СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Традиционно на кафедре кибернетики МИФИ преподавались основы функционального программирования на примере языка Lisp, разработанного в середине XX века, а лабораторные работы проводились на версии μ-Lisp [1]. Однако со времени разработки языка Lisp было создано множество новых теоретических механизмов, формализмов и методологий функционального программирования, венцом чего стала разработка унифицированного стандарта Haskell-98, ставшего в последующем функциональным языком программирования [2].

До сих пор стандарт Haskell-98 остаётся «вехой» в развитии функционального программирования, поэтому каждый, интересующийся этим вопросом, должен знать хотя бы основы нового стандарта и нового языка. Однако несмотря на то, что уже издано много работ на английском языке, использование языка Haskell в России тормозится отсутствием русскоязычной литературы даже в таком «безграничном» источнике, как Интернет.

Настоящий практикум продолжает серию учебно-методической литературы, предназначенной для практической поддержки специальностей и специализаций по искусственному интеллекту.

Парадигма функционального программирования основана на математическом понятии «функция», что позволяет наиболее эффективно создавать программы расчётного характера. Кроме того, функциональное программирование предоставляет возможность эффективно проводить вычисления на уровне символов, а не чисел. Поэтому этот факт нашёл самое явное отражение в искусственном интеллекте [3], [4], [5].

Теоретические основы функционального программирования были заложены ещё в 20-х годах XX столетия после разработки таких мощных вычислительных формализмов, как комбинаторная логика и λ-исчисление [6]. Впоследствии λ-исчисление стало базисом всех разработанных функциональных языков, начиная от первого функционального языка Lisp, заканчивая последней разработкой — языком Haskell-98.

Несмотря на то, что в мире разработки и создания программного обеспечения прочно заняла главенствующую позицию парадигма объектно-ориентированного программирования, что наблюдается особенно в последнее время в связи с появлением новых стандартов распределённого вычисления (CORBA, COM/DCOM, RMI и некоторые другие), принципы функциональных программ всё ещё остаются основополагающими. Ведь даже самые современные технологии программирования не могут обойтись без понятия «функция», поэтому объектно-ориентированное программирование включает в себя некоторые принципы программирования функционального.

Многие алгоритмы могут быть более эффективно реализованы при помощи чистых функций. В первую очередь, это относится к сортировке и поиску. Некоторые задачи, как например задача перевода одного формализованного языка на другой или задача разбора синтаксических конструкций, могут быть решены только при использовании методов функционального программирования.

Поэтому при помощи принципов, заложенных в функциональном программировании, могут решаться такие задачи, как символьные вычисления, вывод на знаниях, обработка формализованных и естественных языков, выявление знаний в базах дан-

ных, создание общающихся между собой программных систем (агентов), а также некоторые другие. Все эти задачи являются традиционными для искусственного интеллекта [7], [8].

Лабораторный практикум состоит из двух частей. Первая часть практикума посвящена описанию инструментального средства HUGS 98, которое является бесплатным программным комплексом для программирования на языке Haskell. Кроме того, в первой части приведены базовые сведения о парадигме функционального программирования — его история, назначение и свойства.

Вторая часть предназначена для выполнения лабораторных работ по курсу «Функциональное программирование» и состоит из четырёх лабораторных заданий, разбитых на части по уровню сложности решаемых задач.

Автор выражает особую благодарность своему научному руководителю Рыбиной Галине Валентиновне, основателю и бессменному руководителю лаборатории «Системы искусственного интеллекта», за предоставленную уникальную возможность взять в свои руки одно из направлений подготовки специалистов по кафедре кибернетики.

Также выражаю благодарность своим коллегам — Клочкову Андрею и Мирошкину Алексею за их помощь и особые советы, которые помогли сделать курс и лабораторный практикум более лёгким для понимания. Особенную благодарность выражаю своей жене Елене за её понимание и поддержку.

ВВЕДЕНИЕ

Созданная в 1998 году спецификация языка Haskell (названного так в честь учёного Хаскелла Карри, одного из основоположников функционального программирования) нашла необычайно широкую поддержку в научных кругах, в первую очередь, Европы и Японии. В связи с этим буквально за несколько месяцев различными исследовательскими группами и коммерческими компаниями было создано несколько реализаций языка Haskell как в виде интерпретаторов, так и в виде компиляторов — бесплатных и коммерческих.

Наиболее интересным инструментальным средством (ИС), которое используется во многих университетах мира при изучении основ функционального программирования, является ИС HUGS 98, включающее в себя собственно интерпретатор языка Haskell стандарта 1998 года (далее Haskell-98), а также интегрированную среду программирования.

Кроме того, ИС HUGS 98 является абсолютно бесплатным программным средством и может быть свободно получено через Интернет по адресу *www.haskell.org*. Это дополнительно способствует распространению рассматриваемого ИС в качестве средства для обучения, хотя оно и обладает некоторыми недостатками по сравнению с коммерческими реализациями языка Haskell-98.

1. ОБЩИЕ СВЕДЕНИЯ О ФУНКЦИОНАЛЬНОМ ПРОГРАММИРОВАНИИ

Прежде чем начать описание собственно функционального программирования, необходимо обратиться к истории науки о программировании.

Как известно, в 40-х годах XX века появились первые цифровые компьютеры, которые программировались при помощи переключения различного рода тумблеров, проводков и кнопок, а впоследствии при помощи перфокарт и перфолент. Число таких переключений достигало нескольких сотен и неумолимо росло с ростом сложности программ. Иногда происходило так, что лаборанты путали стопки перфокарт, что влекло за собой долгие поиски причин неправильной работы созданной программы. Поэтому следующим шагом развития программирования стало создание различных ассемблерных языков с простой мнемоникой. Например:

```
MOV 5, AX
MOV 4, BX
ADD BX
```

Однако даже ассемблеры не могли стать инструментом, удобным для пользования, так как мнемокоды всё ещё оставались слишком сложными, тем более что всякий ассемблер был жёстко связан с архитектурой компьютера, на котором он исполнялся. Таким образом, следующим шагом после ассемблера стали так называемые императивные языки высокого уровня (BASIC, Pascal, C, Modula и прочие, включая объектно-ориентированные). Императивными такие языки были названы по причине того, что главным их свойством является ориентированность, в

первую очередь, на последовательное исполнение инструкций, оперирующих с памятью (т.е. присваиваний), и итеративные циклы. Вызовы функций и процедур, даже рекурсивные, не избавляли такие языки от явной императивности (предписания) [9].

Функциональное программирование основано на совершенно иных принципах. Краеугольным камнем в парадигме функционального программирования является понятие функции. Если вспомнить историю математики, то можно оценить возраст этого понятия. Ему уже около четырехсот лет, и математика придумала бесчисленное множество теоретических и практических аппаратов для оперирования функциями, начиная от обыкновенных операций дифференцирования и интегрирования, заканчивая различного рода функциональными анализами, теориями нечётких множеств и функций комплексных переменных.

Математические функции выражают связь между параметрами (входом) и результатом (выходом) некоторого процесса. Так как вычисление (а в общем случае и программа) — это тоже процесс, имеющий вход и выход, функция является вполне подходящим и адекватным средством описания вычислений. Именно этот простой принцип положен в основу функциональной парадигмы и функционального стиля программирования. Функциональная программа представляет собой набор определений функций. Функции определяются через другие функции или рекурсивно, т.е. через самих себя. В процессе выполнения программы функции получают параметры, вычисляют и возвращают результат, в случае необходимости вычисляя значения других функций. Программируя на функциональном языке, программист не должен описывать порядок вычислений. Ему необходи-

мо просто описать желаемый результат в виде системы функций [10], [11].

Особо следует подчеркнуть, что функциональное программирование, равно как и логическое программирование, нашло большое применение в искусственном интеллекте и его приложениях [7], [8], [12].

В качестве базовой литературы, раскрывающей вопросы функционального программирования, можно посоветовать такие основополагающие работы, как [1], [3], [4].

Расширенный список литературы по функциональному программированию приведён в конце лабораторного практикума.

1.1. История функционального программирования

Широко известно, что теоретические основы императивного программирования были заложены еще в 30-х годах XX века учёными Аланом Тьюрингом и Джоном фон Нейманом. Теория, положенная в основу функционального подхода, также родилась в 20-х – 30-х годах XX столетия. В числе разработчиков математических основ функционального программирования можно назвать Мозеса Шёнфинкеля (Германия и Россия) и Хаскелла Карри (Англия), разработавших комбинаторную логику, а также Алонзо Чёрча (США), создателя λ-исчисления.

Теория так и оставалась теорией, пока в начале 50-х годах XX века Джон МакКарти не разработал язык Lisp [1], [10], [12], который стал первым почти функциональным языком программирования и на протяжении многих лет оставался единственным. Хотя Lisp все еще используется (как, например, и FORTRAN), он уже не удовлетворяет некоторым современным запросам, которые заставляют разработчиков программ взваливать как можно большую ношу на компилятор, облегчив тем самым свой непосильный труд. Необходимость в этом, конечно же, возникла из-за всё более возрастающей сложности программного обеспечения.

В связи с этим всё большую роль начинает играть типизация. В конце 70-х – начале 80-х годов XX века интенсивно разрабатываются модели типизации, подходящие для функциональных языков. Большинство этих моделей включали в себя поддержку таких мощных механизмов, как абстракция данных и полиморфизм. Появляется множество типизированных функциональных языков: ML, Scheme, Hope, Miranda, Clean и многие другие. Вдобавок постоянно увеличивается число их диалектов, применяемых для решения конкретных задач.

В результате вышло так, что практически каждая группа исследователей, занимающаяся функциональным программированием, использовала собственный язык [13], [14]. Это препятствовало дальнейшему распространению этих языков и порождало многочисленные более мелкие проблемы. Чтобы исправить ситуацию, объединенная группа ведущих исследователей в области функционального программирования решила воссоздать достоинства различных языков в новом универсальном функциональном языке. Первая реализация этого языка, названного Haskell в честь Хаскелла Карри, была создана в начале 90-х годов. В настоящее время действителен стандарт Haskell-98 [2].

В первую очередь, большинство функциональных языков программирования реализуются как интерпретаторы, следуя традициям языка Lisp. Интерпретаторы удобны для быстрой отладки программ, исключая длительную фазу компиляции, тем самым укорачивая обычный цикл разработки. Однако с другой

стороны, интерпретаторы в сравнении с компиляторами обычно проигрывают по скорости выполнения в несколько раз. Поэтому помимо интерпретаторов существуют и компиляторы, генерирующие неплохой машинный код (например, Objective Caml) или код на C/C++ (например, Glasgow Haskell Compiler). Практически каждый компилятор с функционального языка реализован на этом же языке.

В данном методическом пособии будет рассмотрена реализация стандарта Haskell-98 в виде интерпретатора HUGS 98.

1.2. Основы функционального программирования

Каждая парадигма программирования отличается теми или иными свойствами, особо выделяемыми исследователями в качестве дифференцирующих свойств, отделяющих одну парадигму от другой. В качестве основных свойств функциональных языков традиционно рассматриваются следующие:

- краткость и простота;
- строгая типизация;
- модульность;
- функции — это значения;
- чистота (отсутствие побочных эффектов);
- отложенные (ленивые) вычисления.

1.2.1. Краткость и простота

Программы на функциональных языках обычно намного короче и проще, чем те же самые программы на императивных языках. Для примера можно сравнить программы на языке C, на некотором абстрактном функциональном языке и на языке Haskell на примере сортировки списка быстрым методом Хоара

(пример, уже ставший классическим при описании преимуществ функциональных языков).

Пример 1. Быстрая сортировка Хоара на языке C:

```
void quickSort (int a[], int l, int r)
{
    int i = l;
    int j = r;
    int x = a[(l + r) / 2];
    do
    {
        while (a[i] < x) i++;
        while (x < a[j]) j--;
        if (i <= j)
        {
            int temp = a[i];
            a[i++] = a[j];
            a[j--] = temp;
        }
    }
    while (i <= j);
    if (l < j) quickSort (a, l, j);
    if (i < r) quickSort (a, i, r);
}
```

Пример 2. Быстрая сортировка Хоара на абстрактном функциональном языке:

```
quickSort ([]) = []
quickSort ([h : t]) = quickSort (n | n ⊆ t, n <= h)
+ [h] + quickSort (n | n ⊆ t, n > h)
```

Пример 2 следует читать так:

1. Если список пуст, то результатом также будет пустой список.

2. Иначе (если список не пуст) выделяется голова (первый элемент) и хвост (список из оставшихся элементов, который может быть пустым). В этом случае результатом будет являться конкатенация (сращивание) отсортированного списка из всех элементов хвоста, которые меньше либо равны голове, списка из самой головы и списка из всех элементов хвоста, которые больше головы.

Пример 3. Быстрая сортировка Хоара на языке Haskell:

```
quickSort [] = []
quickSort (h : t) = quickSort [y | y <- xs, y < x]
++ [h] ++ quickSort [y | y <- xs, y >= x]
```

Как видно, даже на таком простом примере функциональный стиль программирования выигрывает и по количеству написанного кода и по его элегантности.

Кроме того, все операции с памятью выполняются автоматически. При создании какого-либо объекта под него автоматически выделяется память. После того как объект выполнит свое предназначение, он вскоре будет также автоматически уничтожен сборщиком мусора, который является частью любого функционального языка.

Ещё одним полезным свойством, позволяющим сократить программу, является встроенный механизм сопоставления с образцом. Это позволяет описывать функции как индуктивные определения.

Пример 4. Определение N-го числа Фибоначчи:

```
fibb (0) = 1
fibb (1) = 1
fibb (N) = fibb (N – 2) + fibb (N – 1)
```


Одной из сильных сторон функционального программирования является механизм сопоставления с образцом, который позволяет функциональным языкам выходить на более абстрактный уровень, чем традиционные императивные языки (здесь не рассматривается объектно-ориентированная парадигма и ее расширения).

1.2.2. Строгая типизация

Практически все современные языки программирования являются строго типизированными языками (возможно, за исключением языка JavaScript и его диалектов, не существует императивных языков без понятия «тип»). Строгая типизация обеспечивает безопасность. Программа, прошедшая проверку типов, просто не может выпасть в операционную систему с сообщением подобным "access violation", особенно это касается таких языков, как C/C++ и Object Pascal, где применение указателей является типичным способом использования языка. В функциональных языках большая часть ошибок может быть исправлена на стадии компиляции (первого шага интерпретации — кодогенерации), поэтому стадия отладки и общее время разработки программ сокращаются. Вдобавок к этому строгая типизация позволяет компилятору генерировать более эффективный код и тем самым ускорять выполнение программ.

Рассматривая пример с быстрой сортировкой Хоара, можно увидеть, что помимо уже упомянутых отличий между вариантом на языке C и вариантами на абстрактном функциональном языке и на языке Haskell есть ещё одно важное отличие: функция на языке C сортирует список значений типа int (целых чисел), а функции на абстрактном функциональном языке и на языке Haskell — список значений любого типа, который принадлежит

к классу упорядоченных величин. Поэтому последние две функции могут сортировать и список целых чисел, и список чисел с плавающей точкой, и список строк. Можно описать какой-нибудь новый пользовательский тип. Определив для этого типа операции сравнения, возможно без перекомпиляции использовать функции *quickSort* и со списками значений этого нового типа. Это полезное свойство системы типов называется параметрическим или истинным полиморфизмом и поддерживается большинством функциональных языков.

Ещё одной разновидностью полиморфизма является перегрузка функций, позволяющая давать различным, но в чём-то схожим функциям одинаковые имена. Типичным примером перегруженной операции является обычная операция сложения. Функции сложения для целых чисел и чисел с плавающей запятой различны, однако для удобства они носят одно и то же имя (инфиксный знак «+»). Некоторые функциональные языки помимо параметрического полиморфизма поддерживают и перегрузку операций.

В языке C++ имеется такое понятие, как шаблон, которое позволяет программисту определять полиморфные функции, подобные *quickSort*. В стандартную библиотеку C++ STL входит такая функция и множество других полиморфных функций. Но шаблоны языка C++, как и родовые функции языка Ada, на самом деле порождают множество перегруженных функций, которые компилятор должен каждый раз компилировать, что неблагоприятно сказывается на времени компиляции и размере кода. А в функциональных языках полиморфная функция *quickSort* — это одна единственная функция.

В некоторых языках, например в языке Ada, строгая типизация вынуждает программиста явно описывать тип всех значений и функций. Чтобы избежать этого, в строго типизированные функциональные языки встроен специальный механизм, позволяющий компилятору определять типы констант, выражений и функций из контекста. Этот механизм называется механизмом вывода типов. Известно несколько таких механизмов, однако большинство из них являются разновидностями модели типизации Хиндли-Милнера, разработанной в начале 80-х годов XX века. Таким образом, в функциональных языках в большинстве случаев можно не указывать типы функций.

1.2.3. Модульность

Механизм модульности позволяет разделять программы на несколько сравнительно независимых частей (модулей) с чётко определёнными связями между ними. Тем самым облегчается процесс проектирования и последующей поддержки больших программных систем. Поддержка модульности не является свойством именно функциональных языков программирования, однако поддерживается большинством таких языков. Существуют очень развитые модульные императивные языки. В качестве примеров подобных языков можно привести языки Modula-2 и Ada-95.

1.2.4. Функции — это значения

В функциональных языках (равно как и вообще в языках программирования и математике) функции могут быть переданы другим функциям в качестве аргумента или возвращены в качестве результата. Функции, принимающие функциональные аргументы, называются функциями высших порядков или функционалами. Самый известный функционал, это функция *map*. Эта

функция применяет некоторую функцию ко всем элементам списка, формируя из результатов заданной функции новый список. Например, определив функцию возведения целого числа в квадрат как

```
square (N) = N * N,
```

можно воспользоваться функцией map для возведения в квадрат всех элементов некоторого списка:

```
squareList = map (square, [1, 2, 3, 4])
```

Результатом выполнения этой инструкции будет список [1, 4, 9, 16].

1.2.5. Чистота (отсутствие побочных эффектов)

В императивных языках функция в процессе своего выполнения может читать и модифицировать значения глобальных переменных и осуществлять операции ввода/вывода. Поэтому, если вызвать одну и ту же функцию дважды с одним и тем же аргументом, может случиться так, что в качестве результата вычислятся два различных значения. Такая функция называется функцией с побочными эффектами.

Описывать функции без побочных эффектов позволяет, практически, любой язык. Однако некоторые языки поощряют или даже требуют от функции побочных эффектов. Например, во многих объектно-ориентированных языках в функцию-член класса передаётся скрытый параметр (чаще он называется this или self), который эта функция неявно модифицирует.

В чистом функциональном программировании оператор присваивания отсутствует, объекты нельзя изменять и уничтожать, можно только создавать новые путем декомпозиции и синтеза существующих объектов. О ненужных объектах позаботится

встроенный в язык сборщик мусора. Благодаря этому в чистых функциональных языках все функции свободны от побочных эффектов. Однако это не мешает этим языкам имитировать некоторые полезные императивные свойства, такие как исключения и изменяемые массивы. Обычно для этого существуют специальные методы.

Каковы же преимущества чистых функциональных языков? Помимо упрощения анализа программ есть еще одно весомое преимущество — параллелизм. Раз все функции для вычислений используют только свои параметры, мы можем вычислять независимые функции в произвольном порядке или параллельно, на результат вычислений это не повлияет. Причем параллелизм этот может быть организован не только на уровне компилятора языка, но и на уровне архитектуры. В нескольких научных лабораториях уже разработаны и используются экспериментальные компьютеры, основанные на подобных архитектурах. В качестве примера можно привести Lisp-машину.

1.2.6. Отложенные (ленивые) вычисления

В традиционных языках программирования (например, C++) вызов функции приводит к вычислению всех аргументов. Этот метод вызова функции называется *вызов-по-значению*. Если какой-либо аргумент не использовался в функции, то результат вычислений пропадает, следовательно, вычисления были произведены впустую. В каком-то смысле противоположностью *вызова-по-значению* является *вызов-по-необходимости*. В этом случае аргумент вычисляется, только если он нужен для вычисления результата. Примером такого поведения можно взять оператор конъюнкции все из того же C++ (&&), который не вычисля-

ет значение второго аргумента, если первый аргумент имеет ложное значение.

Если функциональный язык не поддерживает отложенные вычисления, то он называется строгим. На самом деле, в таких языках порядок вычисления строго определен. В качестве примера строгих языков можно привести языки Scheme, Standard ML и Caml.

Языки, использующие отложенные вычисления, называются нестрогими. Haskell — нестрогий язык, так же как, например, языки Gofer и Miranda. Нестрогие языки зачастую являются чистыми.

Строгие языки часто включают в себя средства некоторых полезных возможностей, присущих нестрогим языкам, например бесконечных списков. В поставке языка Standard ML присутствует специальный модуль для поддержки отложенных вычислений. А язык Objective Caml, помимо этого, поддерживает дополнительное зарезервированное слово lazy и конструкцию для списков значений, вычисляемых по необходимости.

2. ОСНОВЫ РАБОТЫ С HUGS 98

После запуска ИС HUGS 98 на экране появляется диалоговое окно среды разработчика, автоматически загружается специальный файл предопределений типов и определений стандартных функций на языке Haskell (Prelude.hs) и выводится стандартное приглашение к работе.

Диалоговое окно среды разработчика состоит из главного меню, набора кнопок для быстрого доступа к наиболее часто используемым командам и консоли, где происходит работа с интерпретатором. Необходимо особо отметить, что ИС не позволя-

ет создавать и редактировать файлы с кодами программ, для этого требуется использование любого текстового редактора, поддерживающего обычный стандарт TXT (этим редактором, например, может быть стандартный блокнот Windows).

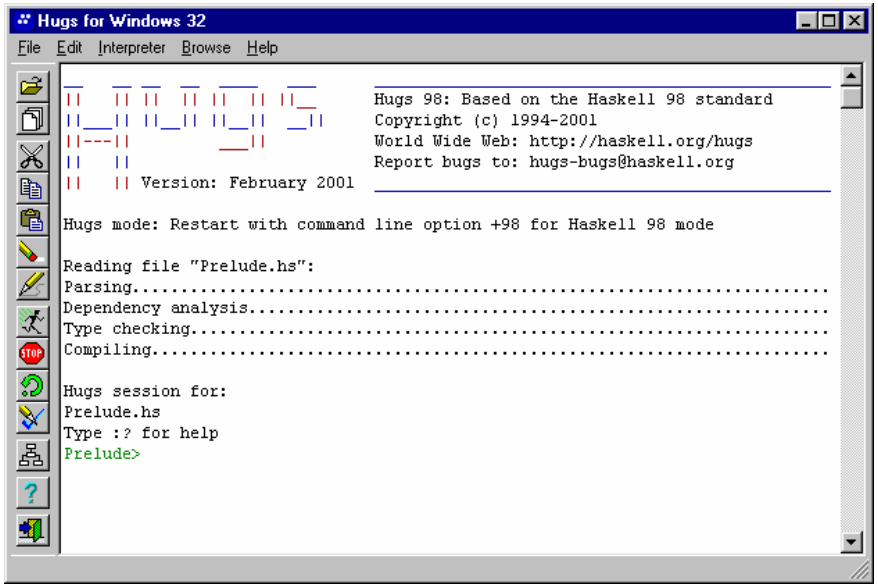

Рис. 1. Главное окно ИС HUGS 98

Главное окно HUGS 98 (рис. 1) обеспечивает доступ ко всем командам ИС, используемым для интерпретации и отладки программ. Кроме того, ИС позволяет вызвать на редактирование открытые модули в стандартном окне Notepad, встроенном в операционную систему Windows любой версии.

2.1. Панель инструментов HUGS 98

На панели инструментов, находящейся слева на главном диалоговом окне, предоставлены кнопки, при помощи которых можно вызвать наиболее часто используемые в процессе разработки команды (конечно, с точки зрения разработчиков ИС

HUGS 98, а не с точки зрения конечного пользователя). Ниже представлены краткие описания всех четырнадцати кнопок, которые можно найти на панели инструментов.

Загрузка модулей из внешних файлов. Позволяет выбрать и открыть файл, из которого загружаются все модули, обнаруживаемые интерпретатором в этом файле.

Вызов менеджера модулей. Менеджер модулей позволяет добавлять, удалять и редактировать загруженные в память ИС программные модули.

Вырезать выделенный текст. Стандартная функция редактирования текстов. Удаляет из редактора выделенный текст и помещает его в буфер обмена операционной системы.

Скопировать выделенный текст в буфер обмена. Стандартная функция редактирования текстов. Копирует выделенный текст в буфер обмена операционной системы.

Вставить текст из буфера обмена. Стандартная функция редактирования текстов. Вставляет в редактируемый тест содержимое буфера обмена операционной системы.

Очистить выбранный текст. Стандартная функция редактирования текстов. Удаляет из редактора выделенный текст, не помещая его в буфер обмена операционной системы.

Запустить внешний редактор текста. Запускает внешний тектовый редактор, зарегистрированный в операционной системе. Для семейства Windows при нажатии на

эту кнопку запускается стандартная программа Notepad.

Запуск на выполнение выражения «*main*». Исполняет функцию main в загруженных модулях (конечно, если такая функция обнаружена в модулях). Если функция *main* не обнаружена ни в одном из загруженных мдулей, то выдаётся ошибка: **ERROR — Undefined variable "main"**.

Остановка исполнения программы. Остановка выполнения любой запущенной функции. Используется, например, для прекращения вычисления бесконечного списка.

Перезагрузка всех файлов текущего проекта. Осуществляет перезагрузку всех файлов с целью загрузить в память интерпретатора все сделанные изменения в коде проекта.

Установка параметров интерпретатора. Вывод на экран диалогового окна установки набора параметров интерпретатора языка Haskell. О параметрах интерпретатора подробно описано в приложении В.

Вывод на экран иерархии классов. На экране появляется иерархия классов текущего проекта, показанная в виде множества прямоугольников с названиями (классы) и связей между ними (отношения наследования).

Вызов справки. Вызывает на экран стандартное диалоговое окно справочной информации. Предполагается, что все справочные файлы присутствуют в каталоге, где установлено ИС (эти файлы не входят в стандартную поставку HUGS 98).

Выход из программы. Осуществляет выход из ИС HUGS 98 в операционную систему.

2.2. Команды консоли HUGS 98

Консоль ИС HUGS 98 предоставляет небольшой набор служебных конструкций, позволяющих управлять работой ИС. Многие из этих команд дублируют действия кнопок на панели инструментов и некоторые пункты главного меню приложения. Однако в любом случае эти команды могут позволить профессиональным пользователям значительно ускорить процесс разработки.

Каждая команда начинается с символа «двоеточие» — «:». Это сделано для того, чтобы отличить встроенные команды от написанных разработчиками функций. Кроме того, ИС позволяет сокращать каждую команду вплоть до одной буквы, набрав только символ «двоеточие» и собственно первую букву команды.

Всего существует девятнадцать команд, ниже представлено подробное описание каждой из них.

:load [<filenames>]

Загружает программные модули из заданных файлов (имена файлов можно разделить пробелом). Дублирует кнопку загрузки модулей на панели инструментов. Если имена файлов отсутствуют, то происходит выгрузка всех модулей, кроме стандартного (Prelude.hs). При повторном использовании команды все ранее загруженные модули выгружаются из памяти интерпретатора.

:also <filenames>

Подгружает дополнительные модули в текущий проект. Имена файлов должны быть разделены пробелами (если указывается более чем один файл).

:reload

Повторяет последнюю выполненную команду загрузки (:load). Позволяет быстро выполнить перезагрузку модуля в случае, если он редактируется во внешнем текстовом редакторе.

:project <filename>

Загружает и использует файл проекта. Загрузить можно только один файл. Файлы проекта используются для объединения разрозненных файлов с кодом. При повторном использовании команды происходит выгрузка всех файлов (как проектных, так и обычных) из памяти интерпретатора.

:edit [<filename>]

Вызывает внешний текстовый редактор для исправления указанного файла. Если имя файла не указано, то на редактирование вызывается последний файл (загруженный или редактированный). Данная команда дублирует кнопку вызова внешнего текстового редактора на панели инструментов.

:module <module>

Устанавливает заданный модуль в качестве текущего для выполнения функций. Эта команда предназначена, в первую, очередь для разрешения коллизий имён.

<expr>

Запуск заданного выражения на выполнение. Например, команда main запустит на выполнение соответствующую функцию — main, что произведёт дублирование кнопки с панели инструментов.

:type <expr>

Выводит на экран тип заданного выражения. Эта команда используется, главным образом, в отладочных целях для быстрого получения типа создаваемого выражения (переменной, функции, сложного объекта).

:?

Выводит на экран список команд с кратким описанием.

:set [<options>]

Позволяет задать параметры ИС с командной строки. Дублирует действие диалогового окна настройки HUGS 98 (описание которого приведено в приложении В). Все возможные параметры этой команды (<options>) выводятся на экран при выполнении этой команды без каких-либо параметров.

:names [pat]

Выводит на экран список всех имён объектов, которые находятся в текущем (если не задано иное) пространстве имён.

:info <names>

Выводит на экран описание заданных имён объектов. Например, для функций выводит их тип вместе с именем заданной функции.

:browse <modules>

Выводит на экран список всех объектов (функций, переменных, типов), определённых в заданных модулях. Имена модулей должны быть разделены пробелом (в случае, если указано более одного имени модуля).

:find <name>

Вызывает на редактирование модуль, содержащий заданное имя. Если заданного имени нет ни в одном из текущих модулей, то выдаётся собщение об ошибке: **ERROR — No current definition for name "<name>"**.

:!<command>

Выходит в операционную систему и выполняет заданную команду. Необходимо особо отметить, что между символом «восклицательный знак» и именем команды операционной системы должен отсутствовать пробел.

:cd <directory>

Изменяет текущий каталог, с которым работает HUGS 98.

:gc

Принудительно запускает на выполнение процесс сборки мусора. После этого выводит на экран статистику о собранных и восстановленных ячейках памяти.

:version

Выводит на экран информацию о версии установленного интерпретатора языка Haskell и ИС HUGS 98.

:quit

Осуществляет выход в операционную систему. Дублирует кнопку на панели инструментов.

2.3. Дополнительные возможности для отладки

ИС HUGS 98 предоставляет разработчику несколько дополнительных возможностей для отладки программ, которые во многих случаях облегчают разработку и позволяют более обширно «оглядеть» созданное приложение. К таким дополнительным возможностям относятся просмотр классов, просмотр зарегистрированных имён объектов, просмотр конструкторов типов и просмотр иерархии классов.

Далее подробно рассматривается каждая из этих дополнительных возможностей, однако перед этим необходимо разъяснить значения обозначений, применяемых во всех инструментах по просмотру. Итак, к таким обозначениям относятся разноцветные пиктограммы прямоугольного формата с определённой буквой внутри. Всего таких пиктограмм девять:

- **Синий прямоугольник с буквой *C*** — обозначение класса (от английского class).
- **Красный прямоугольник с буквой *I*** — обозначение экземпляра класса (от английского instance).
- **Розовый прямоугольник с буквой *M*** — обозначение функции-члена класса (от английского member).
- **Синий прямоугольник с буквой *N*** — обозначение имени функции (от английского name).

- **Красный прямоугольник с буквой *P*** — обозначение примитива (от английского primitive). Примитив отличается от обычной функции тем, что его определения нет в файле, а встроено в интерпретатор. Примитивы присутствуют только в файле Prelude.hs.
- **Зелёный прямоугольник с буквой *C*** — обозначение конструктора типа (от английского constructor).
- **Синий прямоугольник с буквой *D*** — обозначение типа данных (от английского data).
- **Красный прямоугольник с буквой *S*** — обозначение встроенного типа.
- **Розовый прямоугольник с буквой *N*** — обозначение описания нового типа (от английского new type).

2.3.1. Просмотр классов

При помощи инструмента для просмотра классов разработчик может изучить список созданных классов, список функций-членов каждого класса (если они есть) и список экземпляров каждого класса (также, если они есть). Внешний вид этого инструмента можно видеть на рис. 2.

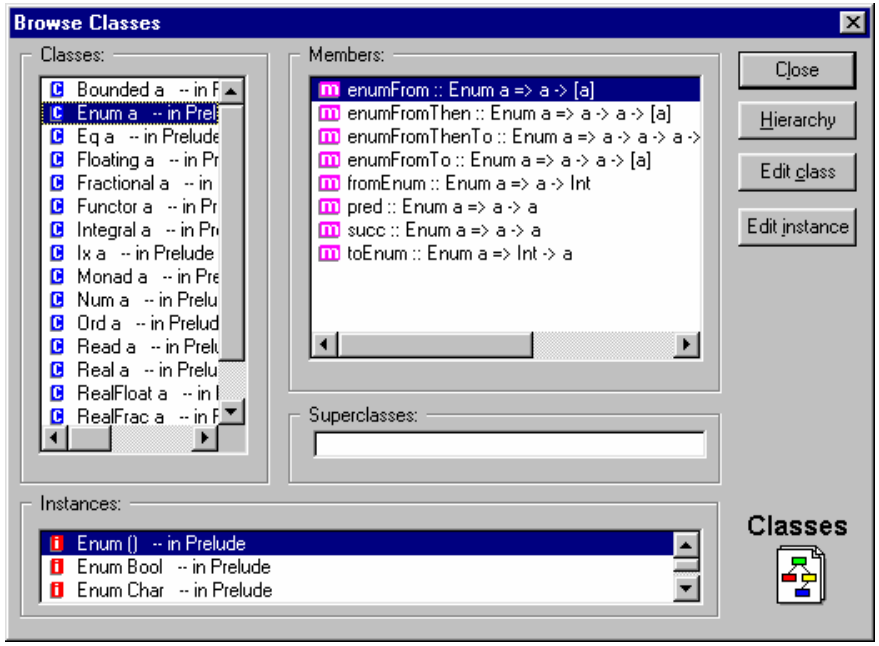

Рис. 2. Диалоговое окно просмотра классов

В левом столбце представлен список созданных классов. При выделении какого-то определённого класса в правом поле появляется список функций-членов класса, а в нижнем поле — список экземпляров класса. В поле «Superclasses» появляются базовые классы для выделенного класса (если они есть).

Это диалоговое окно предоставляет возможность автоматического перехода на описание выделенного класса или выделенного экземпляра класса для их редактирования (при использовании кнопок «Edit class» и «Edit instance» соответственно). Кроме того, можно перейти к просмотру иерархии классов (при помощи нажатия на кнопку «Hierarchy»).

2.3.2. Просмотр зарегистрированных имён объектов

При помощи инструмента для просмотра зарегистрированных имён объектов программист может изучить список всех имён, встречающихся во всех загруженных модулях. К именам относятся имена функций, имена примитивов (тех функций, реализация которых зашита внутри интерпретатора), имена конструкторов данных и имена функций-членов классов. Внешний вид этого инструмента представлен на рис. 3.

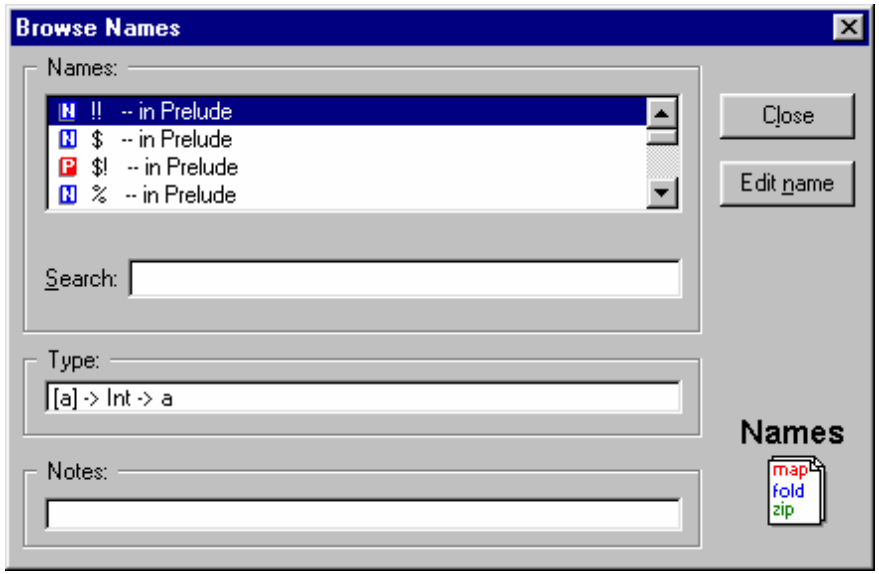

Рис. 3. Диалоговое окно просмотра имён

В верхнем поле представлен список всех имён с соответствующей пиктограммой, обозначающей природу имени. При помощи строки поиска можно осуществить инкрементный поиск по всему списку — при вводе очередной буквы курсор в списке перемещается на первое имя, которое начинается на введённую последовательность символов. В двух нижних полях предостав-

ляется дополнительная информация о выделенном имени — его тип и комментарии (если они присутствуют в описании).

Это диалоговое окно предоставляет разработчику возможность быстро перейти к редактированию выделенного имени при помощи нажатия на кнопку «Edit name».

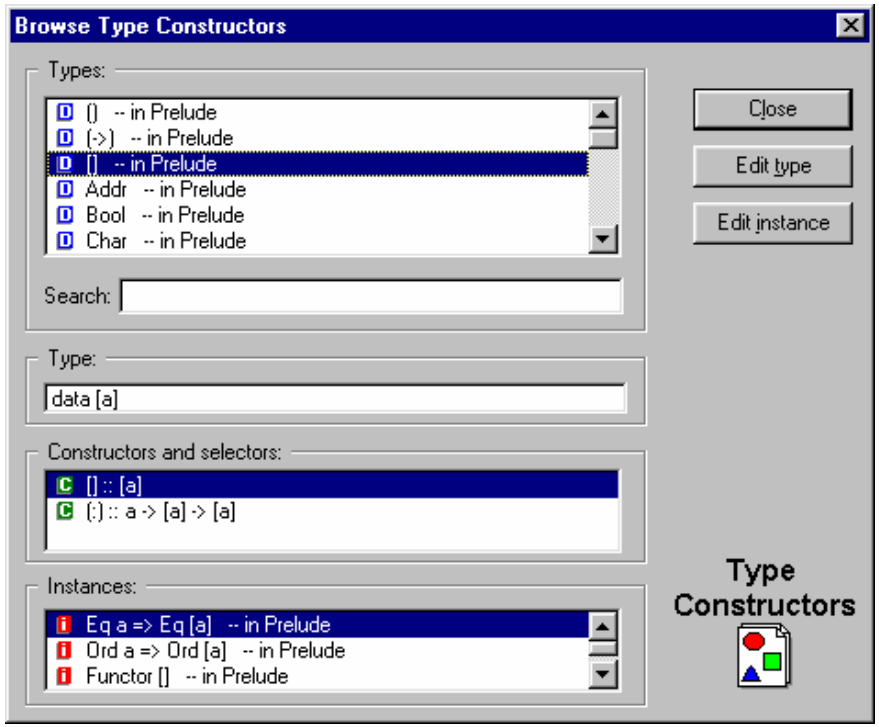

Рис. 4. Диалоговое окно просмотра конструкторов типов

2.3.3. Просмотр конструкторов типов

При помощи инструмента для просмотра конструкторов типов разработчик может изучить список всех конструкторов, которые встречаются во всех загруженных модулях. К конструкторам относятся конструкторы данных (служебное слово data), описания встроенных типов (служебное слово type) и конструкторы новых типов (служебное слово newtype). Внешний вид этого инструмента представлен на рис. 4.

В верхнем поле представлен список всех имён конструкторов типов с соответствующей пиктограммой, обозначающей природу конструктора. При помощи строки поиска можно осуществить инкрементный поиск по всему списку — при вводе очередной буквы курсор в списке перемещается на первое имя, которое начинается на введённую последовательность символов. В поле «Type» приводится определение соответствующего типа. В двух нижних полях предоставляется информация о конструкторах и селекторах выделенного типа, а также об экземплярах типа (если они есть).

При помощи этого диалогового окна разработчик может быстро перейти к редактированию выделенного конструктора (при нажатии на кнопку «Edit type») или к редактированию выделенного экземпляра типа (при помощи нажатия на кнопку «Edit instance»).

2.3.4. Просмотр иерархии классов

Просматривая иерархию классов, программист может увидеть отношения наследования между созданными классами. Необходимо отметить, что алгоритм прорисовки классов и отношений в HUGS 98 несколько неадекватен, поэтому для более полного понимания от программиста требуется либо чутьё, либо способность быстро разбросать все классы по диалоговому окну, создав планарный граф.

Это диалоговое окно можно вызвать не только из главного меню приложения, но и из диалогового окна для просмотра списка классов.

На рис. 5 показана иерархия классов из файла Prelude.hs. Как видно из рисунка в этом файле определены классы эквивалентности (все, которые наследуют свойства класса Eq) и классы-монады (Monad, Functor, Read и Bounded).

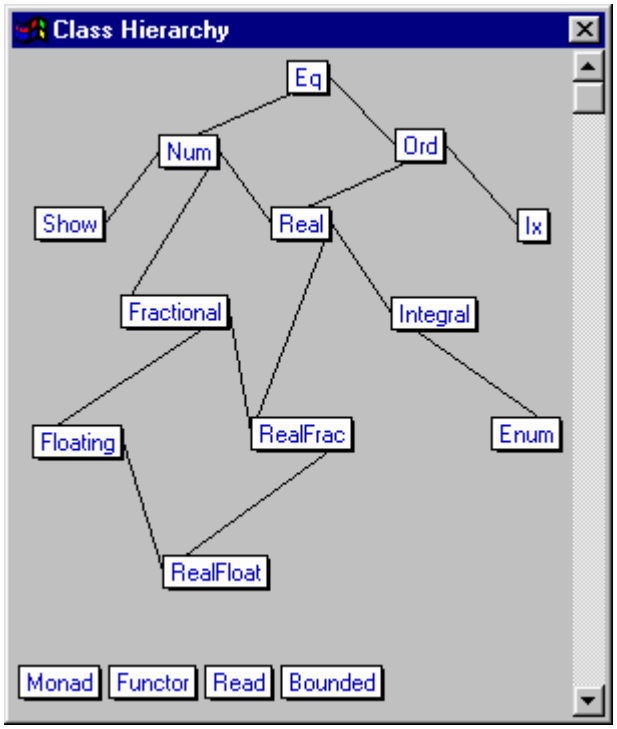

Рис. 5. Иерархия классов из файла Prelude.hs

Работа 1
ОСНОВЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ

Цель: *изучить основные принципы и методы функционального программирования; овладеть такими базисными возможностями функциональной парадигмы программирования, как работа со списками, использование бесконечных списков, использование аккумуляторов, охраны и локальных переменных.*

Предварительная подготовка

В главном каталоге для проведения лабораторных работ по функциональному программированию создать подкаталог, название которого должно соответствовать номеру группы (конечно, если такой подкаталог ещё не создан). В этом каталоге создать ещё один подкаталог, название которого должно соответствовать имени студента.

При помощи стандартного текстового редактора (Notepad для Windows) создать новый файл с расширением HS. Желательно, чтобы в названии файла был отражён номер текущей лабораторной работы.

Задание

1. Конструирование конечных списков (N — количество элементов в списке).
   - 1.1. Список натуральных чисел. $N = 20$.
   - 1.2. Список нечётных натуральных чисел. $N = 20$.
   - 1.3. Список чётных натуральных чисел. $N = 20$.
   - 1.4. Список квадратов натуральных чисел. $N = 30$.
   - 1.5. Список кубов натуральных чисел. $N = 30$.

1.6. Список факториалов. $N = 50$.
1.7. Список степеней двойки. $N = 25$.
1.8. Список степеней тройки. $N = 25$.
1.9. Список треугольных чисел Ферма. $N = 50$.
1.10. Список пирамидальных чисел Ферма. $N = 50$.

2. Конструирование бесконечных списков.
   2.1. Список факториалов.
   2.2. Список квадратов натуральных чисел.
   2.3. Список кубов натуральных чисел.
   2.4. Список пятых степеней натуральных чисел.
   2.5. Список степеней пятёрки.
   2.6. Список степеней семёрки.
   2.7. Список вторых суперстепеней натуральных чисел.
   2.8. Список третьих суперстепеней натуральных чисел.

3. Функции вычисления рядов.
   3.1. $F(x, n) = x^n$
   3.2. $F(n) = \Sigma_{i=1,n}\, i$
   3.3. $F(n) = \Sigma_{j=1,n} (\Sigma_{i=1,j}\, i)$
   3.4. $F(n) = \Sigma_{i=1,p}\, n^{-i}$
   3.5. $F(n) = 2^n$
   3.6. $F(n) = 3^n$
   3.7. $F(n) = e^n = \Sigma_{i=0,\infty} (n^i / i!)$
   3.8. $F(n) = n^n$

4. Функции работы со строками.
   4.1. GetN (*L*, *n*) — функция вычленения *N*-го элемента из заданного списка.
   4.2. ListSumm (L_1, L_2) — функция сложения элементов двух списков. Возвращает список, составленный из сумм элементов списков-параметров. Учесть, что переданные списки могут быть разной длины.
   4.3. OddEven (*L*) — функция перестановки местами соседних чётных и нечётных элементов в заданном списке.
   4.4. Reverse (*L*) — функция, обращающая список (первый элемент списка становится последним, второй — предпоследним, и так далее до последнего элемента).
   4.5. Map (*F*, *L*) — функция применения другой переданной в качестве параметра функции ко всем элементам заданного списка.
   4.6. Reverse_all (*S*) — функция, получающая на вход списочную структуру и обращающая все её элементы, а также её саму.
   4.7. Position (*L*, *A*) — функция, возвращающая номер первого вхождения заданного атома в список.
   4.8. Set (*L*) — функция, возвращающая список, содержащий единичные вхождения атомов заданного списка.
   4.9. Frequency (*L*) — функция, возвращающая список пар (символ, частота). Каждая пара определяет атом из заданного списка и частоту его вхождения в этот список.
   4.10. Insert (*L*, *A*, *n*) — функция включения в список заданного атома на определённую позицию.

5. Функции с аккумулятором. В этом пункте лабораторной работы необходимо написать те же функции, что и в пунктах 3 и 4, но с использованием понятия «накапливающий параметр», если это возможно. Либо доказать (теоретически), что использование аккумулятора невозможно.

Выполнение работы

Преподаватель определяет из пяти пунктов задания два подпункта, и студент выполняет их. Для выполнения пятого пункта задания студент получает для реализации по одной функции из третьего и четвёртого подпунктов. Для каждого задания следует оценить трудоёмкость вычисления и время вычисления (если это возможно). Для бесконечных списков необходимо отметить номер последнего элемента (если интерпретатор остановился из-за нехватки памяти). Для сложных функций необходимо отметить верхнюю границу параметров.

Все полученные результаты записать в журнал лабораторных работ (такой журнал можно вести в виде файла в личном каталоге студента в главном каталоге для проведения лабораторных работ по функциональному программированию).

Вопросы для самоконтроля

1. Перечислить базовые операции работы со списками, определённые в функциональном программировании. Какие из этих операций образуют базис?

2. Как организован список в памяти интерпретатора функционального языка? Каким образом возможно создание и обработка бесконечных списков?

3. Чем списки отличаются от списочных структур? Возможно ли использование функций обработки списков для работы со списочными структурами?

4. Чем «хвостовая» рекурсия отличается от «головной» рекурсии? Почему хорошие трансляторы функциональных языков оптимально работают с «хвостовой» рекурсией?

5. Какой смысл в использовании аккумулятора (накапливающего параметра)?

6. Для чего во многие функциональные языки вкладывается возможность использования локальных переменных и охраны? Поясните на примерах.

7. Что такое «образец»? Что такое «клоз»? Пояснить на примерах.

8. Что такое «каррированная функция»? Для чего в языке Haskell есть возможность использования как каррированных, так и не каррированных функций?

9. Как определяется тип функции? Приведите примеры описания типа каррированных и не каррированных функций на языке Haskell.

10. Для чего необходима конструкция описания λ-выражений на языке Haskell? Каким образом можно применять безымянные функции?

Работа 2
ИЗУЧЕНИЕ ФАЙЛА PRELUDE.HS

Цель: *изучить структуру и реализованные возможности файла начальной загрузки HUGS 98 — Prelude.hs; узнать, какие классы, простые структуры данных и функции для их обработки уже реализованы разработчиками HUGS 98.*

Предварительная подготовка

В системном каталоге ИС HUGS 98 найти файл предопределений Prelude.hs. Скопировать этот файл в личный каталог для проведения лабораторных работ по функциональному програм-

мированию и дать этому файлу новое название, отражающее проведение второй работы (например, Prelude_2.hs).

Задание

1. Воссоздать вручную иерархию классов из файла Prelude.hs.

2. По согласованию с преподавателем выбрать один класс и исследовать его свойства и методы.

3. По согласованию с преподавателем выбрать один экземпляр какого-либо класса и исследовать его реализацию.

4. Составить список всех функций, работающих с натуральными числами. Описать некоторые из них.

5. Составить список всех функций, работающих с действительными числами. Описать некоторые из них.

6. Составить список всех функций, работающих со списками. Описать некоторые из них.

7. Составить список всех функций, работающих с символами и строками. Описать некоторые из них.

8. Составить список всех типов и функций, описывающих примитивы из реализации HUGS 98. Описать некоторые из них.

9. Составить список всех монад. Описать одну из них.

10. Составить список всех функций, работающих с консолью. Описать некоторые из них.

Выполнение работы

По поручению преподавателя студент выполняет одно из представленных заданий.

Все полученные результаты записать в журнал лабораторных работ (такой журнал можно вести в виде файла в личном каталоге студента в главном каталоге для проведения лабораторных работ по функциональному программированию).

Вопросы для самоконтроля

1. Для чего используется файл Prelude.hs?

2. Сколько классов определено в файле Prelude.hs и для чего они используются?

3. Зачем определять несколько экземпляров для класса? Как используются разные экземпляры?

4. Для чего используются «минимальные» определения методов классов? Каким образом можно минимизировать код программ?

5. Для чего используются функции каррирования и декаррирования, определённые в Prelude.hs?

6. Сколько функций работы со списками определено в файле Prelude.hs?

7. Для чего используются функции zip* из файла Prelude.hs?

8. Что такое «монада»?

9. Для каких целей реализована монада *IO*?

10. Можно ли на языке Haskell осуществлять обработку исключительных ситуаций? Если да, то как?

Работа 3
УГЛУБЛЁННОЕ ИЗУЧЕНИЕ ВОЗМОЖНОСТЕЙ ЯЗЫКА HASKELL

Цель: *изучить расширенные возможности языка Haskell; овладеть навыками решения задач на перебор, сортировку и обход бинарных деревьев при помощи методов функционального программирования.*

Предварительная подготовка

После первой лабораторной работы в главном каталоге для проведения лабораторных работ по функциональному программированию уже должны быть созданы каталоги, названия которых соответствуют номерам групп, а также каталоги, названия которых соответствуют именам студентов. Если таких каталогов нет, их необходимо создать (подробнее см. предварительную подготовку к лабораторной работе № 1).

При помощи стандартного текстового редактора (Notepad для Windows) создать новый файл с расширением *hs*. Желательно, чтобы в названии файла был отражён номер текущей лабораторной работы.

Задание

1. Числовые задачи.

1.1. Перевод инфиксой арифметической записи в обратную польскую запись.

1.2. Перевод обратной польской записи в инфиксную арифметическую запись.

1.3. Расстановка скобок в инфиксной арифметической записи в соответствии с общепринятыми приоритетами операций.

1.4. Вычисление арифметического выражения в инфиксной записи.

1.5. Вычисление логического выражения в инфиксной записи.

1.6. Генерация списка «счастливых» шестизначных чисел (т.е. таких, сумма первых трёх цифр которых равна сумме трёх вторых).

1.7. Генерация списка «счастливых» 2N-значных чисел (т.е. таких, сумма первых N чисел которых равна сумме вторых N чисел).

1.8. Генерация всех возможных расстановок знаков арифметических операций и скобок в заданом шестизначном числе так, чтобы результатом полученного выражения было число 100.

1.9. Генерация всех возможных расстановок знаков арифметических операций и скобок в заданном N-значном числе так, чтобы результатом полученного выражения было заданне число M.

2. «Шахматные» задачи.

2.1. Генерация списка всех возможных расстановок максимального количества слонов на доске $N \times N$ клеток таким образом, чтобы ни один из слонов не бил другого. Слоном называется фигура, ходящая по диагонали.

2.2. Генерация списка всех возможных расстановок максимального количества ладей на доске $N \times N$ клеток таким образом, чтобы ни одна из ладей не била другую. Ладьёй назы-

вается фигура, ходящая прямо в горизонтальном или вертикальном направлении.

2.3. Генерация списка всех возможных расстановок минимального количества коней на доске $N \times N$ клеток таким образом, чтобы ни один из коней не бил другого. Конём называется фигура, ходящая на две клетки в прямом направлении, а потом на одну клетку в перпендикулярном изначальному (буквой «Г»).

2.4. Поиск и генерация списка всех возможных расстановок восьми ферзей на шахматной доске (8×8) таким образом, чтобы ни один из ферзей не бил другого. Ферзём называется фигура, ходящая в любом направлении: по горизонтали, вертикали или диагонали.

2.5. Генерация списка всех возможных расстановок N ферзей на доске $N \times N$. В случае, если сгенерированный список получается пустым (например, для $N = 2$), считается, что таких расстановок не существует.

2.6. Поиск такого пути коня на шахматной доске, что в каждой клетке конь побывает один и только один раз.

2.7. Поиск такого пути коня на доске $N \times N$ клеток, что в каждой клетке доски конь побывает один и только один раз. При невозможности найти такой путь считается, что его не существует.

3. Сложные математические задачи.

3.1. Генерация бесконечного списка простых чисел.

3.2. Генерация бесконечного списка совершенных чисел. Совершенным называется такое число, которое равно сумме всех своих делителей.

3.3. Поиск глобального экстремума полинома заданного порядка.

3.4. Генерация списка локальных экстремумов полинома заданного порядка.

3.5. Численое интегрирование методом прямоугольников полинома заданного порядка.

3.6. Численное интегрирование методом трапеций полинома заданного порядка.

3.7. Дифференцирование полинома заданного порядка.

4. Задачи на сортировку и перебор бинарных деревьев.

4.1. Перебрать заданное бинарное дерево методом «левый – корень – правый».

4.2. Перебрать заданное бинарное дерево методом «правый – корень – левый».

4.3. Перебрать заданное бинарное дерево методом «корень – левый – правый».

4.4. Перебрать заданное бинарное дерево заданным методом.

4.5. Отсортировать заданное бинарное дерево.

4.6. Получить количество узлов в заданном бинарном дереве.

4.7. Получить глубину вложенности заданного бинарного дерева.

4.8. Получить ширину заданного бинарного дерева.

Выполнение работы

Преподаватель задаёт студенту для реализации программы для решения одной, двух или трёх задач (желательно из разных разделов) в зависимости от сложности задач и уровня подготовки студента. В случае затруднений в разработке алгоритмов ре-

шения заданных задач, студент должен обратиться к преподавателю.

Все полученные результаты записать в журнал лабораторных работ (такой журнал можно вести в виде файла в личном каталоге студента в главном каталоге для проведения лабораторных работ по функциональному программированию).

Вопросы для самоконтроля

1. Привести трансформационную грамматику для перевода инфиксной арифметической записи в обратную польскую.

2. Привести трансформационную грамматику для перевода обратной польской записи в инфиксную арифметическую.

3. Привести грамматику для вычисления простых арифметических выражений в инфиксной нотации.

4. Привести грамматику для вычисления простых логических выражений в инфиксной нотации.

5. Что такое «бинарное дерево»? Привести примеры.

6. Для чего используются бинарные деревья?

7. Какие бывают способы обходов бинарных деревьев? Чем они различаются? В каких случаях целесообразно использовать тот или иной способ? Привести примеры.

8. Что такое «глубина дерева»?

9. Что такое «ширина дерева»?

10. Привести примерный алгоритм визуализации бинарного дерева.

Работа 4
ВЫПОЛНЕНИЕ СЛОЖНЫХ ЗАДАЧ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Цель: *изучить возможности функциональной парадигмы программирования при решении задач искусственного интеллекта; овладеть методами языка Haskell при решении таких задач; закрепить полученные навыки.*

Предварительная подготовка

После первой лабораторной работы в главном каталоге для проведения лабораторных работ по функциональному программированию уже должны быть созданы каталоги, названия которых соответствуют номерам групп, а также каталоги, названия которых соответствуют именам студентов. Если таких каталогов нет, их необходимо создать (подробнее см. предварительную подготовку к лабораторной работе № 1).

При помощи стандартного текстового редактора (Notepad для Windows) создать новый файл с расширением HS. Желательно, чтобы в названии файла был отражён номер текущей лабораторной работы.

Четвёртую лабораторную работу можно выполнять в домашних условиях, если существует такая возможность, так как она требует большого усердия и высоких временных затрат. Те студенты, которые не могут работать в домашних условиях, смогут за время, отведённое на лабораторную работу, только познакомиться с теоретическим материалом, и, возможно, начать разработку.

Задание

1. Разработка парсера *XML*-документов.

2. Разработка парсера программ на языке Haskell.

3. Разработка парсера для простого языка представления знаний продукционной системы (за описанием грамматики необходимо обратиться к преподавателю).

4. Разработка простейшей машины вывода для продукционной системы.

5. Разработка морфологического анализатора для основных типов слов русского языка.

6. Разработка морфологического анализатора для английских слов (с обработкой лингвистических исключений).

7. Разработка синтаксического анализатора для простых предложений русского языка.

8. Разработка синтаксического анализатора для предложений английского языка.

9. Разработка простого нейрона и небольшой нейросети.

10. Реализация генетических алгоритмов.

Выполнение работы

По поручению преподавателя студент выполняет одно из представленных заданий. При столкновении с теоретическими сложностями студент должен проконсультироваться с преподавателем.

Все полученные результаты записать в журнал лабораторных работ (такой журнал можно вести в виде файла в личном каталоге студента в главном каталоге для проведения лабораторных работ по функциональному программированию).

Вопросы для самоконтроля

1. Что такое «парсер»?

2. Что такое «морфологический анализ»?

3. Что такое «синтаксический анализ»?

4. Что такое «нейронная сеть»?

5. Что такое «генетические алгоритмы»?

СПИСОК ЛИТЕРАТУРЫ

Обязательная

1. Хювёнен Э., Сеппенен И. Мир Lisp'а. В 2-х томах. М.: Мир, 1990.

2. Bird R. Introduction to Functional Programming using Haskell. 2-nd edition, Prentice Hall Press, 1998.

3. Филд А., Харрисон П. Функциональное программирование. М.: Мир, 1993.

4. Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983.

Рекомендуемая

5. Fokker J. Functional programming. Dept. of CS. Utrecht University, 1995.

6. Thompson S. Haskell: The Craft of Functional Programming. 2-nd edition, Addison-Wesley, 1999.

7. Norvig P. Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Los Altos, CA: Morgan Kaufmann, 1992.

8. Russel S. J., Norvig P. Artificial Intelligence: A Modern Approach. Englewood Cliffs, NJ: Prentice-Hall, 1995.

9. Henson M. Elements of functional languages. Dept. of CS. University of Sassex, 1990.

10. Winston P. H., Horn K. P. LISP, 3-rd edition. Reading, MA: Addison-Wesley, 1988.

11. Graham P. On Lisp: Advanced Techniques for Common Lisp. Englewood Cliffs, NJ: Prentice-Hall, 1994.

12. Charniak E., Reisbeck C., McDermott D. Artificial Intelligence Programming, 2-nd edition. Hillsdale, NJ: Lawrence Eribaum, 1987.

13. Бердж В. Методы рекурсивного программирования. М.: Машиностроение, 1983.

14. Джонс С., Лестер Д. Реализация функциональных языков. М.: Мир, 1991.

Приложение

A. Языки функционального программирования

В приложении приведено краткое описание некоторых языков функционального программирования (очень немногих). Дополнительную информацию можно почерпнуть, просмотрев интернет-ресурсы, перечисленные ниже.

- **Lisp** (List processor). Считается первым функциональным языком программирования. Нетипизирован. Содержит массу императивных свойств, однако в общем поощряет именно функциональный стиль программирования. При вычислениях использует *вызов-по-значению*. Сейчас наиболее распространён диалект Common Lisp, ушедший от принципов ФП. Язык сложен в изучении, имеет очень непривычный синтаксис. Существует объектно-ориентированный диалект языка — CLOS.

- **Scheme**. Один из многих диалектов языка Lisp, предназначенный для научных исследований в области информатики. При разработке Scheme был сделан упор на элегантность и простоту языка. Благодаря этому язык получился намного меньше, чем базовая версия языка Lisp. Отличается простотой как самого языка, так и стандартной библиотеки функций, хотя несколько проигрывает в универсальности. Кроме того, здесь точнее соблюдаются принципы функционального программирования.

- **ISWIM** (If you See What I Mean). Функциональный язык-прототип. Разработан Ландиным (США) в 60-х годах для демонстрации того, каким может и должен быть язык функционального программирования. Вместе с языком Ландин разработал и специальную виртуальную машину для исполнения программ на

языке ISWIM. Эта виртуальная машина, основанная на *вызове-по-значению*, получила название SECD-машины. На синтаксисе языка ISWIM базируется синтаксис многих функциональных языков. На синтаксис ISWIM похож синтаксис ML, особенно Caml.

- **ML** (Meta Language). Целое семейство строгих функциональных языков с развитой полиморфной системой типов и параметризуемыми модулями. ML преподаётся во многих университетах мира (в некоторых даже как первый язык программирования).

- **Standard ML**. Один из первых типизированных языков функционального программирования. Содержит некоторые императивные свойства, такие как ссылки на изменяемые значения, и поэтому не является чистым. При вычислениях использует *вызов-по-значению*. Очень интересная реализация модульности. Мощная полиморфная система типов. Последний стандарт языка — Standard ML-97, существует формальное математическое определение синтаксиса, а также статической и динамической семантик языка.

- **Caml Light** и **Objective Caml**. Как и Standard ML принадлежит к семейству ML. Objective Caml отличается от Caml Light, в основном, поддержкой классического объектно-ориентированного программирования. Objective Caml, также как и Standard ML, является строгим, но имеет некоторую встроенную поддержку отложенных вычислений.

- **Miranda**. Функциональный язык, разработанный Дэвидом Тернером (США), в качестве стандартного функционального языка, использовавшего отложенные вычисления. Имеет стро-

гую полиморфную систему типов. Как и ML этот язык преподаётся во многих университетах. Оказал большое влияние на разработчиков языка Haskell.

- **Haskell**. Один из самых распространенных нестрогих языков функционального программирования. Имеет очень развитую систему типизации. Несколько хуже разработана система модулей. Последний стандарт языка — Haskell-98.

- **Gofer** (GOod For Equational Reasoning). Упрощенный диалект языка Haskell. Предназначен для обучения основам функционального программирования.

- **Clean**. Функциональный язык, специально предназначенный для параллельного и распределенного программирования. По синтаксису напоминает Haskell. Чистый. Использует отложенные вычисления. С компилятором поставляется набор библиотек (I/O libraries), позволяющих программировать графический пользовательский интерфейс под Win32 или MacOS.

- **Erlang**. Язык, разработанный компанией Ericsson, ориентирован на сферу коммуникаций; происходит от Пролога, хотя сейчас похож на него лишь внешне. Имеет лёгкий в освоении синтаксис, богатейшую библиотеку, в том числе: переносимый графический интерфейс, СУБД, распределенные вычисления и многое другое; причем всё это доступно бесплатно. Язык является параллельным изначально — возможность параллельной работы нескольких процессов и их взаимодействия заложена на уровне синтаксиса. Недостаток пока только один: компиляция в байт-код, для которого нужен "тяжёлый" и не слишком быстрый интерпретатор.

- **Рефал**. Функциональный язык, разработанный в СССР ещё в семидесятых годах XX века. В некоторых отношениях близок к Прологу. Крайне эффективен для обработки сложных структур данных типа текстов на естественном языке, XML и т.д. Эта эффективность обусловлена тем, что Рефал является единственным языком, использующим двунаправленные списки — это позволяет сократить объём некоторых программ и ускорить их работу (за счёт отсутствия необходимости в сборке мусора). К сожалению, в последнее время разрабатывается не очень активно. С другой стороны, по языку много документации на русском языке; имеется суперкомпилятор — система оптимизации программ на уровне исходных текстов (т.е. фактически оптимизация алгоритма!).

- **Joy**. Чистый функциональный язык, основанный на комбинаторной логике. Внешне похож на Форт: используется обратная польская запись и стек. Пока что находится в стадии развития, хотя уже имеет неплохую теоретическую основу. Существует даже прототипная реализация. Основное преимущество перед другими языками — линейная запись без переменных, что должно резко облегчить автоматизацию написания, проверки и оптимизации программ. Помимо недоразвитости, есть еще один серьезный недостаток: как и у Форта, у Joy страдает читабельность.

Б. Интернет-ресурсы по функциональному программированию

В приложении представлена небольшая (и далеко неполная) подборка ресурсов в Интернете, посвящённых функциональному программированию вообще и различным функциональным языкам в частности.

- **www.haskell.org** — очень насыщенный сайт, посвященный функциональному программированию в общем и языку Haskell в частности. Содержит различные справочные материалы, список интерпретаторов и компиляторов языка Haskell (в настоящий момент все интерпретаторы и компиляторы бесплатны). Кроме того, имеется обширный список интересных ссылок на ресурсы по теории функционального программирования и другим языкам (Standard ML, Clean).

- **cm.bell-labs.com/cm/cs/what/smlnj** — Standard ML of New Jersey. Очень хороший компилятор. В бесплатный дистрибутив помимо компилятора входят утилиты MLYacc и MLLex и библиотека Standard ML Basis Library. Отдельно можно взять документацию по компилятору и библиотеке.

- **www.harlequin.com/products/ads/ml/** — Harlequin MLWorks, коммерческий компилятор Standard ML. Однако в некоммерческих целях можно бесплатно пользоваться версией с несколько ограниченными возможностями.

- **caml.inria.fr** — институт INRIA. «Домашний» сайт команды разработчиков языков Caml Light и Objective Caml. Можно бесплатно скачать дистрибутив Objective Caml, содержащий интерпретатор, компиляторы байт-кода и машинного кода, Yacc и Lex для Caml, отладчик и профайлер, документацию, примеры. Качество компилированного кода у этого компилятора очень хорошее, по скорости опережает даже Standard ML of New Jersey.

- **www.cs.kun.nl/~clean/** — содержит дистрибутив компилятора с языка Clean. Компилятор коммерческий, но допускается бесплатное использование в некоммерческих целях. Из того, что

компилятор коммерческий, следует его качество (очень быстр), наличие среды разработчика, хорошей документации и стандартной библиотеки.

B. Параметры ИС HUGS 98

ИС HUGS 98 предоставляет программисту возможность тонко настраивать интерпретатор и саму ИС под ту или иную задачу. Это возможно при помощи изменения настроек (параметров) ИС. На рис. П1 показано состояние настроек, загружаемых по умолчанию (такой набор параметров действует при первоначальной установке HUGS 98).

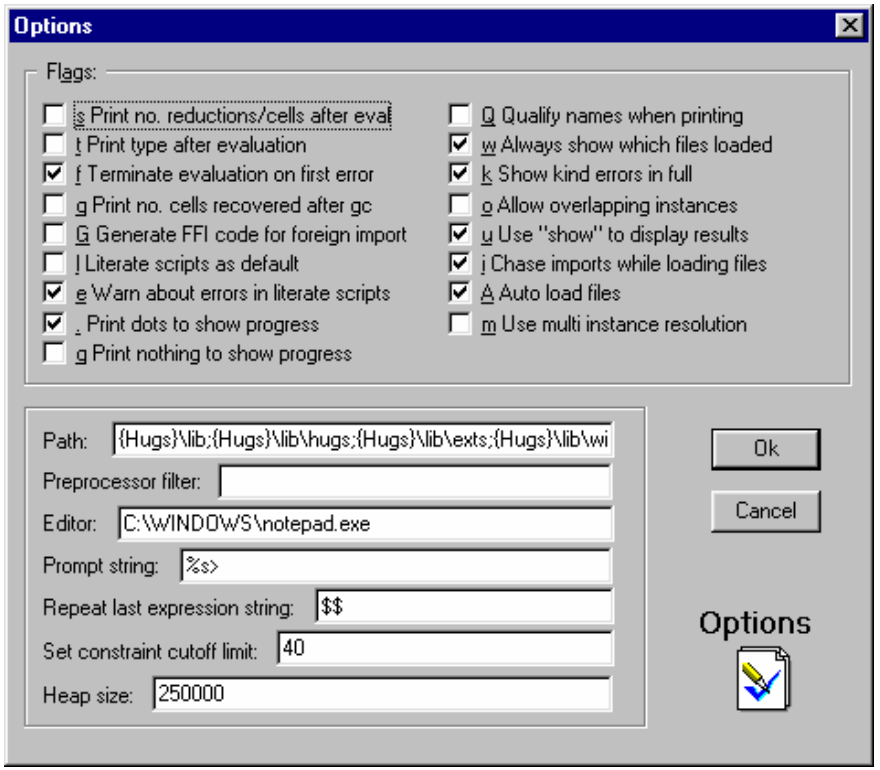

Рис. П1. Диалоговое окно установка параметров ИС

В верхней части представленного диалогового окна находится набор так называемых флагов, значения которых могут быть либо ИСТИНА (флаг установлен), либо ЛОЖЬ (флаг сброшен). Каждый флаг отвечает за тот или иной параметр интерпретатора или самой оболочки.

В нижней части диалогового окна настроек находятся поля ввода внутренних переменных окружения ИС HUGS 98.

Каждый флаг представлен определённой буквой латинского алфавита в верхнем или нижнем ключе. Далее описываются все имеющиеся флаги, обозначаемые соответствующими буквами:

s — распечатывать количество редукций и ячеек памяти после выполнения вычислений;

t — распечатывать тип выражения после его вычисления;

f — прерывать вычисление после первой ошибки;

g — распечатывать количество ячеек памяти, собранный во время сборки мусора;

G — генерация кода FFI для импортированный файлов;

l — оптимизация скриптов по умолчанию;

e — предупреждать об ошибках в оптимизированных скриптах;

. — распечатывать точки для визуализации процесса вычисления;

q — ничего не распечатывать для визуализации процесса вычисления;

Q — квалифицировать имена во время распечатки;

w — всегда показывать названия загруженных файлов;

k — полностью показывать тип и описание ошибок;

o — позволять пересекаться экземплярам классов;

u — использовать функцию «show» для отображения результатов;

i — удалять импортированные файлы при загрузке новых;

A — автоматически загружать файлы;

m — использовать множественную резолюцию экземпляров классов.

Душкин Роман Викторович

Лабораторный практикум по функциональному программированию «Инструментальное средство HUGS 98 для программирования на языке Haskell»

Учебное пособие

Редактор Н. В. Шумакова

Подписано в печать Формат 60x84 116 Объём 4.0 п.л.
Уч.-изд. л. 4.0 Тираж 100 экз. Изд. № 044-1 Заказ

Московский инженерно-физический институт
(государственный университет)

Типография МИФИ. 115409, г. Москва, Каширское шоссе, 31.