

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего
образования

"Московский авиационный институт
(национальный исследовательский университет)"

УТВЕРЖДАЮ

Проректор по учебной работе

Козорез Д.А.
27 июня 2022 г.

РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ (000182852)

Системное программное обеспечение

(указывается наименование дисциплины по учебному плану)

Направление подготовки	09.03.01 Информатика и вычислительная техника
Квалификация выпускника	Бакалавр
Профиль подготовки	Автоматизированные системы обработки информации и управления
Форма обучения	заочная
	(очно, очно-заочное, заочное)
Выпускающая кафедра	МСиИТ
Обеспечивающая кафедра	МСиИТ
Кафедра-разработчик рабочей программы	МСиИТ

Семестр	З.Е.	Трудоемкость, час.	Лекций, час.	Практич. занятий, час.	Лаборат. работ, час.	СРС, час	Экзамен-нов, час.	Форма промежуточног о контроля
9	6	216	8	0	8	164	36	Э
Итого	6	216	8	0	8	164	36	

Москва
2022

РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ

Разделы рабочей программы

1. Цели освоения дисциплины. Перечень планируемых результатов обучения.
2. Место дисциплины в структуре образовательной программы.
3. Структура и содержание дисциплины.
4. Перечень учебно-методического обеспечения для самостоятельной работы обучающихся по дисциплине.
5. Фонд оценочных средств для проведения промежуточной аттестации обучающихся по дисциплине.
6. Перечень основной и дополнительной учебной литературы, необходимой для освоения дисциплины.
7. Перечень ресурсов информационно-телекоммуникационной сети «Интернет», необходимых для освоения дисциплины.
8. Методические указания для обучающихся по освоению дисциплины.
9. Перечень информационных технологий, используемых при осуществлении образовательного процесса по дисциплине.
10. Описание материально-технической базы, необходимой для осуществления образовательного процесса по дисциплине.

Приложения к рабочей программе дисциплины

Приложение 1. Аннотация рабочей программы

Приложение 2. Прикрепленные файлы

Программа составлена в соответствии с требованиями СУОС МАИ, разработанного на основе ФГОС ВО (3++) по направлению 09.03.01 Информатика и вычислительная техника

Авторы программы:

Уханова А. М.

Заведующий обеспечивающей кафедрой МСиИТ

Программа одобрена:

Заведующий выпускающей кафедрой
МСиИТ

Директор выпускающего филиала СТ

1. ЦЕЛИ ОСВОЕНИЯ ДИСЦИПЛИНЫ. ПЕРЕЧЕНЬ ПЛАНИРУЕМЫХ КОМПЕТЕНЦИЙ И РЕЗУЛЬТАТОВ ОБУЧЕНИЯ.

Целью освоения дисциплины Системное программное обеспечение является достижение следующих результатов освоения(РО):

N	Шифр	Результат обучения
1	В-1(ПКР-3.3)	Владеть программными средствами мониторинга системного программного обеспечения
2	З-1(ДПК-2.1)	Знать основные теоретические положения теории системного ПО
3	У-1(ДПК-2.1)	Уметь разрабатывать, тестировать, отлаживать и документировать компиляторы, используя арсенал средств инструментального ПО
4	В-1(ДПК-2.1)	Владеть современными инструментальными средствами для разработки системного программного обеспечения
5	З-1(ДПК-2.2)	Знать способы настройки системы контрол версий и регистрации ошибок, возникающих при решении поставленной задачи
6	У-1(ДПК-2.2)	Уметь использовать способы настройки системы контроля версий и регистрации ошибок, возникающих при решении поставленной задачи
7	В-1(ДПК-2.2)	Владеть способами настройки системы контроля версий и регистрации ошибок, возникающих при решении поставленной задачи

Перечисленные РО являются основой для формирования следующих компетенций:

N	Шифр	Компетенция
1	ДПК-2	Способен организовывать работу по разработке системного программного обеспечения
2	ПКР-3	Способен разрабатывать компоненты системных программных продуктов

Индикаторы достижения компетенций, служащие для проверки сформированности части соответствующей компетенции:

N	Шифр	Индикатор компетенций
1	ПКР-3.3.	Использует программные продукты для измерения характеристик системного программного обеспечения
2	ДПК-2.1	Демонстрирует знание основ теории и методики разработки системного программного обеспечения
3	ДПК-2.1	Демонстрирует знание основ теории и методики разработки системного программного обеспечения
4	ДПК-2.1	Демонстрирует знание основ теории и методики разработки системного программного обеспечения
5	ДПК-2.2	Настраивает системы контроля версий и регистрации ошибок, возникающих при решении поставленной задачи
6	ДПК-2.2	Настраивает системы контроля версий и регистрации ошибок, возникающих при решении поставленной задачи

7	ДПК-2.2	Настраивает системы контроля версий и регистрации ошибок, возникающих при решении поставленной задачи
---	---------	---

2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ.

Дисциплина Системное программное обеспечение является предшествующей и последующей для следующих дисциплин:

N	Предшествующие дисциплины	Последующие дисциплины
---	---------------------------	------------------------

3. СТРУКТУРА И СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

Общая трудоемкость практики составляет 6 зачетных(ые) едениц(ы), 216 часа(ов).

Модуль	Раздел	Лекции	Практич. занятия	Лаборат. работы	СРС	Всего часов	Всего с экзаменами и курсовыми
Системное программное обеспечение (9 семестр).	Введение.	2	0	0	0	2	216
	Основания машинной работы с текстами.	2	0	8	12	22	
	Теория компиляции.	4	0	0	96	100	
	Отладка программ.	0	0	0	8	8	
	Системы контроля версий.	0	0	0	4	4	
	Системы управления сборкой.	0	0	0	8	8	
Всего		8	0	8	128	144	216

3.1. Лекции

№ п/п	Раздел дисциплины	Объем часов	Тема лекции
1	1.1.Введение.	2	Средства и задачи системного программного обеспечения.
2	1.2.Основания машинной работы с текстами.	2	Таблицы кодировки 7/8- битные. Транслитерация кириллицы латиницей.
3	1.2.Основания машинной работы с текстами.		Unicode, UCS, UTF-7/8/16.
4	1.3.Теория компиляции.	2	Способы задания языка. Грамматики.
5	1.3.Теория компиляции.	2	Выводимая цепочка. НФБН.

6	1.3.Теория компиляции.		Вывод цепочки. Левый (правый) вывод.
7	1.3.Теория компиляции.		Однозначность грамматики. Дерево разбора.
8	1.3.Теория компиляции.		Постфиксная (обратная польская) запись.
9	1.3.Теория компиляции.		Левая рекурсия и ее устранение. Рекурсивный спуск.
10	1.3.Теория компиляции.		Иерархия Хомского. Способы задания регулярных языков.
11	1.3.Теория компиляции.		КА, ДКА и НКА. Преобразование НКА в ДКА.
12	1.3.Теория компиляции.		Минимизация КА. Теорема Клини.
13	1.3.Теория компиляции.		Лемма о разрастании для регулярных языков. Программа Lex.
14	1.3.Теория компиляции.		Свойства регулярных языков. Регулярные множества. Стековый (магазинный) КА.
15	1.3.Теория компиляции.		Контекстно-свободные языки. Построение НМКА по заданной грамматике.
16	1.3.Теория компиляции.		Лемма о разрастании для КС-языков. Преобразование КС-языков.
17	1.3.Теория компиляции.		Удаление бесполезных символов и правил. Удаление е-правил.
18	1.3.Теория компиляции.		Устранение циклов и цепных правил. Устранение левой рекурсии.
19	1.3.Теория компиляции.		Нормальные формы Хомского и Грейбаха для КС-языков.
20	1.3.Теория компиляции.		Назначение алгоритмов Кока-Янгера-Касами и Эрли, их свойства.
21	1.3.Теория компиляции.		Множества FIRST и FOLLOW. Алгоритм работы анализатора.
22	1.3.Теория компиляции.		Пример LL(1)- разбора. Разбор снизу-вверх. Сдвиг-свертка.
23	1.3.Теория компиляции.		Граматики простого и операторного предшествования. Линеаризация матрицы предшествования.
24	1.3.Теория компиляции.		Программа Yacc. GLR.
25	1.4.Отладка программ.		Отладка программ.
26	1.4.Отладка программ.		Отладка программ.
27	1.5.Системы контроля версий.		Обзор систем контроля версий.
28	1.5.Системы контроля версий.		Основные возможности программы Subversion.

29	1.6.Системы управления сборкой.		Обзор систем управления сборкой. Основные возможности программы Make.
30	1.6.Системы управления сборкой.		Обзор систем управления сборкой. Основные возможности программы Make.
Итого:		8	

3.2. Содержание лекций

1.1.1. Средства и задачи системного программного обеспечения. (АЗ: 2, СРС: 0)

Тип лекции: Информационная лекция

Форма организации: Лекция

1.2.1. Таблицы кодировки 7/8- битные. Транслитерация кириллицы латиницей. (АЗ: 2, СРС: 4)

Тип лекции: Информационная лекция

Форма организации: Лекция

1.2.2. Unicode, UCS, UTF-7/8/16. (АЗ: 0, СРС: 4)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.3.1. Способы задания языка. Грамматики. (АЗ: 2, СРС: 4)

Тип лекции: Информационная лекция

Форма организации: Лекция

1.3.2. Выводимая цепочка. НФБН. (АЗ: 2, СРС: 4)

Тип лекции: Информационная лекция

Форма организации: Лекция

1.3.3. Вывод цепочки. Левый (правый) вывод. (АЗ: 0, СРС: 4)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.3.4. Однозначность грамматики. Дерево разбора. (АЗ: 0, СРС: 4)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.3.5. Постфиксная (обратная польская) запись. (АЗ: 0, СРС: 4)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.3.6. Левая рекурсия и ее устранение. Рекурсивный спуск. (АЗ: 0, СРС: 4)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.3.7. Иерархия Хомского. Способы задания регулярных языков. (АЗ: 0, СРС: 4)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

- 1.3.8. КА. ДКА и НКА. Преобразование НКА в ДКА. (АЗ: 0, СРС: 4)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа
- 1.3.9. Минимизация КА. Теорема Клини. (АЗ: 0, СРС: 4)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа
- 1.3.10. Лемма о разрастании для регулярных языков. Программа Lex. (АЗ: 0, СРС: 2)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа
- 1.3.11. Свойства регулярных языков. Регулярные множества. Стековый (магазинный) КА. (АЗ: 0, СРС: 2)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа
- 1.3.12. Контекстно-свободные языки. Построение НМКА по заданной грамматике. (АЗ: 0, СРС: 2)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа
- 1.3.13. Лемма о разрастании для КС-языков. Преобразование КС-языков. (АЗ: 0, СРС: 2)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа
- 1.3.14. Удаление бесполезных символов и правил. Удаление е-правил. (АЗ: 0, СРС: 2)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа
- 1.3.15. Устранение циклов и цепных правил. Устранение левой рекурсии. (АЗ: 0, СРС: 2)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа
- 1.3.16. Нормальные формы Хомского и Грейбаха для КС-языков. (АЗ: 0, СРС: 2)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа
- 1.3.17. Назначение алгоритмов Кока-Янгера-Касами и Эрли, их свойства. (АЗ: 0, СРС: 2)**
Тип лекции: Информационная лекция
Форма организации: Самостоятельная работа

1.3.18. Множества FIRST и FOLLOW. Алгоритм работы анализатора. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.3.19. Пример LL(1)- разбора. Разбор снизу-вверх. Сдвиг-свертка. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.3.20. Грамматики простого и операторного предшествования. Линеаризация матрицы предшествования. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.3.21. Программа Yacc. GLR. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.4.1. Отладка программ. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

Описание: Назначение. Основные способы и этапы отладки. Точки останова. Пошаговое исполнение. Значения останова. Работа со стекком. Детали организации работы с подпрограммами на машинном уровне. Создание журнала исполнения. Профилирование.

1.4.2. Отладка программ. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

Описание: Назначение. Основные способы и этапы отладки. Точки останова. Пошаговое исполнение. Значения останова. Работа со стекком. Детали организации работы с подпрограммами на машинном уровне. Создание журнала исполнения. Профилирование.

1.5.1. Обзор систем контроля версий. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.5.2. Основные возможности программы Subversion. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.6.1. Обзор систем управления сборкой. Основные возможности программы Make. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

1.6.2. Обзор систем управления сборкой. Основные возможности программы Make. (АЗ: 0, СРС: 2)

Тип лекции: Информационная лекция

Форма организации: Самостоятельная работа

3.3 Практические занятия

Не предусмотрено учебным планом.

3.4. Лабораторные работы

№ п/п	Раздел дисциплины	Объем часов	Наименование лабораторной работы
1	1.2.Основания машинной работы с текстами.	4	Кодировки.
2	1.2.Основания машинной работы с текстами.	4	Транслитерации.
3	1.3.Теория компиляции.		Конструирование грамматик и автоматов. Алгебра регулярных множеств. Преобразование НКА в ДКА. Минимизация КА.
4	1.3.Теория компиляции.		Распознавание класса языка в иерархии Хомского по грамматике.
5	1.3.Теория компиляции.		Преобразование выражений RPN.
6	1.3.Теория компиляции.		Построение канонических LR-таблиц, их упрощение.
7	1.3.Теория компиляции.		Построение канонических LR-таблиц, их упрощение (часть 2).
8	1.3.Теория компиляции.		Разработка компилятора на бизоне (и флексе) (часть 1).
9	1.3.Теория компиляции.		Разработка компилятора на бизоне (и флексе) (часть 2).
10	1.3.Теория компиляции.		Разработка компилятора на прологе (часть 1).
11	1.3.Теория компиляции.		Разработка компилятора на прологе (часть 2).
12	1.4.Отладка программ.		Использование gdb и gprof.
13	1.6.Системы управления сборкой.		Использование make и svn.
Итого:		8	

3.5.Содержание лабораторных работ

1.2.1. Кодировки. (АЗ: 4, СРС: 2)

Форма организации: Лабораторная работа

1.2.2. Транслитерации. (АЗ: 4, СРС: 2)

Форма организации: Лабораторная работа

1.3.1. Конструирование грамматик и автоматов. Алгебра регулярных множеств. Преобразование НКА в ДКА. Минимизация КА. (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.3.2. Распознавание класса языка в иерархии Хомского по грамматике. (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.3.3. Преобразование выражений RPN. (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.3.4. Построение канонических LR-таблиц, их упрощение. (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.3.5. Построение канонических LR-таблиц, их упрощение (часть 2). (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.3.6. Разработка компилятора на бизоне (и флексе) (часть 1). (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.3.7. Разработка компилятора на бизоне (и флексе) (часть 2). (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.3.8. Разработка компилятора на прологе (часть 1). (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.3.9. Разработка компилятора на прологе (часть 2). (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.4.1. Использование gdb и gprof. (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

1.6.1. Использование make и svn. (АЗ: 0, СРС: 4)

Форма организации: Самостоятельная работа

3.6. Курсовые работы и проекты по дисциплине

1.1. Разработка компилятора по варианту.

Тематика:

Трудоемкость(СРС): 36

Прикрепленные файлы: КР СПО.pdf

3.7. Промежуточная аттестация

1. Экзамен (9 семестр)

Прикрепленные файлы: СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ.docx,
СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ.pdf

4. ПЕРЕЧЕНЬ УЧЕБНО-МЕТОДИЧЕСКОГО ОБЕСПЕЧЕНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

1. Основная и дополнительная литература по дисциплине
2. Ресурсы информационно-телекоммуникационной сети «Интернет».
3. Ресурсы научно-технической библиотеки МАИ.
4. Информационные стенды кафедры.

5. ФОНД ОЦЕНОЧНЫХ СРЕДСТВ ДЛЯ ПРОВЕДЕНИЯ ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

Описание показателей, критерии оценивания компетенций и описание шкал оценивания осуществляются в соответствии с Положением о балльно-рейтинговой системе оценки результатов обучения студентов по дисциплине (Приказ №42 от 04.04.2014 «Об утверждении положения «Рейтинг по дисциплине»).

Для оценивания интегрированных и практико-ориентированных заданий обучающихся используются следующие критерии по 100-балльной шкале:

1. Формулирование представленной информации в виде проблемы;
2. Предложение способа решения проблемы;
3. Обоснование способа решения проблемы;
4. Демонстрация способа решения проблемы.

Оценивание осуществляется по следующей шкале:

100-балльная шкала	Результат освоения
менее 40	Критерий не сформирован
41-70	Критерий четко не выражен
71-100	Критерий выражен четко

Для оценивания ситуационных заданий используется следующая шкала:

100-балльная шкала	Результат освоения
менее 30	обучающийся не может сформулировать проблему, представленную в задании
31-50	обучающийся формулирует поставленную задачу, у него сформированы изолированные знания и умения, однако отсутствуют интегрированные понятия и навыки, в результате чего допущены ошибки в решении и задание не выполнено
51-80	задание выполнено, обучающийся применяет знания для решения поставленной проблемы, однако не сформированы компетенции, вследствие чего обучающийся испытывает затруднения в демонстрации способов решения задачи

81-100	задание выполнено как в теоретическом, так и в практическом плане, обучающийся легко демонстрирует свою компетентность по данному вопросу
--------	---

Фонды оценочных средств, позволяющие оценить результаты обучения, включают в себя:

- вопросы к промежуточной аттестации.

Перечень компетенций и этапы их формирования приведены в следующей таблице:

N	Шифр	Компетенция	Этапы формирования компетенции
1	ДПК-2	Способен организовывать работу по разработке системного программного обеспечения	Знать основные теоретические положения теории системного ПО Уметь разрабатывать, тестировать, отлаживать и документировать компиляторы, используя арсенал средств инструментального ПО Владеть современными инструментальными средствами для разработки системного программного обеспечения Знать способы настройки системы контрол версий и регистрации ошибок, возникающих при решении поставленной задачи Уметь использовать способы настройки системы контроля версий и регистрации ошибок, возникающих при решении поставленной задачи Владеть способами настройки системы контроля версий и регистрации ошибок, возникающих при решении поставленной задачи Семестр - 9
2	ПКР-3	Способен разрабатывать компоненты системных программных продуктов	Владеть программными средствами мониторинга системного программного обеспечения Семестр - 9

Вопросы к промежуточной аттестации

"Системное программное обеспечение"

1. Экзамен (9 семестр)

Прикрепленные файлы: СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ.docx, СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ.pdf

6. ПЕРЕЧЕНЬ ОСНОВНОЙ И ДОПОЛНИТЕЛЬНОЙ УЧЕБНОЙ ЛИТЕРАТУРЫ, НЕОБХОДИМОЙ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

а) Основная литература:

- 1. Системное и прикладное программное обеспечение : учебное пособие / составители И. А. Журавлёва, П. К. Корнеев. — Ставрополь : СКФУ, 2017. — 132 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: <https://e.lanbook.com/book/155253> (дата обращения: 13.12.2021). — Режим доступа: для авториз. пользователей.
- 2. Бохан, К. А. Системное программное обеспечение : учебное пособие / К. А. Бохан. — Рязань : РГРТУ, 2010. — 64 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: <https://e.lanbook.com/book/167994> (дата обращения: 13.12.2021). — Режим доступа: для авториз. пользователей.
- 3. Синаторов, С. В. Информационные технологии : учебное пособие / С. В. Синаторов. - 2-е изд., стер. - Москва : Флинта, 2021. - 448 с. - ISBN 978-5-9765-1717-2. - Текст : электронный. - URL: <https://znanium.com/catalog/product/1304012> (дата обращения: 13.12.2021). – Режим доступа: по подписке.

б) Дополнительная литература:

- 1. Ахо А. Компиляторы. Принципы, технологии, инструменты /А. Ахо, Р. Се-ти, Дж. Ульман, М. Лам — М., СПб., Киев: Вильямс, 2008. — 1185 с.
- 2. Гагарина Л. Г. Введение в теорию алгоритмических языков и компиляторов /Л. Г. Гагарина, Е. В. Кокорева — М.: ИД ФОРУМ, 2011. — 176 с.
- 3. Залогова Л. А. Разработка Паскаль-компилятора /Л. А. Залогова — М.: БИНОМ. Лаборатория знаний, 2012. — 183 с.
- 4. Керниган Б. В. Unix — универсальная среда программирования /Б. В. Керниган, Р. Пайк — М.: Финансы и статистика, 1992.
- 5. В. В. Лидовский Вавилонское кодотворение //Магия ПК, 1/2004.
- 6. В. В. Лидовский Первичная машинная обработка текста: методика и проблематика — депонировано в ИНИОН РАН (N 53656) — 1998.
- 7. Бен Коллинз-Сассман Управление версиями в Subversion [Электронный ресурс] /Бен Коллинз-Сассман, Брайан У. Фитцпатрик, К. Майкл Пилато — Режим доступа: <http://svnbook.red-bean.com/>, 2007.
- 8. Страуструп, Б. Дизайн и эволюция C++ /Б. Страуструп — М.: ДМК Пресс, 2007. - 448 с.
- 9. Charles Donnelly, Richard Stallman Bison — Free Software Foundation, 2009. 173 p.
- 10. Richard M. Stallman, Roland McGrath GNU Make — Free Software Foundation, 2000. 143 p.
- 1. Вирт, Н. Построение компиляторов [Электронный ресурс] /Никлаус Вирт — М.: ДМК Пресс, 2010. — 192 с. Режим доступа: <http://www.znanium.com/bookread.php?book=408433>

7. ПЕРЕЧЕНЬ РЕСУРСОВ ИНФОРМАЦИОННО-ТЕЛЕКОММУНИКАЦИОННОЙ СЕТИ «ИНТЕРНЕТ», НЕОБХОДИМЫХ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

Для обеспечения образовательного процесса по дисциплине обучающимся предоставляется возможность круглосуточного дистанционного индивидуального доступа к электронным библиотечным системам из любой точки, в которой имеется доступ к сети «Интернет».

Наименование ресурса	Интернет-ссылка на ресурс
"ZNANIUM.COM"	
Договор № 4855 эбс/027-1-3200-20 от 08.12.2020 с ООО "ЗНАНИУМ" С «18»12.2020 г. по «17»12.2021 г	http://znanium.com
Договор № эбс/027-1-3026-21 от 22.12.2021 с ООО "ЗНАНИУМ" С «15»12.2021 г. по «31»12.2022 г	https://znanium.com/
Договор № эбс/027-1-2586-22 от 07.12.2022 с ООО "ЗНАНИУМ" С «20»12.2022 г. по «31»12.2023 г	
ООО "Издательство Лань"	
Договор № 027-1-0234-21 от 18.02.2021 года с ООО "Издательство Лань" С «22 »_02. 2021г. по « 21» 02.2022 г	e.lanbook.com
Договор № 027-1-0234-21 от 18.02.2021 года с ООО "ЭБС Лань" С «22 »_02. 2021г. по « 21» 02.2022	
Договор № СЭБ 027-0-0400-21 от 15.09.2021 года с ООО "ЭБС Лань" С «15 »_09. 2021г. по « 14» 09.2024	
Договор № 027-1-0169-22 от 07.02.2022 года с ООО "Издательство Лань" С «22 »_02. 2022г. по « 21» 02.2023 г	
Договор № 027-1-0168-22 от 07.02.2022 года с ООО "ЭБС Лань" С «22 »_02. 2022г. по « 21» 02.2023	
ООО "Электронное издательство ЮРАЙТ"	
Электронная библиотечная система ЮРАЙТ. ЭБС "Легендарные книги"	http://biblio-online.ru , https://biblio-online.ru/catalog/legendary
Договор № 027-1-3191-20 от 04.12.2020г ООО "Электронное издательство ЮРАЙТ" для СПО С «04»12.2020 г. по «03»12.2021	https://urait.ru/
Договор № 027-1-3194-20 от 04.12.2020г. с ООО "Электронное издательства ЮРАЙТ" С «04»12.2020 г. по «03»12.2021 г	https://urait.ru/

Договор № 027-1-3034-21 от 03.12.2021г ООО "Электронное издательство ЮРАЙТ" С «04»12.2021 г. по «03»12.2022 г	https://urait.ru/
Договор № 150-1-3269-21 от 10.12.21 ООО "Электронное издательство ЮРАЙТ" для СПО	https://urait.ru/
Договор № 027-1-2554-22 от 01.12.2022г ООО "Электронное издательство ЮРАЙТ" С «04»12.2022 г. по «03»12.2023 г	
Договор № 5537 от 25.11.2022 ООО "Электронное издательство ЮРАЙТ" для СПО	
Электронная библиотека МАИ	
Электронная библиотека МАИ (собственность МАИ). Лицензионный договор № 0267-НИЧ-13 от 11.12.2013 г. с ООО "Дата Экспресс "на право использования программы для ЭВМ Автоматизированная интегрированная библиотечная система (АИБС) «МегаПро» (для размещения Электронной библиотеки МАИ)	https://elibrary.mai.ru/MegaPro/Web
Электронная библиотека Консорциума аэрокосмических вузов России	
Электронная библиотека Консорциума аэрокосмических вузов России. Соглашение о создании Консорциума вузов России "Национальный объединенный аэрокосмический университет" от 03.09.2012 г. Договор о сетевом взаимодействии от 15.12.2014 г. Соглашение от «03»09.2012 г. бессрочно	
Библиотека РФФИ	
Библиотека РФФИ	http://www.rfbr.ru/rffi/ru/library
Единое окно доступа к образовательным ресурсам	
Единое окно доступа к образовательным ресурсам	http://window.edu.ru/
Polpred.com	
Polpred.com. Обзор СМИ	http://polpred.com
ООО "РУНЭБ"	
Договор № 027-1-3051-20 от 07.12.2020 с ООО "РУНЭБ" С «07»12.2020 г. по «06»12.2028	http://elibrary.ru
Договор № 027-1-2895-21 от 03.12.2021 с ООО "РУНЭБ" С «03»12.2021 г. по «02»12.2039	
Договор № 027-133215-22 от 20.12.2022 с ООО "НЭБ" С «20»12.2022 г. по «19»12.2030	
ООО "Национальный цифровой ресурс "Рукоонт"	
Договор № РКТ-054/20/027-1-1129-20 от 30.05.2020 с ООО "Национальный цифровой ресурс "Рукоонт" С «01»06.2020 г. по «31»05.2021 г	http://text.rucont.ru/
Договор № 027-1-1235-21 от 01.06.2021 с ООО "Национальный цифровой ресурс "Рукоонт" С «01»06.2021 г. по «31»05.2022 г	https://text.rucont.ru/
Договор № 027-1-1467-22 от 09.06.2022 с ООО "Национальный цифровой ресурс "Рукоонт" С «01»06.2022 г. по «31»05.2023 г	https://text.rucont.ru/

ФГБУ "РГБ"	
Договор о предоставлении доступа к Национальной электронной библиотеке (НЭБ) №101/НЭБ/2139 от 13.11.2018г. с ФГБУ" РГБ" С «13»11. 2018 г. по «12» 11. 2023	http://нэб.рф

<p>Национальная подписка на 2022 г с РФФИ Государственного задания</p> <p>Springer Nature: 1. eBook Collection: журналы, книги - https://link.springer.com 2. Коллекция журналов и базы данных Springer Nature: https://link.springer.com</p> <p>Begell House Inc. https://www.dl.begellhouse.com/collections/6764f0021c05bd10.html</p> <p>China Academic Journals (CD Edition) Electronic Publishing House Co., Ltd: https://ar.cnki.net/ACADREF</p> <p>Institute of Electrical and Electronics Engineers: https://ieeexplore.ieee.org/Xplore/home.jsp; https://ieeexplore.ieee.org</p> <p>EBSCO. https://www.search.ebscohost.com/ INSPEC: 1. База данных Academic Search Premier 2. База данных eBook Academic Collection 3. eBook EngineeringCore Collection</p> <p>ORBIT Intelligence - база данных QUESTEL: https://www.orbit.com/</p> <p>SAGE https://journals.sagepub.com/</p> <p>Publication:</p> <p>Wiley: https://onlinelibrary.wiley.com/</p>	<p>https://link.springer.com</p> <p>https://www.dl.begellhouse.com/collections/6764f0021c05bd10.html</p> <p>https://ar.cnki.net/ACADREF</p> <p>https://ieeexplore.ieee.org/Xplore/home.jsp; https://ieeexplore.ieee.org</p> <p>https://www.search.ebscohost.com/</p> <p>https://www.orbit.com/</p> <p>https://journals.sagepub.com/</p> <p>https://onlinelibrary.wiley.com/</p>
---	--

8. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ

Эффективным способом развития творческих способностей студентов при изучении дисциплины является самостоятельная работа, которая нацелена на проработку студентами материала прошедших контактных занятий и подготовку к предстоящим занятиям.

Самостоятельная работа студентов проводится ими в соответствии с собственными возможностями. Можно, однако, рекомендовать групповое изучение материалов, обеспечивающее совместную работу нескольких студентов, что положительно влияет на качество проработки программы курса.

В то же время высокая степень усвоения изучаемой дисциплины достигается при постоянной работе студентов над текущим материалом. В этой связи желательна проработка лекционного материала в день его прочтения, что позволяет, во-первых, оперативно (на следующей лекции) снимать возникающие вопросы и, во-вторых, создавать багаж знаний по дисциплине задолго до промежуточной аттестации.

При подготовке к практическим занятиям также необходима проработка лекционного материала. Это позволит осознанно работать с предлагаемым материалом преподавателем на практическом занятии, а, следовательно, закладывать базу методик и приемов при решении практических задач.

При изучении материала необходимо делать акцент не на зазубривании материала, а на понимании его физической сути, что развивает мышление и позволяет понять методологию изучаемой дисциплины.

9. ПЕРЕЧЕНЬ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, ИСПОЛЬЗУЕМЫХ ПРИ ОСУЩЕСТВЛЕНИИ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

Дисциплина ориентирована на применение компьютерной техники, ресурсов информационно-телекоммуникационной сети "Интернет", электронной библиотеки МАИ для поиска, сбора, хранения, обработки и представления информации.

Программное обеспечение, Интернет-ресурсы, электронные библиотечные системы:

1. Операционная система Linux.
2. Программа-генератор компиляторов yacc или bison.
3. Программа-генератор сканеров lex или flex.
4. Отладчик gdb и графический интерфейс к нему.
5. Профайлер gprof.
6. Транслятор с языка программирования си++ (g++).
7. Система контроля версий subversion.
8. Конвертор кодировок iconv.
9. Программа управления сборки проектов make.
10. www.fepo.ru, ru.wikipedia.org, сайт кафедры МСиИТ.

10. МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ

Использование компьютерного класса для тестирования студентов и проведения лабораторных работ.

Использование проектора и экранов на лекциях и практических занятиях.

Библиотека филиала и электронная библиотека кафедры используются как источник материалов для дополнительных самостоятельных занятий.

Аннотация рабочей программы

Дисциплина "Системное программное обеспечение" является частью "Блока 1 Дисциплины" дисциплин подготовки студентов по направлению подготовки 09.03.01 "Информатика и вычислительная техника". Дисциплина реализуется на Ступино институте "Московский авиационного института (национального исследовательского университета)" кафедрой (кафедрами) МСиИТ.

Дисциплина нацелена на формирование следующих компетенций: ДПК-2, ПКР-3.

Содержание дисциплины охватывает круг вопросов, связанных с: - Изучением студентами основных теоретических положений теории компиляции и разработки ПО и практических методов использования этих положений.

- Созданием у студентов достаточно широкой подготовки в области использования различных инструментальных систем, позволяющей в дальнейшем осуществить специализацию по выбранному профилю.
- Ознакомлением студентов с основными особенностями функционирования, проектирования и создания компиляторов и другого ПО.
- Формированием у студентов научного мышления, правильного понимания оптимальности применимости различных специализированных языков программирования для разработки компиляторов и другого ПО.
- Усвоением основных понятий теории компиляторов и разработки ПО, а также овладением основными методами математического моделирования, широко применяемыми в современной технике.
- Выработкой у студентов владения приемами и навыками решения конкретных задач из разных областей программирования, помогающих в дальнейшем в решении инженерных задач по выбранной специальности.
- Способностью представить адекватную современному уровню знаний научную картину мира на основе знания основных положений, законов и методов естественных наук, математики и программирования.
- Умением использовать математический аппарат и численные методы для моделирования физико-химических процессов и явлений, лежащих в основе нанотехнологий, на вычислительной технике.

Преподавание дисциплины предусматривает следующие формы организации учебного процесса: Лекция, Самостоятельная работа, Лабораторная работа.

Программой дисциплины предусмотрены следующие виды контроля: рубежный контроль в форме и промежуточная аттестация в форме Экзамен (9 семестр).

Общая трудоемкость освоения дисциплины составляет 6 зачетных единиц, 216 часов. Программой дисциплины предусмотрены лекционные (8 часов), лабораторные (8 часов) занятия и (164 часов) самостоятельной работы студента.

Приложение 2
к рабочей программе дисциплины
«Системное программное обеспечение»

Прикрепленные файлы

КР СПО.pdf

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ.pdf

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)

Кафедра “Моделирование систем и информационные технологии”

**РАЗРАБОТКА ТРАНСЛЯТОРА ЯЗЫКА
ПРОГРАММИРОВАНИЯ НА SVN, YACC и C++**

Методические указания к курсовой работе по предмету
“Системное программное обеспечение”

Составитель В. В. Лидовский

Москва 2017

Методические указания к курсовой работе по предмету
“Системное программное обеспечение”
/ сост. В. В. Лидовский. — М.: МАИ, 2017. — с. 32.

© Лидовский В. В.,
составление, 2017

© МАИ, 2017

ВВЕДЕНИЕ

Настоящие методические указания предназначены для обеспечения учебного процесса студентов четвертого курса дневной формы обучения специальности 09.03.01 “ИВТ” при выполнении курсовой работы по предмету “Системное программное обеспечение”.

Цели курсовой работы: научиться использовать на практике программу yacc (bison) и, опционально, программу lex (flex), изучить поэтапный процесс с использованием программы Subversion создания транслятора языка программирования и разработать действующий транслятор.

Для выполнения работы требуются знания языка программирования си++ и основ разбора снизу-вверх. При использовании программы lex или flex потребуется также знакомство с простейшими регулярными выражениями.

1. СИСТЕМНЫЕ ТРЕБОВАНИЯ

Для выполнения курсовой работы необходимы следующие системные компоненты:

- а) генератор компиляторов bison или yacc;
- б) компилятор языка си++ (предпочтительно GNU g++);
- в) программа make;
- г) система версионного контроля Subversion;
- д) (опционально) генератор лексических сканеров flex/lex.

2. ПОСТАНОВКА ЗАДАЧИ

Разработать транслятор языка программирования, поддерживающего стандартную запись арифметических чисел, работу с переменными, основные операторы, процедуры и функции, а также дополнительные, определяемые вариантом задания, возможности.

Работу следует разбить на несколько этапов, начав с разработки простейших возможностей и постепенно переходя к более сложным.

3. ЭТАПЫ РАЗРАБОТКИ ТРАНСЛЯТОРА

3.1. Целочисленный калькулятор

Начнём работать с svn-репозиторием. Выписывем сначала рабочую копию файлов проекта — этот шаг всегда начинает работу при использовании нового компьютера.

```
svn co СЕРВЕР КАТАЛОГ
cd КАТАЛОГ
```

В локальный КАТАЛОГ следует записывать все файлы проекта. Адрес сервера можно узнать у администратора проекта.

Граматику языка взаимодействия с калькулятором можно описать следующим образом — приоритеты операций установим затем явно.

$$\begin{aligned} \text{list} &\rightarrow \varepsilon \mid \text{list nl} \mid \text{list expr nl} \\ \text{expr} &\rightarrow \text{number} \mid \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \mid \text{expr} * \text{expr} \mid \\ &\quad \text{expr} / \text{expr} \mid \text{expr} ^ \text{expr} \mid -\text{expr} \mid (\text{expr}) \end{aligned}$$

В этой грамматике `list` — это список вводимых команд-строк, `nl` — маркер конца строки — клавиша Enter, `expr` — вычисляемое выражение. В уасс принято терминалы обозначать отдельными знаками в апострофах или заглавными буквами, а метасимволы — строчными. Создадим файл `hos.cpp.y`, содержащий описание этой грамматики.

```
%{
#include <iostream>
#include <cctype>
#include <string>
using namespace std;
int yylex(), yyerror(const string&);
}%

%token NUMBER //лексема, терминал
%left '+' '-' //левая ассоциативность
%left '*' '/' //приоритет повышается снизу-вверх
%right '^'

%left UNARYMINUS //унарный минус, псевдотерминал
%%

list: /* пусто */
| list '\n'
| list expr '\n' { cout << $2 << endl; }
;

expr: NUMBER //{$$ = $1;} //действие по-умолчанию
| expr '+' expr {$$ = $1 + $3;}
| expr '-' expr {$$ = $1 - $3;}
| expr '*' expr {$$ = $1 * $3;}
| expr '/' expr {$$ = $1 / $3;}
| expr '^' expr {$$ = $1;
    for (int i = 1; i < $3; i++) $$ *= $1;}
| '-' expr %prec UNARYMINUS {$$ = -$2;} //установка приоритета
| '(' expr ')' {$$ = $2;}
;
%%

int lineno = 1; //for error messages

int yylex () {
```

```

int c;
while ((c = cin.get()) == ' ' || c == '\t');
if (isdigit(c)) {
    cin.unget();
    cin >> yylval;
    return NUMBER;
}
if (c == '\n')
    lineno++;
return c;
}

int yyerror(const string &s) {
    cerr << s << " in " << lineno << endl;
}

main () {
    yyparse();
}

```

Программа на yacc начинается с объявлений на языке `с++`, заключенных между `%{` и `%}`. Этот раздел может отсутствовать. Второй раздел — это правила трансляции: сначала категории лексем (токенов, `tokens`), затем приоритеты и ассоциативность, затем, между маркерами `%%`, грамматика и атрибуты. Последний раздел — это текст на `с++`. Тут должны присутствовать функции `yylex` (лексический анализатор, вызывается `yyparse`, может генерироваться программой `lex`), `yyerror` (вызывается в случае ошибки из `yyparse`) и `main`. Yacc генерирует текст функции `yyparse` и определения связанных с ней программных объектов: переменных, например, `yylval`, макросов, ... Переменная `yylval` типа `YYSTYPE` используется для передачи значения (атрибута) лексемы. Символы `$$`, `$1`, `$2`, ... — это соответственно атрибуты типа `YYSTYPE` (по умолчанию — `int`) нетерминала левой части и занумерованных с 1 компонент правой части продукции.

Приоритет терминала устанавливается вместе с ассоциативностью, порядком следования — описанные выше терминалы имеют меньший приоритет. Приоритет терминала присваивается также правилу, в котором он встречается последним. Если приоритет правила ниже, чем следующего за ним терминала, то выбирается сдвиг, а если не ниже, то свертка. Например, при разборе $5 + 3 * 2$ выбирается сдвиг, так как приоритет `*` выше, чем правила `expr → expr + expr`. Команда `%prec` позволяет явно установить приоритет правила.

Вместо функции `yylex`, если она уже сгенерирована `lex`, можно было бы использовать директиву препроцессора `#include "lex.yy.c"` —

файл `lex.yy.c` получается обработкой файла `hoc.cpp.l` с описанием лексики программой `lex`. Для реализуемого калькулятора лексика может быть описана следующими строками.

```
%option    nouwrap    //системная установка
number    [0-9]+
%%
[ ]        /* пропуск пробелов */
{number}   {istringstream s(yytext); s >> yylval; return NUMBER;}
\n         ++lineno; return '\n';
.          return yytext[0];
```

В регулярных выражениях `lex` знак точки — это любой символ, кроме конца строки. Для использования строковых потоков необходимо добавить заголовок `<sstream>`. Строка `yytext` — это входной текст для `lex`, соответствующий шаблону, регулярному выражению. Каждому шаблону сопоставляются действия. Если действия размещаются в нескольких строках, то их нужно брать в фигурные скобки. Шаблоны перебираются по-порядку — после встречи подходящего все остальные шаблоны игнорируются.

Для сборки используем `makefile`, пример которого приведён в приложение. В этом файле все отступы должны быть получены табуляцией. Команда `make` должна собрать исполнимый файл `hoc`.

Отметим теперь версионизируемые файлы.

```
svn add makefile hoc.cpp.y hoc.cpp.l
```

Имя `hoc.cpp.l` необходимо указывать, только если используется программа `lex`. Зафиксируем теперь изменения в репозитории и обновим рабочую копию. Можно использовать любые подходящие комментарии вместо преведённых.

```
svn ci -m "Сделана версия 1"
svn up
```

3.2. Поддержка вещественных чисел

Для работы с вещественными числами достаточно сделать четыре изменения.

1. Внесем в начало, между `%{` и `%}`, следующую строку, определив тип `YYSTYPE` явно.

```
#define YYSTYPE double
```

2. Изменим правило для возведения в степень на
`| expr '^' expr {$$ = pow($1, $3);}`

— воспользуемся функцией стандартной математической библиотеки.

3. Добавим заголовок `<cmath>`.

4. В `yylex` (для десятичной точки) заменим строку с `isdigit` на

```
if (c == '.' || isdigit(c)) {
```

или в файле-лексике, hoc1.cpp.l, переопределим множество number.

```
number    [0-9]+\.[0-9]*\.[0-9]+
```

Опять зафиксируем изменения в репозитории и обновим рабочую копию.

3.3. Работа с простейшими переменными

Добавим простейшую поддержку работы с переменными, именами переменных будут маленькие английские буквы от а до z, а также механизм восстановления после ошибок. Для хранения переменных введем массив — добавим между `%{` и `%}` следующую строку.

```
double vars[26];
```

Имена переменных получаются целыми числами, поэтому при синтаксическом разборе будут возникать величины типов `int` и `double`, которые передаются через `yylval`. Для реализации такого совмещения естественно использовать объединение — для этого случая предоставляется специальное описание `YYSTYPE`. Добавляем вместо `%token NUMBER` следующий текст.

```
%union {  
    double val; //значения величин  
    int index; //индекс переменной в vars  
}  
%token <val> NUMBER //лексема с типом  
%token <index> VAR  
%type <val> expr //устанавливает тип для выражений  
%right '=' //присваивание, приоритет наименьший
```

Информация в `%type` позволяет автоматически выбирать нужный тип для значения-атрибута символа грамматики.

Добавим к грамматике следующие правила.

```
| VAR {$$ = vars[$1];}  
| VAR '=' expr {$$ = vars[$1] = $3;}
```

Допустимы многократные присваивания типа `a = b = c = 5`.

К лексическому анализатору нужно добавить

```
if (islower(c)) {  
    yylval.index = c - 'a';  
    return VAR;  
}
```

и изменить ввод, прибавив обращение к соответствующему полю `yylval`,
`cin >> yylval.val;`,

или изменить lex-программу так, чтобы в ней появились следующие строки.

```
variable    [a-z]
{number}    {istream s(yytext);
             s >> yyval.val; return NUMBER;}
{variable}  yyval.index = yytext[0] - 'a'; return VAR;
```

Слово `error` зарезервировано — оно дает возможность анализатору “поймать” ошибку синтаксиса и восстановиться после нее. Добавим к грамматике следующее правило.

```
| list error {yyerrork;}
```

Действие `yyerrork` позволяет анализатору вернуться назад к возможности правильного разбора, пропустив ошибочную конструкцию. Для восстановления после ошибки, определяемой пользователем, можно использовать средства `try`, `catch` и `throw` с передачей управления на стандартную функцию `yyparse` после перехвата ошибки. Работа с ошибками — одна из наиболее сложных частей анализатора и рассматриваются только простейшие возможности.

Таким образом модифицируем файл `hos.cpp.y` и, опционально, файл описания лексики (`hos.cpp.l`). Как обычно, фиксируем изменения и обновляем рабочую копию.

3.4. Переменные, стандартные функции и константы

Добавим к калькулятору поддержку переменных, задаваемых идентификаторами, а также стандартных функций и констант. Кроме того, печатать значение выражения будем не всегда, а только когда нужно, например, не будет распечатки после присваивания. Редактируем `hos.cpp.y` и, опционально, `hos.cpp.l`.

Заменим текст после `%}` на следующий.

```
%union {
    double val;
    Symbol *sym; //указатель записи с полем-именем
}
%token <val> NUMBER
%token <sym> VAR BLTIN UNDEF //переменные, встроенные функции
                             //и константы, неопределенность
%type <val> expr assign      //устанавливает тип для выражений и
                             //присваиваний
```

Добавим в правило для `list` следующую строку, чтобы не печатать отдельные присваивания.

```
| list assign '\n'
```

Введем правило для `assign`

```
assign: VAR '=' expr {$$ = $1->val = $3; $1->type = VAR;}
;
```

и удалим соответствующую строку из правила для `expr`.

К expr-продукции добавим следующие строки вместо соответствующей строки.

```
| VAR {  
    if ($1->type == UNDEF)  
        throw "undefined variable: " + *$1->name;  
    $$ = $1->val;  
}  
| assign //для конструкций вида (a=5)*6  
| BLTIN '(' expr ')' {$$ = (*($1->fp))($3);}
```

Использование assign в expr приводит к конфликту сдвиг/свертка, но приоритет у сдвига выше, поэтому отдельные присваивания не сворачиваются к выражению.

Удалим включения заголовков и перепишем первый раздел программы.

```
%{  
#include <iostream>  
#include <cctype>  
#include <string>  
#include <sstream>  
#include <map>  
#include <cmath>  
using namespace std;  
  
int yylex(), yyerror(const string&);  
  
struct Symbol {  
    string *name;  
    short type; //VAR, BLTIN, UNDEF  
    union {  
        double val; //VAR  
        double (*fp)(double); //BLTIN  
    };  
};  
  
struct Flist {  
    string name;  
    double (*fp)(double);  
} flist [] = {"sin", sin}, {"ln", log}, {"sqrt", sqrt},  
             {"arctg", atan}};  
  
struct Clist {  
    string name;  
    double val;  
} clist [] = {"pi", 3.1415926536}, {"e", 2.7182818284},
```

```
 {"phi", 1.6180339887}}};
```

```
map<string, Symbol> names; //общая таблица имен
%}
```

В третьем разделе появится определение функции-инициализатора.

```
void init_names(Flist *p1, Clist *p2) {
    for (; p1 < flist + sizeof(flist)/sizeof(Flist); ++p1) {
        names[p1->name].name = &p1->name;
        names[p1->name].type = BLTIN;
        names[p1->name].fp = p1->fp;
    }
    for (; p2 < clist + sizeof(clist)/sizeof(Clist); ++p2) {
        names[p2->name].name = &p2->name;
        names[p2->name].type = VAR;
        names[p2->name].val = p2->val;
    }
}
```

В yylex вместо раздела с islower появится следующий раздел.

```
if (isalpha(c)) {
    char sbuf[100], *p = sbuf;
    do {
        *p++ = c;
        c = cin.get();
    }
    while (!cin.eof() && isalnum(c));
    cin.unget();
    *p = 0;
    if (names.find(sbuf) == names.end()) {
        names[sbuf].type = UNDEF;
        names[sbuf].name = &(string&)names.find(sbuf)->first;
    }
    Symbol *s = &names[sbuf];
    yylval.sym = s;
    return s->type == UNDEF ? VAR : s->type;
}
```

Можно вместо этого убрать строки с variable и добавить строку

```
symbol      [a-zA-Z][a-zA-Z0-9]*
```

и текст

```
{symbol} {
    string sbuf(yytext);
    if (names.find(sbuf) == names.end()) {
        names[sbuf].type = UNDEF;
        names[sbuf].name = &(string&)names.find(sbuf)->first;
    }
}
```

```

    Symbol *s = &names[sbuf];
    yyval.sym = s;
    return s->type == UNDEF ? VAR : s->type;
}

```

к файлу с лексикой.

Функция `main` с поддержкой генерируемых исключений примет следующий вид.

```

main () {
    init_names(flist, clist);
    while (1) {
        try {
            yyparse();
            break;
        }
        catch (string s) {
            cerr << s << endl;
        }
    }
}

```

Фиксируем изменения в `svn`.

3.5. Генерация кода

Чтобы ввести поддержку работы с операторами и определяемыми пользователем подпрограммами можно использовать два подхода.

1. Реализовать макроподстановки, т. е. сделать язык, похожий на препроцессор `си/си++`. Такой подход используется также в языках `TeX`, `MetaFont` и некоторых других. Он требует значительного усложнения лексического сканера.

2. Реализовать компилятор в простой стековый код и интерпретатор этого кода. В `пи`-системах, `яве`, платформе `.NET` подобный подход приводит к генерации отдельных файлов с независимыми от аппаратуры кодами, которые потом исполняют специальные, относительно простые интерпретаторы. Во многих языках сценариев (`оук`, `перл`, `питон`, `рубин` и др.) подобный код генерируется по ходу исполнения программы и тут же выполняется интерпретатором.

Трансформируем программу-калькулятор так, чтобы введенные пользователем строки сначала переводились в стековый код, который затем исполняется. Например, выражение `x=4+y` трансформируется в код

<code>constpush</code>	записать в стек константу
<code>4</code>	
<code>varpush</code>	записать в стек переменную
<code>y</code>	символ <code>y</code>

eval	вычислить значение символа
add	сложить числа на вершине стека
varpush	
x	
assign_code	присвоить символу на вершине стека следующее значение за ним в стеке значение
pop_code	убрать элемент с вершины стека
STOP	маркер конца программы, останавливает интерпретатор

Программа становится большой и ее поэтому лучше разделить на части. Сначала изменяем файл `hoc.cpp.y` — существенно меняются лишь атрибуты грамматики, а также обработка лексем-чисел. Числа теперь заносятся в таблицу символов, что позволяет хранить одинаковые числа-константы в одном месте памяти. Вместо стандартной обработки ошибок через `errgr-errgrk`, изменив `ууerrgr`, перейдем к полностью контролируемой пользователем обработке ошибок. В `main` теперь будут друг за другом запускаться синтаксический анализатор, управляющий компиляцией, и интерпретатор.

```
%{
#include <cctype>
#include <sstream>
#include <map>
#include "hoc.h"
#define code2(c1,c2) code(c1);code(c2)
#define code3(c1,c2,c3) code(c1);code(c2);code(c3)
}%
%union {
    Symbol *sym;    //указатель записи
    Inst *inst;     //указатель команды стекового кода
}
%token <sym> VAR BLTIN UNDEF NUMBER
%right '='
%left '+' '-'
%left '*' '/'
%right '^'
%left UNARYMINUS
%%
list:
| list '\n'
| list assign '\n' {code2(pop_code, STOP); return 1;}
| list expr '\n' {code2(print, STOP); return 1;}
;
assign: VAR '=' expr {code3(varpush, (Inst) $1, assign_code);}
;
```

```

expr: NUMBER {code2(constpush, (Inst) $1);}
| VAR {code3(varpush, (Inst) $1, eval);}
| assign
| BLTIN '(' expr ')' {code2(bltin, (Inst)$1->fp);}
| expr '+' expr {code(add);}
| expr '-' expr {code(sub);}
| expr '*' expr {code(mul);}
| expr '/' expr {code(div);}
| expr '^' expr {code(power);}
| '-' expr %prec UNARYMINUS {code(negate_code);}
| '(' expr ')'
;
%%

```

Определения `lineno`, `Flist`, `Clist`, `init_names` остаются такими же.

Добавляем к `yulex` новый раздел для работы с числами вместо прежнего.

```

if (c == '.' || isdigit(c)) {
    cin.unget();
    double w;
    cin >> w;
    ostringstream oss;
    oss << w;
    string s(oss.str());
    if (names.find(s) == names.end()) {
        names[s].type = NUMBER;
        names[s].name = &(string&)names.find(s)->first;
        names[s].val = w;
    }
    yylval.sym = &names[s];
    return NUMBER;
}

```

Можно вместо этого подправить описание лексики.

```

{number} {
    istringstream iss(yytext);
    double w;
    iss >> w;
    string s(yytext);

```

Функции `yuerrog` и `main` изменяются незначительно.

```

int yyerror(const string &s) {
    ostringstream oss;
    oss << s << " in " << lineno << endl;
    throw oss.str();
}

```

```

main () {
    init_names(flist, clist);
    while (1) {
        try {
            for (initcode(); yyparse(); initcode())
                execute(prog);
            break;
        }
        catch (string s) {
            cerr << s << endl;
        }
    }
}

```

Затем создадим файл code.cpp с реализацией машины-интерпретатора. Программа записывается в массив prog и при исполнении использует стек на основе массива stack.

```

#include "hoc.h"
#include "y.tab.h"

#define NSTACK 5000 //размер стека
static Datum stack[NSTACK]; //защита абстракции -
                             //модульной видимостью
static Datum *stackp; //указатель вершины стека

#define NPROG 2000 //макс. размер программы
Inst prog[NPROG]; //память
Inst *progp; //первое свободное место для создаваемого кода
Inst *pc; //program counter, счетчик команд при выполнении

void initcode() {
    stackp = stack;
    progp = prog;
}

Inst* code(Inst f) { //занести одну команду в память
    Inst *saved_progp = progp;
    if (progp >= prog + NPROG)
        throw (string) "program too big";
    *progp++ = f;
    return saved_progp;
}

void execute(Inst *p) { //выполнить команды
    for (pc = p; *pc != STOP;)
        ((*pc++))(); //это весь интерпретатор!
}

```

```

}

//работа со стеком
void push(Datum d) {
    if (stackp >= stack + NSTACK)
        throw (string) "stack overflow";
    *stackp++ = d;
}

Datum pop() {
    if (stackp <= stack)
        throw (string) "stack underflow";
    return *--stackp;
}

//команды для интерпретатора
void pop_code() {
    pop();
}

void constpush() { //записать константу в стек
    Datum d;
    d.val = ((Symbol*) *pc++)->val;
    push(d);
}

void varpush() { //записать переменную в стек
    Datum d;
    d.sym = (Symbol*) *pc++;
    push(d);
}

void sub() { //вычитание
    Datum d1 = pop(), d2 = pop();
    d2.val -= d1.val;
    push(d2);
}

    Аналогично реализуются add, mul, div, power и negate_code.
void bltin() {
    Datum d = pop();
    d.val = (*(double*)(double))(*pc++)(d.val);
    push(d);
}

void assign_code() {
    Datum d1 = pop(), d2 = pop();

```

```

    d1.sym->val = d2.val;
    d1.sym->type = VAR;
    push(d2);
}

void eval() {
    Datum d = pop();
    if (d.sym->type == UNDEF)
        throw "undefined variable: " + *d.sym->name;
    d.val = d.sym->val;
    push(d);
}

void print() {
    Datum d = pop();
    cout << d.val << endl;
}

```

Значение функции code в данной версии программы не используется. Для согласования файлов создадим файл-заголовок hос.h с общими и предварительными объявлениями.

```

#include <iostream>
#include <string>
#include <cmath>
using namespace std;

struct Symbol {
    string *name;
    short type; //VAR, BLTIN, UNDEF, NUMBER
    union {
        double val; //VAR, NUMBER
        double (*fp)(double); //BLTIN
    };
};

int yylex(), yyparse(), yyerror(const string &);

union Datum { //тип стека интерпретатора
    double val;
    Symbol *sym;
};

typedef void (*Inst)();
#define STOP 0
extern Inst prog[];

Datum pop();

```

```

Inst* code(Inst);
void initcode(), push(Datum), pop_code(), execute(Inst*),
    constpush(), varpush(), add(), sub(), mul(), div(), power(),
    negate_code(), bltin(), assign_code(), eval(), print();

```

Более сложный процесс сборки программы нужно отразить в файле makefile. Цель clean позволяет, вызвав make clean, очистить каталог от временных файлов.

```

hoc: hoc.cpp.y hoc.h code.cpp makefile
    -lex hoc.cpp.l
    yacc -d hoc.cpp.y
    g++ -c code.cpp
    g++ -c y.tab.c
    g++ y.tab.o code.o -o hoc

```

clean:

```
rm -f y.tab.[ch] lex.yy.c hoc *.o
```

Ключ -d приводит yacc к генерации файла y.tab.h, содержащего определения генерируемых yacc символов. Ключ -v можно использовать для распечатки LALR(1)-множеств пунктов с действиями, переходами и отметками конфликтов, а также суммарной справочной информации. Вместо yacc можно вызывать bison с ключом -y.

Необходимо добавить файлы code.cpp и hoc.h к версионизируемым файлам проекта, зафиксировать изменения в репозитории и обновить рабочую копию.

3.6. Операторы

Добавим к языку поддержку операторов if, while, печати и составного, а также 6 операций сравнения и 3 логических.

```

%token <sym> VAR BLTIN UNDEF NUMBER PRINT WHILE IF ELSE
%type <inst> assign expr oper operlist while if cond endop
%right '='
%left OR
%left AND
%left GT GE LT LE EQ NE  //> >= < <= == !=
%left '+' '-'
%left '*' '/'
%right '^'
%left NOT

```

Добавим к list-продукции информацию об операторах.

```
| list oper '\n' {code(STOP); return 1;}
```

Оператор и связанные с ним понятия определяются правилами.

```

oper: expr {code(pop_code);} //$$=$1 происходит автоматически
| PRINT expr {code(prexpr); $$ = $2;}

```

```

| while cond oper endop {
    ($1)[1] = (Inst) $3; //переход на тело цикла
    ($1)[2] = (Inst) $4;} //переход на выход из цикла
| if cond oper endop { //if без else
    ($1)[1] = (Inst) $3; //переход на часть после then
    ($1)[3] = (Inst) $4;} //переход на следующий оператор
| if cond oper endop ELSE oper endop { //if с else
    ($1)[1] = (Inst) $3;
    ($1)[2] = (Inst) $6; //переход на else-часть
    ($1)[3] = (Inst) $7;}
| '{' operlist '}' {$$ = $2;}
;
cond: '(' expr ')' {code(STOP); $$ = $2;}
;
while: WHILE {$$ = code3(while_code, STOP, STOP);}
        //STOP - резервирование места для перехода
;
if: IF {$$ = code(if_code); code3(STOP, STOP, STOP);}
;
endop: {code(STOP); $$ = prog;}
;
operlist: {$$ = prog;}
| operlist '\n'
| operlist oper
;

```

К assign-продукции добавим оператор $$$ = \3 ; и внесём следующие изменения в expr-продукцию.

```

NUMBER {$$ = code2(constpush, (Inst) $1);}
| VAR {$$ = code3(varpush, (Inst) $1, eval);}
| BLTIN '(' expr ')' {$$ = $3; code2(bltin, (Inst)$1->fp);}
| '-' expr %prec NOT {$$ = $2; code(negate_code);}
| '(' expr ')' {$$ = $2;}

```

Эти изменения вызваны необходимостью отслеживать адрес генерируемых выражений. Сюда также добавляются строки с определениями новых операций.

```

| expr GT expr {code(gt);}
| expr GE expr {code(ge);}
| expr LT expr {code(lt);}
| expr LE expr {code(le);}
| expr EQ expr {code(eq);}
| expr NE expr {code(ne);}
| expr AND expr {code(and_code);}
| expr OR expr {code(or_code);}
| NOT expr {$$ = $2; code(not_code);}

```

Лексический анализатор потребует вспомогательной функции.

```
int follow(int expect, int ifyes, int ifno) {
    int c = cin.get();
    if (c == expect)
        return ifyes;
    cin.unget();
    return ifno;
}
```

Добавим к ууlex следующий новый раздел.

```
switch (c) {
    case '>': return follow('=', GE, GT);
    case '<': return follow('=', LE, LT);
    case '!': return follow('=', NE, NOT);
    case '=': return follow(c, EQ, c);
    case '|': return follow(c, OR, c);
    case '&': return follow(c, AND, c);
}
```

Вместо этого можно добавить к hoc.cpp.l следующие строки.

```
>=    return GE;
\<=   return LE;
==     return EQ;
!=     return NE;
>      return GT;
\<     return LT;
\\||   return OR;
&&     return AND;
!      return NOT;
```

Нужно добавить служебные слова к списку символов. Добавим массив данных

```
struct Klist {
    string name;
    int kval;
} klist [] = {"if", IF}, {"else", ELSE}, {"while", WHILE},
{"print", PRINT}};
```

и соответствующий цикл к initnames, у которой появится третий аргумент (p3).

```
for (; p3 < klist + sizeof(klist)/sizeof(Klist); ++p3) {
    names[p3->name].name = &p3->name;
    names[p3->name].type = p3->kval;
}
```

В main вызов этой функции должен быть дополнен третьим аргументом — klist.

Добавим к hoc.h объявления следующих новых функций.

```
void gt(), ge(), lt(), le(), eq(), ne(), and_code(), or_code(),
not_code(), while_code(), if_code(), preexpr();
```

Также изменим следующую строку.

```
extern Inst prog[], *progp;
```

Добавим в конец code.cpp определения новых функций.

```
void gt() { //greater than
    Datum d1 = pop(), d2 = pop();
    d2.val = (d2.val > d1.val);
    push(d2);
}
```

```
//Функции ge, lt, le, eq и ne
```

```
void and_code() {
    Datum d1 = pop(), d2 = pop();
    d2.val = (d2.val && d1.val);
    push(d2);
}
```

```
//Функции or_code и not_code
```

```
void preexpr() {
    Datum d = pop();
    cout << d.val << endl;
}
```

```
void while_code() {
    Datum d;
    Inst *savepc = pc;
    execute(pc + 2); //условие
    d = pop();
    while (d.val) {
        execute(*(Inst**)savepc); //тело
        execute(savepc + 2); //условие
        d = pop();
    }
    pc = *(Inst**)(savepc + 1); //следующий оператор
}
```

```
void if_code() {
    Inst *savepc = pc;
    execute(pc + 3); //условие
    Datum d = pop();
    if (d.val)
```

```

    execute(*(Inst**)savepc); //then-часть
else if (*(savepc + 1)) //есть ли else
    execute(*(Inst**)(savepc + 1)); //else-часть
pc = (*(Inst**)(savepc + 2)); //следующий оператор
}

```

Разберем исполнение while_code (if_code — аналогичен). Код для этого оператора имеет следующий вид.

```

        НАЧАЛО - КОД while_code
pc      ПЕРЕХОД НА ТЕЛО ЦИКЛА
pc + 1  ПЕРЕХОД НА ВЫХОД
pc + 2  УСЛОВИЕ
        ТЕЛО

```

Для генерации кода этого оператора в память записываются три слова: собственно код и два приведенных перехода. Используется рекурсивный вызов execute — каждая часть оператора (условие и тело) заканчиваются STOP.

Рассмотрим генерацию кода для выражения if(a>3)print a else a=7.

```

if_code
адрес позиции L1
адрес позиции L2
адрес позиции L3
varpush
a
eval
constpush
3
gt
STOP
L1  varpush
    a
    eval
    prexpr
    STOP
L2  constpush
    7
    varpush
    a
    assign_code
    pop_code
    STOP
L3

```

Скорректируем makefile и проведём сборку hoc. Затем, например, можно вычислить факториал 7.

```
f=i=1
while(i<7){i=i+1 f=f*i}
f
```

Цикл можно было бы записать более кратко, например, `while(i<7) f=f*(i=i+1)`.

В заключении этапа работаем с Subversion.

3.7. Определяемые процедуры и функции

Добавим поддержку подпрограмм, чтобы можно было, например, определить

```
func fac() {if($1<2)return 1 else return $1*fac($1-1)}
proc hello() while($1>0){print $1," Hello\n" $1=$1-1}
```

и вызвать `hello(fac(7))`. Добавим также поддержку строк, расширим возможности оператора `print` аргументом-списком и добавим оператор ввода данных.

Изменим тип `yylval`.

```
%union {
    Symbol *sym;
    Inst *inst;
    int nargs; //число аргументов подпрограммы, long для x86_64
}
```

Появятся новые категории лексем

```
%token <sym> STRING FUNCCALL PROCCALL RETURN FUNCDEF PROCDEF
%token <sym> READ
%token <narg> ARG
```

и новые данные.

```
%type <inst> prlist begin
%type <sym> subrname
%type <narg> arglist
```

В `list`-правиле появится строка определения новой подпрограммы.

```
| list deffn '\n'
```

В `assign`-продукции появится случай для работы с параметрами — они могут использоваться как локальные переменные и им можно присваивать значения. Параметры именуются `$1`, `$2`, ...

```
| ARG '=' expr {
    defonly("$"); code2(argassign, (Inst) $1); $$ = $3;}

```

В `opreg`-продукции изменится правило с `PRINT` на

```
| PRINT prlist {$$ = $2;}

```

— теперь печатать можно не только отдельные выражения, но и списки из выражений и строк. Сюда также будут добавлены правила для возврата из подпрограмм и вызова процедур.

```
| RETURN {defonly("return"); code(procret);}
| RETURN expr {defonly("return"); code(funcrct); $$ = $2;}
| PROCCALL begin '(' arglist ')' {
    $$ = $2;
    code3(call, (Inst) $1, (Inst) $4);}

```

Появятся продукции для begin, prlist, deffn, subname (устанавливает, что именем подпрограммы может быть имя переменной или подпрограммы, т.е. эти объекты можно переопределять) и arglist (подсчет числа аргументов подпрограммы).

```
begin: {$$ = progp;}
;
prlist: expr {code(prexp);}
| STRING {$$ = code2(prstr, (Inst) $1);}
| prlist ',' expr {code(prexp);}
| prlist ',' STRING {code2(prstr, (Inst) $3);}
;
deffn: FUNCDEF subname {$2->type = FUNCCALL; indef = 1;}
    '(' ')' oper {
        code3(constpush, (Inst)&names.begin()->second,
                                                    funcrct);
        define($2);
        indef = 0;}
| PROCCDEF subname {$2->type = PROCCALL; indef = 1;}
    '(' ')' oper {code(procret); define($2); indef = 0;}
;
subname: VAR
| FUNCCALL
| PROCCALL
;
arglist: {$$ = 0;}
| expr {$$ = 1;}
| arglist ',' expr {$$ = $1 + 1;}
;

```

В expr-продукции появятся варианты для аргумента подпрограммы (можно использовать только внутри подпрограммы), вызова функции и операции ввода данных в заданную переменную со стандартного потока ввода.

```
| ARG {defonly("$"); $$ = code2(arg, (Inst) $1);}
| FUNCCALL begin '(' arglist ')' {
    $$ = $2;
    code3(call, (Inst) $1, (Inst) $4);}
| READ '(' VAR ')' {$$ = code2(varread, (Inst) $3);}

```

В список (массив klist) ключевых слов нужно добавить следующие.

```
{"func", FUNCDEF}, {"proc", PROCDEF}, {"return", RETURN},
{"read", READ}
```

Для работы со стандартными escape-последовательностями языка си добавим перед yylex функцию backslash.

```
int backslash(int c) {
    static string transtab = "b\\bf\\fn\\nr\\rt\\t";
    if (c != '\\') return c;
    c = cin.get();
    if (transtab.find(c) != string::npos)
        return transtab[transtab.find(c) + 1];
    return c;
}
```

В yylex появится раздел для работы со строками, последовательностями знаков в кавычках, и раздел для работы с аргументами подпрограмм.

```
if (c == '$') {
    int n = 0;
    while (isdigit(c = cin.get()))
        n = n*10 + c - '0';
    cin.unget();
    if (n == 0)
        throw (string) "strange $...";
    yylval.narg = n;
    return ARG;
}
if (c == '"') { //строка в кавычках
    char sbuf[100], *p;
    for (p = sbuf; (c = cin.get()) != '"'; p++) {
        if (c == '\\n')
            throw (string) "missing quote";
        if (p > sbuf + sizeof(sbuf)) {
            *p = 0;
            throw "string too long " + string(sbuf);
        }
        *p = backslash(c);
    }
    *p = 0;
    yylval.sym = new Symbol;
    yylval.sym->name = new string(sbuf);
    return yylval.sym->type = STRING;
}
```

Если использовать lex/flex, то нужно добавить к файлу с описанием лексики аналогичные разделы.

```

\[0-9]+  {
    istringstream iss(yytext + 1);
    int n;
    iss >> n;
    if (n == 0)
        throw (string) "strange $...";
    yylval.narg = n;
    return ARG;
}
\[^\"]*\" {
    string istr(yytext + 1), so("\\b\\n\\r\\t\\f"),
        si[] = {"\\b", "\\n", "\\r", "\\t", "\\f"};
    istr.erase(istr.length() - 1);
    int i, p;
    for (i = 0; i < so.length(); i++)
        while ((p = istr.find(si[i])) != string::npos)
            istr.replace(p, 2, so.substr(i, 1));
    yylval.sym = new Symbol;
    yylval.sym->name = new string(istr);
    return yylval.sym->type = STRING;
}

```

В code.cpp появятся определения новых переменных. Возврат из подпрограмм организуем не синтаксически, а семантически, при помощи returning.

```

Inst *progbase = prog; //начало текущей подпрограммы
int returning; //1, если находимся в режиме возврата
static Datum *bp; //указатель базы локальных переменных

```

В initcode добавятся следующие строки.

```

fp = frame;
returning = 0;
progp = progbase; //вместо progp = prog;

```

Незначительно изменится цикл в execute.

```

for (pc = p; *pc != STOP && !returning;
    (*(pc++))());

```

Изменения, связанные с returning, произойдут в while_code.

```

while (d.val) {
    execute(*(Inst**)savepc); //тело
    if (returning)
        break;
    execute(savepc + 2); //условие
    d = pop();
}
if (!returning)

```

```
    pc = *(Inst**)(savepc + 1); //следующий оператор
и в if_code.
```

```
    if (!returning)
        pc = *(Inst**)(savepc + 2); //следующий оператор
```

Теперь при печати с помощью print не нужно всегда печатать переход на новую строку — убираем endl из prxprg. Для печати строк понадобится новая функция.

```
void prstr() { //печать строки
    cout << *(((Symbol*) *pc++)->name);
}
```

Определим функции для работы с подпрограммами.

```
void define(Symbol *sp) {
    sp->deffn = (Inst) progbase;
    progbase = prog;
}
```

```
void call() { //вызов подпрограммы
    Datum d;
    d.sym = (Symbol*)(pc + 2); //адрес возврата
    push(d);
    d.sym = (Symbol*)bp; //адрес базы локальных параметров
    push(d);
    bp = stackp; //новое значение базы
    execute((Inst*)((Symbol*)*pc->deffn)); //обращение к
        //записи в таблице имен для вызова подпрограммы
    returning = 0;
}
```

```
void ret() { //общий возврат из подпрограммы
    Datum d = pop();
    bp = (Datum*) d.sym;
    d = pop();
    pc = (Inst*)d.sym;
    stackp -= (long)*(pc - 1); //удаление аргументов
    returning = 1;
}
```

```
void funcrct() { //возврат из функции
    Datum d = pop(); //значение функции
    Symbol *sp = *((Symbol**)(stackp - 2) - 2);
    //обращение к таблице имён через сохранённый pc
    if (sp->type == PROCCALL)
        throw *sp->name + " (proc) returns value";
    ret();
}
```

```

    push(d);
}

void procret() { //возврат из процедуры
    Symbol *sp = (*(Symbol**)(stackp - 2) - 2);
    if (sp->type == FUNCCALL)
        throw *sp->name + " (func) returns no value";
    ret();
}

double *getarg() { //возвращает указатель на аргумент
    int nargs = (long)*pc++; //номер аргумента
    long nargs = (long)*(Symbol**)(bp - 2) - 1);
        //обращение к коду через bp и сохранённый pc
    if (narg > nargs)
        throw (*(Symbol**)(bp - 2) - 2)->name
            + " not enough arguments";
    return &(bp - nargs + nargs - 3)->val;
}

void arg() { //перенести аргумент в стек операций
    Datum d;
    d.val = *getarg();
    push(d);
}

void argassign() { //записать значение с вершины стека
    Datum d; //в аргумент
    push(d = pop());
    *getarg() = d.val;
}

```

Определим также функцию для ввода данных.

```

void varread() { //считывает значения для переменных
    Datum d;
    Symbol *var = (Symbol*) *pc++;
    if (!(cin >> var->val))
        throw "non-number read into " + *var->name;
    d.val = var->val;
    var->type = VAR;
    push(d);
}

```

Отразим в `hoc.h` изменения в текстах программы. В определении `Symbol` появится поле `deffn`.

```

void (*deffn)(); //FUNCDEF, PROCDEF

```

Декларация внешних переменных приобретет следующий вид.

```
extern Inst prog[], *progp, *progbase;  
extern int indef;
```

К списку функций нужно добавить следующие.

```
defonly(const string&), define(Symbol*), call(), procret(),  
funcrct(), arg(), argassign(), prstr(), varread();
```

Корректируем makefile и запускаем сборку. Проверяем, набрав в файл fib.hoc текст

```
func fib() {  
    if ($1 < 3) return 1 else return fib($1 - 1) + fib($1 - 2)  
}  
fib(32)
```

и запустив hoc<fib.hoc с командой строки.

Подсчет числа аргументов у подпрограмм вводится только для контроля за стеком, например, для отслеживания случаев, когда внутри подпрограммы запрашивается аргумент с номером, большим общего числа аргументов.

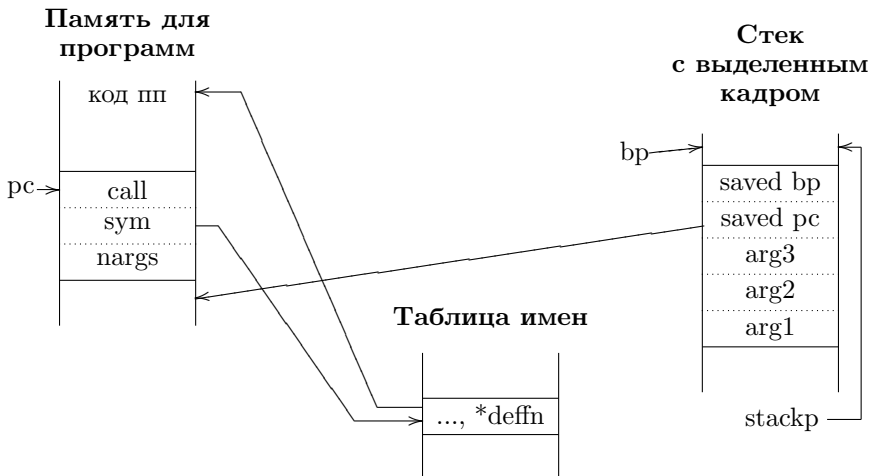
Для организации вызовов подпрограмм с параметрами в стеке создается специальный кадр подпрограммы. В кадре хранятся адрес возврата, адрес базы параметров, указатель на запись в таблице имен для подпрограммы (для лучших сообщений об ошибке) и число аргументов при вызове (для означенного ранее контроля). Аргументы и результат функции также сохраняются в стеке.

В deffn-продукции использовалась возможность помещать семантику в середину синтаксического правила. Это использовалось для проверки допустимости синтаксических конструкций, возможных только внутри подпрограмм: работа с параметрами и возврат. Функция defonly (только в определении) генерирует исключение при встрече таких конструкций вне тела подпрограмм. Работа defonly основана на переменной indef — она устанавливается в 1 вначале определения и в 0 в конце. Это случай, когда правильность конструкции проще проверить через семантику, а не синтаксис. Использование переменной returning также упрощает синтаксис — после ее установки все операторы до конца подпрограммы пропускаются без исполнения.

Подпрограммы заносятся в начало массива prog, вызывая смещение начала кода для текущих выражений (progbase), поэтому в вызове execute в main нужно заменить параметр. При переопределении подпрограмм память от старого кода не освобождается. При сбросе синтаксического анализатора, например, при ошибке, подпрограммы остаются в памяти.

Процесс вызова подпрограммы с тремя аргументами можно изобразить приведённой схемой.

Проверочные примеры можно добавить к проекту. Фиксируем изменения и обновляем рабочую копию. Отчет о проведенных фиксациях можно получить командой `svn log`.



4. ПОРЯДОК ВЫПОЛНЕНИЯ И ОТЧЕТ ПО КУРСОВОЙ РАБОТЕ

4.1. Порядок выполнения курсовой работы

После каждого этапа выполнения работы следует проводить фиксацию изменений. На сложных этапах (операторы, подпрограммы) можно проводить дополнительные фиксации промежуточных результатов.

После подготовки всех необходимых на текущем этапе текстов программ следует при помощи `make` собрать соответствующий этап у транслятор, а затем провести его тестирование.

4.2. Отчет по курсовой работе

Отчет по работе выполняется на отдельных бланках или листах и должен содержать:

- 1) цель исследования;
- 2) краткую характеристику реализованного языка;
- 3) листинг программ;
- 4) описание части программы, реализующей определенное задание (изменения в базовой программе, новые типы, переменные, подпрограммы и т. п.);
- 5) выводы.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Назначение созданного транслятора.
2. Характеристики программы yacc/bison (lex/flex).
3. Способы реализации операторов и подпрограмм.
4. Как реализовать поддержку стандартной функции \log_2 — двоичного логарифма?
5. Как добавить поддержку новой стандартной константы?
6. Как добавить поддержку простейших, задаваемых цепочкой строчных английских букв переменных, в имени у которых значимыми являются только первые два символа, к hoc версии 2?
7. Сформулируйте как можно больше конкретных расширений для hoc.
8. Какие массивы проще для реализации на уровне поддержки синтаксиса обычные в стиле си или ассоциативные? Поясните примерами.
9. Чем текстовые файлы для make принципиально отличаются от большинства тестовых форматов на основе чистого текста (plain text)?
10. В чем недостатки стандартного способа контроля за ошибками?
11. Как зависит скорость поиска символа в таблице имен компилятора от размера этой таблицы: линейно, логарифмически, экспоненциально, никак? Эффективно ли таблица имен использует память?
12. Разобрать исполнение if_code.
13. Описать процесс возврата из подпрограммы.
14. Почему присваивание не всегда сворачивается к выражению?
15. Построить дерево разбора для заданной синтаксической конструкции hoc.
16. Написать код, генерируемый при разборе заданной синтаксической конструкции.
17. Какое значение возвращается при обращении к функции, не возвращающей значения?

6. ВАРИАНТЫ РАБОТ

Курсовой работа по теме “Разработка транслятора языка программирования на SVN, Yacc и C++” имеет 16 вариантов заданий, отличающихся друг от друга расширениями к описанному языку hoc:

1. Добавить поддержку си-оператора do и поддержку комментариев `si++`.
2. Реализовать поддержку си-оператора for, используя скобки вместо точек с запятой, например, `for(i=1(i<10)i=i+1)`.

3. Добавить поддержку си-операций инкремента и декремента как постфиксной, так и в префиксной формах.
4. Реализовать составные присваивания в стиле си, а именно $+=$, $-=$, $*=$, $/=$, $\wedge=$.
5. Реализовать поддержку операторов си break и continue.
6. Исправить операции $\&\&$ и $||$, сделав так, чтобы их второй аргумент вычислялся только в случае необходимости.
- 7-8. Добавить поддержку локальных переменных со статическим/автоматическим распределением памяти, используя служебное слово static/auto.
9. Реализовать поддержку многомерных ассоциативных массивов с возможностью удаления отдельных элементов, отдельных размерностей и всего массива в целом.
10. Реализовать операторы unless (отличается от if только инвертированием условия и отсутствием else-альтернативы) и until (отличается от while только инвертированием условия).
11. Добавить поддержку констант, реализовав конструкцию инициализации со служебным словом const. Встроенные переменные превратить в константы.
12. Добавить распечатку получаемого после компиляции кода в виде листинга мнемокодов.
13. Добавить распечатку выполняющихся инструкций мнемокодами.
14. Реализовать очистку памяти от ставших ненужными символьных строк.
15. Реализовать освобождения памяти от старого кода при переопределении подпрограмм.
16. Сделать размер стека выражений или массива для программ динамическим.

ЛИТЕРАТУРА

1. Ахо А., Сети Р., Ульман Д. Компиляторы / Ахо А., Сети Р., Ульман Д. — М., СПб., Киев: Вильямс, 2003.
2. Керниган Б. В. , Пайк Р. Unix — универсальная среда программирования / Керниган Б. В. , Пайк Р. — М.: Финансы и статистика, 1992.
3. Страуструп Б. Язык программирования C++ / Страуструп Б. — М.: Binom Publisher, СПб: Невский диалект, 1999.

ПРИЛОЖЕНИЕ 1

Пример файла makefile для начальной версии программы

```
hoc: hoc.cpp.y hoc.cpp.l makefile
    -lex hoc.cpp.l
    yacc hoc.cpp.y
    g++ -o hoc y.tab.c

hoc.cpp.l:
    echo >hoc.cpp.l

clean:
    rm -f y.tab.c lex.yy.c hoc
```

ПРИЛОЖЕНИЕ 2

Некоторые дополнительные команды системы Subversion

svn st	— файлы, изменившиеся с последней фиксации (опция -q скрывает неверсионированные файлы)
svn log	— журнал фиксаций
svn diff ФАЙЛ	— показ результатов текущего редактирования по файлу
svn diff	— показ результатов текущего редактирования по всем версионированным файлам
svn revert ФАЙЛ	— отказ от всех изменений с момента последней фиксации в заданном файле
svn cat ФАЙЛ -r ВЕРСИЯ	— печать содержимого файла заданной версии
svn ps svn:keywords Id ФАЙЛ	— установка поддержки раскрытия значения \$Id\$ в заданном файле
svn ren ФАЙЛ1 ФАЙЛ2	— версионированное переименование файла
svn list	— содержимое репозитория, соответствующего рабочей копии
svn list СЕРВЕР	— содержимое репозитория на заданном сервере
svn del ФАЙЛ	— версионированное удаление файла
svn info ФАЙЛ	— информация о текущем репозитории

ОГЛАВЛЕНИЕ

Введение	3
1. Системные требования	3
2. Постановка задачи	3
3. Этапы разработки транслятора	3
3.1. Целочисленный калькулятор	3
3.2. Поддержка вещественных чисел	6
3.3. Работа с простейшими переменными	7
3.4. Переменные, стандартные функции и константы	8
3.5. Генерация кода	11
3.6. Операторы	17
3.7. Определяемые процедуры и функции	22
4. Порядок выполнения и отчет по курсовой работе	29
4.1. Порядок выполнения курсовой работы	29
4.2. Отчет по курсовой работе	29
5. Контрольные вопросы	30
6. Варианты работ	30
Литература	31
Приложение 1	32
Приложение 2	32

Владимир Викторович Лидовский

**РАЗРАБОТКА ТРАНСЛЯТОРА
ЯЗЫКА ПРОГРАММИРОВАНИЯ НА SVN, YACC и C++**

Методические указания к курсовой работе по предмету
“Системное программное обеспечение”

Ведущий редактор *О.В. Филиппова*

Технический редактор *О.В. Филиппова*

Оригинал-макет подготовлен в пакетах Plain-TeX и Xy-pic

Подписано в печать ??.2017 Формат 60x84/16
Печать на ризографе. Усл. печ. л. 2. Уч.-изд. л. 2,19
Тираж ? экз. Заказ № ?

Издательско-типографский центр МАИ
Москва

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ: ВОПРОСЫ К ЭКЗАМЕНУ

- 1) Компьютерная обработка текстов. Таблицы кодировки 7/8-битные.
 - 2) Транслитерация кириллицы латиницей. Unicode, UCS, UTF-7/8/16.
 - 3) Способы задания языка. Грамматики. Выводимая цепочка. НФБН.
 - 4) Вывод цепочки. Левый (правый) вывод. Однозначность грамматики. Дерево разбора.
 - 5) Постфиксная (обратная польская) запись. Левая рекурсия и ее устранение. Рекурсивный спуск. Пример компилятора рекурсивного спуска.
 - 6) Иерархия Хомского. Способы задания регулярных языков. Регулярные множества.
 - 7) КА. ДКА и НКА. Преобразование НКА в ДКА.
 - 8) Минимизация КА. Лемма о разрастании для регулярных языков. Программа Lex/Flex. Свойства регулярных языков. Регулярные множества.
 - 9) Стековый (магазинный) КА. Контекстно-свободные языки. Построение НМКА по заданной грамматике и наоборот. Лемма о разрастании для КС-языков.
 - 10) Преобразование КС-языков. Удаление бесполезных символов и правил. Удаление ϵ -правил. Устранение циклов и цепных правил.
- Устранение
левой рекурсии.
- 11) Нормальные формы Хомского и Грейбаха для КС-языков. Назначение алгоритмов Кока-Янгера-Касами и Ерли, их свойства. $LL(k)$ -языки и грамматики.
 - 12) Множества FIRST и FOLLOW. Алгоритм работы анализатора. Пример $LL(1)$ -разбора.
 - 13) Разбор снизу-вверх. Сдвиг-свертка. Грамматики простого и операторного предшествования.
 - 14) $LR(k)$ -языки и грамматики. Схема работы $LR(k)$ -анализатора.
 - 15) Пример $LR(1)$ -разбора. $SLR(1)$ и $LALR(1)$ языки.
 - 16) Построение канонических $LR(1)$ -таблиц.
 - 17) Построение $SLR(1)$ и $LALR(1)$ -таблиц. Программа Yacc/Bison. Устранение конфликтов. GLR-грамматики.
 - 18) Синтаксически управляемая трансляция.
 - 19) Организация вызова подпрограмм и прерываний. Команды ассемблера CALL, INT, RET, ENTER и LEAVE. Регистры SP и BP. Соглашения паскаля и си.
 - 20) Отладка и профилирование программ.
 - 21) Пример создания простой программы-калькулятора [yacc/bison].
 - 22) Поддержка односимвольных переменных [yacc/bison].
 - 23) Поддержка функций и констант [yacc/bison].
 - 24) Генерация кода [yacc/bison].
 - 25) Поддержка операторов [yacc/bison].
 - 26) Поддержка определения новых подпрограмм [yacc/bison].
 - 27) Система восстановления после ошибок и поддержка работы с символьными строками [yacc/bison].
 - 28) Системы версионного контроля. Основные особенности.
 - 29) Программа subversion.
 - 30) Программы для управления проектами.
 - 31) Использование языка пролог для работы с КС-языками.
 - 32) ОК-грамматики. Примеры работы.