МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. М. В. Ломоносова

Факультет вычислительной математики и кибернетики

**Майлингова О. Л., Манжелей С. Г., Соловская Л. Б.**

# Прототипирование программ на языке Scheme

*(методическое пособие по практикуму)*

**Москва**
2001

УДК 681.142
ББК 22.18
М14

**Майлингова О.Л., Манжелей С.Г., Соловская Л.Б. Прототипирование программ на языке Scheme (методическое пособие по практикуму).** Издательский отдел факультета ВМиК МГУ (лицензия ЛР№ 040777 от 23.07.96), 2001. - 83 с.

Рецензенты: доцент Пильщиков В.Н., к.ф.-м.н.
доцент Мансуров Н.Н., к.ф.-м.н.

Печатается по решению Редакционно-издательского Совета факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова

Пособие содержит необходимую информацию по языку Scheme и приемам функционального программирования. Предлагаются методические указания по быстрой разработке прототипов программ, их пошаговой доработке и документированию. Приведены примеры заданий практикума и рекомендации по их выполнению. Методическое пособие предназначено для студентов 3-го курса факультета ВМиК.

ISBN 5-89407-004-X

© Издательский отдел факультета вычислительной математики и кибернетики МГУ им. М.В.Ломоносова, 2001

## Введение

Предлагаемое методическое пособие предназначено для студентов 3-го курса факультета ВМиК в поддержку практикума, проводимого кафедрой системного программирования. Целью практикума является знакомство студентов с функциональным подходом в программировании на примере языка Scheme (современного диалекта Lisp), приобретение навыков быстрого прототипирования при создании программ, разработка библиотек функций и их использование при решении конкретных задач.

Язык Scheme включает многие элементы логического и функционального программирования. Уступая в эффективности таким языкам, как C++, Scheme отличается простотой и обладает богатыми встроенными возможностями манипулирования сложными структурами данных, такими как списки, векторы, деревья, что позволяет минимизировать затраты на создание программ. Легко расширяемый язык, Scheme дает возможность быстро создавать работающие прототипы программ, позволяющие оценить степень удачности того или иного решения, уточнить требования заказчика. Будучи интерпретируемым языком, Scheme позволяет работать со структурами данных в интерактивном режиме, используя окно интерпретатора, и как следствие, исключительно удобен для экспериментов со сложными структурами данных и проверке новых подходов к их анализу. Поэтому изучение возможностей языка целесообразно проводить на примере задач, связанных с обработкой данных, имеющих достаточно сложную структуру, с одной стороны, и использующих нетрадиционные методы решения, с другой. Именно к этому классу относятся задачи, решаемые с использованием новых аналитических методов, основанных на эволюционных процессах, и методов анализа графов.

Задание практикума состоит из двух частей: разработки библиотеки (генетического алгоритма или работы с графами) и решения с помощью библиотеки одной из предложенных задач. Выполнение задания практикума предполагает использование современных подходов в технологии программирования, а именно создание повторно используемых компонент и пошаговое уточнение программ с построением последовательности прототипов.

Разработка библиотеки на Scheme воплощает концепцию создания повторно используемого кода. Будучи расширяемым языком (это значит, что пользовательские функции ничем существенно не отличаются от встроенных), Scheme как нельзя лучше подходит для создания и использования библиотек. Хороший стиль программирования на Scheme как раз и состоит в создании многократно используемых пользовательских

функций, объединенных в библиотеки. Обладая достаточно сложной функциональностью, библиотека может иметь очень простой внешний (пользовательский) интерфейс. При проектировании сложных библиотек не менее важен внутренний интерфейс, определяющий структуру, модульность самой библиотеки. В частности, хорошо продуманная внутренняя структура должна позволять легко заменять (перегружать) внутренние функции, допускающие различную реализацию (например, различные алгоритмы сортировки массивов, различные модели естественного отбора).

Решение конкретной задачи - это пример создания собственного небольшого приложения, демонстрирующего функциональность, гибкость и удобство использования библиотеки. В соответствии со «спиральной» моделью жизненного цикла программы, проектирование приложения целесообразно осуществлять поэтапно по мере уточнения (усложнения) задачи. При этом на каждом шаге создается работающий прототип программы, который затем может быть расширен. Постановка предлагаемых задач уже включает элементы последовательного уточнения.

Пособие содержит необходимую информацию о языке Scheme и приемах функционального программирования. Предлагаются методические указания по быстрой разработке прототипов программ, их пошаговой доработке и документированию. В пособии приводятся примеры заданий практикума и рекомендации по их выполнению.

Пособие состоит из четырех глав. Первая глава посвящена описанию моделей разработки программного обеспечения, их анализу и сравнению. Основное внимание уделено моделям, использующим прототипирование. Вторая глава содержит необходимую информацию о языке Scheme и приемах функционального программирования. В третьей главе даны методические указания по быстрой разработке прототипов программ, использующих генетические алгоритмы и графы, их пошаговой доработке и документированию. В четвертой главе приведены примеры заданий практикума и рекомендации по их выполнению.

# Глава 1

# Модели процесса разработки программного обеспечения

Основной задачей моделирования процесса разработки программного обеспечения или программной системы является определение последовательности этапов разработки и определение критериев перехода от одного этапа к другому. Модель процесса не полностью описывает все действия, которые моделирует. Модели процессов разработки важны, прежде всего, потому, что они определяют стратегию выполнения основных задач, возникающих при реализации некоторого проекта, таких как разбиение на этапы, развитие системы, прототипирование, верификация. Существуют различные модели, которые могут быть использованы разработчиками программного обеспечения для создания их систем эффективно и квалифицированно. Рассмотрим некоторые из них.

## *Кодирование и использование*

Одна из первых моделей, которая использовалась на ранних этапах разработки программ, состояла из двух фаз: написание некоторого кода и его внедрение. Основные проблемы этой модели очевидны:

- после некоторого количества модификаций код становился столь неструктурированным, что дальнейшее его сопровождение становилось невозможным. Это определило необходимость включение в процесс разработки до начала кодирования фазы *проектирования;*
- часто оказывалось, что даже хорошо спроектированный программный продукт слабо удовлетворял потребности пользователя, что приводило либо к полному отказу от продукта, либо к его дорогостоящей модификации. Это потребовало включения в процесс разработки фазы *определения требований* еще до начала проектирования;
- как правило, оказывалось сложно использовать код из-за того, что он был недостаточно хорошо протестирован, и не предусматривал возможность дальнейшей модификации. Поэтому возникла потребность явно разделить фазы кодирования и тестирования, и планировать *тестирование,* развитие и *сопровождение* системы еще на ранних этапах ее разработки.

### *Пошаговая разработка*

Уже в 1956 году, опыт разработки больших программных систем привел к осознанию существовавших проблем и к разработке пошаговой модели для их решения. Модель определяла процесс создания программных продуктов как выполнение последовательности шагов, таких как общее проектирование, детальное проектирование, спецификация кода, кодирование, тестирование, общее тестирование, оценка системы.

### *Водопадная модель*

В 60-е годы был констатирован «кризис программного обеспечения», что подтолкнуло к внимательному исследованию причин "неудачной жизни программ". Наиболее важной причиной оказалась недооценка важности этапа формулирования требований. Ошибки, связанные с несоответствием системы требованиям заказчиков, оказались самыми дорогими и приводящими к нежизнеспособности разработанного продукта. Для того чтобы решать многочисленные проблемы, возникающие при разработке больших программных систем, была определена в 1970 году и уточнена в 1976 году классическая *водопадная модель* разработки программного обеспечения (Рис. 1). Эта модель учитывает существование обратной связи между фазами разработки, определяет способы описания результатов фаз разработки.

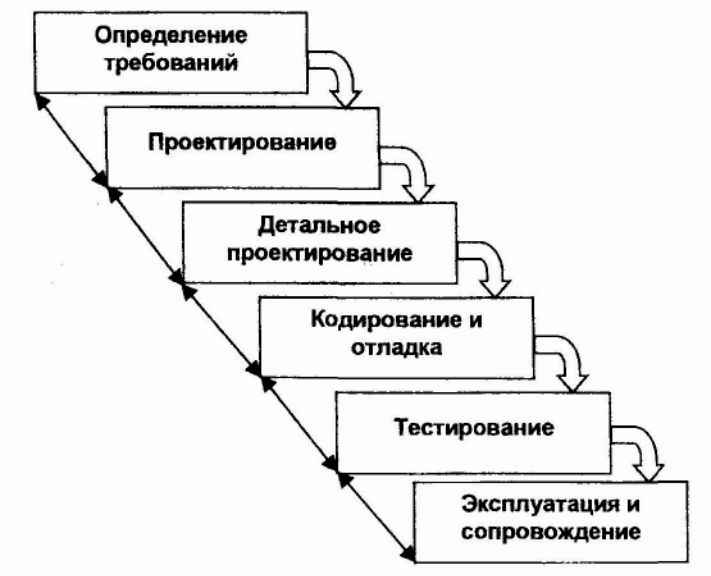



Рис. 1. Водопадная модель процесса разработки

Водопадная модель предполагает «фазированную» разработку программного продукта. При этом определение требований выполняется до начала разработки системы (еще до проектирования), а проектирование (определение взаимодействия компонент системы) происходит до начала кодирования. Все фазы разработки тщательно документируются, а документация используется для тестирования и сопровождения системы. Таким образом, снижается стоимость разработки и сопровождения.

Водопадную модель принято считать классической моделью разработки, но и она не является идеальной. На практике не всегда возможно четко разграничить фазы процесса разработки. Как правило, стоимость программного обеспечения превышает запланированную, система изготавливается позже, чем требуется, к тому же не всегда надежно и в соответствии с потребности пользователей.

### *Анализ моделей разработки*

Для анализа различных моделей разработки за основу возьмем тот факт, что требования пользователя постоянно изменяются. Таким образом, разрабатываемая система должна стремиться к достижению постоянно движущейся цели.

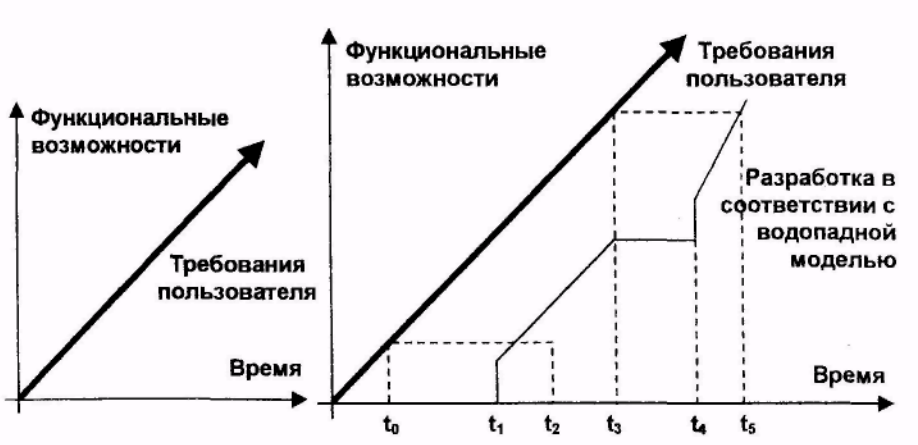



Рис. 2. Изменение требований к системе во времени

Рис. 3. Функциональные возможности системы (приближение к требованиям) во времени при использовании водопадной модели разработки

Представим рост требований пользователя прямой (Рис. 2). На осях преднамеренно не указан масштаб и единицы измерения, так как рост требований не является линейной и непрерывной во времени функцией.

На Рис. 3 представлено, что происходит при традиционной разработке системы. В момент времени to возникает потребность в программном обеспечении и начинается его разработка при относительно неполном понимании потребностей заказчика. В момент времени $t_1$ усилия разработчиков приводят к появлению работоспособной версии системы, которая не только не удовлетворяет потребностей времени $t_1$, но она не удовлетворяет потребностей времени to, так как разработка началась до уточнения требований. Далее система совершенствуется (между $t_1$ и $t_3$), и удовлетворяет начальным требованиям ($t_2$) и некоторым новым. В точке времени $t_3$ стоимость дальнейшего усовершенствования системы столь велика, что принимается решение разработать новую систему (опять на основании неточно понятых требований), ее разработка завершена в момент времени U и цикл повторяется.

Развитием (модификацией) водопадной модели стали модели, использующие прототипирование, повторное использование компонент и автоматическую генерацию кода по спецификациям.

## *Модели, использующие прототипирование*

Под *прототипированием* понимается разработка некоторого предварительного варианта системы с целью исследования некоторых свойств этой системы. Часто основной целью создания прототипа является получение информации от пользователя о том, как по его представлению система должна работать. На основе этой информации могут быть изменены спецификации требований к системе, что способствует увеличению гарантий правильности работы окончательного варианта системы. Создание прототипа может также служить целям исследования некоторых проблемных задач, альтернативных решений, принятых при проектировании или реализации системы. Назначением прототипа обычно является получение требуемой информации как можно быстрее и с минимальными издержками, поэтому чаще всего при его создании внимание сосредоточивается на некоторых сторонах создаваемой системы при полном игнорировании остальных сторон. Например, можно совершенно не обращать внимания на эффективность и рабочие характеристики системы и полностью отвлечься от некоторых функций системы. Что же касается исследуемых с помощью прототипирования аспектов поведения системы, то тут созданный прототип должен быть совершенно реальным. Таким образом, прототипирование может снижать стоимость разработки за счет частичных реализаций системы.

Концепция прототипирования открыла возможность для участия пользователей в процессе разработки системы, прежде всего, при анализе требований и спецификации системы. Основной проблемой начальных фаз разработки системы является то, что пользователю по текстовому описанию спецификации сложно представить реальное функционирование системы. Также сложно бывает понять, являются ли спецификации полными и непротиворечивыми. Более того, опыт показал, что ошибки, допущенные при анализе требований, обнаруживаются в последнюю очередь, на этапе тестирования, и поэтому их исправление самое дорогое. Для решения этих проблем используются различные подходы, разрабатываются языки спецификаций, графические средства и их комбинации. Но они больше помогают разработчику, чем пользователю.

## Быстрое прототипирование

Прототипирование на этапе спецификации требований решает некоторые проблемы взаимопонимания пользователей и разработчиков. Имея прототип, пользователь может испытать систему. Если не разрабатывать прототип, то прототипом становится первая версия системы. Она будет заменена другой, в которой будут исправлены ошибки первой.

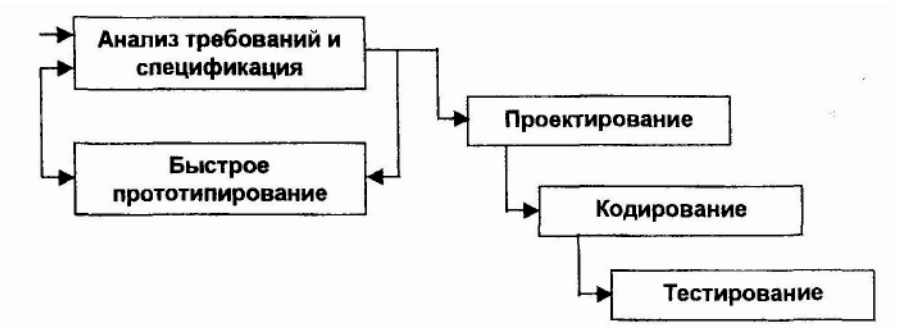



Рис. 4. Быстрое прототипирование

Для того чтобы прототип был эффективным инструментом, он должен удовлетворять следующим условиям:

- прототип системы должен быть работающей системой, которая помогает понять, изучить, пересмотреть спецификацию требований;
- прототип должен быть дешевле самой системы (не более 10% стоимости);
- прототип должен быть разработан быстро (до утверждения спецификации системы, так как цена ошибки растет со временем ее появления).

Использование быстрого прототипирования приводит к следующей модели процесса разработки (Рис. 4).

1. Предварительный анализ и спецификация требований.
2. Разработка и реализация прототипа.
3. Испытание прототипа.
4. Итеративное уточнение прототипа.
5. .Уточнение спецификации.
6. Разработка и реализация окончательного варианта системы.

*Предварительный анализ и спецификация требований* состоит из разработки спецификации, которая, по мнению разработчика, удовлетворяет требованиям заказчика.

При *разработке и реализации прототипа* основное внимание уделяется быстроте реализации и минимизации стоимости. Это достигается за счет реализации, прежде всего пользовательского интерфейса, для того, чтобы пользователь получил представление о системе, тогда как эффективность реализации функциональных возможностей пока не имеет значения. Прототип может быть реализован небольшим коллективом разработчиков. Язык реализации прототипа выбирается таким, чтобы на нем было удобно быстро программировать. Например, для этой цели подходят интерпретируемые языки, так как их использование ускоряет процесс сборки и обнаружения ошибок. Как правило, выбираются языки, имеющие встроенные средства работы со сложными структурами данных. Также для реализации прототипа выбирают инструментальную систему, в которой есть инструменты, способствующие быстрой реализации действующей системы. Существенным преимуществом является возможность повторного использования компонент.

*Изучение прототипа.* Каждый из пользователей может самостоятельно работать с прототипом и выражать свои замечания. Для этого нужно в графике разработки выделить время. Необходима обратная связь - учет замечаний пользователей. Как правило, больше пользователей принимает участие в изучении прототипа, чем в изучении спецификации. Документация, требуемая для изучения прототипа не столь объемна, как документация системы.

*Итеративное уточнение прототипа.* Желательно как можно быстрее реагировать на замечания пользователя и модифицировать прототип. И снова его отдавать для изучения пользователю. Итеративное уточнение и эксперименты с прототипом продолжаются в зависимости от стоимости, времени и замечаний. Пока цена уточнения не достигнет предела.

*Уточнение спецификаций.* На этапе уточнения прототипа уточняются требования пользователя, спецификации требований совершенствуются в соответствии с замечаниями пользователя и становятся базисом для дальнейшей разработки и реализации системы.

*Разработка, кодирование и тестирование системы* происходят в соответствии со стандартной моделью процесса разработки. Основное внимание при производстве основной системы уделяется, прежде всего, методам структурной разработки, структурного программирования, систематического тестирования и созданию полной документации. Таким образом, основное внимание направлено на обеспечение *сопровождения системы.*

## Инкрементное прототипирование

Рассмотрим другой подход, при котором прототип составляет ядро разрабатываемой системы. При этом подходе сначала реализуется ядро системы, ее основные функциональные возможности, а затем система постепенно наращивается.

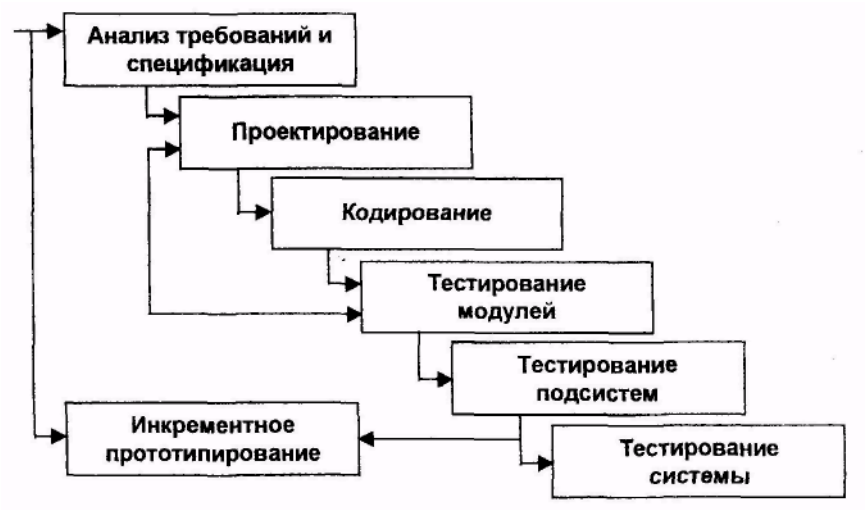



Рис. 5. Инкрементное прототипирование

Преимуществом *инкрементного прототипирования* является приятный для разработчиков момент, что система уже функционирует. Наращиваемые версии системы могут быть использованы в качестве прототипа для тестирования основной части системы, пользовательского интерфейса, или для тестирования ключевых алгоритмов. Работающая

часть системы может быть использована для получения замечаний пользователя.

Модель разработки с использованием инкрементного прототипирования (Рис. 5) включает следующие этапы.

1. Анализ требований и спецификация.
2. Разработка архитектуры - разбиение на модули и определение межмодульных интерфейсов.
3. Инкрементная реализация модулей. Детальная разработка каждого модуля, который предполагается включить в систему. Кодирование и тестирование модулей.
4. Инкрементная интеграция модулей в систему. Тестирование возникающих подсистем.
5. Повторение шагов 3 и 4 для каждого приращения системы. В некоторых случаях происходит уточнение архитектуры. Также могут уточняться спецификации требований.
6. Тестирование системы с точки зрения соответствия функциональной спецификации.
7. Прием в эксплуатацию - тестирование пользователями.

После того, как разработана архитектура системы, детально проектируются все модули. Так как межмодульный интерфейс определен на этапе разработки архитектуры, каждый из модулей может проектироваться независимо. Поэтому нет необходимости детально проектировать до начала этапа кодирования все модули. Таким образом, ядро системы может функционировать до того, как некоторые из модулей детально спроектированы.

### *Сравнение моделей, использующих прототипирование*

| Быстрое прототипирование | Инкрементное прототипирование |
|---|---|
| Подход к «быстрому и грязному» созданию частичной реализации системы на этапе определения требований. | Реализация части системы, не все требования к которой точно известны. Для уточнения требуется эксперимент с работоспособной версией системы. |
| Начинаем с того, что требует уточнения. | Начинаем с того, что не требует уточнения. |
| Используется для уточнения требований пользователя. | Используется для уточнения непонятых требований. |
| Приводит к полным и правильным спецификациям требований, хотя фаза определения требований | Приводит к раннему появлению первой версии системы. |

| | |
|---|---|
| занимает в результате больше времени, чем разработка формальных спецификаций. | |
| Качеству разработки не уделяется особое внимание, так как система экспериментальная. | Основное внимание уделяется качеству разработки. Поэтому не столь важна скорость разработки. |
| Строится модель системы, демонстрирующая понимание требований заказчика исполнителем для четкого определения требований к системе. Как правило, строится несколько прототипов системы. | Происходит наращивание работающей части системы новыми компонентами. В результате прототип становится системой. В данном подходе существенное значение имеет процесс разработки архитектуры системы. |

На Рис. 6, Рис. 7 отображено сравнение моделей, использующих прототипирование («жирная линия»), с традиционной моделью разработки. Использование быстрого прототипирования на ранних этапах разработки программных продуктов повышает степень понимания реальных потребностей пользователя в момент времени to.

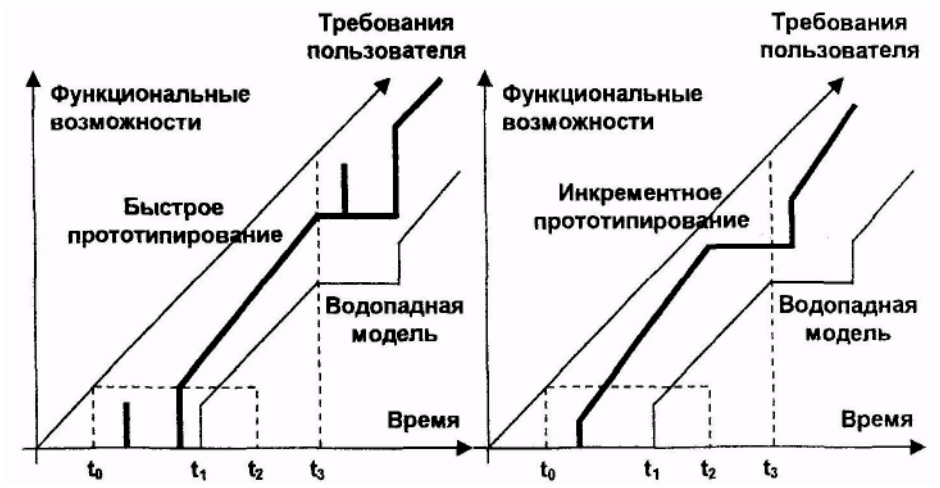



Рис. 6. Быстрое прототипирование и традиционная разработка

Рис. 7. Инкрементное прототипирование и традиционная разработка

Таким образом, сама модель разработки не сильно изменяется, однако использование прототипа влияет на конечный результат разработки. На Рис 6 показано, что в момент времени $t_1$ функциональные возможности системы больше, чем при ее разработке в соответствии с водопадной

моделью. На Рис. 6 показан также сам прототип как вертикальная линия, отображающая ограниченные возможности системы, вскоре после времени to. Нет оснований предполагать, что время, в которое система может быть использована без внесения изменений ($t_3$ - $t_1$), отличается от времени использования системы при традиционной разработке.

При инкрементном прототипировании, первый прототип появляется рано и демонстрирует реализацию тех требований, которые, предположительно, были поняты, а также поведение системы в целом. Каждая следующая версия прототипа охватывает новую область требований пользователя и уточняет предыдущие возможности. В результате решение приближается к реализации потребностей пользователя. Конечно, через некоторое время также требуется глобальная реконструкция системы. При описываемом подходе хорошо спроектированная система позволяет заменять одни компоненты другими и равномерно приближаться к требуемому результату.

Одним из существенных требований к прототипу является минимальное время его реализации. Как разработать прототип быстрее, чем систему? Рассмотрим модели разработки, основанные на повторном использовании программных компонент и использовании автоматической генерации кода (прототипа) на основе спецификаций требований.

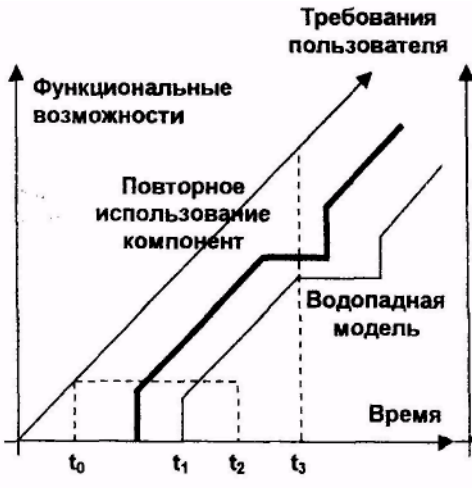



Рис. 8. Повторное использование компонент

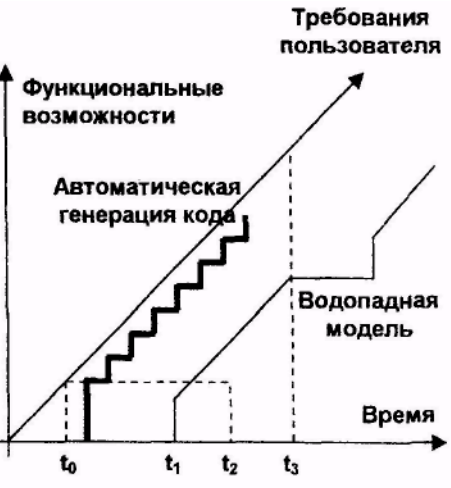



Рис. 9. Автоматическая генерация кода

### *Повторное использование компонент*

Прототипирование снижает стоимость разработки путем частичных реализаций системы. Повторное использование программных компонент является подходом, при котором стоимость разработки системы снижается за счет использования существующих проектных решений или существующего кода в новых программных продуктах. Разработчиков программного обеспечения часто обвиняют в постоянном «изобретении колеса». Это происходит отчасти потому, что существует немного инструментов, позволяющих применять накопленный опыт. Требуются технические приемы и инструменты для создания, хранения, каталогизации и поиска компонент, пригодных для многократного использования. Результатом будет сокращение времени разработки новых систем и увеличение их надежности. На Рис. 8 показано, как повторное использование компонент («жирная линия») влияет на время разработки системы, хотя сама модель может оставаться, по сути, традиционной.

### *Автоматическая генерация кода*

Термин автоматическая генерация кода используется для описания преобразования требований или спецификации проекта в выполняемый код. Процесс преобразования может быть алгоритмическим или основанным на технике баз знаний. Разные поколения разработчиков программного обеспечения используют этот термин для описания преобразования с языка более высокого уровня в используемый язык программирования. Этим термином обозначали «автоматическую» трансляцию языка ассемблера в машинный код (ассемблирование), перевод текста на языке программирования в машинный код (компиляцию). В настоящее время этот термин употребляется для описания преобразования с языка спецификаций в язык программирования.

При реализации такого подхода требования записываются на некотором формальном языке спецификации, например SDL, и система строится автоматически. В результате время разработки системы существенно сокращается, также как и стоимость разработки. Эффект автоматического построения системы показан на Рис. 9 («жирная линия»). В идеальной ситуации, при каждом внесении изменений в спецификацию требований новая система (или прототип) генерируется автоматически, и функциональные возможности системы постоянно соответствуют требованиям.

### *Спиральная модель*

Использование прототипирования приводит к меньшему объему кодирования, более простому использованию. Традиционный подход характеризуется лучшей согласованностью, большей функциональностью, более высокой степенью переносимости и более простой интеграцией Из этого следует, что требуется смешанный подход, позволяющий по некоторому подмножеству требований быстро разрабатывать прототипы, на основе которых уточняются основные требования.

Спиральная модель была разработана на основе опыта усовершенствования водопадной модели. Она рассматривает существующие модели разработки как частные случаи, и описывает возможности комбинирования существующих приемов (Рис. 10). Радиус представляет собой общую стоимость разработки в текущий момент времени, угол отображает прогресс при выполнении витка спирали. Модель предполагает, что на каждом витке выполняется та же последовательность шагов для каждого фрагмента продукта и для любого уровня его разработки. Типичный цикл состоит из следующих шагов:


Рис. 10. Спиральная модель

- определение целей разработки, средств реализации, ограничений,
- оценка альтернатив с учетом целей и ограничений;
- исследования степени риска (прототипирование, моделирование, опрос пользователей и т.п.),

• на основе оценки риска определяется следующий шаг разработки, более детальное прототипирование, или переход к следующему шагу разработки в соответствии с водопадной моделью.

Спиральная модель представляет собой обобщение подходов к разработке программных систем, основанное на разработке спецификаций, использовании прототипирования, построении моделей, использовании автоматической генерации системы.

Использование прототипов при разработке программных систем позволяет обнаруживать ошибки в требованиях к системе на более ранних стадиях разработки. Заметим, что цена ошибки растет со временем ее обнаружения. Заметим также, что в моделях, не использующих прототипирование, действующая система появляется на достаточно поздних этапах жизненного цикла. Поэтому, при наличии в ней существенных недостатков, такая система обречена стать, по сути, прототипом, только очень дорогим.

Язык Scheme (диалект языка программирования Lisp) отличается простотой синтаксиса и обладает богатыми встроенными возможностями манипулирования сложными структурами данных, такими как списки, векторы, деревья, что позволяет минимизировать затраты на создание программ. Легко расширяемый язык, Scheme дает возможность быстро создавать работающие прототипы программ, позволяющие оценить степень удачности того или иного решения. Будучи интерпретируемым языком, Scheme позволяет работать со структурами данных в интерактивном режиме и как следствие, исключительно удобен для экспериментов со сложными структурами данных и проверке новых подходов к их анализу. Поэтому, этот язык может быть использован для разработки прототипов систем.

# Глава 2

# Разработка программ на Scheme

Для быстрой разработки прототипов программ необходим инструмент, гибкий и удобный в использовании, например, язык Scheme. Scheme - это один из диалектов функционального языка программирования Lisp, предложенного Джоном Маккарти в конце 50-х годов. По давности использования Lisp является вторым языком программирования (старше - только Фортран).

Долгое время язык Lisp развивался неформально, живо реагируя на требования пользователей. Такое демократичное развитие в сочетании с гибкостью и элегантностью начальной концепции позволило языку Lisp адаптироваться к современным тенденциям проектирования программ. В настоящее время Lisp представляет собой семейство диалектов, объединенных общей концепцией и называемых Lisp-культурой. Из существующих диалектов, Scheme наиболее близок к классическому языку Lisp Он был разработан в 1975 в Массачусетском Университете Джеральдом Суссманом (Guy Lewis Steele Jr. Gerald Jay Sussman, MIT Artificial Intelligence Lab). В 1990 году был принят стандарт языка Scheme IEEE Standard 1990

## *Элементы языка программирования Scheme*

Язык Scheme является интерпретируемым языком. Программа на Scheme состоит из выражений, имеющих значения. Вычисляя выражения, интерпретатор выполняет один и тот же цикл: считывает выражение, вычисляет выражение, возвращает результат Выражения языка Scheme строятся из следующих лексем.

### Числа

На языке Scheme числа представляются в десятичной системе счисления. Вещественные числа с плавающей точкой записываются в виде <p>e<k>, обозначающем число $p\ 10^k$. Например, 123e10 представляет собой число $123*10^{10}$, -123.45e-1 - число $-123.45\ 10^{-1}$.

### Идентификаторы

В языке Scheme определены следующие ключевые слова:

```
and           else               or
begin         error              quote
cond          if                 sequence
cons-stream   lambda             set!
define        let                the-environment
delay         make-environment   unquote
```

Идентификаторы, не являющиеся ключевым словом, могут быть использованы в качестве имен. Большие и малые буквы не различаются, за исключением строковых констант. Примеры идентификаторов-имен:

```
Q      <=?          list->vector     +

soup   a34kTMNs     the-word-recursion-has-many-meanings
```

Foo обозначает тот же объект, что и FOO.

## Пробельные символы и комментарии

Пробельные символы - пробелы, знаки табуляции и знаки конца строки. Используются в качестве разделителей между именами и числами, или для форматирования текста. Они могут входить в строковые константы. Комментарии ограничены знаком «;» слева, и концом строки справа.

### Специальные символы и их использование

Следующие символы могут быть использованы в качестве «буквы» в идентификаторах:

```
.  *  /  <  =  >  !  ?  :  $  %  _  &  ~  ^
```

«.», «+» и «-» используются при записи чисел и идентификаторов. Точка также используется в записи составных данных типа точечная пара. «(» и «)» являются ограничителями списков и выражений. «'» определяет символьные данные. «"» ограничивает строковые константы. «\» используется для включения специального символа в строковую константу. «[», «]», «{» и «}» зарезервированы для расширений языка. «#t» и «#f» являются представлением логических констант *истина* и *ложь*. «#\» используется для задания ASCII символа в числовом виде. «# (» определяет векторную константу.

### Выражения

Выражением на языке Scheme называется фрагмент кода, значение которого может быть вычислено. Существуют следующие типы выражений: константы, имена, списки из имен и констант в скобках, обозначающие вызовы процедур и специальные формы.

Получая выражение, интерпретатор его вычисляет и возвращает результат, например, для отображения на экран. Ответ интерпретатора будем записывать после символа ->. Примером простейшего выражения является число. На ввод числа

```
486
```

интерпретатор отвечает

```
-» 486
```

Для того чтобы построить более сложные выражения, числа и встроенные процедуры (например, + или *) можно объединять в списки. Построенные таким образом выражения будут иметь значение равное результату *применения процедуры к* аргументам. Например:

```
(+ 137 349)      -> 486
(- 100 -334)     -> 434
(* 5 99)         -> 495
(/ 10 5}         -> 2
(+ 2.1 10)       -> 12.7
```

Приведенные выше примеры выражений, построенных в виде списков, ограниченных скобками, означают *вызов процедуры*. Самый левый элемент списка представляет оператор, остальные элементы - операнды. Значение выражения получается в результате применения процедуры, заданной оператором, к аргументам, которыми являются значения операндов.

Способ записи выражения, при которой оператор находится слева от своих операндов, называется *префиксной записью* выражения. Такая запись имеет ряд преимуществ. Например, позволяет легко описывать процедуры, допускающие произвольное количество аргументов. Например:

```
(+ 11)              -> 11
(+ 21  35  12  7)   -> 75
(*25  4  12)        -> 1200
```

Неопределенности не возникает, так как оператор определен самым левым элементом списка, а выражение ограничено скобками. Другим преимуществом префиксной записи является элегантное представление сложных выражений в виде списков, содержащих в качестве элементов другие списки.

```
(+  (*  3  5)  (-  10  6) )        ->  19
```

В принципе, нет ограничения на глубину вложенности, и на общую сложность выражения, которое может вычислить интерпретатор. Значение выражения

```
(+ (* 3 (+ (* 2 4) ( + 3 5 ) ) )  (+ (- 10 7) 6) )
```

интерпретируется как 57. Для большей наглядности оно может быть записано в форматированном виде

```
(+ (* 3
      (+ (* 2 4) (+
         3 5)
      )
      ( + (- 10 7)
   6 ) ) )
```

## Область действия имен

В любом языке программирования имена используются для обращения к тем или иным вычислительным объектам. В Scheme имя определяет переменную, значением которой является некоторый объект языка (число, процедура, и т.п.). Для именования используется специальная форма define. Запись

```
(define size 2)
```

приводит к тому, что интерпретатор *связывает* значение *2* с именем size Так как define это специальная форма, не имеющая вычисляемого значения, ответ интерпретатора на такое выражение зависит от реализации. После того, как имя size было связано с числом 2, к этому объекту можно обращаться по имени:

```
size            ->  2
(*  5 size)     ->  10
```

Примеры использования define:

```
(define pi 3.14159)
(define radius 10)
(* pi (* radius radius))                  ->  314.159
(define circumference (* 2 pi radius))
circumference                             ->  62.8318
```

Форма define является простейшим средством абстракции языка. Она позволяет использовать имена для обращения к результатам составных процедур, например, вычисления площади круга, и именовать все более крупные объекты. Эта возможность упрощает инкрементную разработку и тестирование программ. Как правило, программы на Scheme состоят из большого количества относительно простых процедур.

Любой идентификатор, который не является ключевым словом, может быть использован как имя. Совокупность пар <имя, значение> называется *окружением (контекстом)*. Scheme является языком с блочной структурой, поддерживающим концепцию статического (раннего) связывания. Область видимости имени определяется выражением (блоком), в котором оно было определено. Определение имени вне какого-либо выражения означает, что имя глобальное, то есть значение, связанное с именем, доступно во всех выражениях, в которых это имя не переопределено. Заметим, что предопределенные (встроенные) процедуры определены в так называемом глобальном контексте.

## Символьные данные

Одним из основных типов объектов в языке Scheme являются *символьные константы*. Значением символьной константы является сама эта константа. Можно сказать, что символьная константа - это текстовое представление объекта Scheme. Примером символьной константы является символ abc. значением которого является abc. Другими примерами символьных констант могут служить числа и логические константы: 123, 78e-3, #t.

Различие между именами и символьными константами заключается в том, что с именем связано некоторое значение, отличное от этого имени, а значением символьной константы является она сама. Для указания того, что идентификатор должен рассматриваться как символьная константа, используется оператор quote.

Значением (quote <datum>) является <datum>. При этом <datum> может быть текстовым представлением любого объекта Scheme. Заметим, что значение <datum> при этом не вычисляется, то есть quote блокирует вычисление. Вместо quote также можно использовать знак «'» («кавычка»). То есть, записи (quote <datum>) или '<datum> эквивалентны.

```
(quote a)              ->  a
(quote # (a b          ->  #(a Ь c)
c)) (quote (+ 1        ->   (+1
2))                    2)
```

```
'a                  ->  a
'#(a ь c)           ->  # (a ь c)
'(+1 2)             ->  (+ 1 2)
'(quote a)          ->  (quote a)
''                  a   ->      (quote a)
```

В отличие от символьных констант различают *строковые константы,* которые записываются в двойных кавычках. Например, "abc".

Значением числовых, строковых и логических констант являются сами константы, поэтому выделять их из контекста (с помощью quote или символа «кавычка») не обязательно.

```
'"abc"    ->  "abc"
"abc"     ->  "abc"
'145932   ->  145932
145932    ->  145932
'#t       ->  #t
#t        ->  #t
```

Но quote необходимо для отличия имен переменных от символьных констант:

```
(define a 3) (define abc
(+12))

a         ->  3
(quote a) ->  a
'a        -^  a
abc       ->  3
'abc      ->  abc
b         ->  ошибка - неопределенный объект
 'b        -> b
(+1 2)    ->  3 •(+1
2)   ->  (+1 2)
```

## Общие правила вычисления выражений

Рассмотрим последовательность действий, которую выполняет интерпретатор при вычислении выражений.

1. Вычислить подвыражения.
2. Применить процедуру, которая является значением самого левого подвыражения, к аргументам, которые получились в результате вычисления остальных выражений.

Рассмотрим вычисление выражения (* (+ 2 (* 4 6)) (+ 3 5 7)). Правило вычисления выражения будет применено для вычисления значений четырех различных вызовов процедур. Предположим, что повторное применение правила (1) привело к тому, что требуется вычислить значение так называемого примитива, например, числа, встроенного оператора или другого имени. В этих случаях действуют следующие правила:

- значением констант являются константы (числовые, логические, строковые);
- значением встроенного оператора являются последовательности машинных команд, которые реализуют соответствующие операции;
- значением имени являются объекты, связанные с этими именами в текущем контексте.

Заметим, что эти правила вычисления выражений не справедливы для определения имен (define). Вычисление (define x 3) не *применяет* define к аргументам x и 3, а *связывает* x с 3. То есть, define не является вызовом процедуры,

Такие исключения из общих правил вычисления называются *специальными формами.* Каждая специальная форма имеет свои правила вычисления. Разные типы выражений (с соответствующими правилами вычислений) определяют синтаксис языка. По сравнению с другими языками программирования Scheme имеет очень простой синтаксис, то есть, количество специальных форм не велико.

## *Определение процедур*

В этом разделе на примере работы с числовыми данными будут рассмотрены правила построения процедур. Пусть, например, необходимо определить операцию возведения в квадрат. Средствами языка Scheme это можно сделать следующим образом:

```
(define (square x) (* x x))
```

При этом выполняются два действия: создание процедуры и связывание ее с именем. Позже будет рассмотрен способ создания процедуры без имени. Общий вид определения процедуры можно записать следующим образом:

```
(define (<name> <formal parameters>) <body>)
```

В этой записи <name> представляет собой имя, которое будет связано с определением процедуры в соответствующем контексте. < formal

parameters> определяют имена формальных параметров, которые локализованы в теле процедуры. <body> представляет собой выражение, которое получит значение в результате применения процедуры, когда формальные параметры будут заменены фактическими. В общем случае, тело процедуры может быть последовательностью выражений. Интерпретатор вычисляет последовательно все выражения, и в качестве результата выдает значение последнего выражения, которое было вычислено. Определив процедуру square, ее можно использовать.

```
(square 21)        ->  441
(square (+2 5))    -> 49
(square (square 3)) ->  81
```

Определенную ранее процедуру можно также использовать при определении следующих процедур. Например, выражение $x^2 + y^2$ может быть записано как

(+ (square x) (square y)) **и**

**представлено в виде процедуры.**

```
(define (sum-of-squares x y)  (+  (square x)  (square y)))
(sum-of-squares 3 4 )          ->  25
```

В свою очередь процедуру sum-of-squares можно использовать в качестве блока для построения следующих процедур:

```
(define (f a)  (sum-of-squares  (+ a 1)  (*  a 2 ) ) )
(f 5)                          ->  136
```

Составные процедуры используются так же, как и встроенные процедуры. По внешнему виду нельзя определить, является ли процедура встроенной в интерпретатор, или определенной пользователем.

## Модель подстановок при применении процедур

Для того чтобы вычислить выражение, оператор которого является именем составной процедуры, интерпретатор выполняет те же действия, которые были описаны выше. То есть, интерпретатор вычисляет значения подвыражений и применяет процедуру к аргументам. Способ применения примитивных процедур к аргументам встроен в интерпретатор. Для того чтобы применить составную процедуру к аргументам необходимо вычислить выражение, являющееся телом процедуры, заменяя каждый формальный параметр соответствующим аргументом. Рассмотрим пример вычисления (f 5), где f - процедура, определенная выше. Начинаем с восстановления тела процедуры f:

(sum-of-squares (+ a 1) (* a 2)) Далее

**заменяем формальный параметр, на аргумент 5:**

(sum-of-squares ( + 5 1 ) (* 5 2})

Таким образом, задача сведена к вычислению выражения с двумя операндами и оператором sum-of-squares. Вычисление этого выражения включает три подзадачи. Сначала необходимо вычислить оператор, для того, чтобы получить процедуру, которую нужно применить, а затем вычислить значения операндов, чтобы получить аргументы процедуры. Вычисляя ( + 5 1 ) и (* 5 2) получаем 6 и 10 соответственно. Далее требуется применить sum-of-squares к аргументам 6 и 10. Получаем

(+ (square 6) (square 10))

Затем в соответствии с определением square получаем

(+ (* 6 6) (* 10 10) )

что сводится к вычислению (+ 36 100) и приводит к результату 136. Описанный процесс называется *моделью подстановок* при применении процедуры к аргументам.

## Аппликативный и нормальный порядок

Согласно описанию вычисления выражений, интерпретатор сначала вычисляет оператор и операнды, а потом применяет полученную процедуру к полученным аргументам. Это не единственный способ выполнения вычисления. Альтернативным способом является модель, в которой операнды не вычисляются до тех пор, пока они не требуются. Вместо этого, сначала производится подстановка всех подвыражений, пока не будет получено выражение, состоящее только из примитивных операторов, после чего выполняются все вычисления. Рассмотрим этот метод на примере вычисления (f 5). На первом шаге раскрываются все подвыражения

```
(sum-of-squares ( + 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2) ))
```

и далее происходят свертки

```
{+   (*  6  6)   (*  10  10) )
( + 36  100)
136
```

Заметам, что вычисление выражений (+ 5 1) и (* 5 2 ) выполняется дважды, в соответствии со сверткой выражения (* x x), и заменой каждого x на (+ 5 1) и (* 5 2 ) .

Метод «полностью подставь, потом сворачивай», известен как *нормальный порядок* вычисления. Метод «вычисли аргументы, потом примени», который, как правило, использует интерпретатор, называется *аппликативным порядком* вычисления. Для приведенных выше примеров, оба метода дают одинаковые результаты. Однако далее будут рассмотрены случаи, при которых порядок вычисления может привести к разным результатам. Scheme использует аппликативный порядок вычислений, отчасти потому, что он повышает эффективность, в частности, позволяет один раз вычислять общие подвыражения.

## Условные выражения и предикаты

В языке Scheme поддерживается набор специальных форм, играющих ту же роль, что и управляющие конструкции в традиционных языках программирования. В частности для записи условных выражений применяется форма cond. С ее использованием вычисление модуля целого числа может быть представлено в следующем виде:

```
(define  (abs x)
      (cond       ((> x 0)  x)
                  ((= x 0)  0)
                  ((< x 0)  (- x))))
```

В общем случае условное выражение имеет вид

```
(cond  (<p1> <e2>)
       (<p2> <e2>)

       (<pn> <en>) )
```

где за ключевым словом cond следует список пар выражений $<p_i>$ $<e_l>$, заключенных в скобки. Первое выражение в каждой паре является предикатом, то есть выражением, которое интерпретируется как истинное или ложное. В Scheme поддерживаются константы #f и #t, обозначающие соответственно ложь и истину. При вычислении значения не ложь является истиной. Условное выражение вычисляется следующим образом. Сначала вычисляется предикат <pi>. Если его значение ложь, то вычисляется предикат $<p_2>$. Если значение предиката $<p_2>$ также ложь, то вычисляется

предикат <$p_э$>. Процесс продолжается до тех пор, пока не будет найден предикат<$p_i$ >, значением которого будет истина. В этом случае, значением условного выражения будет значение соответствующего выражения <$e_i$> Если ни один из предикатов не имеет значение истина, значение условного выражения не определено.

Процедура abs использует примитивные предикаты >, <, =. Она также использует оператор «минус», который при использовании с одним аргументом означает изменение знака числа. Процедуру abs можно определить и другим способом:

```
(define   (abs x)
      (cond  ( (< x  0)   (-x) )
             (else x ) ) )
```

else является специальным символом, который можно использовать вместо предиката в последней альтернативе cond. В случае невыполнения ни одного из условий альтернатива else будет определять значение условного выражения. Вместо else можно использовать любое выражение, которое всегда имеет истинное значение (например, константа #t).

**Рассмотрим еще один способ определения процедуры** abs.

```
(define   (abs x)   ( i f   (< x  0)   (-x)   x) )
```

В этом определении используется специальная форма if, представляющая собой ограниченный тип условного выражения, который можно использовать для рассмотрения ровно двух альтернатив. Общая форма if выражения имеет вид:

```
(if <predicate> <consequent> <alternative>)
```

При вычислении if выражения интерпретатор вычисляет <predicate>. При получении значения истина интерпретатор вычисляет значение <consequent> и оно становится значением if выражения, иначе вычисляется значение <alternative> и оно становится значением if выражения. Небольшим отличием между if и cond выражениями является то, что часть <$e_i$> каждого cond предложения может быть *последовательностью выражений,* и результатом тогда становится значение последнего вычисленного выражения. В if выражении <consequent> и <alternative> должны быть *простыми выражениями.*

Наряду с примитивными предикатами, такими, как <, >, = в языке имеются логические операции, которые позволяют строить составные предикаты. Наиболее часто используемыми являются and, or и not.

(and **<$e_1$>** . . . <$e_n$>)

В форме and интерпретатор вычисляет слева направо выражения <$e_1$>. Если одно из вычисленных значений выражений стало ложным, то значение and выражения будет ложь, а остальные выражения не вычисляются. Если все выражения имеют значения истина, то и значением and выражения будет истина.

**(or <$e_1$>** . . . <$e_n$>)

В форме or интерпретатор вычисляет выражения <$e_1$> слева направо. Если одно из вычисленных выражений стало истинным, то это значение и будет значением or выражения, а остальные выражения вычисляться не будут. Если все выражения имеют значения ложь, то значением всего or выражения также будет ложь.

(not <e>)

Значением not выражения будет истина, если значение выражения <e> ложь. Заметим, что and и or выражения являются специальными формами, так как не все их подвыражения вычисляются, not является обычной процедурой.

В качестве примера использования логических операций рассмотрим вычисление условия *5 < x < 10.*

```
(and   (< x 5)   (< x 10) )
```

Другим примером может быть определение предиката «больше или равно»:

```
(define   (>= x y)   (or   (> x y)   (= x y) ) )
```

или

```
(define   (>= x y)   (not   (< x y) ) )
```

*Упражнения* 1 . Что будет выдано интерпретатором в ответ на следующие выражения?

```
10
(+534)
(- 91) (/6 2)
(+ (* 24) (-4 6))
(define a 3)
(define b (+ a 1))
(+ a b (* a b) )
(= a b)
(if (and (> b a) (< b (* a b) ) ) b a)
(cond ((= a 4) 6) ((= b 4) (+ 6 7 a) ) (else 25))
(+ 2 (if (> b a) b a) )
(* (cond ((> a b) a) ( (< b a) b) (else -1) ) (+ a 1) )
```

2. Определить процедуру от трех параметров-чисел, которая вычисляет сумму квадратов двух больших чисел.

3 В соответствии с моделью вычисления выражений описать поведение следующей процедуры:

(define (a-plus-abs-b a b) (if (> b 0) + -) a b)) 4.

**Пусть определены следующие процедуры:**

(define (p) (p))
(define (test x y) (if (= x 0) 0 y) )

Определить поведение интерпретатора при вычислении значения выражения (test o (p)) в случае апшшкативного порядка вычисления и в случае нормального порядка вычисления. Замечание: вычисление if выражения происходит одинаково в обоих случаях. Вычисляется предикат, и в зависимости от его результата вычисляется первое или второе выражение.

## Пример. Нахождение квадратного корня методом Ньютона

Квадратный корень *y* из числа *x* может быть определен следующим образом: /*корень*/*k = y,* если *y > = 0* и $y^2 = x$. Для вычисления корня воспользуемся методом приближений Ньютона, по которому, имея приближенное значение *y* можно получить более точное приближение к корню из *x,* взяв среднее значение *y* и *x/y.* Вычислять корень из 2, например, можно следующим образом.

Предположим, что начальное приближение .y = /. Тогда:

| Приближение *y* | *x/y* | Среднее */y, y)* |
|---|---|---|
| *1* | *2/1=2* | *(2+1)/2=1.5* |
| *1.5* | *1.5/2=1.3333* | *(1.3333+ 1.5J/2-1. 4167* |

| | | |
|---|---|---|
| *1.4167* | *2/1.4167=1.4118* | *(1.4168+1.4118)/2=1.4142* |
| *1.4142* | | |

Основной процедурой будет процедура sqrt-iter, определенная от аргументов guess (приближение) и x (число, квадратный корень которого она находит). Если текущее приближение является достаточно хорошим, то процесс вычисления заканчивается и результатом становится текущее значение guess, иначе вычисление продолжается с улучшенным значением guess.

```
(define {sqrt-iter guess x)
    (if (good-enough? guess x) guess (sqrt-
        iter (improve guess x) x))}
```

Улучшение приближения получаем как среднее значение x/guess и guess.

```
(define (improve guess x) (average guess (/ x guess)))
```

где

```
(define (average x y) (/ (+ x y) 2))
```

Необходимо также определить допустимую погрешность вычисления, предикат good-enough?. Заметим, что имена предикатов, как правило, заканчиваются знаком «?». Для первого варианта решения задачи допустимую погрешность можно определить как $|guess^2 - x| < eps$.

```
(define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
```

Однако лучше этот предикат определить как модуль разности значений guess на предыдущем и текущем шаге. В качестве упражнения предлагается переопределить этот предикат.

Наконец, нужно определить саму процедуру sqrt. Допустим, что мы всегда будем начинать с приближения 1.

```
(define (sqrt x) (sqrt-iter 1.0 x))
```

После определения всех процедур можно использовать sqrt, как обычную процедуру.

```
(sqrt 9)                    ->  3.00009155413138
(sqrt (+100 37))            ->  11.704699917758145
```

```
(sqrt (+ (sqrt 2)(sqrt 3)))    ->  1.7739279023207S92
(square (sqrt 1000))           ->  1000.000369924366
```

Заметим, что необходимая процедура полностью определена, несмотря на то, что не были определены никакие средства языка поддерживающие итерационный процесс (например, операторы цикла или перехода). Процедура sqrt-iter не использует никаких специальных конструкций кроме вызова процедуры.

Рассмотрим наиболее важные моменты построения процедуры sqrt.

Задача вычисления квадратного корня разбита на несколько подзадач, а именно: достаточно ли хорошее приближение найдено, как улучшить приближение и т.д. Каждая их подзадач реализуется отдельной процедурой. Важность стратегии декомпозиции состоит не только в разбиении задачи на подзадачи. Существенно, что каждая из процедур выполняет независимую задачу и может быть использована и в других процедурах. Например, при определении процедуры good-enough? была использована процедура square. При этом процедура рассматривалась как «черный ящик». То есть детали реализации процедуры скрыты, а возможности использования определяются ее интерфейсом. При использовании процедуры интересует то, *что* она делает, а не то, *как.* В этом смысле square представляет собой так называемую процедурную абстракцию.

*Упражнения*

5. Почему необходима специальная форма для if выражения? Почему его нельзя определить как обычную процедуру с помощью cor.d?

```
(define  (new-if predicate  then-clause  else-clause)
      (cond  (predicate then-clause) (else else-
      clause)))
```

Примеры

```
(new-if  (=  2   3)   0  5}        ->   5
(new-if  (=  1   1)   0  5)        ->   0
```

дают правильный ответ. Что произойдет, если использовать new-if в процедуре sqrt-iter? Объясните.

```
(define (sqrt-iter guess x!
    (new-if (good-enough? guess x) guess (sqrt-
          iter (improve guess x) x)))
```

6. Переопределите предикат good-enough? чтобы он приводил к правильному вычислению корня для больших и малых чисел.

7. Метод Ньютона для вычисления кубического корня основан на том, что, имея приближенное значение *y* кубического корня *x,* уточнение можно получить по формуле $(x/y^2 + 2y)/3$. Используя приведенную формулу, определите процедуру cube-root, которая вычисляет кубический корень.

## Внутренние определения и блочная структура

Имена формальных параметров локализованы в теле процедуры. Как видно из рассмотренного примера, для реализации процедуры square потребовалось определить несколько вспомогательных процедур. Во избежание возможной коллизии имен при разработке больших библиотек принято локализовать вспомогательные процедуры внутри основной. Таким образом, можно переписать определение процедуры sqrt.

```
(define (average x y) (/ (+ x y) 2))

(define (sqrt x)
    (define (good-enough? guess x)
       (< (abs (- square guess) x)) 0.001))
    (define (improve guess x)
          (average guess (/ x guess)))
    (define (sqrt-iter guess x)
          (if (good-enough? guess x)
                guess
                (sqrt-iter (improve guess x) x)))
    (sqrt-iter 1.0 x) )
```

Такая структура процедур называется *блочной* и является в основном правильным путем решения проблем. Но его можно еще усовершенствовать. Так как переменная x является *локальной (связанной)* в определении процедуры sqrt, и процедуры good-enough?, improve и square-iter находятся в области видимости x, то нет необходимости передавать x в качестве параметра каждой из процедур. Имя x можно использовать как *глобальную (свободную)* переменную во всех вложенных определениях. Такой подход называется *лексическим связыванием.*

```
(define (sqrt x)
     (define (good-enough? guess)
          (< (abs (- square guess) x)) 0.001))
     (define (improve guess)
          (average guess (/ x guess)))
     (define (sqrt-iter guess)
```

```
      (if (good-enough? guess)
          guess
          (sqrt-iter (improve guess))))
(sqrt-iter 1.0))
```

Идея блочной структуры возникла в языке программирования Algol-60. Она поддерживается во многих языках программирования и является одним из основных методов разработки больших программ.

## Процедуры и процессы, которые они порождают

Процедура является шаблоном локального развития вычислительного процесса. Она определяет, как текущее состояние процесса получается на основе предыдущего состояния. Опишем наиболее типичные шаблоны процессов, порождаемых процедурами в Scheme.

### *Линейная рекурсия и итерация*

Рассмотрим задачу вычисления факториала натурального числа n! = *n (n - 1) (n - 2) ... 3 2 1*. Существует много различных способов вычисления факториала. Например, можно представить *n!* как *n (n - 1)!*. Таким образом, добавив утверждение, что *1! = 1,* можно получить следующее определение процедуры:

```
(define (factorial n)
    (if (= n 1) 1 (* n (factorial (- n 1 ) ) ) ) )
```

Для того чтобы представить процесс, который выполняется при вычислении значения (factorial 6) можно использовать одну из моделей подстановок, описанных выше. Например,

```
(factorial 6)
(* 6 (factorial 5))
(* б (* 5 (factorial 4)))
(* 6 (* 5 (* 4( factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 {* 3 (* 2 1))))) )
(* 6 (* 5 (* 4 (* 3 2)))))
(* 6 (* 5 (* 4 6))))
(* 6 (* 5 24»
(* 6 120)
720
```

Такой процесс представляет собой линейную рекурсию

Вычисление факториала можно также описать как умножение *1* на 2 на *3* и т.д. до *n*. При этом необходимо накапливать произведение и изменять

текущий счетчик от *1* до *n.* Получаем следующие правила для вычисления факториала:

*произведение* <- *счетчик* * *произведение*
*счетчик* <- *счетчик* + *1*

Тогда факториал n! - это значение произведения, полученное при значении счетчика, превысившего *n.*

```
(define   (factorial n)
        (define   (iter product counter) (if
        (>  counter n)     product
              (iter (*  counter product)    (+  counter  I ) ) ) )
              (iter l 1 ) )
```

Вычислительный процесс, соответствующий данной процедуре, можно представить следующим образом.

```
(factorial 6)
(iter    1 1)
(iter    1 2)
(iter    2 3)
(iter    6 4)
(iter   24 5)
(iter  120 6)
(iter  720 7)
720
```

Такой процесс является линейным и итеративным.

Сравним первый и второй процессы. Оба вычисляют одну и ту же математическую функцию в одной и той же области определения. Каждый выполняет количество шагов, пропорциональное *n.* Оба процесса выполняют одну и ту же последовательность умножений и получают одну и ту же последовательность промежуточных результатов. Рассмотрим подробнее первый процесс. В нем происходит накопление цепочки отложенных операций (в рассмотренном примере отложенных умножений). Сжатие цепочки происходит, когда эти операции начинают выполняться. Тип процесса, который порождает цепочку отложенных операций, называется *рекурсивным процессом.* Выполнение процесса требует, чтобы интерпретатор хранил трассу операций, которые должны будут выполниться позже. Для вычисления л/ длина цепочки отложенных умножений, которую требуется хранить, пропорциональна я, как и количество шагов вычисления. Такой процесс называется *линейным рекурсивным* процессом.

На каждом шаге второго процесса нам необходимо хранить текущие значения счетчика, произведения и конечного значения счетчика. Такой процесс называем *итеративным*. В общем случае, итеративным называется процесс, состояния которого могут быть описаны фиксированным количеством переменных состояния, правилами перехода из одного состояния в другое и правилом окончания процесса. Для вычисления n! количество шагов пропорционально я. Такой процесс называется *линейным итеративным* процессом.

В итеративном процессе, переменные содержат полное описание состояния процесса в каждой точке. Если мы прервем вычисление между состояниями, то для возобновления вычисления интерпретатору необходимо задать значения трех переменных. В рекурсивном процессе присутствует дополнительная «скрытая» информация, используемая интерпретатором, и не содержащаяся в переменных программы, которая определяет «где находится процесс» при выполнении цепочки отложенных операций. Чем длиннее цепочка, тем больше требуется информации.

Сравнивая итерацию и рекурсию, нельзя путать рекурсивный *процесс* и рекурсивную *процедуру*. Когда мы говорим о рекурсивной процедуре, то имеем в виду факт синтаксиса, обращение процедуры к самой себе. При описании рекурсивного процесса как некоторой схемы, например, линейной рекурсии, мы имеем в виду развитие процесса, а не синтаксис написания процедуры. Рекурсивная процедура (например, iter) может порождать итеративный процесс.

Проблемы в понимании отличий процесса от порождающей его процедуры возникают потому, что реализация большинства языков программирования, например Ada, Pascal или C спроектирована таким образом, что выполнение любой рекурсивной процедуры требует большого количества памяти, которое растет при каждом вызове процедуры, даже когда процесс, порождаемый процедурой, является в принципе итеративным. Как следствие, описать итеративный процесс на таких языках возможно только с помощью специальных конструкций для записи циклов. Реализация Scheme выполняет итеративный процесс на константном пространстве, даже когда он описан рекурсивной процедурой.

*Упражнения*

8. Ниже приведены две процедуры, которые определяют сложение двух положительных целых чисел в терминах процедур inc и dec.

```
(define (+ a b) (if (= a 0) b (inc (+ (dec a) b) ) } )
(define (+ a b) (if (= a 0) b (+ (dec a) (inc b) ) ) )
```

Используя модель подстановок, опишите поведение процессов порожденных процедурами при вычислении (+ 4 5). Какими являются процессы - итеративными или рекурсивными?

### *Символьные выражения*

Определим класс символьных выражений. Каждая из следующих строк содержит символьное выражение.

```
(JOHN SMITH 33 YEARS)
((DAVE 17) (MARY 24) (ELIZABETH 6 ) )
((my house) has (large (light windows)))
```

Символьные выражения состоят из элементов, которые называются атомами, объединенных в список с помощью скобок. Если изменить положение и количество скобок, то вместе с этим изменится структура и смысл выражения. Атомы бывают символьные и числовые. Логические константы также считаются атомами. Символьный атом рассматривается как одно неделимое целое. К символьным атомам применяется только одна операция - сравнение. Общие правила построения символьных выражений

1. Атом есть символьное выражение.
2. Последовательность символьных выражений, заключенная в скобки, является символьным выражением.

### Списки. Селекторы и конструкторы

Для того чтобы оперировать символьными выражениями, рассмотрим операции, с помощью которых можно выделять из выражений подвыражения *(селекторы)* и составлять выражения из имеющихся выражений *(конструкторы).*

Процедура car может быть применена к списку и ее результатом является первый (самый левый) элемент в списке. Этот элемент может быть либо атомом, либо списком. Процедура car от атома не определена.

```
(car ' (a b c) )          -> a
(car ' ( (a) b c d) )     -> (a)
(car 'a)                  -> ошибка
(car ' () )               -> ошибка
```

Процедура cdr от списка возвращает в результате список, полученный из исходного отбрасыванием его первого элемента. Для атома процедура cdr

не определена. Если исходный список состоит из единственного атома, то значением процедуры cdr для этого списка является специальный атом ni 1, обозначающий пустой список.

```
(cdr  '( (a)  b  c d) )   ->    ( b   c
d )
(cdr  '( a ) )              ->    nil
(cdr  '() )                 ->   ошибка
```

При помощи процедур car и cdr можно получить любой элемент списка Например, третий элемент списка x задается выражением.

```
(car   (cdr   (cdr x)))
```

Пусть x есть список (A B C D). Тогда (cdr x) есть (B C D), а (cdr (cdr x)) есть (C D). Таким образом, (car (cdr (cdr x))) есть элемент c, который является третьим в исходном списке.

В библиотеку языка Scheme входят композиции процедур car и cdr. То есть для того, чтобы выбрать элемент из списка, можно воспользоваться библиотечными процедурами:

```
(caar <list>)
(cadr <list>)

(cdddar <list>)
(cddddr <list>)
```

Например, процедура caddr определяется как

```
(define (caddr x) (car (cdr (cdr x)))))
```

В состав библиотеки Scheme входят и более мнемоничные аналоги процедур выбора элемента из списка:

```
(first <list>)
(second <list>)

(eighth <list>)
```

Результатом этих процедур являются соответственно первый, второй, ..., восьмой элемент заданного списка.

```
(first ' ( (a b) c d) )          ->  (a b)
(second ' ( (a b) c d) )          -> c

(eighth ' (a b c d e f g h) )    -> h
```

С помощью процедуры cons можно объединить два символьных выражения в новое символьное выражение. Исходные компоненты выражения (аргументы процедуры cons) можно получить с помощью процедур car и cdr. Если второй аргумент процедуры cons является списком или специальным атомом nil, то результатом cons будет список, полученный включением первого аргумента в качестве первого элемента списка. Если у есть список длины и то (cons x y) будет списком длины *n + 1*. Атом nil удобно рассматривать как список нулевой длины, или пустой список. Это символьный атом, но он играет также специальную роль, обозначая пустой список.

```
(define x  (cons  1  2) )
(car x)                    ->  1
(cdr x)                    ->  2

(define   y   (cons   3   4)   )
(define   z   (cons   x   y)   )
(car (car z) )             -> 1
(car  (cdr  z) )           ->  3

(cons  'a  '( ) )          ->  (a)
(cons  '( a )   '(b c d) )  ->  ( ( a )   b c d)
(cons  "a"  '( b e ) )      ->   ("a" b c)
```

Для того чтобы построить список из элементов можно последовательно воспользоваться процедурой cons:

```
(cons 1 (cons 2 (cons 3 (cons 4 nil)))) ->  ( 1 2 3  4)
```

В Scheme есть также специальная процедура для построения списков

(list **<a$_1$> <a$_2$> <a$_3$>** ... <a$_n$>)

которая эквивалентна

(cons <a$_1$> (cons <a$_2$> (cons <a$_3$> ... (cons <a$_n$> n i l) . . . ) ) )

**Например,**

```
(list  1  2  3  4 )          -»  (1  2  3  4)
(list  'a  (+  3  4)  'c )   -»  (a  7  c)
(list)                       ->  ()
```

Для обработки списков произвольной вложенности, необходимо уметь распознавать, является ли значение символьного выражения атомом или списком, и устанавливать равенство атомов. Для этого можно использовать следующие предикаты.

(atom? <obj>)

**Предикат возвращает значение истина, если <obj> (любой объект данных языка) является атомом.**

```
(atom? 3)       -> #t
(atom? '        -> #t
(atom? car)     -> #t
(atom? '(a  3   -> #f
(atom? '        -> #t
(atom? nil)     -> #t
```

**Узнать, является ли выражение списком, можно с помощью предиката**

(list? <obj>)

**Предикат возвращает значение истина, если <ob j > является списком.**

```
(list? 'a)          -> #f
(list? '())         -> #t
(list? '(a b) )     -> #t
(list? '(a . b))    -> #f
```

**Проверка списка на пустоту:**

(null? <obj>)

**Предикат возвращает значение истина, если <obj> является пустым списком.**

**Для сравнения выражений используют предикаты eq?, сравнивающий значения двух атомов, и equal?, производящий поэлементное сравнение списков. В последнем случае элементы списка сравниваются по значению, тогда как первый предикат производит сравнение адресов в памяти своих аргументов.**

```
(equal?  '(a b)  '(a b))     -> #t
(equal?  3 3 )               -> #t
(equal?  'a  'a)             -> #t
(eq?  '(a b)  '(a b ) )      -> #f
(eq?  3 3 )                  -> #t
```

## Процедуры со списками

**Рассмотрим, как определить процедуру list-ref от двух аргументов, списка items и числа n, которая возвращает *n-ый* элемент списка.**

**Элементы списка пронумерованы, начиная с *1*. Для *n = 1* list-ref возвращает первый элемент списка, полученный в результате применения процедуры car. Иначе list-ref возвращает *(n-l)-ый* элемент списка, полученный в результате применения процедуры cdr к исходному списку.**

```
(define  (list-ref items n)
     (if  (= n 1)
          (car items)
          (list-ref  (cdr items)  (- n 1 ) ) ) )
(define squares  (list 1 4  9 16 25))
(list-ref squares 4)        ->  16
```

**Процедура вычисления длины списка (количества элементов в нем) строится по той же схеме.**

```
(define (length items)
     (if (null? items) 0
          (+ 1 (length (cdr items)))))
(length squares)    ->  5
```

**Длину списка можно вычислить и итеративно:**

```
(define (length items)
     (define (length-iter a count)
          (if (null? a) count
               (length-iteg (cdr a) (+ 1 count))))
(length-iter items 0))
```

**Процедура append строит из двух списков новый, добавляя в начало второго списка элементы первого:**

```
(define (append list1 Iist2) (if (null? listl) list2 (cons
     (car listl) (append (cdr listl) list2))))
```

***Упражнения***

**9. Определить процедуру last-element от одного аргумента, непустого списка, результатом которой является последний элемент заданного списка.**

```
(last-pair  (list  1  2  3  4 ) )        ->  4
```

**10. Определить процедуру reverse, которая по заданному списку строит новый список, элементы которого расположены в обратном порядке.**

```
(reverse (list 1 2 3 4 ) ) - >   ( 4 3 2 1 )
```

## Точечные пары

В общем случае, вторым аргументом процедуры cons может быть как список, так и просто атом. И в первом и во втором случае получаем объект типа *точечная пара.*

```
(cons 'a 3)                 ->  (a . 3)
(cons ' (a Ь) 'c)           ->  ((a Ь) . c)
(cons 'a (b) )              ->  (a Ь)
```

Список является частным случаем пары, в которой самый правый элемент является атомом nil.

(pair? <obj>)

Предикат возвращает значение истина, если <obj> является парой. Заметим, что пустой список является списком, но не является парой Пустой список - это атом.

```
(pair? '(a . b))      -> #t
(pair? '(a b c))      -> #t
(pair? '())           -> #f
(pair? '\#(a b))      -> #f
```

Значение выражения (cons $e_1$ $e_2$) представляется точечной парой ($v_1$.$v_2$), где $v_1$ и $v_2$ суть символьные выражения для значений $e_1$ и $e_2$, независимо от того, какого типа эти значения. Соглашения о правилах записи точечных пар позволяют также использовать и списочную нотацию:

- точка, за которой непосредственно следует открывающая скобка, может быть опущена, так же опускаются открывающая и соответствующая закрывающие скобки:

($v_1$. ($v_2$. ($v_3$ … ($v_k$ . $v_{k+1}$) . . . ) записывается в виде ($v_1$ $v_2$ … $V_k$.$v_{k+1}$)

- точка, за которой непосредственно следует атом nil, также может быть опущена, это относится и к атому nil:

($v_1$ $v_2$ ... $V_k$.nil) записывается в виде ($v_1$ $v_2$ … $v_k$)

Из этого следует, что значение выражения (cons a (cons b (cons c nil))) представляется структурой (a. (b. (c. nil))), которая может быть последовательно сведена к структурам

(a b . (c . nil))
(a b c . nil) (a b
c)

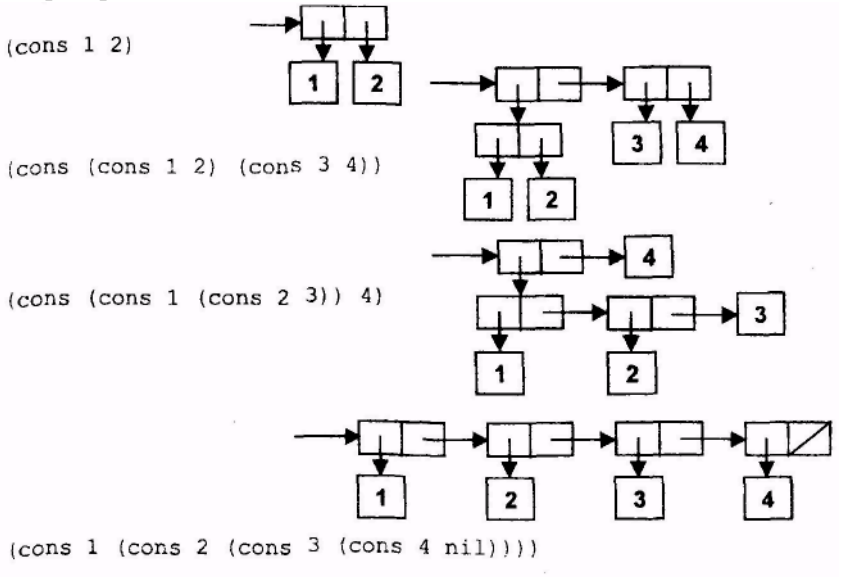
Точечная запись используется в тех случаях, когда, например, необходимо сослаться на первые два элемента списка произвольной длины. В записи (x1 x2 . y) X1 обозначает первый элемент, $x_2$ - второй, y - оставшийся список.

## Графическое представление составных объектов

Объекты можно представлять в виде указателя на некоторый бокс. Бокс простого объекта содержит текстовое представление этого объекта. Бокс для составного объекта (типа пара) - двойной. Левая часть содержит указатель на первый элемент пары (car пары), в правой части находится указатель на второй элемент пары (cdr пары).

Пара является универсальным составным типом для построения любых структур данных. Соответственно, объекты произвольно сложной структуры можно построить с помощью процедуры cons.

Например,



Символьное выражение можно представить в виде двоичного дерева. Если элементы - атомы, то они являются листьями, если это точечные пары, то они являются поддеревьями. Рекурсия является естественным приемом для обработки деревьев. Рассмотрим процедуру count-leaves, которая в качестве результата выдает количество листьев в дереве. В отличие от процедуры length процедура count-leaves считает элементы-атомы на всех уровнях вложенности.

```
(define x (cons (list 1 2) (list 3 4)))
(length x)                          -> 2
(count-leaves x)                    -> 4
(list x x)                          -> ( ( (1 2) 3 4) ((1 2) 3
4))
(length (list x x))                 -> 2
(count-leaves (list  x x))          -> 8
```

Реализация процедуры count-leaves может опираться на следующие утверждения.

1. Количество листьев в пустом дереве равно нулю.
2. Количество листьев в дереве, состоящем из единственного атома, равно единице.
3. Количество листьев в любом другом дереве может быть получено как сумма листьев правого (car) и левого (cdr) поддеревьев.

```
(define (count-leaves x)
     (cond ((null? x) 0)
     ((atom? x) 1)
          (else (+ (count-leaves (car x))
               (count-leaves (cdr x))))))
```

*Упражнения*

11. Вычислите выражение (list 1 (list 2 (list 3 4) ) ) . Определите результат работы интерпретатора, а также приведите соответствующее графическое представление структуры.

12. Определите необходимые комбинации вызовов процедур car и cdr, в результате которых для данных списков можно будет получить элемент 7.

```
(1  3  (5  7)  9)
((7))
(1  (2  (3  (4  (5  ( 6 7 ) ) ) } ) )
```

13. Пусть x и y определены как списки.

```
(define x   (list  1  2  3  ) )
(define y   (list 4  5  6  ) )
```

Вычислите выражения:

```
(append x y)
(cons x y)
(list x y)
```

14. Модифицируйте процедуру reverse, так чтобы получить процедуру deep-reverse, которая по заданному списку, с элементами любой структуры, строит список, элементы которого расположены в обратном порядке. При этом если элементом списка является список, то в полученном списке элементы подсписка расположены также в обратном порядке. Например:

```
(define x   (list   (list  1  2)   (list  3 4 ) ) )
x                        ->   ( (1   2)   (3   4))
(reverse x)              ->   ( (3 4)   (1   2 ) )
(deep-reverse  x)        ->   ((4   3)   (2   1) )
```

15. Определите процедуру fringe, которая по дереву, заданному в виде списка, строит список, содержащий листовые элементы заданного дерева, расположенные слева направо.

```
(define x   (list   (list  1   2)    (list  3  4 ) ) )
(fringe  x)                 ->    (1   2   3  4)
(fringe   (list x x) )   ->    (  1  2  3  4  1  2  3  4  )
```

### *Процедуры высшего порядка*

Часто одни и те же *схемы программ,* отличающиеся друг от друга набором используемых процедур, могут быть использованы для решения различных задач. Для того чтобы определять такие схемы, необходимы процедуры, допускающие использование аргумента процедурного типа, или возвращающие значение типа процедура. Процедуры, которые используют, таким образом, другие процедуры, называются *процедурами высшего порядка.*

#### Процедуры как аргументы

Рассмотрим следующие три примера процедур. Первая процедура вычисляет сумму целых чисел от a до b:

```
(define (sum-integers a b)
     (if (> a b) 0 (+ a (sum-integers (+ a 1) b) ) ) )
```

**Вторая вычисляет *сумму* квадратов целых чисел на том же отрезке:**

```
(define (sum-squares a b)
     (if (> a b) 0 (+ (square a) (sum-squares (+ a 1) b)
))  )
```

**Третья вычисляет сумму ряда: 1/1*3+1/5*7+1/9*11+…**

```
(define (pi-sum a b) (if (> a b) O ( + (/1.0 (* a ( + a
     2 ) ) )  (pi-sum (+ a 4) Ь) ) ) )
```

**Все три процедуры могут быть описаны одной общей схемой. Они отличаются друг от друга только именем процедуры, способом вычисления слагаемого и определением следующего значения a. Любая из них может быть получена из следующего конструктивного шаблона:**

```
(define (<name> a b)
     (if (> a b) 0 ( + (<term> a) (<name> (<next> a) b))))
```

**Математики для записи вычислений сумм используют /*знак суммы*/E-обозначения. В программировании *концепция* суммирования может быть представлена в виде процедуры.**

```
(define (sum term a next b)
     (if (> a b) 0 ( + (term a) (sum (next a) b) ) ) )
```

**Процедура sum имеет 4 аргумента: нижнюю и верхнюю границу суммирования (a, b) и процедуры term и next. Процедуру sum можно использовать так же, как и все предшествующие. Например, для определения описанных выше процедур суммирования.**

```
(define (inc n) (+ n 1))
(define (sum-squares a b)
(sum square a inc b))

(define (identity x) x)
(define (sum-integers a b)
     (sum identity a inc b))

(define (pi-sum a b)
   (define (pi-term x) (/ 1.0 (* x (+ x 2)))))
   (define (pi-next x) (+ x 4)) (sum pi-term a
   pi-next b))
```

В качестве еще одного примера рассмотрим процедуру, которая прибавляет единицу к каждому элементу списка, состоящего из чисел.

```
(define  (inc1-list x)
     (if  (eq?  x nil)  nil
          (cons  (+  (car  x)  1)  (incl-list  (cdr x)))))
```

Имеющая аналогичную структуру процедура mod2-list вычисляет по исходному списку x список остатков от деления каждого из его элементов на 2.

```
(define  (mod2-list x)
     (if  (eq?  x nil)  nil
          (cons  (mod  (car x)  2)  (mod2-list  (cdr x)))))
```

Определим обобщенную процедуру, в которой конкретная операция, которая выполняется над каждым элементом списка, является параметром этой процедуры.

```
(define  (map x  f)
     (if  (eq?  x nil)  nil
          (cons  (f  (car x)  2)  (map  (cdr x)))))
```

Можно переопределить процедуры inc-list и mod2-list через map следующим образом:

```
(define  (inc1  x)  (+ x  1) )
(define  (mod2 x)  (mod x 2))
(define  (incl-list x)  (map x  incl))
(define  (mod2-list x)  (map x mod2))
```

Процедуры, которые используют другие процедуры в качестве аргументов, являются примерами процедур высшего порядка. Разумное использование процедур высшего порядка может привести к простой записи сложных программ.

В качестве другого примера определим процедуру reduce со следующими параметрами x - список, g - бинарная операция, a - константа. Процедура по списку (x1 $x_2$ … $x_k$) вычисляет значение $g(x_1, g(x_2 \ldots g(x_k, a) \ldots))$. Например, (reduce x + 0) вычисляет сумму элементов списка x.

```
(define (reduce x g a)
     (if (eq? x nil) a
          (g (car x) (reduce (cdr x) g a ))))
```

Применяя эту функцию, можно определить операции суммирования и произведения:

```
{define (sum x) (reduce x + 0))
(define (mult x) (reduce x * 1)
```

Еще одна процедура filter по заданному списку строит список, состоящий из элементов заданного списка, удовлетворяющих определенному условию.

```
(define (filter predicate sequence)
     (cond ((null? sequence) nil)
          ((predicate (car sequence))
          (cons (car sequence)
               (filter predicate (car sequence))))
     (else (filter predicate (cdr sequence)))))
```

Например:

```
(filter odd? (list 1 2 3 4 5 ) )     -> (135) Другой
```

пример процедуры высшего порядка - процедура accumulate.

```
(define (accumulate op initial sequence) (if (null?
     sequence) initial (op (car sequence) (accumulate
     op initial (cdr sequence)))))
```

Примеры ее использования:

```
(accumulate + 0 (list 1 2 3 4 5 ) )       -> 15
(accumulate * 1 (list 1 2 3 4 5 ) )      ->  120
(accumulate cons nil (list 1 2 3 4 5 ) )  ->  ( 1 2 3 4 5 )
```

*Упражнения*

**16.Переопределите процедуры** map, append, length, count-leaves C **помощью определенной выше процедуры** accumulate.

```
(define (map p sequence)
     (accumulate (lambda (x y) <??>) nil sequence)

(define (append seql seq2)
          (accumulate cons <??> <??>))

(define (length sequence)
          (accumulate <??> 0 sequence))
```

```
(define (count-leaves t)
          (accumulate <??> <??> (map <??> <??>)))
```

17. Определенная выше процедура sum порождает линейный рекурсивный процесс. Переопределите процедуру sum так, чтобы процесс, порожденный процедурой, был итеративным. Для этого заполните <??> в следующем определении.

```
(define   (sum term a next b)
      (define   (iter a result) (if
      <??> <??>
                (iter <??> <??>)))
      (iter <??> <??>))
```

## lambda выражения

Определенным неудобством при использовании процедур высших порядков является необходимость давать имена процедурам, используемым как фактические параметры, например, pi-next, при определении процедуры pi-sum. Более удобно было бы иметь возможность определять безымянную «процедуру, которая возвращает значение аргумента, увеличенное на 4» . Это можно сделать, используя специальную форму, называемую *lambda выражением:*

```
(lambda   (x)   (+ x 4) )
```

Таким образом, определение pi-sum можно записать без использования лишних имен:

```
(define (pi-sum a b)
     (sum (lambda (x) (/ 1.0 (* x (+ x 2) ) ) ) a
     (lambda (x) (+ 4 x) )
     Ы)
```

В общем случае, синтаксис lambda выражения следующий:

```
(lambda <formals> <body>)
```

где < formals > представляет собой список имен формальных параметров, а <body> является последовательностью из одного или более выражений. Значением lambda выражения является процедура, с которой не связано никакое имя. По существу, определение процедуры (define (plus4 x) (+ x 4 ) ) эквивалентно определению (define plus4 (lambda (x) (+ 4 x)) ).

Как любое выражение, значением которого является процедура, lambda выражение может быть использовано в качестве оператора.

```
(lambda  (x)  (+ x x))                                  ->  процедура
((lambda  (x y z)  (+ x y  (square z) ) )  1  2  3)   ->   12
```

Формальные параметры lambda выражения могут быть описаны одним из следующих способов:

- ($variable_1$ ... $variable_n$) - процедура имеет фиксированное количество аргументов. При вызове процедуры происходит замена формальных параметров фактическими аргументами;
- variable - процедура может иметь произвольное количество аргументов. При вызове процедуры последовательность аргументов преобразуется в список;
- ($variable_1$ . . . $variable_{n-1}$ . $variable_n$) - процедура должна иметь хотя бы *n-1* аргументов. При вызове процедуры происходит замена *n-1* формальных параметров фактическими аргументами, остальные аргументы преобразуются в список.

```
((lambda x  x)  3  4  5  6)              ->  (3  4 5  6)
((lambda  (x y  .  z)  z)  3  4  5  6)   ->  (5  6)
```

## Создание локальных переменных

Для создания локальных переменных в процедуре можно использовать lambda выражения. Рассмотрим пример вычисления функции

$$f(x, y)=x(1+ xy)^2 + y(1-y) + (1+xy)\ (1 -y)$$

Ее можно записать как

$$a = 1 + xy\ b = 1-y$$
$$f(x,y)=-xa^2 + yb + ab$$

При определении процедуры, вычисляющей *f* потребуются не только переменные x и y, но также переменные для хранения промежуточных значений a, b. Можно определить вспомогательную процедуру f-helper и локализовать эти переменные:

```
(define (f x y)
     (define (f-helper a b)
          (+ (* x (square a)) (* y b) (* a b) } )
     (f-helper (+ 1 {* x y}(- 1 y) )
```

Можно использовать lambda выражение для определения анонимной процедуры с такой же возможностью локализации переменных. В этом случае тело процедуры f представляет собой просто вызов процедуры:

```
(define  (f x  y)
      ((lambda  (a b)  ( +  (*  x  (square  a))
            (*  y b)  (*  a b)) ) ( + 1  (* x y) )
            (- 1  y) ) )
```

Существует специальная форма, называемая let, для определения локальных переменных. С помощью let процедуру f можно записать так:

```
(define  (f x y)
      (let  ( ( a  ( + 1  (* x y) ) )  (b
            (- 1 y) ) )  (+  (*  x
            (square a) ) (*  y b)  (*
            a b) ) ) )
```

Общий синтаксис let выражения:

```
(let <bindings> <body>)
```

где <bindings> **имеет вид**

( (<var1> <exp1>)  (<$var_2$> <$exp_2$>)  . . .  (<$var_n$> <$exp_n$>) )

Семантика let выражения следующая, переменная <var1> связывается со значением выражения <exp1>, переменная <$var_2$> - со значением выражения <$exp_2$>, ... и переменная <$var_n$> - со значение выражения <$exp_n$>. Первая часть let выражения является списком пар (имя - значение). Тело let выражения вычисляется в контексте имен, имеющих область видимости, ограниченной let выражением. По существу, let выражение является альтернативной записью конструкции

((lambda  (<$var_1$>...<$var_n$>)  <body>)  <exp1>...<$exp_n$>)

Для спецификации локальных переменных интерпретатору не требуется никаких новых механизмов вычисления. Из альтернативной записи let выражения следует, что областью видимости переменных, определенных в let выражении является тело let. Поэтому:

- let выражение позволяет локализовать переменные там, где они используются. Например, если значением x является 5, то значением выражения (+ (let ((x 3)) (+ x (* x 10))) x) будет 38. Значение x в теле let равно з, значение let выражения равно 33. Однако вне let выражения x имеет значение 5,

- значения аргументов вычисляются вне let выражений. Это имеет значение, если выражение, которое определяет значение локальных переменных, зависит от переменных с теми же именами, что и локальные. Например, пусть x имеет значение 2, тогда (let ((x 3) (y (+ x 2 ) ) ) (* x y)) будет иметь значение 12, так как внутри тела let выражения x будет иметь значение 3, а y - значение 4 (которое получается как внешнее значение x, увеличенное на 2).

Заметим, что при вычислении lambda выражений глобальные переменные вычисляются в месте определения, а не в месте вызова процедуры. Такой механизм вычисления называется *статическим связыванием.* Например:

```
(let   ((n  1))
       (let   ((f   ((lambda  z)    (+  z n)))
       (let   ( ( n   2 ))    (f  3 ) ) ) )
```

При вычислении выражения сначала выполняется локальное связывание n с 1, затем вычисляется lambda выражение, связывая f со значением типа процедура (z + *1),* после этого переопределяется n и затем вызывается f с фактическим параметром з. Так как f связано с определением процедуры (+ z 1), то получаем значение 4.

*Упражнения*

18. Пусть имеется определение процедуры (define  (f g)  (g 2 ) ) . Тогда:

```
(f  square)                                  ->   4
(f   (lambda   (z)   (*   z   (+   z   1) ) ) )      ->   6
```

Что произойдет при вычислении (f f). Объясните.

## Процедуры как возвращаемые значения

В языке Scheme можно определить процедуру (высшего порядка), которая возвращает результат типа процедура. Рассмотрим определение:

```
(define   (incn n)    ((lambda x)(+ x n)))
```

где incn является процедурой, которой в качестве аргумента передается целое число n. Результат применения этой процедуры является процедурой, которая увеличивает на n передаваемый ей аргумент. Например, (incn 3) есть процедура, которая выдает в качестве результата значение аргумента, увеличенное на з. Таким образом,

```
( (incn  3 ) 2 )      ->   5
```

Можно **переопределить процедуру** incl-list **как**

```
(define (incl-list x) (map x (incn 1))
```

Рассмотрим пример процедуры высшего порядка, аргументы которой являются процедурами, и результатом которой также является процедура. Определим процедуру compose. Она принимает в качестве аргументов две процедуры и выдает в качестве результата процедуру, которая применяет последовательно обе исходные процедуры.

```
(define (compose f g) ((lambda x) (f (g x))))
```

Таким образом, она применяет f к результату g. Преимущество процедур высшего порядка состоит в том, что они позволяют разрабатывать программы в терминах более высоких абстракций, чем элементарные конструкции языка. С помощью процедур высшего порядка с такими абстракциями можно работать, так же как и с элементарными конструкциями языка.

В языках программирования существуют ограничения на некоторые действия с элементами языка. Элементами «первого класса» называются конструкции языка, которые могут:

- иметь имя;
- быть переданы в качестве аргументов процедуры,
- быть возвращены в качестве значения процедуры;
- из которых можно строить структуры данных.

В Scheme процедуры являются объектами первого класса, что существенно увеличивает выразительную силу языка. А возможность создавать новые процедуры во время выполнения других процедур, обеспечивает возможность программирования, управляемого данными.

## *Присваивание*

До настоящего момента, все рассматриваемые процедуры можно было считать спецификациями, описывающими требуемый вычислительный процесс. При вызове процедуры происходило вычисление значения выражения для заданных аргументов. При обращении к процедуре с одними и теми же аргументами мы всегда получали один и тот же результат.

Однако разработку программ можно вести в терминах объектов и их состояний, которые изменяются во времени. Состояние объекта может быть описано набором переменных состояния, которые определяют поведение объекта в данный момент времени. Значение переменных состояния изменяется во время вычисления. Для описания изменения значений переменных в языках программирования используется *оператор присваивания.*

При использовании оператора присваивания процесс применения процедур к аргументам не может быть описан с помощью модели подстановок. Оператор присваивания позволяет описывать систему как набор взаимодействующих объектов, каждый из которых может находиться в определенном состоянии и изменять свое состояние. С другой стороны, для описания работы с объектами и оператором присваивания нет простой математической модели.

Программирование без использования оператора присваивания называется *функциональным программированием.* Программирование, в котором используется оператор присваивания, называется *императивным программированием.*

До сих пор переменные Scheme использовались только для *именования* значений. При использовании присваивания и возможности изменять значение переменной, имя должно указывать на некоторую область памяти, в которой хранится значение. Можно описать модель, в которой контекст (окружение) является той областью, в которой хранится значение.

### Оператор присваивания

```
(set!  variable expression)
```

В результате выполнения процедуры set! вычисляется значение выражения <expression>. Значение переменной <variabie> (которая должна быть уже определена с помощью define) становится равным значению выражения <expression>. При этом значение самого выражения set! не определено.

```
(define x 2}     ->  не определено
(+ x 1)          ->  3
(set!  x 4)      ->  не определено
(+ x 1)          ->  5
```

Рассмотрим процедуру вычисления факториала числа. Ее можно записать в императивном стиле:

```
(define (factorial n)
     (let ((product 1) (counter 1)
          (define (iter)
               (if (> counter n) product
                    (begin (set! product (* counter
                product))
     (set! counter (+ counter 1)) (iter)))) (iter)))
```

**При этом порядок записи операторов присваивания играет существенную роль.**

## Изменение структуры данных

**Существуют функции с побочным эффектом, изменяющие внутреннюю структуру данных.**

```
(set-car! <pair> <obj>)
```

**Метапеременная <pair> должна иметь тип пара. Происходит переопределение пары, первым элементом которой становится указатель на <obj >. Значение выражения set-car! не определено.**

```
(set-cdr! <pair> <obj>)
```

**Метапеременная <pair> должна иметь тип пара. Происходит переопределение пары, вторым элементом которой становится указатель на <obj >. Значение выражения set-cdr! не определено.**

## Последовательность выражений

**Если синтаксис языка требует наличие ровно одного выражения, как, например в if выражении, можно воспользоваться объединяющими выражениями sequence и begin.**

```
(sequence expression1 expression2 ...)
(begin expression1 expression2 ...)
```

**При вычислении этих выражений происходит последовательное (слева направо) вычисление подвыражений. Результатом является значение последнего подвыражения.**

```
(begin (set! x 5) (+ x 1) )        -> 6
(sequence (print "4 plus 1 equals ")
     (print ( + 4 1 ) ) )          -> не определено
```

В последнем случае будет напечатано «4 plus 1 equals 5». А значение всего выражения не определено, так как значение последнего выражения print также не определено.

# Глава 3

## Решение задач и механизмы эволюции

Традиционные способы решения задач основаны либо на детерминированных моделях (существует четкий алгоритм вычисления неизвестных параметров по исходным данным, например, теорема Пифагора), либо на вероятностных моделях, применяемых в сочетании со статистическим подходом (исходные данные получаются по выборкам наблюдаемых значений). При работе с такими моделями используются хорошо известные математические методы: методы линейного программирования, градиентные методы, полный перебор.

Использование детерминированных моделей несомненно является «наилучшим» способом: используется найденная и доказанная формула, которая во всех случаях дает решение поставленной задачи. Но, к сожалению, подходящие формулы существуют далеко не для всех задач. Тогда на помощь приходят методы перебора, традиционно используемые при решении широкого класса задач, имеющих комбинаторную природу.

Сущность любой комбинаторной задачи можно сформулировать следующим образом:

«Найти на множестве *X* элемент *x,* удовлетворяющий совокупности условий *K(x),* в предположении, что пространство поиска *X* содержит некоторое конечное число различных точек».

Если требуется найти элемент, принадлежащий некоторому конечному множеству, можно перебирать все элементы, пока не найдется подходящий. Этот метод прост, надежен и хорошо работает, если мощность множества невелика. Такой метод полного перебора можно усовершенствовать, вместо рассмотрения всего элемента *y.* можно ограничиться некоторым его фрагментом, достаточным для проверки выполнения (невыполнения) условия *K(x).* При этом наиболее благоприятным является случай, когда можно определить градиент.

Суть градиентного метода состоит в том, что для скорейшего достижения вершины надо продвигаться по самому крутому склону. Каждой конкретной точке поискового пространства ставится в соответствие некоторая числовая оценка, достигающая экстремума в целевой вершине, и на каждом шаге выбирается действие, которое дает максимальное приращение оценочной функции. К градиентным методам относятся симплекс-метод в линейном программировании, A*-процедура, методы

последовательных приближений в численном анализе, минимаксные методы, метод ветвей и границ, динамическое программирование [8].

Но для многих практических задач (например, прогнозирование курса валют, оптимальное распределение инвестиций) применение традиционных методов решения связано с рядом трудностей. В частности, исходных данных может быть недостаточно для описания полной модели, алгоритм решения задачи может быть вовсе неизвестен, а использование математических методов решения может привести к неприемлемым затратам вычислительных ресурсов и времени. Также стоит отметить, что математические методы решения накладывают существенные ограничения на рассматриваемые модели. Например, симплекс-метод применим только к линейным функциям, а градиентные методы в чистом виде не пригодны для задач глобальной оптимизации, статистические же методы хорошо развиты только для одномерных случайных величин, а вычисление многомерных статистических моделей зачастую требует использования эвристических подходов.

Поэтому среди современных аналитических технологий определенную популярность приобретают методы, имитирующие процессы развития природы (рассматриваемые ранее только в теории искусственного интеллекта). В частности, принципы работы человеческого мозга послужили основой для создания теории нейронных сетей. А процессы, описанные Ч. Дарвином в его теории эволюции, легли в основу так называемых генетических алгоритмов. Нейронные сети, генетические алгоритмы и их комбинации являются наиболее изученными и проработанными методами из этого семейства аналитических технологий.

Эксперименты показали, что механизмы генетической эволюции успешно используются при решении задач, для которых не существует четких алгоритмов и пространство возможных решений велико. Например, задачи оптимизации, прогнозирования, классификации, разного рода комбинаторные задачи. По шкале сложности, принятой в теории алгоритмов, такие задачи, как правило, относятся к классу NP-полных задач. На практике, эти методы применяются для эвристического сокращения полного перебора в экспертных системах, распознавания образов, прогнозирования доходов, составления оптимальных маршрутов, расписаний, проектирования и оптимизации сетей связей и т.д.

### *Генетические алгоритмы*

*Генетический алгоритм* (ГА) представляет собой метод поиска оптимальных решений, основанный на имитации эволюционных

процессов развития популяции хромосом (смена поколений, естественный отбор, наследование, мутация).

Основные понятия генетического алгоритма:

1. *Ген* - элементарная единица информации (например, 0 и 1).
2. *Хромосома* - последовательность генов (например, битовая последовательность из 0 и 1), кодирующая некоторую полезную информацию.
3 *Популяция* - множество хромосом, которое обрабатывается с помощью алгоритмов репродукции, изменчивости, генетической композиции. Наиболее важные среди них - скрещивание генетического материала, содержащегося в родительских хромосомах и случайные мутации данных в индивидуальных хромосомах.
4. *Скрещивание* - в популяции случайным образом выбираются две хромосомы-родители; произвольный номер гена разбивает каждую хромосому на две части; два потомка получаются в результате обмена соответствующими составными частями хромосом-родителей; размер хромосом при этом не изменяется.
5. *Мутация* - в популяции случайным образом выбирается одна хромосома; в ней выбирается некоторый ген, значение которого заменяется произвольным из области допустимых значений.
6. *Пригодность хромосомы* - неотрицательное число, определяющее степень пригодности хромосомы для решения задачи.
7. *Оценочная функция* - функция одного аргумента, которая для каждой хромосомы возвращает значение ее пригодности.
8. *Естественный отбор-выбор* «лучших» хромосом для участия в дальнейшей репродукции популяции.

Для решения задачи с помощью ГА сначала строится исходная популяция, представляющая собой множество возможных решений, полученных произвольным образом и закодированных в виде хромосом. Как правило, задачи, решаемые ГА, имеют большое пространство возможных решений. Начальная популяция состоит из некоторого их подмножества. Например, для поиска экстремума функции на отрезке, в качестве исходной популяции можно взять все числа из этого отрезка, расположенные на расстоянии некоторого фиксированного интервала друг от друга, или просто случайные числа из этого отрезка.

Главная проблема построения алгоритмов генетической эволюции - это кодирование информации. В клетке любого живого организма хромосома представляет собой молекулу ДНК (Дезоксирибонуклеиновая Кислота). Одна молекула ДНК - это цепочка, состоящая из нуклеотидов четырех типов (A, T, C и G), представляющих минимальную единицу генетической информации. В программировании традиционно элементарной единицей

информации является бит. По аналогии с ДНК, в программной системе, использующей ГА, хромосома может представлять собой битовую последовательность. Для достаточно широкого класса задач, такое представление дает хорошие результаты. Например, в задаче нахождения экстремума функции на отрезке, битовая хромосома может интерпретироваться как двоичная запись аргумента функции. Для более сложных задач роль гена могут играть элементы, устроенные более сложно, чем один бит. Заметим только, что в результате операций репродукции, могут получаться произвольные комбинации генов, которые должны иметь осмысленную интерпретацию.

Репродукция популяции происходит в условиях естественного отбора, предполагающего выживание и размножение «лучших», наиболее приспособленных к конкретным условиям, хромосом. Для реализации концепции отбора необходимо ввести способ сопоставления (сравнения) различных хромосом в соответствии с их возможностями решения поставленной задачи. В программной системе роль окружающей среды играет оценочная функция. Оценочная функция для каждой хромосомы возвращает число, определяющее степень ее пригодности. Собственно говоря, именно в оценочной функции содержится алгоритм расшифровки хромосомы. Например, при поиске максимума на отрезке роль оценочной функции может играть сама функция, заданная в условии задачи. Стоит отметить, что при использовании важно не просто абсолютное значение оценочной функции, а возможность применения ее как меры для сопоставления хромосом. Для осуществления более точного и гибкого отбора оценочная функция должна иметь достаточно широкую область значений (0 и 1, да и нет, будет явно недостаточно).

В процессе регенерации популяции участвуют только хромосомы, обладающие достаточно высокой степенью пригодности. Выбор «хороших» хромосом-родителей может происходить по разным алгоритмам. Простейший (но дающий хорошие результаты) алгоритм заключается в упорядочение всех хромосом по убыванию их пригодности и отбрасывании худшей половины. После этого, из оставшейся части хромосомы для дальнейшей регенерации популяции могут выбираться случайным образом. Другой алгоритм выбора хромосом-родителей носит название «рулетки». Хромосомы также упорядочиваются по убыванию их пригодности, затем пригодности всех хромосом суммируются. Для выбора каждой хромосомы-родителя берется случайное число из интервала от нуля до этой Суммы. Перебирая упорядоченную последовательность хромосом, снова суммируется их пригодность. При этом выбирается та хромосома из последовательности, на пригодности которой сумма достигает заданное число. Очевидно, что при таком алгоритме, вероятность быть выбранными значительно выше у хромосом, расположенных в начале последовательности и, следовательно, имеющих

большую степень пригодности. Заметим, что и в первом, и во втором случаях одна хромосома может несколько раз принимать участие в регенерации популяции.

На первом шаге ГА, начальная популяция порождается случайным образом. Каждая следующая популяция получается в результате репродукции с участием «лучших» хромосом. При этом та или иная модель отбора, определяет, каким образом следует строить популяцию следующего поколения. Возможна полная замена популяции, «старые» хромосомы из предыдущего поколения «умирают», давая место новым. Не менее популярна так называемая стратегия «элитизма», при которой несколько лучших хромосом переходят в следующее поколение без изменений. В любом случае каждое следующее поколение будет в среднем лучше предыдущего. Будем считать, что размер популяции при смене поколений остается постоянным.

Смена поколений в популяции является итеративным процессом. В качестве решения задачи оптимизации будет выбрана наилучшая хромосома из последней популяции. При этом решение об остановке ГА может быть принято в соответствии с различными стратегиями. Например, при инициализации генетического эксперимента можно задать максимальное число итераций. В этом случае независимо от получаемого на каждом шаге результата ГА проработает фиксированное число раз. Если же ГА работает в интерактивном режиме, то пользователь после каждой итерации может получать наилучшую хромосому, оценивать соответствующее ей решение задачи и принимать решение об остановке или продолжении генетического эксперимента. Более гибкий алгоритм строится по аналогии с численными методами и основывается на задании точности, с которой необходимо найти решение. Процесс останавливается, когда пригодность хромосом перестает заметно улучшаться (т.е. пригодность наилучшей хромосомы из последней популяции отличается от пригодности наилучшей хромосомы из предыдущей не более чем на величину заданной точности). Также по аналогии с терминологией, принятой в численных методах, можно говорить о сходимости ГА. Скорость сходимости ГА для каждой конкретной задачи определяется многими факторами. Во-первых, на работу алгоритма влияют его параметры:

- интенсивная мутация снижает скорость алгоритма и может серьезно повлиять на сходимость;
- слишком малая вероятность мутации затрудняет выход алгоритма из локальных экстремумов;
- большие популяции сходятся дольше;
- при малом размере популяции возрастает вероятность остановки в локальном экстремуме.

Также скорость сходимости ГА зависит от модели отбора, алгоритма выбора хромомосом-родителей.

В простейшей модели ГА будем рассматривать только две стратегии репродукции популяции - скрещивание двух хромосом и мутацию Скрещивание определяет поступательное развитие популяции и состоит в наследование генетического материала достаточно «хороших» родителей. В результате скрещивания доля «хороших» хромосом в общей массе будет возрастать.

Поступательное развитие в условиях некоторой среды в общем случае не является монотонным и непрерывным. На пути к глобальному экстремуму возможно бесконечное множество локальных. Выбирая лучших в небольшой окрестности локального экстремума, существует опасность вырождения (зацикливания) ГА. И глобальный экстремум, являющийся конечной целью оптимизации, никогда не будет достигнут Мутация привносит необходимый элемент случайности, позволяющий преодолеть локальный экстремум. При этом мутации должно подвергаться достаточно небольшое количество хромосом, чтобы разброс значений пригодности хромосом не был слишком велик, и ГА достаточно быстро сходился. Очевидно, что мутации могут быть как полезными, так и ухудшающими хромосомы. В первом случае механизм наследования (скрещивание) закрепит это свойство в следующих поколениях. В противном же случае, механизм естественного отбора сократит влияние регрессирующих хромосом на остальную популяцию, и со временем нежелательное свойство будет удалено из популяции.

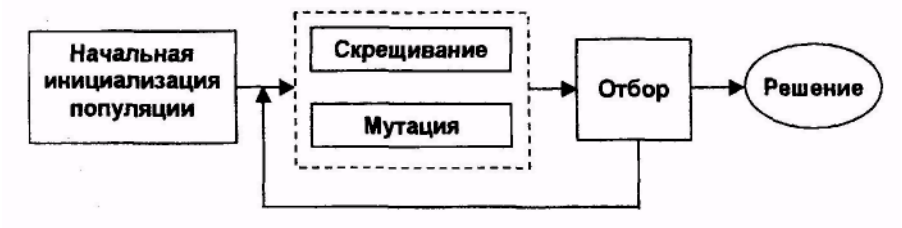



Рис. 11. Генетический алгоритм

Возможностью мутации выгодно отличает ГА от градиентных алгоритмов, которые в чистом виде пригодны только для унимодальных задач (целевая функция имеет единственный локальный, он же и глобальный, экстремум). В то же время к достоинствам метода градиентного спуска относится высокая скорость нахождения экстремума. В свою очередь, полный перебор хотя и гарантирует нахождение действительно оптимального решения, но требует слишком больших вычислительных затрат. ГА можно

рассматривать как комбинацию техники полного перебора (мутация, скрещивание) и градиентных методов (естественный отбор).

Итак, последовательность шагов генетического алгоритма следующая (Рис. 11). Случайным образом инициализируется популяция, и все хромосомы сравниваются в соответствии с выбранной функцией оценки. Далее (возможно многократно) выполняется процедура репродукции популяции. Хромосомы-родители выбираются случайным образом пропорционально степени их пригодности. Репродукция происходит либо индивидуально для одного родителя путем мутации хромосомы, либо для двух родителей путем скрещивания. Получившиеся хромосомы оцениваются в соответствии с заданной оценочной функцией и помещаются в популяцию. В качестве решения выбирается хромосома с наибольшей степенью пригодности из последней полученной популяции.

### *Графы*

Одним из наиболее распространенных способов представления данных, позволяющих описывать сложные зависимости между элементами, является граф (и его частные случаи, имеющие самостоятельное значение, - деревья, бинарные деревья, списки и т.д.). Представление информации в виде графов и алгоритмы работы с ними широко применяются при построении компиляторов, различных автоматов, решении комбинаторных задач и т.д.

Алгоритмы работы с графами давно известны и хорошо проработаны. Но каждая программная система, использующая методы анализа графов, сталкивается с выбором внутреннего представления для их хранения и обработки. Эффективность работы с графами, которые для реальных задач могут содержать большое число узлов и ребер, во многом определяется удачностью выбора внутреннего представления. Граф представляет собой совокупность множества вершин и ребер *G(V, E)*. Как правило, информационные структуры, соответствующие вершинам (V) и ребрам графа (E), имеют последовательное представление. Для отображения *E —> V * V* (каждому ребру ставится в соответствие пара вершин - начало и конец) используются различные способы: матрицы смежности, матрицы инцидентности, списочные структуры. Каждый способ имеет определенные достоинства и недостатки, и выбор определяется спецификой конкретной задачи.

В реальных задачах исходная информация для инициализации конкретного графа отличается большим объемом и хорошей структурированностью. Она может быть получена как непосредственно от пользователя, так и из другого приложения. В качестве внешнего представления удобно

использовать текстовый файл, содержащий описание всех вершин и ребер графа. Такое внешнее представление также можно использовать для обмена данными между разными приложениями. Как следствие, при работе с графами присутствует предварительный этап преобразования информации о конкретном графе во внутреннее представление системы. Преобразование графа из внешнего формата во внутреннее представление и обратно, а так же функции доступа к отдельным элементам графа и их свойствам могут быть оформлены в виде библиотеки.

### *Пример решения задачи*

Рассмотрим последовательность шагов создания программы с использованием приемов быстрого прототипирования на примере решения задачи о нахождении пути между двумя вершинами на графе [5].

Жизненный цикл любой программы включает анализ требований, проектирование и создание реализации, удовлетворяющей этим требованиям. Назовем эту реализацию первым прототипом программы. Для простых задач первый прототип может быть и последним. В общем же случае, требования анализируются еще раз, уточняются и детализируются. В соответствии с этим пересматривается проект программы и создается второй прототип. Заметим, что важность этапа проектирования заметно возрастает при таком подходе. Для использования преимуществ технологии прототипирования программа должна быть гибкой, легко расширяемой и хорошо документированной.

И первый, и второй прототип - это законченные работающие программы. Первый прототип может отличаться от второго детальностью требований, эффективностью. Второй прототип строится на основе первого, и может рассматриваться как его расширение. В общем случае, «спиральная» модель жизненного цикла программы может включать несколько прототипов.

Для поиска пути на графе существует две стратегии: поиск вглубь и поиск вширь. Алгоритм поиска вглубь предполагает последовательное рассмотрение всех возможных путей, начинающихся из стартовой вершины. На первом шаге текущей является стартовая вершина, и рассматриваются все вершины, непосредственно связанные с ней. Из этих вершин выбирается одна (например, первая) и становится текущей. Если текущая вершина является искомой, то алгоритм останавливается. Иначе из списка вершин, непосредственно связанных с данной, выбирается следующая текущая вершина. Процесс повторяется, пока искомая вершина не будет достигнута или все возможности поиска по этому пути не будут исчерпаны (достигнута точка возврата). Если при этом искомая вершина

**не была достигнута, то алгоритм возвращается на предыдущий шаг, и в качестве текущей выбирается следующая вершина из списка (backtracking). Заметим, что на каждом шаге достаточно хранить информацию только об одном текущем пути. Алгоритм прост и подразумевает рекурсивную реализацию. К достоинствам этого алгоритма можно отнести минимальные расходы памяти на поддержание рекурсии. Альтернативный алгоритм поиска вширь просматривает все возможные пути одновременно, начиная со стартовой вершины, и пока искомая вершина не будет достигнута на одном из путей, либо все пути не будут отвергнуты.**

**С точки зрения реализации поиск вглубь является более простым алгоритмом, и поэтому для первого прототипа программы логично использовать именно его. Переменная nodes задает список всех вершин графа с их координатами. В переменной paths хранится список ребер графа. Функция init-length строит новый список, содержащий информацию о ребрах графа и их длинах, вычисленных на основе координат вершин с помощью функций slow-path-length и dist-between-points. Функция find-connected-nodes возвращает список вершин, непосредственно связанных с заданной. Функция depth ищет путь между двумя заданными вершинами, используя алгоритм поиска вглубь.**

```
(define (y-coord x) (truncate (cadr x)))
(define (x-coord x) (truncate (car x)))

(define nodes '
     (
          (n1 (60 402))
          (n2 (50 310))
          (n3 (110 60))
          (n4 (220 375))
          (n5 (192 220))
          (пб (260 44))
          (n7 (305 300))
          (n8 (347 404))
          (n9 (351 262))
          (nl0 (400 110))
          (nil (415 404) )))

(define paths '
     (
          (n1 n2) (n2 n3) (n3 n5) (n3 n6) (n6 nl0)
          (n9 nl0) (n7 n9) (n1 n4) (n4 n2) (n5 n8)
          (n8 n4) (n7 nil)))

(define (init-lengths pathlist)
     (let ((new-path-list '())
          (pathlength 0)
          (path-with-length '())) (do
     ((path pathlist (cdr path)))
```

```
          ((null? path))
          (set! pathlength (slow-path-length (car path)))
(set! path-with-length
               (append (car path) (list pathlength)))
          (set! new-path-list
               (cons path-with-length new-path-list)))
         new-path-list))

; "As the crow flies" distance between
; the starting and ending nodes on a path:

(define (slow-path-length path)
     (let ((node1 (car path))
          (node2 (cadr path))) (let
     ((nl (assoc node1 nodes))
          (n2 (assoc node2 nodes))) (dist-between-
     points (cadr nl) (cadr n2)))))

; Calculate the Cartesian distance between points:

(define (dist-between-points point1 point2)
     (let ((x-dif (- (X-coord point2) (X-coord point1)))
     (y-dif (- (Y-coord point2) (Y-coord point1)))) (sqrt
     (+ (* x-dif x-dif)  (* y-dif y-dif)))))

; Change the global path list to include distance between
; adjacent nodes:

(set! paths (init-lengths paths)) ;

(pp paths)

(define (find-connected-nodes a-node)
     (let ((ret-list '()))
          (do ((l paths (cdr l)))
          ((null? l))
     (let ((path (car l))) ; (node1 node2 distance)=path
     (if (equal? a-node (car path))
          (set! ret-list (cons (cadr path) ret-list)))
     (if (equal? a-node (cadr path))
          (set! ret-list (cons (car path) ret-list)))))
     ret-list))

; (find-connected-nodes 'n2)

(define (depth path goal) (if (equal? (car
(last-pair path)) goal) path ; done with
search (let ((new-nodes (find-connected-
nodes
                    (car (last-pair path)))) (keep-
     searching #t) (ret-val '())) (do ((nn new-
     nodes (cdr nn)))
```

```
            ( (or
            (null? nn)
            (equal? keep-searching #f)))
            (if (not (member (car nn) path))
            (let ((temp (depth
                     (append path (list (car nn)))
                 goal)))
            (if (not (null? temp))
                 (begin
(set!  ret-val temp) (set!  keep-searching #f) ) ) ) ) )
ret-val)))
```

**Функции testl и test2 используются для проверки работы функции depth. Функция testl изображает на экране граф, а функция test2 рисует на нем найденный между двумя вершинами путь.**

```
;  Test code with graphics support:

(load "Graph.ss") (define
(testl) (open-gr) (pen-width
1) (do  ((p paths  (cdr p)))
      ((null? p ) )
    (display "(car p)=") (display (car p)) (newline)
    (let ((from (cadr (assoc (caar p) nodes))) (to
    (cadr (assoc (cadar p) nodes)))) (plot-line
  (x-coord from) (y-coord from)
  (x-coord to) (y-coord to)
  "black"))) (do ((n nodes (cdr
  n)))
      ((null? n))
    (let ((n-val (cadar n)))
      (plot-string
        (+ 2 (x-coord n-val)) (y-coord
        n-val) (symbol->string (caar
        n))))))

(define (test2)
  (define (draw-path pi)
    (pen-width 3)
    (let ((nodel (cadr (assoc (car pi) nodes))))
      (set! pi {cdr pi)) (do ((p pi (cdr p)))
      ((null? p))
        (plot-line (x-coord nodel)
                   (y-coord nodel)
                   (x-coord (cadr (assoc (car p) nodes)))
                   (y-coord (cadr (assoc (car p) nodes)))
```

```
                    "red")
        (set! nodel (cadr (assoc (car p) nodes))))))
    (draw-path (depth ' (n1) 'nil)))


(testl)
(test2)
```

**Чтобы проследить последовательность рекурсивных вызовов функции depth, можно воспользоваться встроенной функцией Scheme trace.**

**Функция depth проста в реализации и для небольших графов эффективна. Но если первый выбранный путь является ошибочным и граф достаточно велик, то потеря в эффективности может быть существенна. Программа рекурсивно просматривает неверный путь до конца, и затем поднимается по каждой точке возврата. Для реальных поисковых задач пространство поиска велико и требуется определенная устойчивость по времени поиска в зависимости от исходных данных. В среднем стратегия поиска вширь является более эффективной, чем поиск вглубь. Поэтому требования к программе, реализующей поиск пути в графе, дополняются требованием эффективности для «средних» случаев поиска. Созданный первый прототип программы уже не удовлетворяет этому требованию.**

**Для построения следующего прототипа программы, отвечающего расширенным требованиям, можно воспользоваться уже написанной программой, просто заменив в ней функцию depth на функцию breadth. Все вспомогательные функции и функции рисования графа остаются без изменения, и могут быть повторно использованы в следующем прототипе программы.**

```
(define  (breadth  start-node  goal-node) (let
  ((visited-list  (list  start-node))
  (search-list  (list
       (list  start-node  start-node  0 . 0 ) ) ) )

   (define  (next  s-list) (let
     ((new-s-list  ' ())) (do  ((1
     s-list  (cdr 1)))
     .  ((null?  1)) (let
     ((path  (car  1))) (let
     ((last-node  (car
                 (last-pair (except-last-pair path)))))
          (let ((connected-nodes
              (find-connected-nodes last-node)))
          (do ((n connected-nodes (cdr n)))
              ((null? n))
            (if (not (member (car n) visited-list))
                (begin
                  (set! new-s-list (cons (append
```

```
                    (except-last-pair path)
                    (list (car n)) (list (+
                      (car (last-pair path)) ; old distance
                      (slow-path-length (list (car (last-pair
                    (except-last-pair path))) (car n))
                  ) ))) new-s-list)) (set! visited-
                  list
                    (cons (earn) visited-list))))))))))
    new-s-list))

    (define (found-goal-node?)
  (let ((good-path '())) (do
((1 search-list (cdr 1)))
((null? 1)
     (not (null? good-path))) (if
  (member goal-node (car 1))
      (begin
   (set!
    good-path
    (car 1))
(display "Found a good path:")
(display good-path) (newline))))
good-path))
    (let ((a-good-path •()))
      (do ((iter 0 (+ iter 1) ) )
  ((or
(equal? iter (length nodes)) (not
(null? a-good-path) ) ) ) (set!
search-list (next
search-list))
(newline)
(display "search level=") (display iter) (newline)
(display "current visited list:") (newline) (display
visited-list) (newline) (display "current search
list:") (newline) (display search-list) (newline)
(set! a-good-path (found-goal-node?))) (cdr a-good-
path))))
```

**Функция breadth, реализующая стратегию поиска вширь, несколько сложнее предшествующей функции depth из-за необходимости на каждом шаге хранить информацию обо всех рассматриваемых путях.**

**Печать последовательности шагов функции breadth, позволяет убедиться в том, что реализованная стратегия осуществляет поиск вширь. Несмотря**

**на то, что найденный путь совпадает с путем, найденным с помощью предыдущей реализацией программы, и результаты работы функции test2 совпадают для первого и второго прототипов.**

```
» (breadth 'n1 'nil)

search level=0
current visited list:
(n4 n2 n1)
current search list:
((n1 n1 n4 162.2621335986927) (n1 n1 n2
92.5418824100742))

search level=1 current
visited list:
(n3 n8 n4 n2 n1)
current search list:
((n1 n1 n2 n3 349.64108505372303) (n1 n1 n4 n8
292.53108615454926))

search level=2
current visited list:
(n6 n5 n3 n8 n4 n2 n1)
current search list:
((n1 n1 n2 n3 n6 500.49200483878764) (n1 n1 n2 n3 n5
529.4298499974759))

search level=3 current visited list: (n10
n6 n5 n3 n8 n4 n2 n1) current search list:
((n1 n1 n2 n3 n6 n10 655.2692641503545))

search level=4 current visited list: (n9 nl0
n6 n5 n3 n8 n4 n2 nl) current search list:
((n1 n1 n2 n3 n6 nl0 n9 814.9721132169882))

search level=5 current
visited list:
(n7 n9 nl0 n6 n5 n3 n8 n4 n2 n1)
current search list:
((n1 n1 n2 n3 n6 nl0 n9 n7 874.6378487776934))

search level=6 current
visited list:
(n11 n7 n9 nl0 n6 n5 n3 n8 n4 n2 nl)
current search list:
((n1 n1 n2 n3 n6 nl0 n9 n7 n11 1026.0181645390235))
Found a good path:(nl nl n2 n3 n6 nl0 n9 n7 n11
1026.0181645390235)
```

# Глава 4

# Постановка задач и методические указания

Задание практикума состоит из двух частей: разработки библиотеки (генетического алгоритма или работы с графами) и решения с помощью библиотеки одной из предложенных задач.

Разработка библиотеки предполагает создание повторно используемых компонент. Обладая достаточно сложной функциональностью, библиотека может иметь очень простой внешний (пользовательский) интерфейс. При проектировании сложных библиотек не менее важен внутренний интерфейс, определяющий структуру, модульность самой библиотеки. В частности, хорошо продуманная внутренняя структура должна позволять легко заменять (перегружать) внутренние функции, допускающие различную реализацию (например, различные модели естественного отбора).

Решение конкретной задачи - это пример создания собственного небольшого приложения, демонстрирующего функциональность, гибкость и удобство использования библиотеки. В соответствии со «спиральной» моделью жизненного цикла программы, проектирование приложения целесообразно осуществлять поэтапно по мере уточнения (усложнения) задачи. При этом на каждом шаге создается работающий прототип программы, который затем может быть расширен. Постановка предлагаемых задач уже включает элементы последовательного уточнения.

### *Генетический алгоритм*

Задание - написать библиотеку генетического алгоритма (ГА) и с ее помощью решить поставленную задачу.

Для реализации предлагается простейшая модель генетического алгоритма. Хромосомы кодируются с помощью 0 и 1. Методы репродукции включают одноточечное скрещивание и мутацию хромосом. Размер популяции при смене поколений не изменяется. Длина хромосом также остается постоянной. Вероятность мутации задается как параметр генетического эксперимента.

Решение об окончании работы генетического алгоритма принимается либо при достижении заданной точности, либо алгоритм осуществляет заданное число итераций и останавливается. В качестве окончательного

решения выбирается «лучшая» хромосома в последней популяции. При этом предполагается возможность просмотра всего процесса пошагового решения (т.е. для каждой промежуточной популяции выводится «лучшая» хромосома и ее пригодность). Просмотр пошагового решения может также включать визуализацию процесса решения (например, диаграмму пригодности).

Реализация библиотеки ГА включает спецификацию, код и набор тестов для демонстрации работоспособности библиотеки. Решение задачи включает отображение задачи на понятия ГА, наличие нескольких, последовательно уточняемых прототипов программы и визуализацию результатов и процесса пошагового решения.

## *Список задач*

**1. Нахождение максимума функции.**
Найти экстремум функции на заданном промежутке.

1-ый прототип: функция и промежуток заданы:f(x) = *x*+ |*sin (32 x)*|, *x*∈ *[0, pi)*.

2-ой прототип: произвольная функция от двух переменных.

**2. Построение треугольника по трем отрезкам.**
Отрезки произвольной длины перемещаются на экране, пока не получится треугольник.

1-ый прототип: две точки фиксированы в системе координат, одна свободно перемещается на плоскости.

2-ой прототип: все три точки свободно перемещаются (ориентация треугольника на плоскости произвольна).

**3. Составное число.**
Задано положительное целое число *N.* Существуют ли такие целые числа *m* и n *(m, n > 1),* что *N = m*n*?

1-ый прототип: числа *m, n, N* представимы с помощью разрядной сетки машины.

2-ой прототип: числа *m, n, N.* не представимы с помощью разрядной сетки машины (иными словами встроенные арифметические функции не могут быть использованы).

**4. Линейная делимость.**
Даны целые положительные числа a, c. Существует ли такое целое *x,* что *c* делится нацело на *a -x* + 1?

1-ый прототип: числа a, *c* представимы с помощью разрядной сетки машины.

2-ой прототип: числа *a, c* не представимы с помощью разрядной сетки машины (иными словами встроенные арифметические функции не могут быть использованы).

**5. Упаковка в контейнеры.**

Заданы конечное множество *U* предметов, размер $s(u) \in Z^+$ каждого предмета $u \in U$, положительное целое число *K,* и положительные целые число $B1, \ldots, B_k$ - вместимости контейнеров. Существует ли такое разбиение множества U на непересекающиеся подмножества $U1, U2, \ldots, U_k$, что сумма размеров предметов из каждого подмножества м, не превосходит $B_j$?

1-ый прототип: все контейнеры одинаковой вместимости ($B_i - B_i \; \forall \; i,j$).

2-ой прототип: контейнеры разной вместимости.

**6. Рюкзак.**

Заданы конечное множество *U* предметов, размер $s(u) \in Z^+$ и стоимость $v(u) \in Z^+$ каждого $u \in U$, положительные целые числа *B и K.* Существует ли *N* таких различных подмножеств $U' \subseteq U$, что £ „ $_{\pounds}$ *y s(u)* $< B$ и £ „ $_{e}$ у $v(u) > Kl$

1-ый прототип: нахождение одного такого подмножества.

2-ой прототип: нахождение *N* таких различных подмножеств.

**7. Подпоследовательность чисел.**

Даны натуральные числа *m, a;, …, a,,.* В последовательности *a,, .., a,,* выбрать такую подпоследовательность $a,j, a,_2, \ldots, a_{ik}$ *(0 <= И < (2 < ... < ik < n),* что $a_{ц} + \ldots$ ч- $a\& = m$.

1-ый прототип: нахождение одной такой подпоследовательности.

2-ой прототип: нахождение *N* таких различных подпоследовательностей.

**8. Раскрашиваемость графа в *k* цветов.**

Задан граф *G = (V, E).* Можно ли *G* раскрасить в *k* цветов? Другими словами, существует функция/ *У -> { 1, 2, …, k},* такая, что *f(u) ?f(v),* если *{u, v} ∈ E?*

1 -ый прототип: $k = 2$.

2-ой прототип: $k > 4$.

**9. Группы предметов.**

Имеется *n* предметов, веса которых равны a1,…, an. Разделить эти предметы на *m* групп так, чтобы общие веса групп были максимально близки.

1-ый прототип: $m = 2$.

2-ой прототип: $m > 2$.

**10. Доминирующее множество.**
Заданы граф $G$ = (V, E) и целое число $K$, $K < |V|$. Существует ли такое подмножество $V' \subseteq V$, что $|V| \leq K$ и каждая вершина v $\in V \setminus V'$ соединена ребром по крайней мере с одной вершиной из $V'$?

1-ый прототип: нахождение одного подмножества.

2-ой прототип: нахождение $N$ таких различных подмножеств.

**11. Гамильтонов цикл (путь).**
Задан граф G = *(V, E).* Имеется ли в G гамильтонов цикл (путь)?

1-ый прототип: гамильтонов цикл (необходимо пройти через все вершины).

2-ой прототип: гамильтонов путь (необходимо пройти через все ребра).

**12. Поиск пути в графе.**
Задан граф G = *(V, E) с* двумя выделенными вершинами *s, t* $\in V$, длина *1(e)* $\in Z^+$ каждого ребра *e* $\in E$ и положительные целые числа *B и K.* Существует ли в G *K* различных путей от s до *t,* веса которых не превосходят B?

1-ой прототип: нахождение одного пути, вес которого не превосходит *B.*

2-ий прототип: нахождение в G *K* различных путей от *s* до t, веса которых не превосходят *B.*

**13. Сельский почтальон.**
Заданы граф G = *(V, E),* длина *l(e)* $\in Z^+$ каждого ребра *e* $\in E$, подмножество $E' \subseteq E$ и граница $B \in Z^*$ . Существует ли в G цикл, включающий каждое ребро из E' и имеющий длину не более *B?*

1-ый прототип: нахождение цикла, включающего каждое ребро из E'.

2-ой прототип: нахождение цикла, включающего каждое ребро из $E'$ и имеющий длину не более *B.*

## *Обработка графов*

Задание - написать библиотеку работы с графом, решить поставленную задачу и визуализировать результат решения.

На первом этапе задание практикума предполагает создание библиотеки работы с графами, включающей процедуру преобразования графа из внешнего формата во внутреннее представление и обратно. Внешний формат будем считать фиксированным.

Формат описания графа:

```
Graph ::=
     Start
     GraphNodes
     GraphEdges
     Finish

Start : : =
      [ORIENTED]
      [COLORED]
      [WEIGHTED]
      [WITH CONDITIONS]
     #GRAPH DESCRIPTION
     OptName

OptName ::= Name |

GraphNodes ::=
     iNODES
     StartNode
     [Node}*

StartNode ::= #START NODE Name [COLOR ColorName]

Node ::= Name [COLOR ColorName]

GraphEdges ::=
     #EDGES
     {Edge}*

Edge :=
      EdgeName ':' Name Name
      [WEIGHT NaturalNumber]
      [CONDITION Input]
      [COLOR ColorName]

EdgeName := OptName

Input ::= Name

Finish ::= #END GRAPH DESCRIPTION

OptName, Name - последовательность из букв и цифр
NaturalNumber - последовательность из цифр
ColorName - один из цветов 16-цветной палитры
```

**Пример описания графа:**

```
ORIENTED
COLORED
WITH CONDITIONS
#GRAPH DESCRIPTION ExampleGraph
#NODES
#START NODE A COLOR RED
```

```
B COLOR WHITE C
COLOR MAGENTA D E
QQ COLOR BLACK
#EDGES
El : A  B CONDITION si
E2 : D  E CONDITION s2
   : D QQ CONDITION si
#END GRAPH DESCRIPTION
```

Этот пример описывает ориентированный раскрашенный граф с действиями. Первые два ребра поименованы: E1 и E2. При ориентированности графа порядок вершин в описании ребра существенен: ребро ориентировано от первой вершины ко второй. Если граф раскрашенный, то вершины без явного указания цвета имеют цвет по умолчанию (WHITE). Граф с действиями используется для представления автомата с конечным числом состояний. В этом случае вершина соответствует состоянию автомата, ребро - переходу в другое состояние, а условие - текущему значению из управляющей входной последовательности. В рассматриваемом примере из состояния A в состояние в можно перейти, только считав на входе строку s1.

Между любыми двумя лексемами во входном файле может быть произвольное число разделителей (пробелов и табуляций). Процедура преобразования графа из внешнего формата во внутреннее представление должна быть устойчива по отношению к ошибкам во входном файле, и уметь диагностировать ошибки ввода.

Несмотря на ограничения, накладываемые на свойства графа в каждой конкретной задаче, библиотека считывания должна уметь обрабатывать полный формат описания графа. Внутреннее представление графа также строится для полного формата. Если для решения конкретной задачи существенно более простое представление, оно может быть получено путем преобразования из общего (библиотечного) представления средствами прикладной программы.

Помимо функций считывания графа из файла и записи в файл, интерфейс библиотеки работы с графом должен предоставлять методы доступа к внутреннему представлению и его модификации. Например:

- получить все вершины;
- получить свойства вершины по ее имени;
- получить все ребра, выходящие из данной вершины,
- получить свойства ребра по его имени;
- изменить цвет вершины;

и т.д.

Решение конкретной задачи должно включать визуализацию графа с указанием имен вершин и обозначенным решением. Например, если необходимо найти путь на графе, то вершины и ребра, входящие в этот путь, должны быть выделены цветом.

Реализация библиотеки обработки графов включает спецификацию внутреннего представления и методов доступа к элементам графа, код библиотеки и набор тестов на считывание графа для демонстрации работоспособности библиотеки. Решение задачи включает описание алгоритма, наличие нескольких, последовательно уточняемых прототипов и визуализацию результатов решения.

### *Список задач*

**1. Минимальный путь в графе.**
Дана схема городского движения: дороги и перекрестки. Как добраться от одного заданного перекрестка до другого, проезжая минимальное количество перекрестков (найти самый короткий путь между заданными вершинами графа)?

1-ый прототип, найти все пути между заданными вершинами графа. Выдать все найденные пути в порядке возрастания длины.

2-ой прототип: найти минимальный путь между заданными вершинами графа.

Ограничения на тип входного графа: не ориентированный, не раскрашенный, без весов, без условий.

**2. Оптимальный обход графа.**
Дана схема городского движения: дороги и перекрестки. На перекрестках - светофоры. Ремонтная бригада каждый день объезжает все светофоры с проверкой. Выработать оптимальный маршрут для бригады, предполагая, что после работы она возвращается в исходное место (у одного из светофоров).

1-ый прототип: найти произвольный обход графа, начиная со стартовой вершины, выдавать путь и его длину.

2-ой прототип, найти кратчайший обход в графе.

Ограничения на тип входного графа, не ориентированный, не раскрашенный, без весов, без условий.

**3. Самый «дешевый» путь между двумя вершинами на графе с весами.**
Дана схема транспортного сообщения между *N* городами: заданы цены проездов из i-го города в *j-й* город, 1<=i, *j* <= *N,* если города связаны

напрямую. Как наиболее дешево добраться из одного заданного города в другой?

1-ый прототип: найти все пути между заданными вершинами графа и выдать все найденные пути в порядке возрастания весов.

2-ой прототип: найти самый «дешевый» путь между заданными вершинами графа.

Ограничения на тип входного графа: не ориентированный, не раскрашенный, без условий.

**4. Самый «дешевый» обход графа с весами.**

Начальник поручил рядовому коммивояжеру объехать *N* городов для реализации товара и вернуться обратно с выручкой. Города некоторым образом связаны транспортным сообщением (известны цены билетов из *i*-го города в j-й, 1 <= *i, j* <= *N,* если города связаны напрямую). Как объехать все города, истратив минимум денег на дорогу?

1-ый прототип: найти произвольный обход графа, начиная со стартовой вершины, выдать путь и его стоимость.

2-ой прототип: найти самый «дешевый» обход в графе, начав со стартовой вершины графа и вернувшись в нее.

Ограничения на тип входного графа: не ориентированный, не раскрашенный, без условий.

**5. Раскраска двудольного графа.**

Раскрасить вершины двудольного графа в черный и белый цвета. (Граф называется двудольным, если все его вершины можно раскрасить в два цвета так, что каждое ребро графа соединяло бы вершины разного цвета.)

1-ый прототип: раскрасить граф в черный и белый цвет и распечатать список вершин с их цветами.

2-ой прототип: выяснить, является ли граф двудольным и если является, то раскрасить его вершины в черный и белый цвет.

Ограничения на тип входного графа: не ориентированный, без весов, без условий.

**6. Минимальная тестовая последовательность для конечного автомата.**

Регулярная грамматика представляется конечным автоматом, который в свою очередь представляется в виде ориентированного графа с условиями. Для простоты ограничимся лево-регулярной грамматикой. Граф с условиями (диаграмма состояний) содержит условие перехода на каждом из своих ребер, вершины являются состояниями конечного автомата. Этот граф удобно использовать для проверки принадлежности цепочки к языку, задаваемому грамматикой: выходя из стартовой вершины (стартового состояния), считывая по одному символу из цепочки и выполняя соответствующий переход в другое состояние, нужно, исчерпав цепочку, оказаться в специальном стоп-состояний. Задача заключается в

нахождении цепочки минимальной длины, разбор которой покрывает все ребра графа.

1-ый прототип: найти какую-нибудь входную последовательность, применение которой к конечному автомату позволило бы пройти по всем ребрам графа, и распечатать ее.

2-ой прототип: найти минимальную входную последовательность и распечатать ее. Ограничения на тип входного графа: не раскрашенный, без весов.

**7. Удаление циклов в ориентированном графе.**

Программа на Си состоит из набора модулей, которые связаны по импорту переменных. Возможны нежелательные кольцевые зависимости, когда, например, модуль А использует какую-нибудь переменную VI из модуля 8, который в свою очередь использует переменную V2 из модуля А. Отношение межмодульной зависимости транзитивно. Предложить вариант минимальной модификации программы (в терминах модулей: модуль А отказывается от использования переменной VI из модуля В), чтобы исключить кольцевые зависимости.

1-ый прототип: найти и напечатать все циклы.

2-ой прототип: определить минимальное количество ребер, которые разрывают все циклы.

Ограничения на тип входного графа: не раскрашенный, без весов, без условий.

**8. Недостижимые вершины в ориентированном графе.**

Дана схема дорог и перекрестков, причем все дороги - с односторонним движением. Стартуя из заданного перекрестка, найти все перекрестки, к которым нельзя подъехать.

1-ый прототип: найти и напечатать хотя бы одну вершину, не достижимую из стартовой вершины.

2-ой прототип: найти и напечатать все вершины и ребра, не достижимые из стартовой вершины.

Ограничения на тип входного графа: не раскрашенный, без весов, без условий.

**9. Принадлежность цепочки языку.**

Регулярная грамматика представляется конечным автоматом, который в свою очередь представляется в виде ориентированного графа с условиями. Для простоты ограничимся лево-регулярной грамматикой. Граф с условиями (диаграмма состояний) содержит условие перехода на каждом из своих ребер, вершины являются состояниями конечного автомата. Этот граф удобно использовать для проверки принадлежности цепочки к языку, задаваемому грамматикой: выходя из стартовой вершины (стартового состояния), считывая по одному символу из цепочки и выполняя соответствующий переход в другое состояние, нужно, исчерпав цепочку,

оказаться в специальном стоп-состоянии. Используя представление грамматики в виде графа с условиями, осуществить проверку заданной цепочки. Восстановить по графу текст грамматики.

1-ый прототип: проверить цепочку на принадлежность к языку. описываемому графом и распечатать трассу обхода.

2-ой прототип: по графу с условиями сгенерировать и распечатать текст грамматики. Ограничения на тип входного графа: не раскрашенный, без весов.

## Литература


1. Haasan Gomaa. The Impact of prototyping on software system engineering in lEEEComputer Society press tutorial 1990, 543-552.

2. Barry W Boehm. A Spiral Model of Software Development and Enhancement in Software Engineering Project Management, 1987, 120-142.

3. Alan M. Davis, Edward H. Bersoff, Edward R.Comer A Strategy for Comparising Alternative Software Development Life Cycle Models in IEEE Transactions on Software Eng. Vol. SE-14, №10, Oct.1988,1453-1461.

4. Harold Abelson, Gerald J. Sussman, Julie Sussman. Structure and Interpretation of Computer Programs. MIT Press. 1996.

5. Mark Watson. Programming in Scheme. Springer-Verlag. 1996.

6. М.Гэри, Д.Джонсон. Вычислительные машины и труднорешаемые задачи, М., Мир, 1982.

7. А.Ахо, Дж.Хопкрофт, Дж.Ульман, Построение и анализ вычислительных алгоритмов, М., Мир, 1979.

8. Э.Рейнгольд, Ю.Нивергельт, Н.Део, Комбинаторные алгоритмы: теория и практика, М., Мир, 1980.

9. В.Липский, Комбинаторика для программистов, М., Мир, 1988.

10. Э.Майника, Алгоритмы оптимизации на сетях и графах, М., Мир, 1981.

## Содержание