data-list-register-names, data-list-register-values, data-disassemble, data-read-memory-bytes и data-write-memory-bytes (с опцией bank, задающей имя банка памяти для чтения)
2. Для реальной ВМ – изучить функцию системного вызова ptrace
    a. Механизм запуска дочернего процесса под наблюдением управляющего процесса
    b. Способ получения информации о сигналах и изменении состояния наблюдаемого процесса
    c. Способ влияния на ход потока управления в наблюдаемом процессе
       (останова, трассировки, продолжения выполнения)
    d. Способы инспекции состояния наблюдаемого процесса (регистров и памяти)
    e. libopcodes или любую готовую библиотеку по выбору для распознавания инструкций

Порядок выполнения:

1. Реализовать консольное приложение-инспектор в диалоговом командном режиме
    a. Использовать программные интерфейсы ОС для запуска инспектируемого дочернего процесса
    b. Реализовать вводимые пользователем команды с выводом соответствующей информации в человекочитаемо сформатированном виде:
        i. Чтение состояния регистров
        ii. Чтение заданного количества байт и заданного банка памяти по заданному адресу
        iii. Чтение и дизассемблирование заданного количества байт по заданному адресу
        iv. Выполнение одной инструкции
        v. Продолжение выполнения программы
        vi. Постановка и снятие точек останова по заданному адресу инструкции
2. Дополнить разработанную в третьем задании программу для формирования секции с информацией о структуре программы, представленной в формируемом бинарном модуле
    a. Использовать псевдоинструкции определения данных для кодирования нужной информации
    b. С метками в качестве элементов данных описать соответствие строк исходного текста программы (см. задание 1) адресам инструкций в банке оперативной памяти (см. задание 3), для каждой подпрограммы описать диапазон адресов, где представлены её инструкции
    c. Для каждой подпрограммы сформировать информацию, описывающую сигнатуры (имена и списки аргументов с типами), аргументы и локальные переменные (имена с типами и информацию об их расположении в памяти во время выполнения кода подпрограммы)
3. Дополнить консольное приложение-инспектор рядом команд, использующих информацию о структуре инспектируемой программы и реализовать вводимые пользователем команды с выводом соответствующей информации в человекочитаемо сформатированном виде:
    a. Чтение состояния аргументов и локальных переменных текущей выполняемой подпрограммы (см. п.2.c) с отображением элементов массивов

    b. Выполнение последовательности инструкций, представляющих собой одно выражение в исходном тексте подпрограммы (см. п.2.b)
    c. Отображение окрестности текущего положения в тексте подпрограммы в соответствии с адресом следующей выполняемой инструкции (значение регистра счетчика команд)
    d. Постановка и снятие точек останова по номеру строки в исходном тексте (см. п.2.b)
    e. Отображение стека вызовов в виде списка имён подпрограмм по порядку их вызова друг другом от точки входа до текущей выполняемой подпрограммы
4. Дополнить разработанную в третьем задании программу, внеся соответствующие изменения в каждый из её модулей для поддержки пользовательских типов и полиморфизма по варианту
    a. Описать необходимые структуры данных для представления информации о пользовательских типах, их членах (полях и методах) и отношениях
    b. Добавить поддержку новых синтаксических конструкций в модуль, реализующий синтаксический анализ из задания 1 (см. дополнения к синтаксической модели)
    c. Добавить поддержку новых операций для работы с членами типов в дереве операций, формируемом модулем из задания 2, отвечающим за построение графа потока управления
    d. Поддержать формирование необходимых последовательностей инструкций в линейном коде для добавленных операций в модуле, реализованном в задании 3
    e. Используя псевдоинструкции определения данных, добавить в секцию с информацией о структуре программы информацию, описывающую пользовательские типы и их поля (имена и типы значений)
    f. В консольном приложении-инспекторе при выводе информации о значениях аргументов и локальных переменных поддержать вывод значений в полях пользовательских типов
5. Результаты тестирования представить в виде отчета, в который включить:
    a. В части 3 привести описание изменений в структурах данных и логике затронутых модулей (пп.4.a-d), внутреннюю организацию секции с информацией о структуре программы (п.2, п.4.e)
    b. В части 4 описать программный интерфейс и особенности реализации программы-инспектора
    c. В части 5 привести примеры исходных текстов, соответствующие ассемблерные листинги и примера вывода запущенных тестовых программ, а также вывод ряда команд программы-инспектора на разных этапах работы инспектируемой программы (п.3)

Варианты к заданию 1

Вариант	**Входной язык**
1	1
2	2
3	3
4	4
5	1
6	2
7	3
8	4
9	1
10	2
11	3
12	4
13	1
14	2
15	3
16	4
17	1
18	2
19	3
20	4
21	1
22	2
23	3
24	4
25	1
26	2
27	3
28	4
29	1
30	2
31	3
32	4

Общая часть синтаксической модели для всех вариантов:

```
identifier: "[a-zA-Z_][a-zA-Z_0-9]*"; // идентификатор

str: "\"[^\"\\]*(?:\\.[^\"\\]*)*\""; // строка, окруженная двойными кавычками
char: "'[^']'";                      // одиночный символ в одинарных кавычках
hex: "0[xX][0-9A-Fa-f]+";            // шестнадцатеричный литерал
bits: "0[bB][01]+";                  // битовый литерал
dec: "[0-9]+";                       // десятичный литерал
bool: 'true'|'false';                // булевский литерал

list<item>: (item (',' item)*)?; // список элементов, разделённых запятыми
```

Вариант 1

```
source: sourceItem*;

typeRef: {
      |builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';
      |custom: identifier;
      |array: typeRef '[' (',')* ']';
};

funcSignature: typeRef? identifier '(' list<argDef> ')' {
      argDef: typeRef? identifier;
};

sourceItem: {
      |funcDef: funcSignature (statement.block|';');
};

statement: {
      |var: typeRef list<identifier ('=' expr)?> ';'; // for static typing
      |if: 'if' '(' expr ')' statement ('else' statement)?;
      |block: '{' statement* '}';
      |while: 'while' '(' expr ')' statement;
      |do: 'do' block 'while' '(' expr ')' ';';
      |break: 'break' ';';
      |expression: expr ';'
};

expr: { // присваивание через '='
      |binary: expr binOp expr; // где binOp - символ бинарного оператора
      |unary: unOp expr; // где unOp - символ унарного оператора
      |braces: '(' expr ')';
      |call: expr '(' list<expr> ')';
      |indexer: expr '[' list<expr> ']';
      |place: identifier;
      |literal: bool|str|char|hex|bits|dec;
};
```

Вариант 2

```
source: sourceItem*;

typeRef: {
      |builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';
      |custom: identifier;
      |array: 'array' '[' (',')* ']' 'of' typeRef;
};

funcSignature: identifier '(' list<argDef> ')' (':' typeRef)? {
      argDef: identifier (':' typeRef)?;
};

sourceItem: {
      |funcDef: 'method' funcSignature (body|';') {
            body: ('var' (list<identifier> (':' typeRef)? ';')*)? statement.block;
      };

statement: {
      |if: 'if' expr 'then' statement ('else' statement)?;
      |block: 'begin' statement* 'end' ';';
      |while: 'while' expr 'do' statement;
      |do: 'repeat' statement ('while'|'until') expr ';';
      |break: 'break' ';';
      |expression: expr ';';
};

expr: { // присваивание через ':='
      |binary: expr binOp expr; // где binOp - символ бинарного оператора
      |unary: unOp expr; // где unOp - символ унарного оператора
      |braces: '(' expr ')';
      |call: expr '(' list<expr> ')';
      |indexer: expr '[' list<expr> ']';
      |place: identifier;
      |literal: bool|str|char|hex|bits|dec;
};
```

Вариант 3

```
source: sourceItem*;

typeRef: {
      |builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';
      |custom: identifier;
      |array: typeRef '(' (',')* ')';
};

funcSignature: identifier '(' list<argDef> ')' ('as' typeRef)? {
      argDef: identifier ('as' typeRef)?;
};

sourceItem: {
      |funcDef: 'function' funcSignature (statement* 'end' 'function')?;
};

statement: {
      |var: 'dim' list<identifier> 'as' typeRef;// for static typing
      |if: 'if' expr 'then' statement* ('else' statement*)? 'end' 'if';
      |while: 'while' expr statement* 'wend';
      |do: 'do' statement* 'loop' ('while'|'until') expr;
      |break: 'break';
      |expression: expr ';';
};

expr: { // присваивание через '='
      |binary: expr binOp expr; // где binOp - символ бинарного оператора
      |unary: unOp expr; // где unOp - символ унарного оператора
      |braces: '(' expr ')';
      |callOrIndexer: expr '(' list<expr> ')';
      |place: identifier;
      |literal: bool|str|char|hex|bits|dec;
};
```

Вариант 4

```
source: sourceItem*;

typeRef: {
      |builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';
      |custom: identifier;
      |array: typeRef 'array' '[' dec ']'; // число - размерность
};

funcSignature: identifier '(' list<arg> ')' ('of' typeRef)? {
      arg: identifier ('of' typeRef)?;
};

sourceItem: {
      |funcDef: 'def' funcSignature (statement* 'end')?;
};

statement: { // присваивание через '='
      |if: 'if' expr 'then' statement ('else' statement)?;
      |loop: ('while'|'until') expr statement* 'end';
      |repeat: statement ('while'|'until') expr ';';
      |break: 'break' ';';
      |expression: expr ';';
      |block: ('begin'|'{') (statement|sourceItem)* ('end'|'}');
};

expr: {
      |binary: expr binOp expr; // где binOp - символ бинарного оператора
      |unary: unOp expr; // где unOp - символ унарного оператора
      |braces: '(' expr ')';
      |call: expr '(' list<expr> ')';
      |slice: expr '[' list<range> ']' { // индексация или срез массива
            ranges: expr ('..' expr)?; // from index, to
      };
      |place: identifier;
      |literal: bool|str|char|hex|bits|dec;
};
```

Варианты к заданию 3

Вариант	**Типизация**	**Код**	**Банки памяти**
1	Статическая	Стековый	1: общая RAM
2	Динамическая	Стековый	2: код, данные
3	Статическая	Стековый	3: код, константы, данные
4	Динамическая	Стековый	4: код с константами, данные
5	Статическая	Стековый	5: код с данными, стек
6	Динамическая	Стековый	6: код, константы, данные, стек
7	Статическая	Регистровый двухадресный	1: общая RAM
8	Динамическая	Регистровый двухадресный	2: код, данные
9	Статическая	Регистровый двухадресный	3: код, константы, данные
10	Динамическая	Регистровый двухадресный	4: код с константами, данные
11	Статическая	Регистровый двухадресный	5: код с данными, стек
12	Динамическая	Регистровый двухадресный	6: код, константы, данные, стек
13	Статическая	Регистровый трёхадресный	1: общая RAM
14	Динамическая	Регистровый трёхадресный	2: код, данные
15	Статическая	Регистровый трёхадресный	3: код, константы, данные
16	Динамическая	Регистровый трёхадресный	4: код с константами, данные
17	Статическая	Регистровый трёхадресный	5: код с данными, стек
18	Динамическая	Регистровый трёхадресный	6: код, константы, данные, стек
19	Динамическая	Стековый	1: общая RAM
20	Статическая	Стековый	2: код, данные
21	Динамическая	Стековый	3: код, константы, данные
22	Статическая	Стековый	4: код с константами, данные
23	Динамическая	Стековый	5: код с данными, стек
24	Статическая	Стековый	6: код, константы, данные, стек
25	Динамическая	Регистровый двухадресный	1: общая RAM
26	Статическая	Регистровый двухадресный	2: код, данные
27	Динамическая	Регистровый двухадресный	3: код, константы, данные
28	Статическая	Регистровый двухадресный	4: код с константами, данные
29	Динамическая	Регистровый двухадресный	5: код с данными, стек
30	Статическая	Регистровый двухадресный	6: код, константы, данные, стек
31	Динамическая	Регистровый трёхадресный	1: общая RAM
32	Статическая	Регистровый трёхадресный	2: код, данные
33	Динамическая	Регистровый трёхадресный	3: код, константы, данные
34	Статическая	Регистровый трёхадресный	4: код с константами, данные
35	Динамическая	Регистровый трёхадресный	5: код с данными, стек
36	Статическая	Регистровый трёхадресный	6: код, константы, данные, стек

Для вариантов с со статической типизацией – указания типов во всех синтаксических конструкциях обязательны.

Для вариантов с динамической типизацией – указания типов опциональны и могут игнорироваться, тип переменной соответствует последнему присвоенному значению.

При работе с реальной ВМ (второй сценарий при подготовке к выполнению) – указания типов при описании импортируемых функций и отсутствии у них тела обязательны во всех вариантах.

Варианты к заданию 4

Вариант	**Способ представления пользовательских типов**	**Полиморфизм для подпрограмм**	**Расширение типов**
1	По ссылке	Перегрузка	Шаблонизациия
2	По ссылке	Перегрузка	Интерфейсы
3	По ссылке	Переопределение	Шаблонизация
4	По ссылке	Переопределение	Интерфейсы
5	По значению	Перегрузка	Шаблонизациия
6	По значению	Перегрузка	Интерфейсы
7	По значению	Переопределение	Шаблонизация
8	По значению	Переопределение	Интерфейсы
9	По ссылке	Переопределение	Шаблонизациия
10	По ссылке	Переопределение	Интерфейсы
11	По ссылке	Перегрузка	Шаблонизация
12	По ссылке	Перегрузка	Интерфейсы
13	По значению	Переопределение	Шаблонизациия
14	По значению	Переопределение	Интерфейсы
15	По значению	Перегрузка	Шаблонизация
16	По значению	Перегрузка	Интерфейсы
17	По ссылке	Перегрузка	Интерфейсы
18	По ссылке	Перегрузка	Шаблонизация
19	По ссылке	Переопределение	Интерфейсы
20	По ссылке	Переопределение	Шаблонизация
21	По значению	Перегрузка	Интерфейсы
22	По значению	Перегрузка	Шаблонизация
23	По значению	Переопределение	Интерфейсы
24	По значению	Переопределение	Шаблонизация
25	По ссылке	Переопределение	Интерфейсы
26	По ссылке	Переопределение	Шаблонизация
27	По ссылке	Перегрузка	Интерфейсы
28	По ссылке	Перегрузка	Шаблонизация
29	По значению	Переопределение	Интерфейсы
30	По значению	Переопределение	Шаблонизация
31	По значению	Перегрузка	Интерфейсы
32	По значению	Перегрузка	Шаблонизация

Для вариантов с перегрузкой – переопределение методов базового типа запрещено, все методы в одной цепочке наследования имеют разные сигнатуры.

Для вариантов с переопределением – перегрузка запрещена, все методы в цепочке наследования, кроме переопределяемых, имеют разные имена, а участвующие в переопределении имеют полностью одинаковые сигнатуры.

Для вариантов с шаблонизацией – в качестве параметров шаблона разрешены только типы, количество типов-параметров может быть произвольным и задаётся при определении пользовательского шаблонного типа.

Для вариантов с интерфейсами – для пользовательских разрешено реализовывать несколько интерфейсов, поддержка расширения между интерфейсами не обязательна.

Для вариантов на базе учебной ВМ – синтаксические конструкции для объявления внешних функций не используются.

Для вариантов на базе реальной ВМ – поддержка внешних функций обязательна через статическое или динамическое связывание по выбору. Синтаксические конструкции для определения внешних функций поддерживать по мере необходимости.

Прямое наследование от одного предка разрешено во всех вариантах.

Полиморфизм подпрограмм разрешён только для методов пользовательских типов.

Контроль областей видимости по желанию.

Дополнения к синтаксической модели

Дополнительные синтаксические конструкции для выражений ввести самостоятельно:

1. Для инстанцирования пользовательского типа – с использованием префиксного ключевого слова или используя имя типа в качестве имени конструирующей функции
2. Для обращения к члену пользовательского типа (полю или методу) – на основе бинарного оператора с использованием ещё не задействованного символа по выбору (`'.'`, `'->'` и т.п.)

Общая часть синтаксической модели для всех вариантов:

```
customTypeKw: 'class'|'interface';
```

Вариант 1

```
sourceItem: {
  |funcDef: importSpec? funcSignature (statement.block|';') {
    importSpec: 'extern' '(' dllName (',' dllEntryName)? ')';
    dllName: str; // имя DLL, содержащей импортируемую функцию
    dllEntryName: str; // имя точки входа при отличии от объявляемого
  };
  |classDef: customTypeKw identifier extension? '{' member* '}' {
    member: modifier? (funcDef|field);
    field: typeRef? list<identifier> ';';
    modifier: 'public'|'private';
    extension: templated? (':' list<identifier>)?;
    templated: '<' list<identifier> '>';
  };
};
```

Вариант 2

```
sourceItem: {
  |funcDef: 'method' funcSignature (body|';'|importSpec) {
    body: varsSpec? statement.block;
    varsSpec: 'var' (list<identifier> (':' typeRef)? ';')*;
    importSpec: 'from' (dllEntryName 'in')? dllName ';';
    dllName: str; // имя DLL, содержащей импортируемую функцию
    dllEntryName: str; // имя точки входа при отличии от объявляемого
  };
  |classDef:customTypeKw identifier extension? funcDef.varsSpec 'begin' member* 'end'{
    member: modifier? (funcDef);
    modifier: 'public'|'private';
    extension: (':' identifier)? templated? ('implements' identifier ';')*;
    templated: ('generic' identifier ';')*;
  };
```

```
};
```

Вариант 3

```
sourceItem: {
  |funcDef: 'function' funcSignature statement* 'end' 'function';
  |externFuncDef: 'declare' 'function' funcSignature 'lib' dllName ('alias'
dllEntryName)? {
    dllName: str; // имя DLL, содержащей импортируемую функцию
    dllEntryName: str; // имя точки входа при отличии от объявляемого
  };
  |classDef: customTypeKw identifier extension? member* 'end' 'class' {
    member: modifier? (funcDef|field|externFuncDef);
    field: list<identifier> ('as' typeRef)?;
    modifier: 'public'|'private';
    extension: template? 'inherits' ('<' list<identifier> '>')?;
    templated: '(' 'of' list<identifier> ')';
  };
};
```

Вариант 4

```
sourceItem: {
  |funcDef: 'def' funcSignature statement* 'end';
  |externFuncDef: 'import' funcSignature 'native' dllName ('entry' dllEntryName)? {
    dllName: str; // имя DLL, содержащей импортируемую функцию
    dllEntryName: str; // имя точки входа при отличии от объявляемого
  };
  |classDef: customTypeKw identifier extension? member* 'end' {
    member: modifier? (funcDef|field|externFuncDef);
    field: 'var' list<identifier> ('of' typeRef)? ';';
    modifier: 'public'|'private';
    extension: templated? ('extends' identifier)? ('implements' list<identifier>)?;
    templated: '<' list<identifier> '>';
  };

};
```