

**Pawn embedded scripting language  
(Встроенный скриптовый язык Pawn)**

**Оглавление**

**Предисловие. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1**

**Введение в учебник. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3**

**Данные и декларации. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61**

**Функции . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 0,70**

**Препроцессор. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 0,93**

**Общий синтаксис. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97**

**Операторы и выражения. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105**

**Заявления . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113**

**Директивы. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118**

**Предлагаемая библиотека функций. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125**

**Подводные камни: отличия от C . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134**

**Разные советы. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137**

**Приложения . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148**

**A: Сообщения об ошибках и предупреждения. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148**

**Б: Компилятор. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 169**

**С: Обоснование. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175**

**Д: Лицензия. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 0,182**

**Индекс. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 0,185**

**Foreword  
------------------------------------------------------------------------------------------**

«PAWN» — это простой, бестиповый, 32-битный «скриптовый» язык с C-подобным синтаксисом. Скорость выполнения, стабильность, простота и небольшой размер были важными критериями дизайна как для языка, так и для интерпретатора/абстрактной машины, на которой работает PAWN-программа.

Приложение или инструменты не могут делать все для всех пользователей. Это и оправдывает разнообразие редакторов, компиляторов, операционных систем и многих других программных комплексов, это также объясняет наличие обширной конфигурации параметров и макросов, языков сценариев в приложениях. Мои собственные приложения содержали множество маленьких “языков(скорее всего – скриптов, написанных на разных языках)”, большинство из них были очень простыми, некоторые были обширными… и большинство нужд могло было быть решено общим целевым языком со специальной библиотекой. Отсюда…Pawn.   
  
Язык Pawn был разработан как гибкий язык для манипулирования объектами в хост-приложении. Набор инструментов(компилятор, абстрактная машина), были написаны так, чтобы их можно было легко расширять и запускать на разных программно-аппаратных архитектурах (software/hardware architecures).

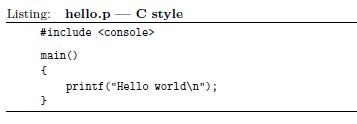
Pawn является потомком оригинального Small C Рона Кейна и Джеймса Хендрикса, который, в свою очередь, был подмножеством С. Некоторые из модификаций, которые я сделал, для Small C, например, удаление системы типов и замена указателей по ссылкам были настолько фундаментальны, что я могу назвать свой язык диалектом С или подмножеством С. Поэтому я убрал букву C из названия и использовал 'Small' для имени языка в моей публикации в Dr. Dobb’s Journal и в последующие годы. В течение разработки и сопровождения продукта я получил много запросов для изменений. Одним из часто запрашиваемым изменением было — другое название языка. Слово “small”, было таким распространенным словом, что мешало поиску информацию об этом языке сценариев. Смена названия произошла вместе со значительным изменением языка: поддержка “states” (и state machines).  
Я в долгу перед Роном Каином и Джеймсом Хендриксом(а совсем недавно Энди Юэнь) и журналу доктора Добба, чтобы сдвинуть дело с мертвой точки. Хотя я должен коснуться почти каждой строки исходного кода несколько раз, происхождения от C все еще ясно видно.  
Подробный трактат о целях проектирования и компромиссах содержится в приложении C, здесь я хотел бы резюмировать несколько ключевых моментов. Как написано в предыдущем абзаце, пешка предназначена для настройки приложений(путем написания скрипта), а не для написания приложений. Пешка плохо структурирует данные, потому что программы, написанные на пешке предназначены для манипулирования объектами(текстом, спрайтами, потоками, запросами…) в хост-приложении, но программе намерено отказан прямой доступ к любым данным за пределами своей абстрактной машины. Единственное, что программы на пешке должны манипулировать объектами в хост-приложении путем вызовов подпрограмм, так называемых “собственных функций”, которые хост-приложения обеспечивает.  
  
Пешка гибка в ключевой области: вызов функций, пешка поддерживает значения по умолчанию для любого из аргументов фцнкции(не только последнего), callby-ссылка, а также вызов по значению, и “именованный”, а также “позиционный” аргументы функции. Пешка не имеет механизм проверки типов, в силу того, что это бестиповый язык, но предлагает взамен механизм проверки классификации, называемый метками. Система тегов особенно удобна для аргументов функций, потому что каждый аргумент может указывать несколько допустимых тегов.  
Для любого языка сила или слабость заключается не в отдельных чертах, а в и их сочетании. Для пешки, я чувствую, что комбинации имени аргументов – позволяют указывать аргументы функции в любом порядке, и значения по умолчанию, что позволяет пропустить указание аргументов, которые не интересуют, смешать в удобный и ‘наглядный’ способ вызова собственной функции для управления объектами в главном приложении.

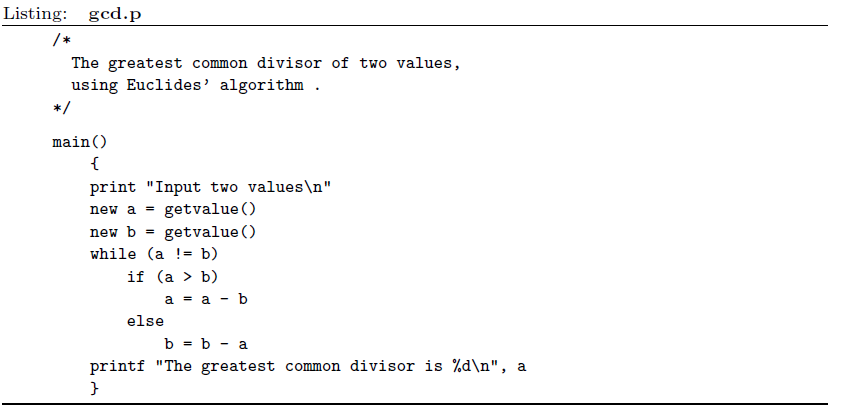
Введение в учебник  
  
Pawn – это простой язык программирования с синтаксисом напоминающим Язык программирования “C”. Программа пешки состоит из набора функций и набора переменных.  
Переменные – это объекты данных. Функции содержат инструкции(называемые операторами), которые работают с объектами данных или которые выполняют задачи.  
Первая программа почти на любом языке программирования – это та, которая печатает  

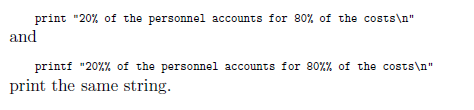

В этом руководстве предполагается, что вы знаете, как запустить pawn-программу, если нет, обратитесь к руководству по применению(дополнительные советы на странице 169).  
Программа Pawn начинает выполнение в функции “входа” – почти во всех случаях. В примерах данного руководства эта функция ввода называется основной. Здесь функция main содержит только одну инструкцию, которая находится на строке ниже самого заголовка функции. Разрывы строк и отступы незначительны; вызов функции print также может находиться в той же строке, что и заголовок функции main.

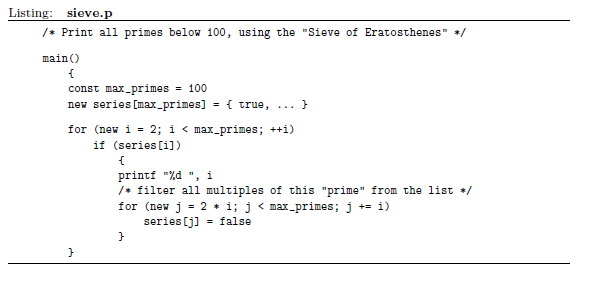
Определение функции требует, чтобы за именем функции следовала пара круглых скобок. Если функция принимает параметры, их объявления появляются в круглых скобках. Функция main не принимает круглых скобок. Правила отличаются для вызова функции(или вызова функции); круглые скобки необязательны при вызове функции печати.  
Единственным аргументом функции печати является строка, которая должна заключена в двойные кавычки. Символы \n в конце строки образуют escape-последовательность, в данном случае они обозначают символ ‘новой строки'. Когда print встречает escape-последовательность новой строки, она перемещает курсор в первый столбец следующей строки. Нужно использовать escape-последовательность \n, чтобы вставить ‘новую строку’ в строку, потому что строка не может переноситься на несколько строк.

4

Pawn- это язык, чувствительный к регистру: прописные и строчные буквы считаются разными буквами. Было бы ошибкой написать функцию printf в приведенном выше примере как PrintF. Ключевые слова и предопределенные символы, такие как название функции ‘main’ должны быть набраны строчными буквами.  
Если вы знаете язык C, вам может показаться, что приведенный выше пример не очень похож на эквивалентную программу “Hello world” на C/C++. Однако пешка также может быть очень похожа на C. Следующий пример программы также имеет допустимый синтаксис пешки(и он имеет ту же семантику, что и предыдущий пример):  


Эти первые примеры также показывают некоторые различия между pawn и языком C:  
\* обычно нет необходимости включать какой-либо системный “заголовочный файл”;  
\* точка с запятой необязательна(кроме случаев написания нескольких операторов в одной строке);  
\* когда тело функции представляет собой одну инструкцию, фигурные скобки (для составных инструкций) необязательны;  
\* если вы не используете результат функции в выражении или присваивании, круглые скобки вокруг аргумента функции необязательны.  
  
Кроме того, несколько предыдущих пунктов относятся к необязательным синтаксисам. Вы сами выбираете, какой синтаксис использовать: ни один из стилей не является “устаревшим” или “считающимся вредным”. Примеры в этом руководстве располагают фигурные скобки и используют углубление, известное как “стиль кузнеца”, но пешка – это язык свободного формата и другие стили отступов так же хороши.  
  
Поскольку pawn разработан как язык расширения для приложений, набор функций/библиотека, которыми располагает программа pawn, зависит от основного приложения. В результате язык пешка не имеет внутреннего знания о какой-либо функции. Функция печати, используемая в этом первом примере должна быть доступна хост-приложению и быть “объявлена” для парсера пешки. Однако, предполагается, то все хост-приложения обеспечивают минимальный набор общих функций, таких как print и printf.  
  
9 страница  
  
В некоторых средах дисплей или терминал должны быть включены, прежде чем на него можно будет вывести какой-либо текст. Если это так, вы должны добавить вызов функции “консоль” перед первым вызовом функции print или printf. Функция консоли также позволяет указать характеристики устройства, такие как количество строк и столбцов дисплея. Примеры программ в этом руководстве не используют консольные функции, потому что многие платформы не требуют или не предоставляют их.  
  
Арифметика  
Фундаментальными элементами большинства программ являются вычисления, решения(условное выполнение), итерации(циклы) и переменные для хранения входных данных, выходных данных и промежуточных результатов. Следующий пример программы иллюстрирует многие из этих концепций.  
Программа вычисляет наибольший общий делитель двух значений, используя алгоритм, изобретенный Евклидом.  
  
Функция main теперь содержит больше, чем оператор “print”. Когда тело функции содержит более одного оператора, эти операторы должны быть заключены в фигурные скобки – символы { и }. Это группирует инструкции в один составной оператор. Понятие группировки операторов в составном операторе также применимо к телам инструкции if-else и циклов.  
  
New создает переменную. Имя переменной следует за новым. Обычно, но не обязательно, присваивать значение переменной уже в момент ее создания. Переменные должны быть объявлены до того, как они будут использованы в выражении. Функция getvalue (также распространенная предопределенная функция) считывает значение с клавиатуры и возвращает результат. Обратите внимание, что pawn – это язык без типов, все переменные – числовые ячейки, которые могут содержать целочисленное значение со знаком.  
  
За именем функции getvalue следует пара круглых скобок. Это необходимо, потому что значение, которе возвращает getvalue, хранится в переменной. Обычно аргументы (или параметры) функции указываются в круглых скобках, но функция getvalue(используемая в этой программе) не принимает никаких явных аргументов. Если вы не присваиваете результат функции переменной или не используете его в выражении другим способом, круглые скобки необязательны. Например, результат операторов print и printf не используется. Вы по-прежнему можете использовать круглые скобки вокруг аргументов, но это не обязательно.  
  
Инструкции цикла, такие как “while”, повторяют одну инструкцию до тех пор, пока условие цикла (выражение в скобках) “true”. Можно выполнить несколько инструкций в цикле, сгруппировав их в составной оператор. В инструкции if-else есть одна инструкция для “истинного” предложения и одна для “ложного”.  
Обратите внимание, что некоторые операторы, такие как while и if-else, содержат(или “складывают”) другую инструкцию, а в случае if-else даже две другие инструкции.  
Полный комплект – это, опять же, единая инструкция. То есть:  
\* операторы присваивания “a = a – b” ниже if и “b = b – a” ниже else находятся операторы;  
\* оператор if-else объединяет эти два оператора присваивания и образует сам по себе единый оператор;  
\* оператор while сворачивает вокруг оператора if-else и снова образует единый оператор.

Обычно вложенность операторов делают явным, устанавливая отступ для любых вложенных операторов под оператором в исходном тексте. В примере “Наибольший общий делитель” отступ левого поля увеличивается на четыре символа пробела после оператора while и снова после ключевых слов if и else.  
Операторы одного уровня, такие как printf вызовы и цикл while имеют одинаковый отступ.  
  
Условие цикла для цикла while: “(a != b)” символ != является оператором “не равно”. То есть инструкция if-else повторяется до тех пор, пока “a” равно “б”. Хорошей практикой является создание отступа для инструкций, которые выполняются под управлением другого оператора, как это сделано в предыдущем примере.  
  
Вызов printf в нижней части примера отличается от вызова print сразу после открывающей фигурной скобки {. “f” в printf означает отформатировано, что означает, что функция может форматировать и печатать числовые значения и другие данные(в указанном пользователем формате), а также обычный текст.  
Символ %d в строке представляет собой токен, указывающий позицию и формат, в котором должен быть напечатан последующий аргумент функции printf. Во время выполнения токен %d заменяется значение переменной ‘a’ (второй аргумент printf).  
  
Функция print может печатать только текст; это быстрее, чем prinf. Если хочешь напечатать ‘%’ на дисплее, то ты должен использовать print, или ты должен удвоить его в строке printf.  
То есть:  


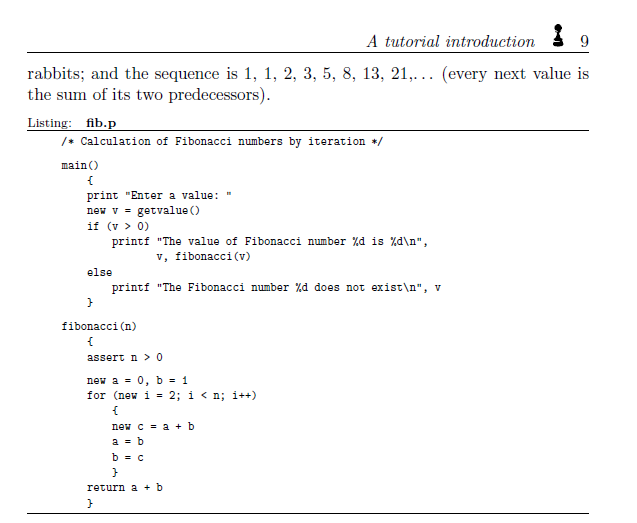
Массивы и константы  
Помимо простых переменных размером в одну ячейку, pawn поддерживает “переменные-массивы”, которые содержат множество ячеек/значений. В следующем примере программа отображает последовательность простых чисел, используя хорошо известное “решето Эратосфена”.  
Программа также вводит еще одну новую концепцию:  
символические константы.  
Символьные константы выглядят как переменные, но их нельзя изменить.  
  
Когда программа или подпрограмма имеет встроенный фиксированный предел, хорошей практикой является создание для нее символической константы. В предыдущем примере символ max\_primes является константой со значением 100. Программа использует символ max\_primes три раза после его определения: в объявлении переменной series и в обоих циклах for. Если бы мы адаптировали программу для печати всех простых чисел меньше 500, то теперь нужно было бы изменить только одну строку.

Как и простые переменные, массивы могут быть инициализированы при создании. pawn предлагает удобное сокращение для инициализации всех элементов фиксированным значением: всем сотням элементов массива «series» присваивается значение «истина» — без необходимости, чтобы программист ввел слово «истина» сто раз. Символы true и false являются предопределенными константами.

Когда простая переменная, такая как переменные i и j в примере с решетом простых чисел, объявляется в первом выражении цикла for, переменная действительна только внутри цикла. Объявление переменных имеет свои правила; это не утверждение, хотя и выглядит таковым. Одно из этих правил состоит в том, что первое выражение цикла for может содержать объявление переменной.  
  
Оба цикла for также вводят новые операторы в свое третье выражение. Оператор ++ увеличивает свой операнд на единицу; это означает, что ++i равно i = i + 1. Оператор += добавляет выражение справа к переменной слева; то есть j += i равно j = j + i.  
Существует проблема «отклонения от единицы», о которой вам нужно знать, если вы работаете с массивами. Первый элемент в массиве серий — это серия [0], поэтому, если массив содержит элементы max\_primes, последним элементом в массиве будет серия [max\_primes-1]. Если max\_primes равно 100, то последним элементом будет серия[99]. Доступ к серии [100] недействителен.  
  
• Функции

Большие программы разделяют задачи и операции на функции. Использование функций повышает модульность программ, а функции, если они хорошо написаны, переносимы в другие программы. В следующем примере реализована функция для вычисления чисел из ряда Фибоначчи.

Последовательность Фибоначчи была открыта Леонардо «Фибоначчи» из Пизы, итальянским математиком 13 века, величайшим достижением которого было

популяризация для западного мира индийско-арабских цифр. Цель последовательности состояла в том, чтобы описать рост популяции (идеализированных)  
  


Инструкция утверждения в верхней части функции Фибоначчи заслуживает отдельного упоминания о состоянии «утверждение»; он защищает от «невозможных» или недопустимых условий. ative число Фибоначчи недействительно, и оператор assert помечает его как ошибку программиста, если такое случается. Утверждения должны отмечать только ошибки программиста, а не ошибки пользовательского ввода.

Реализация определяемой пользователем функции не сильно отличается от реализации функции main. Однако функция fibonacci демонстрирует две новые концепции: она получает входное значение через параметр и возвращает значение (у нее есть «результат»).

Параметры функции объявляются в заголовке функции; единственный параметр в этом примере — «n». Внутри функции параметр ведет себя как локальная переменная, но значение которой передается извне при вызове функции. Оператор return завершает функцию и устанавливает результат функции. Он не должен появляться в самом конце функции; разрешены ранние выезды.

Основная функция примера Фибоначчи вызывает предопределенные «родные» функции, такие как getvalue и printf, а также определяемую пользователем функцию fibonacci. С точки зрения вызова функции (как в функции main) нет никакой разницы между пользовательскими и собственными функциями.

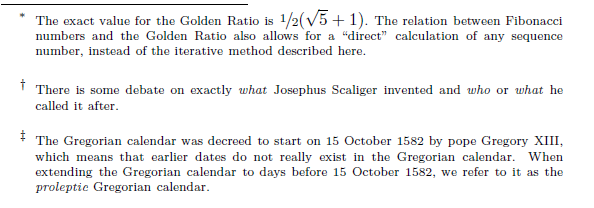
Последовательность чисел Фибоначчи описывает удивительное разнообразие природных явлений. Например, два или три набора спиралей в ананасах, сосновых шишках и подсолнухах обычно имеют последовательные числа Фибоначчи от 5 до 89 в качестве количества спиралей. Числа, которые естественным образом встречаются в паттернах ветвления (например, у растений), действительно являются числами Фибоначчи.

Наконец, хотя последовательность Фибоначчи не является геометрической последовательностью, чем дальше расширяется последовательность, тем ближе отношение между последовательными элементами приближается к золотому сечению, равному 1,618. . .∗, что так часто встречается в искусстве и архитектуре.  
  
• Вызов по ссылке и вызов по значению

Даты являются особенно богатым источником алгоритмов и процедур преобразования, потому что календари, к которым относится дата, знали такое разнообразие во времени и во всем мире.

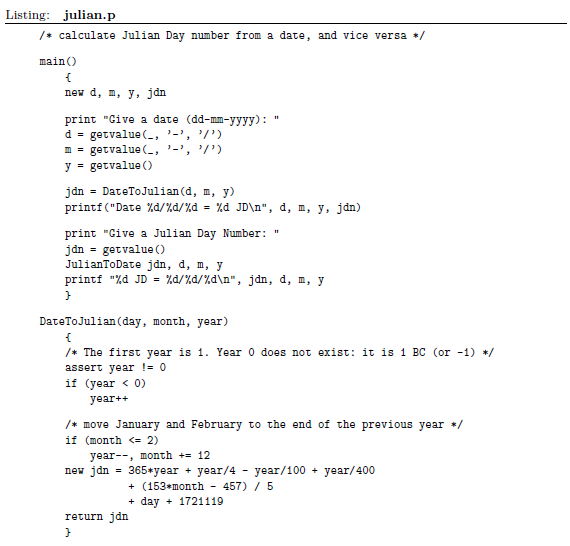
«Юлианское число дней» приписывается Иосифу Флавию Скалигеру† и подсчитывает количество дней с 24 ноября 4714 г. до н.э. (пролептический григорианский календарь‡).

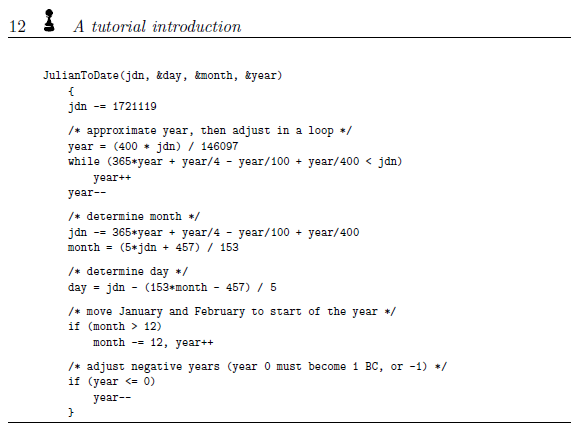
Скалигер выбрал эту дату, потому что она знаменовала совпадение трех

хорошо зарекомендовавшие себя циклы: 28-летний солнечный цикл (по старому юлианскому календарю), 19-летний метонический цикл и 15-летний индикационный цикл (периодические налоги или правительственные поборы в Древнем Риме), а также отсутствие литературы или письменной истории было известно, что оно предшествовало этой конкретной дате в далеком прошлом. Скалигер использовал эту концепцию для согласования дат в исторических документах.  
  


позже астрономы использовали его, чтобы легче вычислять интервалы между двумя событиями.

Числа юлианских дней (иногда обозначаемые единицей «jd») не следует путать с юлианскими датами (количество дней с начала того же года) или с юлианским календарем, введенным Юлием Цезарем. Ниже приведена программа, которая вычисляет номер дня по юлианскому календарю по дате (пролептического) григорианского календаря и наоборот. Обратите внимание, что в пролептическом григорианском календаре первый год — это 1 год нашей эры (Anno Domini), а годом ранее — 1 год до нашей эры (до Рождества Христова): нулевого года не существует! Программа использует отрицательные значения года для лет до нашей эры и положительные (ненулевые) значения для лет нашей эры.





Функция main начинается с создания переменных для хранения дня, месяца и года, а также рассчитанного числа дней по юлианскому календарю. Затем он считывает дату – три вызова getvalue – и вызывает функцию DateToJulian для вычисления номера дня. После вычисления результата main печатает введенную вами дату и номер дня по юлианскому календарю для этой даты. Теперь сосредоточимся на функции DateToJulian…  
  
В верхней части функции DateToJulian она увеличивает значение года, если это отрицательно; это делается, чтобы справиться с отсутствием «нулевого» года в пролептический григорианский календарь. Другими словами, функция DateToJulian изменяет аргументы своей функции (позднее она также изменяет месяц). Внутри функции аргумент ведет себя как локальная переменная: вы можете изменить его. Однако эти модификации остаются локальными для функции DateToJulian. Функция main передает значения d, m и y в DateToJulian, который сопоставляет их с аргументами своей функции день, месяц и год соответственно. Хотя DateToJulian изменяет год и месяц, он не изменяет y и m в функции main; он изменяет только локальные копии y и m. Эта концепция называется «колл по стоимости».  
  
В примере намеренно используются разные имена для локальных переменных в функциях main и DateToJulian, чтобы упростить приведенное выше объяснение. Переименование основных переменных d, m и y в день, месяц и год соответственно не меняет дела: тогда у вас просто есть две локальные переменные с именем day, две с именем month и две с именем year, что вполне допустимо в pawn.  
  
Остальная часть функции DateToJulian, с точки зрения пешечного языка, представляет собой неинтересную арифметику.

Возвращаясь ко второй части функции main, мы видим, что теперь она запрашивает номер дня и вызывает другую функцию, JulianToDate, чтобы найти дату, совпадающую с номером дня. Функция JulianToDate интересна тем, что она принимает один входной аргумент (число дней по юлианскому календарю) и должна вычислять три выходных значения: день, месяц и год. Увы, функция может иметь только одно возвращаемое значение, то есть оператор return в функции может содержать только одно выражение. Чтобы решить эту проблему, JulianToDate специально запрашивает, чтобы изменения, которые он вносит в некоторые аргументы своей функции, копировались обратно в переменные вызывающей функции. Затем в main переменные, которые должны содержать результат JulianToDate, передаются в качестве аргументов.

на ДжулианТоДате.

Функция JulianToDate помечает соответствующие аргументы для «копирования обратно в вызывающую программу», добавляя к ним префикс с символом &. Аргументы с & копируются обратно, аргументы без is — нет. «Обратное копирование» — не совсем правильный термин. Аргумент, помеченный символом &, передается в функцию особым образом, что позволяет функции напрямую изменять исходную переменную. Это называется «вызов по ссылке», а аргумент, который его использует, называется «аргументом-ссылкой».

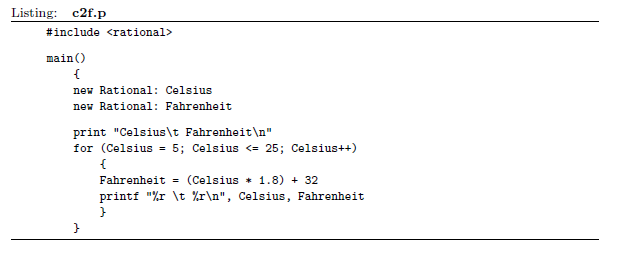
Другими словами, если main передает y функции JulianToDate, которая сопоставляет ее с годом аргумента своей функции, а JulianToDate меняет год, тогда JulianToDate действительно меняет y. Только с помощью ссылочных аргументов функция может напрямую изменять переменную, объявленную в другой функции.

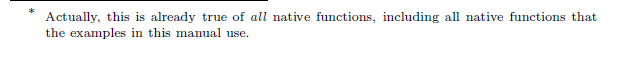
Подводя итог использованию вызова по значению по сравнению с вызовом по ссылке: если функция имеет одно выходное значение, вы обычно используете оператор return; если функция имеет больше выходных значений, вы используете ссылочные аргументы. Вы можете комбинировать их внутри одной функции, например, в функции, которая возвращает свой «нормальный» вывод через ссылочный аргумент и код ошибки в возвращаемом значении.

Кроме того, многие настольные приложения используют преобразования в числа юлианского дня и обратно (или их разновидности), чтобы удобно рассчитать количество дней между датами или для расчета даты, наступающей через 90 дней.  
  
• Рациональное число

Все расчеты, сделанные до этого момента, включали только целые числа — целые значения. Pawn также поддерживает числа, которые могут содержать дробные значения: они называются «рациональными числами». Однако то, включена ли эта поддержка, зависит от хост-приложения.

Рациональные числа могут быть реализованы как числа с плавающей или фиксированной точкой. Арифметика с плавающей запятой обычно используется для общих и научных вычислений, в то время как арифметика с фиксированной запятой больше подходит для финансовой обработки и приложений, где ошибки округления не должны вступать в игру (или, по крайней мере, они должны быть предсказуемы). В наборе инструментов pawn есть модули с плавающей запятой и с фиксированной запятой, а также подробности (и компромиссы) для этих модулей в соответствующей документации. Проблема, однако,

что хост-приложение может реализовывать либо плавающую, либо фиксированную точку, либо обе, либо ни одну из них. \* Приведенная ниже программа требует, чтобы по крайней мере любой из видов поддержки рациональных чисел был доступен; он не запустится, если основное приложение вообще не поддерживает рациональные числа.  
  


Пример программы преобразует таблицу градусов Цельсия в градусы Фаренгейта. Первая директива этой программы — импортировать определения поддержки рациональных чисел из включаемого файла. Файл «рациональный» включает поддержку либо чисел с плавающей запятой, либо чисел с фиксированной запятой, в зависимости от того, что доступно.  
  


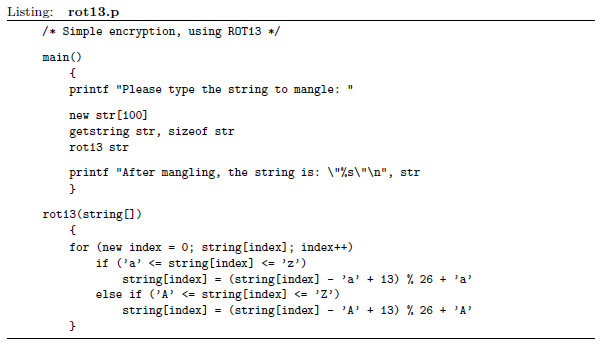
Переменные Цельсия и Фаренгейта объявляются с тегом «Rational:»

Имена тегов: 67 между ключевым словом new и именем переменной. Имя тега обозначает назначение переменной, ее разрешенное использование и, как особый случай для рациональных чисел, расположение ее памяти. Тег Rational: сообщает синтаксическому анализатору пешки, что переменные Цельсия и Фаренгейта содержат дробные значения, а не целые числа.

Уравнение для получения градусов Фаренгейта из градусов Цельсия: 

Программа использует значение 1,8 для частного 9/5. Когда включена поддержка рациональных чисел, pawn поддерживает значения с дробной частью после запятой.  
Единственное другое нетривиальное изменение по сравнению с более ранними программами заключается в том, что строка формата для функции printf теперь имеет заполнители переменных, обозначаемые «%r» вместо «%d». Заполнитель %r печатает рациональное число в позиции; %d предназначен только для целых чисел («целые числа»).  
Я использовал включаемый файл «рациональный», а не «плавающий» или «фиксированный» в

попытайтесь сделать пример программы переносимым. Если вы знаете, что хост-приложение поддерживает арифметику с плавающей запятой, возможно, будет удобнее «#включить» определения из файла float и использовать тег Float: вместо Rational — при этом также следует заменить %r на % f в вызове printf. Дополнительные сведения о поддержке с фиксированной и плавающей запятой см. в примечаниях по применению «Библиотека поддержки фиксированной точки» и «Библиотека поддержки операций с плавающей запятой», которые доступны отдельно.  
  
• Пешка Strings не имеет встроенного типа «string»; строки символов

хранятся в массивах с условием, что элемент массива за последним допустимым символом равен нулю. Таким образом, работа со строками эквивалентна работе с массивами. Среди самых простых схем шифрования есть та, которая называется «ROT13» — на самом деле алгоритм довольно «слабый» с криптографической точки зрения. Наиболее широко он используется в общедоступных электронных форумах (BBS, Usenet) для сокрытия текстов от случайного чтения, например решения головоломок или загадок. ROT13 просто «поворачивает» алфавит на половину его длины, то есть на 13 символов. Это симметричная операция: дважды применив ее к одному и тому же тексту, вы обнаружите оригинал.  
  


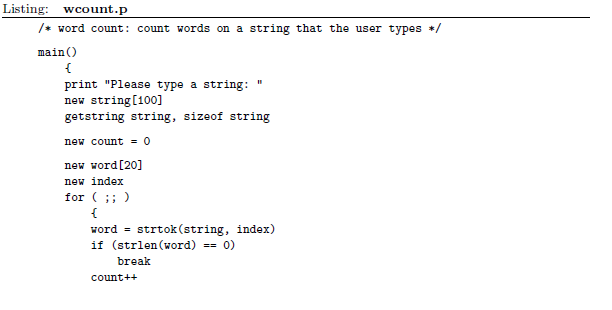
В заголовке функции rot13 параметр string объявлен как массив, но без указания размера массива — в квадратных скобках нет значения. Когда вы указываете размер массива в заголовке функции, он должен соответствовать размеру фактического параметра в вызове функции. Отсутствие спецификации размера массива в заголовке функции снимает это ограничение и позволяет вызывать функцию с массивами любого размера. Затем у вас должны быть какие-то другие средства определения (максимального) размера массива. В случае строкового параметра можно просто найти нулевой терминатор.

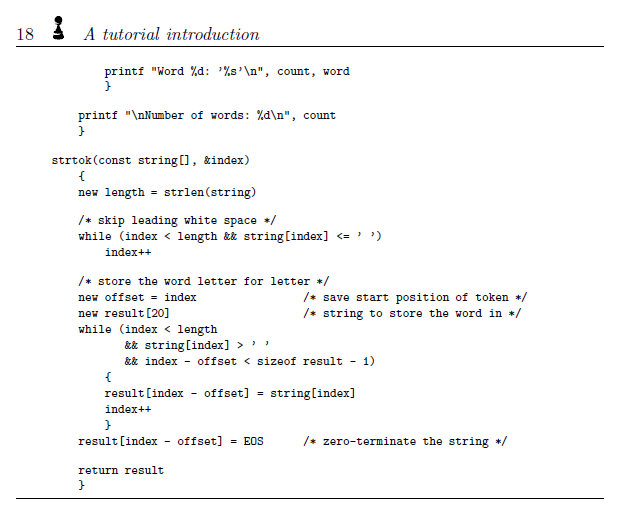
Цикл for, обходящий строку, типичен для функций обработки строк. Обратите внимание, что условием цикла является «string[index]». Правило истинного/

Ложное условие в pawn заключается в том, что любое значение является «истинным», кроме нуля. То есть, когда ячейка массива в строке [индекс] равна нулю, это «ложь», и цикл прерывается. Алгоритм ROT13 меняет только буквы; цифры, знаки препинания и специальные символы остаются без изменений. Кроме того, прописные и строчные буквы должны обрабатываться отдельно. Внутри цикла for два оператора if отфильтровывают интересующие символы. Примечателен способ, которым второй if связан с предложением «else» первого if, поскольку это типичный метод проверки нескольких непересекающихся условий.  
  
Ранее в этой главе обсуждалась концепция «вызова по значению» и «вызова по ссылке». Когда вы работаете со строками или вообще с массивами, обратите внимание, что pawn всегда передает массивы по ссылке. Это делается для экономии памяти и повышения производительности — массивы могут быть большими структурами данных, и передача их по значению требует создания копии этой структуры данных, что занимает как память, так и время. Благодаря этому правилу функция rot13 может изменять свой параметр функции (называемый в примере «строкой») без необходимости объявлять его в качестве аргумента ссылки.  
  
Еще одним интересным моментом являются условия в двух операторах if. Первый if, например, содержит условие «'a' <= string[index] <= 'z'», что означает, что выражение истинно, если (и только если) оба 'a' <= string[index] и string[index] <= 'z' верны. В комбинированном выражении операторы отношения называются «сцепленными», поскольку они связывают несколько сравнений в одно условие.

Наконец, обратите внимание, как последний printf в функции main использует управляющую последовательность \" для вывода двойной кавычки. Обычно двойная кавычка завершает литеральную строку; управляющая последовательность "\"" вставляет в строку двойную кавычку.

Продолжая тему строк и массивов, ниже приведена программа, которая разделяет строку текста на отдельные слова и подсчитывает их. Это простая программа, которая показывает несколько новых возможностей языка пешек.





Функция main сначала отображает сообщение и извлекает строку, которую должен ввести пользователь. Затем он входит в цикл: запись «for (;;)» создает цикл без инициализации, без приращения и без проверки — это бесконечный цикл, эквивалентный «пока (истина)». Однако там, где синтаксический анализатор пешек выдаст вам предупреждение, если вы наберете «пока (истина)» (что-то вроде «избыточный

тестовое выражение; всегда верно»), «for (;;)» проходит анализатор без предупреждения.

Типичное использование бесконечного цикла — это случай, когда вам нужен цикл с проверкой в ​​середине — гибрид между while и do. . .while цикл, так сказать. pawn не поддерживает циклы с проверкой в ​​середине напрямую, но вы можете имитировать их, закодировав бесконечный цикл с условным разрывом. В этом примере программы цикл:

⋄ получает слово из строки —code перед тестом ;

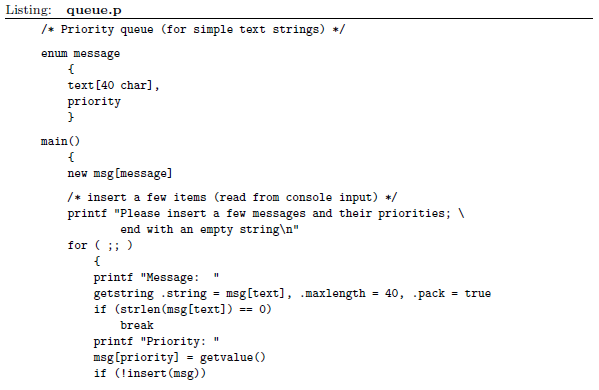
⋄ проверяет, доступно ли новое слово, и прерывает цикл, если нет — проверка посередине;

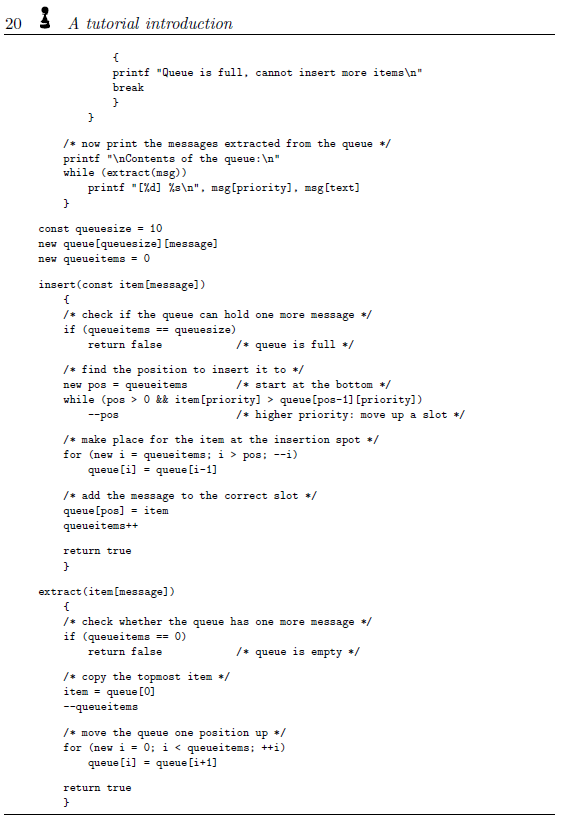
⋄ печатает слово и его порядковый номер —код после проверки.

Как видно из строки «word = strtok(string, index)» (и объявления переменной word), pawn поддерживает присваивание массивов и функции, возвращающие массивы. Парсер пешки проверяет, что массив, возвращаемый strtok, имеет тот же размер и размерность, что и переменная, которой он присвоен.

Функция strlen — нативная функция (предопределенная), а strtok — нет: ее нужно реализовать самим. Функция strtok была вдохновлена одноименной функцией из C/C++, но она не изменяет исходную строку. Вместо этого он слово в слово копирует символы из исходной строки в локальный массив, который затем возвращает.  
  
• Массивы и перечисления (структурированные данные)

В бестиповом языке мы могли бы присвоить некоторым элементам массива другое назначение, чем другим элементам того же массива. pawn поддерживает перечисляемые константы с расширением, позволяющим имитировать некоторые функции, которые другие языки реализуют с помощью «структур» или «записей».

Пример, иллюстрирующий перечисления и массивы, длиннее, чем предыдущие программы Pawn, и в нем также показаны некоторые другие функции, такие как глобальные переменные и именованные параметры.  
  




В верхней части листинга программы находится объявление сообщения перечисления. Это перечисление определяет две константы: текст, равный нулю, и приоритет, равный 10 (при условии, что ячейка 32-разрядная). Идея перечисления

заключается в быстром определении списка символических констант без дубликатов. По умолчанию каждая константа в списке на 1 больше предыдущей, а самая первая константа в списке равна нулю. Однако вы можете указать дополнительное приращение для константы, чтобы преемник имел значение 1 плюс это дополнительное приращение. Текстовая константа задает дополнительное приращение в 40 символов. В pawn char — это оператор, он возвращает количество ячеек, необходимых для хранения упакованной строки из указанного количества символов. Предполагая 32-битную ячейку и 8-битный символ, 10 ячеек могут содержать 40 упакованных символов.

Непосредственно в начале функции main объявляется новая переменная массива с размером сообщения. Сообщение символа является именем перечисления. Это также константа со значением последней константы в списке перечисления плюс необязательный дополнительный приращение для этого последнего элемента. Таким образом, в этом примере сообщение равно 11. Другими словами, массив msg объявлен для хранения 11 ячеек.

Далее в main идут две петли. Цикл for считывает строки и значения приоритета из консоли и вставляет их в очередь. Приведенный ниже цикл while извлекает элемент за элементом из очереди и выводит информацию на экран. Следует отметить, что цикл for сохраняет и строку, и номер приоритета (целое число) в одной и той же переменной msg; действительно, функция main объявляет только одну переменную. Функция getstring сохраняет текст сообщения, которое вы набираете, начиная с массива msg[текст], в то время как значение приоритета сохраняется (путем присваивания несколькими строками ниже) в msg[priority]. Функция printf в цикле while также считывает строку и значение из этих позиций.

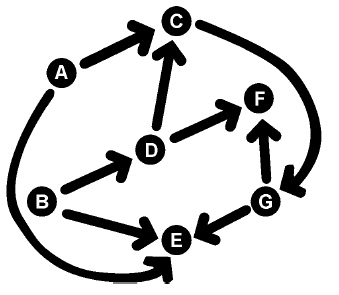
В то же время массив msg является сущностью сам по себе: он целиком передается функции вставки. Эта функция, в свою очередь, говорит ближе к концу «queue[queueitems] = item», где item — это массив с сообщением о размере, а очередь — это двумерный массив, который содержит элементы размера очереди с сообщением о размере. Объявление очереди и размера очереди находится чуть выше функции вставки. В примере реализована «очередь с приоритетом». Вы можете вставить несколько сообщений в очередь, и когда все эти сообщения имеют одинаковый приоритет, они извлекаются из очереди в одном и том же порядке. Однако, когда сообщения имеют разные приоритеты, сообщение с наивысшим приоритетом выходит первым. «Интеллект» для этой операции находится внутри функции вставки: она сначала определяет позицию нового сообщения для добавления, затем перемещает несколько сообщений на одну позицию вверх, чтобы освободить место для нового сообщения. Функция извлечения просто всегда извлекает первый элемент очереди и сдвигает все остальные элементы на одну позицию вниз.

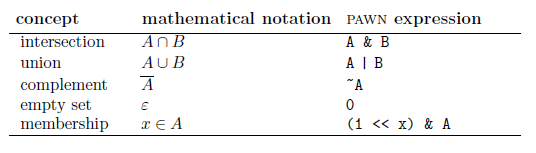
Обратите внимание, что обе функции вставки и извлечения работают с двумя общими переменными, очередью и элементами очереди. К переменной, объявленной внутри функции, например к переменной msg в функции main, можно получить доступ только из этой функции. «Глобальная переменная» доступна для всех функций, и эта переменная объявлена вне области видимости любой функции. Переменные по-прежнему должны быть объявлены до того, как они будут использованы, поэтому функция main не может получить доступ к переменным очереди и элементам очереди, но как вставка, так и извлечение могут. Извлечение функции возвращает сообщения с наивысшим приоритетом через элемент аргумента функции. То есть он изменяет аргумент своей функции, копируя первый элемент массива очереди в элемент. Функция вставки копирует в другом направлении и не меняет элемент аргумента своей функции. В таком случае рекомендуется пометить аргумент функции как «const». Это помогает синтаксическому анализатору пешек как проверять наличие ошибок, так и генерировать более качественный (более компактный и быстрый) код.

Последнее замечание по этому последнему примеру — вызов getstring в функции main: обратите внимание, как параметры атрибутируются с описанием. Первый параметр помечен как «.string», второй — как «.maxlength», а третий — как «.pack». Функция getstring получает «именованные параметры», а не позиционные параметры. Порядок, в котором перечислены именованные параметры, не важен. Именованные параметры удобны для указания и расшифровки длинных списков параметров.  
• Битовые операции для управления «множествами»

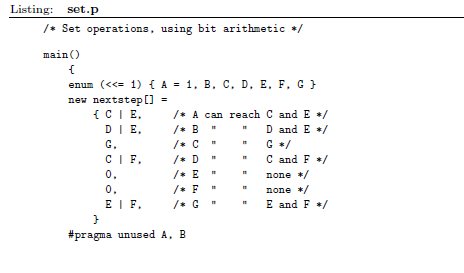
Несколько алгоритмов легче всего решить с помощью «операций над множествами», таких как пересечение, объединение и инверсия. Например, на рисунке ниже мы хотим разработать алгоритм, который возвращает нам точки, до которых можно добраться из какой-либо другой точки за заданное максимальное количество шагов. Например, если мы попросим его вернуть точки, которые могут быть достигнуты за два шага, начиная с B, алгоритм должен вернуть C, D, E и F, но не G, потому что G делает три шага из B. Наш подход состоит в том, чтобы сохраните для каждой точки на графике набор других точек, которых она может достичь за один шаг — это набор «next\_step». У нас также есть набор «результатов», в котором хранятся все точки, которые мы нашли до сих пор. Начнем с того, что установим набор результатов равным набору next\_step для точки отправления. Теперь у нас в результирующем наборе есть все точки, до которых можно добраться за один шаг. Затем для каждой точки в нашем наборе результатов мы создаем объединение набора результатов и набора next\_step для этой точки. Этот процесс повторяется в течение определенного количества циклов.

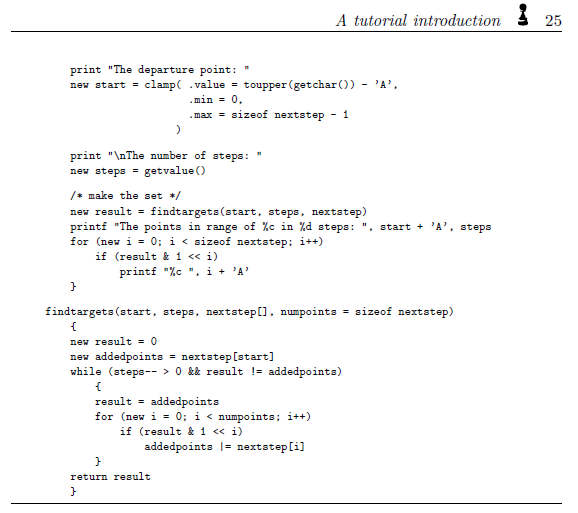
Пример может прояснить описанную выше процедуру. Когда отправной точкой является B, мы начинаем с установки набора результатов в D и E — это точки, которых можно достичь из B за один шаг. Затем мы проходим через набор результатов. Первая точка, с которой мы сталкиваемся в наборе, — это D, и мы проверяем, какие точки могут быть достигнуты из D за один шаг: это C и F. Итак, мы добавляем C и F в результирующий набор. Мы знали, что точки, которые могут быть достигнуты из D за один шаг, — это C и F, потому что C и F находятся в наборе next\_step для D. Итак, что мы делаем, так это объединяем набор next\_step для точки D с результирующим набором. Слияние называется «объединением» в теории множеств. Это обрабатывает D. Исходный результирующий набор также содержал точку E, но набор next\_step для E пуст, поэтому точка больше не добавляется. Таким образом, новый набор результатов теперь содержит C, D, E и F.



Набор — это контейнер общего назначения для элементов. Единственная информация, которую множество содержит об элементе, это то, присутствует ли он в множестве или нет. Порядок элементов в наборе не имеет значения, и набор не может содержать один и тот же элемент несколько раз. Язык пешек не предоставляет тип данных «множество» или операторы, которые работают с множествами. Однако наборы, содержащие до 32 элементов, можно моделировать с помощью битовых операций. Для хранения статуса «присутствует/отсутствует» требуется всего один бит, поэтому 32-битная ячейка может поддерживать статус для 32 элементов набора — при условии, что каждому элементу назначена уникальная битовая позиция.  
  
Связь между операциями над множествами и побитовыми операциями представлена в следующей таблице. В таблице заглавная буква обозначает набор, а строчная буква — элемент из этого набора.  
  


Чтобы проверить принадлежность, то есть узнать, содержит ли набор определенный элемент, создайте набор только с одним элементом и выполните пересечение. Если результат равен 0 (пустой набор), элемент отсутствует в наборе. Нумерация битов обычно начинается с нуля; младший бит — это бит 0, а старший бит в 32-битной ячейке — это бит 31. Чтобы создать ячейку с установленным только битом 7, сдвиньте значение 1 влево на семь — или в выражении пешки: «1 << 7» .

Ниже представлена программа, реализующая описанный ранее алгоритм поиска всех точек, до которых можно добраться с определенного отправления за заданное количество шагов. Алгоритм полностью находится в функции findtargets.  
  




Оператор enum сразу под заголовком основной функции объявляет константы для узлов от A до G, но с изюминкой. Обычно перечисление начинается с нуля; здесь значение первой константы, A, явно установлено равным 1. Более примечательным является выражение «(<<= 1)» между ключевым словом enum и открывающей фигурной скобкой, с которой начинается список констант: оно определяет «побитовый сдвиг приращение. По умолчанию каждая константа в списке перечислений получает значение, которое на 1 выше, чем у предыдущей, но вы можете указать, что каждая последующая константа в перечислении имеет значение, которое равно:

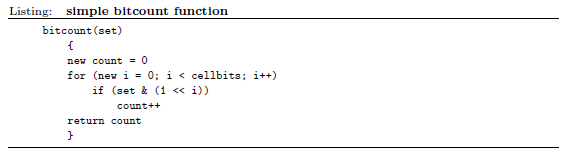
⋄ его предшественник увеличился на любое значение (не только на 1) — например, «(+= 5)»;

⋄ его предшественник, умноженный на любое значение — например, «(\*= 3)»;

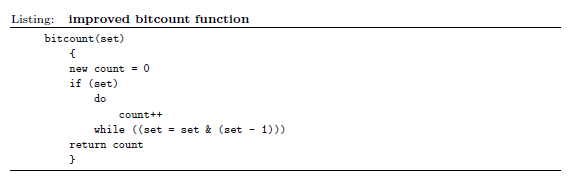
⋄ его предшественник сдвинут влево на любое значение — например, «(<<= 1)»;

Обратите внимание, что в двоичной арифметике сдвиг влево на один бит эквивалентен умножению на два, а это означает, что «(\*= 2)» и «(<<= 1)» делают одно и то же.

При работе с наборами типичная всплывающая задача — определить количество элементов в наборе. Простая функция, которая делает это, приведена ниже:



При размере ячейки 32 бита цикл этой функции повторяется 32 раза, проверяя один бит на каждой итерации. С помощью магии двоичной арифметики мы можем свести его к циклу только для количества битов, которые «установлены». То есть следующая функция выполняет итерацию только один раз, если входное значение имеет только один установленный бит:

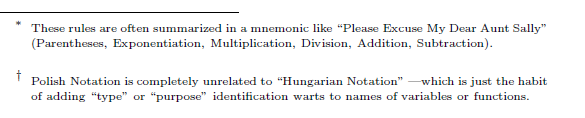


Простой калькулятор RPN

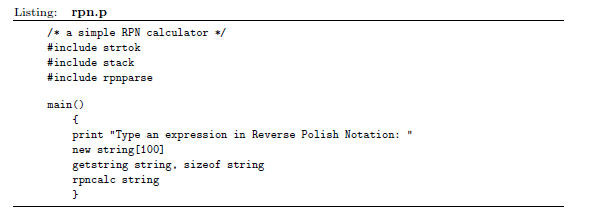
Общепринятая математическая запись с такими выражениями, как «26−3×(5+2)», известна как алгебраическая запись. Это компактная нотация, и мы к ней привыкли. pawn и, безусловно, большинство других языков программирования используют алгебраическую нотацию для своих программных выражений. Однако у алгебраической записи есть несколько недостатков. Например, иногда требуется, чтобы порядок операций был сделан явным путем заключения части выражения в круглые скобки. Выражение в начале этого абзаца можно переписать, убрав круглые скобки, но за счет увеличения его длины почти вдвое. На практике алгебраическая нотация дополняется правилами уровня старшинства, согласно которым, например, умножение предшествует сложению и вычитанию.\* Уровни старшинства значительно уменьшают потребность в круглых скобках, но не позволяют полностью их избежать. Хуже того, когда количество операторов становится большим, иерархию уровней приоритета и конкретный уровень приоритета для каждого оператора становится трудно запомнить — вот почему язык с большим количеством операторов, такой как APL, полностью избавляется от уровней приоритета.

Примерно в 1920 году польский математик Ян Лукасевич продемонстрировал, что, помещая операторы перед их операндами, а не между ними, уровни приоритета становятся излишними и круглые скобки никогда не нужны. Эта нотация стала известна как «польская нотация».† Чарльз Хэмблин позже предложил помещать операторы за операндами, назвав это «обратной польской нотацией». Преимущество обратного порядка заключается в том, что операторы перечислены в том же порядке, в котором они должны выполняться: при чтении операторов слева направо у вас также есть операции, которые необходимо выполнить в этом порядке. Алгебраическое выражение из начала этого раздела будет читаться в rpn как:



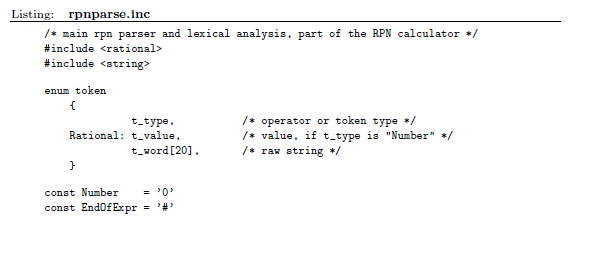
Если смотреть только на операторы, у нас есть: сначала сложение, затем умножение и, наконец, вычитание. Операнды каждого оператора читаются справа налево: операнды оператора + — это значения 5 и 2, операнды оператора × — результат предыдущего сложения и значение 3 и так далее.  
Полезно представить, что значения складываются в стопку, где операторы берут один или несколько операндов с вершины стопки и помещают результат обратно на вершину стопки. При чтении выражения rpn значения 26, 3, 5 и 2 «складываются» в указанном порядке. Оператор + удаляет из стека два верхних элемента (5 и 2) и возвращает сумму этих значений обратно — теперь стек читается как «26 3 7». Затем оператор × удаляет 3 и 7 и помещает произведение значений в стек — стек равен «26 21». Наконец, оператор − вычитает 21 из 26 и сохраняет единственное значение 5, конечный результат выражения, обратно в стек.  


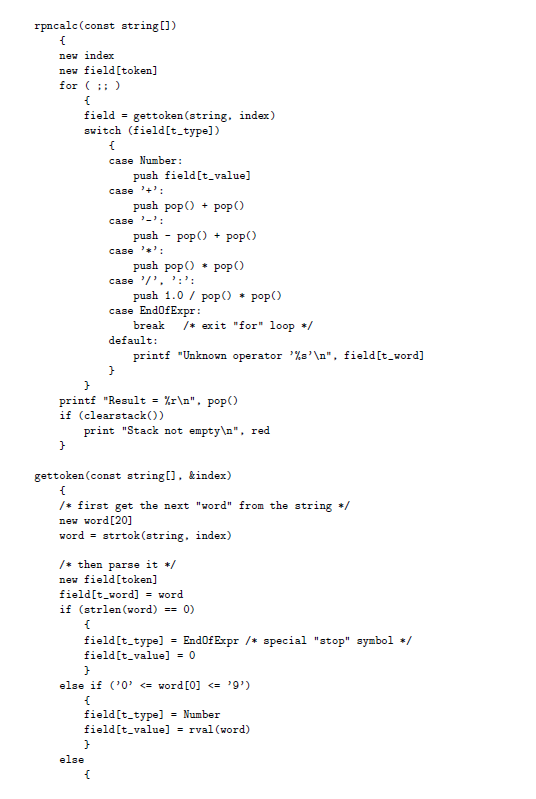
Обратная польская запись стала популярной, потому что ее было легко понять и легко реализовать в (ранних) калькуляторах. Это также открывает путь к операторам с более чем двумя операндами (например, интегрирование) или операторам с более чем одним результатом (например, преобразование между полярными и декартовыми координатами). Основная программа для калькулятора обратной польской записи приведена ниже:

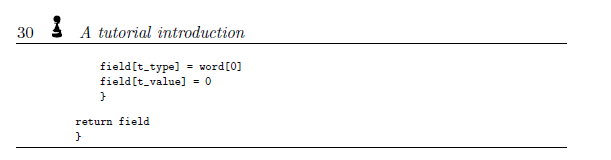


Основная программа сама по себе содержит очень мало кода; вместо этого он включает требуемый код из трех других файлов, каждый из которых реализует несколько функций, которые вместе создают калькулятор rpn. Когда программы или сценарии становятся больше, обычно рекомендуется распределить реализацию по нескольким файлам, чтобы упростить обслуживание.

Сначала функция main выводит приглашение и вызывает встроенную функцию getstring для чтения выражения, которое вводит пользователь. Затем он вызывает пользовательскую функцию rpncalc для выполнения реальной работы. Функция rpncalc реализована в файле rpnparse.inc, воспроизведенном ниже:







Калькулятор rpn использует поддержку рациональных чисел, и rpnparse.inc включает в себя «рациональный» файл для этой цели. Почти все операции над рациональными числами скрыты в арифметике. Единственными прямыми ссылками на рациональные числа являются код формата «%r» в операторе printf в нижней части функции rpncalc и вызов наполовину функции рационального вывода gettoken.

Первым примечательным элементом в файле rpnparse.inc является объявление enum, где один элемент имеет тег (t\_value), а другой элемент имеет размер (t\_word). Функция rpncalc объявляет переменное поле как массив, используя символ перечисления в качестве его размера. За экраном это объявление делает больше, чем просто создает массив с 22 ячейками:

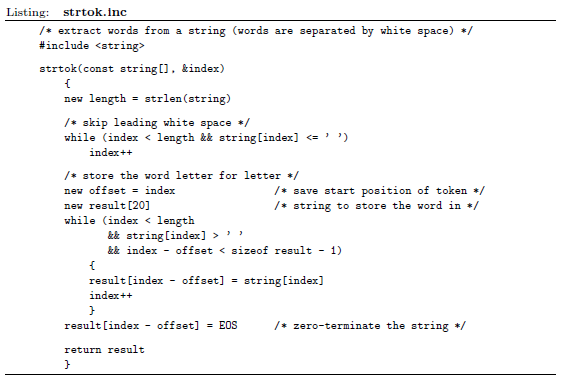
Тег индекса массива устанавливается на имя тега «токен:». Это означает, что вы можете индексировать массив с любым из элементов перечисления, но не со значениями, имеющими другой тег. Другими словами, с field[t\_type] все в порядке, но field[1] дает диагностику синтаксического анализатора.

Имя тега перечисления имеет приоритет над именем тега массива. переменная, если есть. Переменная поля не помечена, но поле[t\_value] имеет тег Rational:, потому что элемент перечисления t\_value объявлен как таковой. Следовательно, это позволяет вам создать массив, элементы которого имеют разные имена тегов.

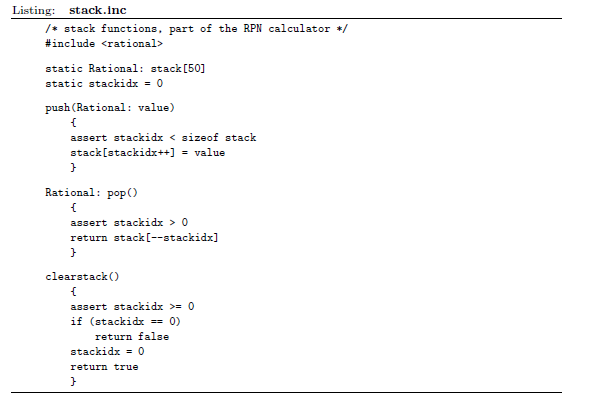
Когда элемент перечисления имеет размер, элемент массива, указанный этим элементом, иногда рассматривается как подмассив. В rpncalc выражение «field[t\_type]» — это одна ячейка, «field[t\_value]» — это одна ячейка, а «field[t\_word]» — это одномерный массив из 20 ячеек. Мы видим это, в частности, в строке: printf "Неизвестный оператор '%s'\n", field[t\_word], где код формата %s ожидает строку — массив с нулевым завершением.

Если вы знаете C/C++ или Java, вы можете взглянуть на оператор switch. Оператор switch во многом отличается от других языков, которые его предоставляют. Случаи, например, не проваливаются, что, в свою очередь, означает, что оператор break для случая EndOfExpr выходит из охватывающего цикла, а не из переключателя.

В верхней части цикла for в функции rpncalc вы найдете инструкцию «field = gettoken(string, index)». Как уже было показано в программе wcount.p («счетчик слов») на стр. 17, функции могут возвращать массивы. Еще интереснее аналогичная строка в функции gettoken: field[t\_word] = word, где word — массив из 20 ячеек, а field — массив из 22 ячеек. Однако, поскольку поле перечисления t\_word объявлено как имеющее размер 20 ячеек, «field[t\_word]» считается подмассивом из 20 ячеек, что точно соответствует размеру массива word.



Функция strtok такая же, как и в примере с wcount.p. Он реализован в отдельном файле для программы-калькулятора рпн. Обратите внимание, что функция strtok в том виде, в каком она здесь реализована, может обрабатывать только слова, содержащие до 19 символов, причем 20-й символ является завершающим нулем. По-настоящему многоразовая реализация функции strtok общего назначения будет передавать массив назначения в качестве параметра, чтобы она могла обрабатывать слова любого размера. Поддержка как упакованных, так и распакованных строк также была бы полезной функцией функции общего назначения. Обсуждая достоинства обратной польской нотации, я упомянул, что стек одновременно помогает «визуализировать» алгоритм, а также является удобным методом реализации синтаксического анализатора rpn. Этот пример калькулятора rpn использует стек с вездесущими функциями push и pop. Для проверки ошибок и сброса стека есть третья функция, очищающая стек.



Файл stack.inc снова включает файл ratio. это технически

не обязательно (в rpnparse.inc уже включены определения для поддержки рациональных чисел), но это также не приносит никакого вреда, и ради повторного использования кода лучше сделать так, чтобы любой файл включал определения библиотек, которые он зависит от. Обратите внимание, что две глобальные переменные stack и stackidx объявлены как «статические» переменные; используя ключевое слово static вместо new. Это делает глобальные переменные «видимыми» только в этом файле. Для всех остальных файлов в большом проекте стек символов и stackidx невидимы, и они не могут (случайно) изменять переменные. Это также позволяет другим модулям объявлять свои собственные частные переменные с этими именами, что позволяет избежать конфликта имен. Калькулятор rpn на самом деле все еще довольно маленькая программа, но она настроена так, как если бы это была более крупная программа. Он также был разработан для демонстрации набора элементов пешечного языка, и программу-пример можно было бы реализовать более компактно.

• Программирование, управляемое событиями

Все программы-примеры, разработанные до сих пор в этой главе, использовали модель программирования, управляемую потоком: они начинаются с main, а код определяет, что делать и когда запрашивать ввод. Эта модель программирования проста для понимания и хорошо подходит для большинства языков программирования, но она также не подходит для многих ситуаций «реальной жизни». Довольно часто программа не может просто обрабатывать данные и предлагает пользователю вводить данные только тогда, когда они готовы для него/нее. Вместо этого именно пользователь решает, когда вводить данные, а программа или сценарий должны быть готовы обработать их за приемлемое время, независимо от того, что они делали в данный момент.

Приведенное выше описание предполагает, что программа должна иметь возможность прерывать свою работу и выполнять другие действия, прежде чем приступить к исходной задаче. В ранних реализациях такая функциональность действительно была реализована: многозадачная система, в которой одна задача (или поток) управляла фоновыми задачами, а вторая задача/поток находилась в цикле, постоянно запрашивая пользовательский ввод. Однако это тяжеловесное решение. Более легкая реализация адаптивной системы называется моделью программирования, управляемой событиями.

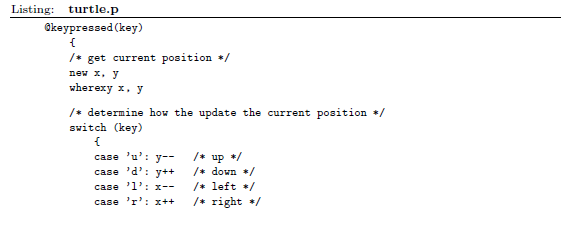
В модели программирования, управляемой событиями, программа или сценарий разбивает любую длинную (фоновую) задачу на короткие управляемые блоки, а между ними она доступна для ввода. Однако вместо того, чтобы программа опрашивала входные данные, хост-приложение (или какая-либо другая подсистема) вызывает функцию, связанную с событием, но только в том случае, если событие происходит.

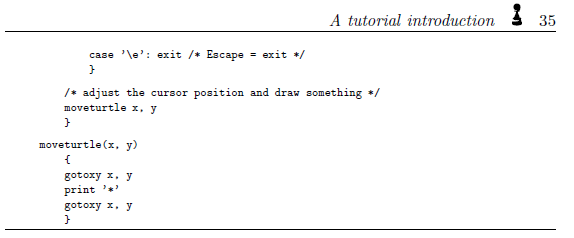
Типичным событием является «вход». Обратите внимание, что ввод исходит не только от людей-операторов. Входные пакеты могут поступать по последовательным кабелям, сетевым стекам, внутренним подсистемам, таким как таймеры и часы, и любому другому оборудованию, которое может быть подключено к вашей системе. Многие аппараты, производящие ввод, просто отправляют его. Поступление такого ввода является событием, как и нажатие клавиши. Если вы не перехватите событие, некоторые из них могут быть сохранены во внутренней системной очереди, но как только очередь переполняется, события просто отбрасываются.

pawn напрямую поддерживает модель, управляемую событиями, поскольку она поддерживает несколько точек входа. Единственной точкой входа в программу, управляемую потоком, является main; программа, управляемая событиями, имеет точку входа для каждого события, которое она фиксирует. По сравнению с моделью, управляемой потоком, программы, управляемые событиями, часто кажутся «снизу вверх»: вместо того, чтобы ваша программа обращалась к хост-приложению и решала, что делать дальше, ваша программа вызывается извне, и от нее требуется реагировать адекватно и оперативно.

pawn не указывает стандартную библиотеку, поэтому нет никакой гарантии что в конкретной реализации такие функции, как printf и getvalue. Хотя предполагается, что каждая реализация предоставляет минимальный консольный/терминальный интерфейс с этими функциями, их доступность в конечном счете зависит от реализации. То же самое относится и к общедоступным функциям — точкам входа для скрипта. Какие общедоступные функции поддерживает хост-приложение, зависит от реализации. Поэтому сценарий в этом разделе может не работать на вашей платформе (даже если все предыдущие сценарии работали нормально). Инструменты в стандартном дистрибутиве системы pawn поддерживают все сценарии, разработанные в этом руководстве, при условии, что ваша операционная система или среда поддерживают стандартные функции терминала, такие как установка положения курсора.

Первым языком программирования, который был разработан исключительно для обучения детей концепциям программирования, был «Лого». Этот диалект LISP сделал программирование наглядным благодаря тому, что маленький робот, «черепаха», едет по полу под управлением простой программы. Затем эта концепция была скопирована для перемещения (обычно треугольного) курсора на дисплее компьютера, опять же под управлением программы. Новым было то, что черепаха теперь оставляла за собой след, позволяющий создавать рисунки, правильно программируя черепаху — это стало известно как графика черепах. Термин «графика черепахи» также использовался для интерактивного рисования с помощью клавиш со стрелками на клавиатуре и «черепашка» для текущей позиции. Этот метод рисования изображений на компьютере был популярен до появления мыши.





Точка входа вышеуказанной программы — @keypressed — она вызывается при нажатии клавиши. Если вы запустите программу и не нажмете ни одной клавиши, функция @keypressed никогда не запустится; если вы наберете десять клавиш, @keypressed запустится десять раз. Сравните это поведение с main: функция main запускается сразу после запуска скрипта и запускается только один раз.

По-прежнему разрешено добавлять основную функцию в программу, управляемую событиями: в этом случае основная функция будет служить для одноразовой инициализации. Простым дополнением к этому примеру программы является добавление основной функции, чтобы очищать окно консоли/терминала при входе и, возможно, устанавливать начальное положение «черепахи» в центре.

Поддержка функциональных клавиш и других специальных клавиш (например, клавиш со стрелками) сильно зависит от системы. На терминалах ANSI эти клавиши производят другой код, чем в «DOS-ящике» Windows. Чтобы программа-пример оставалась переносимой, я использовал обычные буквы («u» для «вверх», «l» для «влево» и т. д.). Это не означает, однако, что специальные ключи находятся за пределами возможностей пешки.

В сценарии «черепашка» клавиша «Escape» завершает работу хост-приложения.

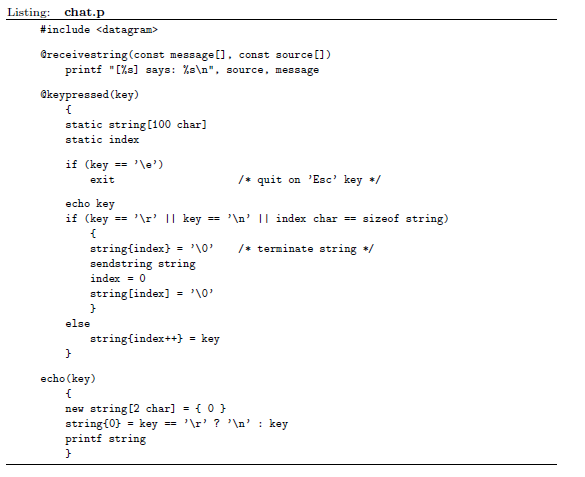
через выход инструкции. Для простого хоста времени выполнения пешки это будет

действительно работа. В хост-приложениях, в которых сценарий является надстройкой, или в хост-приложениях, встроенных в устройство, сценарий обычно не может завершить работу хост-приложения.

• Несколько событий

Преимущества модели программирования, управляемой событиями, для создания реактивных программ становятся очевидными при наличии нескольких событий. На самом деле модель, управляемая событиями, полезна только в том случае, если у вас более одной точки входа; если ваш сценарий просто обрабатывает одно событие, он может также войти в цикл опроса для этого единственного события. Чем больше событий необходимо обработать, тем сложнее становится модель программирования, управляемая потоком. Приведенный ниже сценарий реализует простую программу «чата», использующую только два события: одно для отправки и одно для получения. Сценарий позволяет пользователям в сети (или, возможно, по другому соединению) обмениваться однострочными сообщениями.

Сценарий зависит от хост-приложения, чтобы обеспечить собственные и общедоступные функции для отправки и получения «датаграмм» и для ответа на введенные ключи. Как хост-приложение отправляет свои сообщения, по последовательной линии или с использованием TCP/IP, хост-приложение может решить само. Инструменты в стандартной рассылке пешек передают сообщения по сети TCP/IP и допускают «широковещательный» режим, так что более двух человек могут общаться друг с другом.



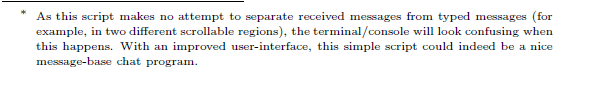
Основная часть приведенного выше скрипта обрабатывает полученные нажатия клавиш в строку и отправляет эту строку после просмотра клавиши ввода. Клавиша «Escape» завершает работу программы. Функция echo служит для визуальной обратной связи о том, что вводит пользователь: она строит строку с завершающим нулем из ключа и печатает ее.

Несмотря на свою простоту, этот сценарий имеет интересное свойство: в нем нет фиксированного или предписанного порядка, в котором сообщения должны быть отправлены или получены — нет схемы запрос-ответ, где каждый хост по очереди говорит и слушает. Новое сообщение может быть получено даже в то время, когда пользователь набирает свое собственное сообщение.∗

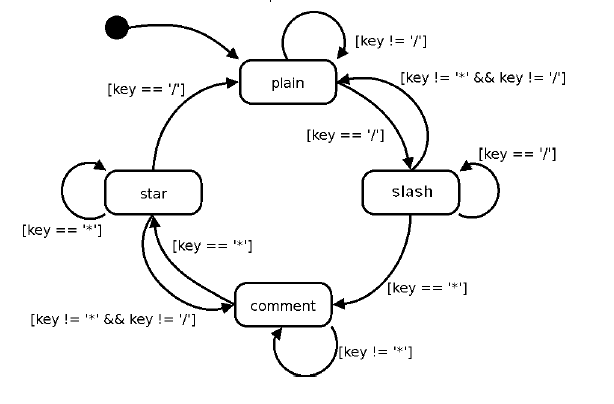
• Государственное программирование

В программе, основанной на событийной модели, события поступают индивидуально, и на них также индивидуально реагируют. Однако иногда событие является частью последовательного потока, который необходимо обрабатывать по порядку. Примерами являются протоколы передачи данных, например, по последовательной линии. Каждое событие может нести команду, фрагмент данных, который является частью большего файла, подтверждение или другие сигналы, которые участвуют в протоколе. Чтобы поток событий (и пакеты данных, которые они несут) имел смысл, программа, управляемая событиями, должна следовать точному протоколу установления связи.

Чтобы придерживаться протокола, программа, управляемая событиями, должна реагировать на каждое событие в соответствии с (недавней) историей событий, полученных ранее, и ответами на эти события. Другими словами, обработка одного события может установить «условие» или «среду» для обработки любого одного или нескольких последующих событий.

Простая, но весьма эффективная абстракция для построения реактивных систем, которые должны следовать (частично) последовательным протоколам, — это «автомат» или конечный автомат. Поскольку количество состояний обычно конечно, теория часто ссылается на такие автоматы, как автоматы с конечным состоянием или конечные автоматы. В автомате контекстом (или условием) события является его состояние. Поступившее событие может обрабатываться по-разному в зависимости от состояния автомата, и в ответ на событие автомат может перейти в другое состояние — это называется переходом. Переход, иначе говоря, как реакция автомата на событие в контексте его состояния. 

Автоматы очень распространены в программном обеспечении, а также в механических устройствах (вы можете увидеть жаккардовый ткацкий станок как раннюю конечную машину). Автоматы с конечным числом состояний являются детерминированными (т. е. предсказуемыми в поведении), а их относительно простая конструкция допускает простую реализацию из «диаграммы состояний».



На диаграмме состояний состояния обычно изображаются кружками или прямоугольниками со скругленными углами, а стрелки обозначают переходы. Поскольку переходы — это реакция автомата на события, стрелку также можно рассматривать как событие, «которое что-то делает». Предполагается, что событие/переход, не определенное в конкретном состоянии, не имеет никакого эффекта — оно просто игнорируется. Закрашенная точка представляет состояние входа, которое ваша программа (или хост-приложение) должна установить при запуске. Обычно на диаграмме состояний опускают все стрелки событий, которые возвращаются в одно и то же состояние, но для предыдущего рисунка я решил сделать реакцию на все события явной.

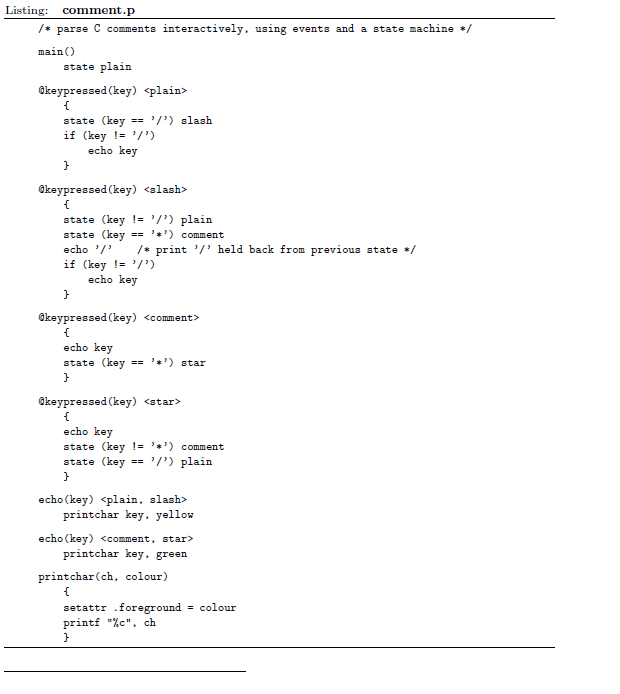
Эта диаграмма состояний предназначена для «разбора» комментариев, которые начинаются с «/\*» и заканчиваются на «\*/». Существуют состояния для простого текста и текста внутри комментария, а также два состояния для предварительного входа в комментарий или выхода из него. Автомат предназначен для интерактивного разбора комментариев из символов, которые пользователь набирает на клавиатуре. Поэтому единственными событиями, на которые реагирует автомат, являются нажатия клавиш. На самом деле есть только одно событие («нажатие клавиши»), а переключение состояний определяется параметром события: клавишей. pawn поддерживает автоматы и состояния непосредственно в языке. Каждой функции† может быть назначено одно или несколько состояний.



Pawn также поддерживает несколько автоматов, и каждое состояние является частью определенного автомата.

Следующий сценарий реализует предыдущую диаграмму состояний (в

одиночный, анонимный, автомат). Чтобы отличить обычный текст от комментариев, оба они выводятся разным цветом.



Функция main устанавливает начальное состояние в main и завершает работу; вся логика управляется событиями.

Когда ключ поступает в состояние, программа проверяет косую черту.

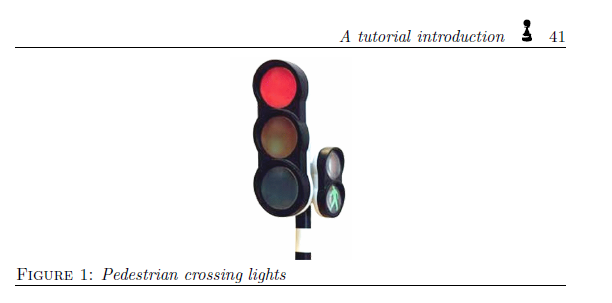
и условно печатает полученный ключ. Взаимодействие между состояниями «просто» и «слэш» демонстрирует сложность, типичную для автоматов: вы должны решить, как реагировать на событие, когда оно приходит, не имея возможности «заглянуть вперед» или отменить ответы на более ранние события. Это обычно имеет место в системах, управляемых событиями: вы не знаете, какое событие вы получите следующим, и когда вы его получите, и какой бы ни была ваша реакция на текущее событие, есть хороший шанс, что вы не сможете стереть его в будущем. событие и сделать вид, что его никогда не было.

В нашем конкретном случае, когда приходит косая черта, это может быть началом последовательности комментариев («/\*»), но это не обязательно так. Таким образом, мы не можем решить при приеме символа косой черты, каким цветом его печатать. Следовательно, мы удерживаем его. Однако в скрипте нет глобальной переменной, говорящей о задержании символа — фактически, кроме параметров функции, в этом скрипте вообще не объявлена никакая переменная. Информация о удерживаемом персонаже «спрятана» в состоянии автомата.

Как видно из сценария, изменения состояния могут быть условными. Условие является необязательным, и вы также можете использовать общую конструкцию if-else для изменения состояния.

Зависимость от состояния не зарезервирована для функций события. Другие функции также могут иметь объявления состояния, как демонстрирует функция echo. Когда функция будет иметь одинаковую реализацию для нескольких состояний, вам просто нужно написать одну реализацию и упомянуть все применимые состояния. Для функции echo существует две реализации для обработки четырех состояний.\*

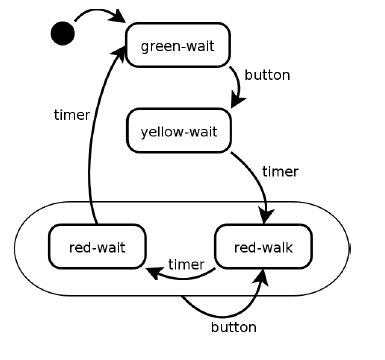
При этом автомат должен быть готов обрабатывать все события в любом состоянии. Как правило, автомат не контролирует ни то, какие события прибывают, ни то, когда они прибывают, поэтому отсутствие обработки события в каком-либо состоянии может привести к неправильным решениям. Поэтому часто бывает так, что некоторые события имеют смысл только в нескольких определенных состояниях и что они должны вызывать ошибку или процедуру «сброса» во всех остальных случаях. Функция для обработки события в таком состоянии «ошибка» может содержать множество имен состояний, если вы укажете их явно. Есть более короткий путь: не упоминая никакого имени в угловых скобках, функция соответствует всем состояниям, которые не имеют явной реализации где-либо еще. Так, например, вы можете использовать подпись «echo(key) <>» для любой из двух реализаций (но не для обеих).



Предопределен один анонимный автомат. Если программа содержит более одного автомата, остальные должны быть явно упомянуты как в классификаторе состояния функции, так и в инструкции состояния. Для этого добавьте имя автомата перед именем состояния и разделите имена автомата и состояния двоеточием. То есть «parser:slash» обозначает косую черту состояния автоматического синтаксического анализатора. Функция может быть частью только одного автомата; вы можете совместно использовать одну реализацию функции для нескольких состояний одного и того же автомата, но вы не можете использовать эту функцию для состояний разных автоматов.

• Функции входа и теория автоматов

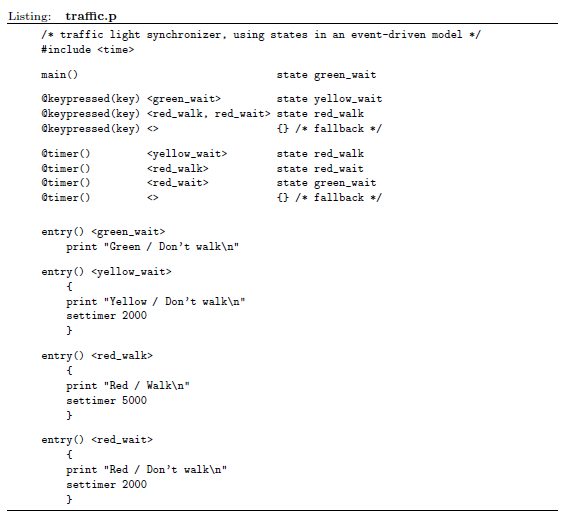
Конечные автоматы и основа «теории автоматов» происходят из механического проектирования и пневматических/электрических коммутационных схем (использующих реле, а не транзисторы). Типичными примерами являются монетоприемники, схемы управления светофорами и связи. В этих приложениях надежность и предсказуемость имеют первостепенное значение, и было обнаружено, что эти цели лучше всего достигаются, когда действия (выход) привязываются к состояниям, а не к событиям (вход). В этой схеме вход в состояние вызывает активность — события вызывают изменения состояния, но не выполняют других операций. В системе освещения пешеходного перехода освещение для транспортных средств и пешеходов должно быть синхронизировано. Технически существует шесть возможных комбинаций, но, очевидно, сочетание зеленого сигнала светофора и знак «ходьба» для пешеходов — это рецепт катастрофы. Мы также можем сразу отбросить комбинацию желтый/ходьба как слишком опасную. Таким образом, осталось обработать четыре комбинации. На рисунке ниже показана диаграмма состояния светофоров пешеходного перехода. Весь процесс активируется кнопкой и работает по таймеру.



Когда время состояния красный/ходьба истекает, состояние не может немедленно вернуться к зеленому/ожиданию, потому что пешеходам, которые в этот момент заняты переходом дороги, нужно некоторое время, чтобы расчистить дорогу — состояние красный/ожидание позволяет это сделать. В целях демонстрации этот пешеходный переход имеет дополнительную функциональность: когда пешеход нажимает кнопку, когда сигнал светофора уже горит красным, время, отведенное пешеходу на переход, увеличивается. Если состояние красное/ожидание и кнопка нажата, она снова переключается на красный/ходьба. Окружающая рамка вокруг состояний красный/ходьба и красный/жди обработки события кнопки — это просто условное обозначение: я мог бы также нарисовать две стрелки из любого состояния обратно в красный/ходьба. Однако исходный код скрипта (приведенный ниже) отражает такое же удобство записи.

В реализации на языке пешек функции событий теперь всегда имеют один оператор, который является либо изменением состояния, либо пустым оператором. События, не вызывающие изменение состояния, на диаграмме отсутствуют, но их необходимо обрабатывать в сценарии; следовательно, «возвратные» функции события ничего не делают. Вывод, в этом примере программы только сообщения, выводимые на консоль, выполняется в записи специальных функций. Запись функции может рассматриваться как основная для состояния: она неявно вызывается при входе в состояние, к которому она присоединена. Обратите внимание, что функция входа вызывается также при «переходе» в состояние, в котором автомат уже находится: когда состояние

is red\_walk вызов @keypressed устанавливает состояние в red\_walk (в котором он уже находится) и запускает функцию входа red\_walk — это повторный вход в состояние.



Этот пример программы имеет дополнительную зависимость от хост-приложения/среды: в дополнение к функции события «@keypressed» хост также должен предоставлять настраиваемое событие «@timer». Из-за функций синхронизации сценарий включает системный файл time.inc в верхней части сценария. Все функции событий с изменениями состояния находятся в верхней части скрипта. Функции расположены так, чтобы каждая занимала одну строку, чтобы предложить табличную структуру. Все изменения состояния в этом примере являются безусловными, но условные изменения состояния также могут использоваться с входными функциями. Нижняя часть — это функции событий.

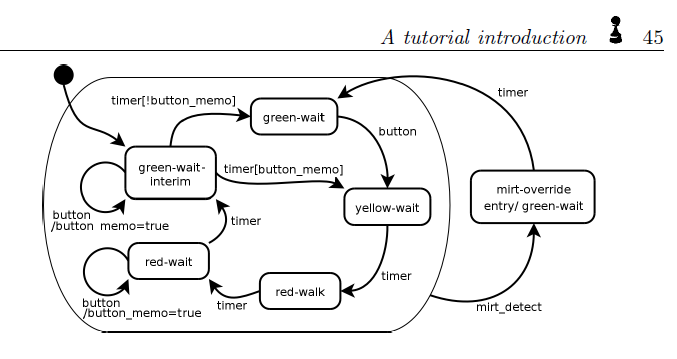
Существует два перехода в состояние red\_walk — или три, если вы рассматриваете влияние нескольких состояний на одну функцию события просто как удобство записи: из yellow\_wait и из комбинации red\_walk и red\_wait. Все эти переходы проходят через одну и ту же функцию входа, тем самым сокращая и упрощая код.

В теории автоматов автомат, который связывает действия с записями состояния, как, например, в этом примере со светофором для пешеходов, называется «автоматом Мура»; автомат, который связывает деятельность с (зависящими от состояния) событиями или переходами, называется «автоматом Мили». Интерактивный анализатор комментариев на стр. 39 представляет собой типичный автомат Мили. Эти два типа эквивалентны: автомат Мили может быть преобразован в автомат Мура и наоборот, хотя автомату Мура может потребоваться больше состояний для реализации того же поведения. На практике модели часто бывают смешанными, с общей конструкцией «автомата Мура» и несколькими «состояниями Мили», где это сохраняет состояние.

• Переменные состояния

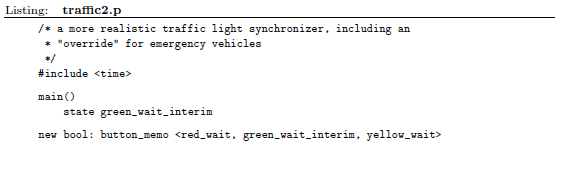
Модель светофора в предыдущем примере не очень реалистична (ее единственная цель — продемонстрировать некоторые свойства программирования состояний с помощью пешки). Первое, чего не хватает, — это степени справедливости: пешеходы не должны иметь возможность блокировать автомобильное движение на неопределенный срок. Автомобильное движение должно видеть зеленый свет на период некоторой минимальной продолжительности после того, как пешеходы получили свое время для перехода дороги. Во-вторых, многие светофоры имеют своего рода возможность дистанционного управления, так что аварийный транспорт (скорая помощь, пожарная машина и т. д.) может включить зеленый свет на своем пути. Хорошо известным примером такого дистанционного управления является система mirt (мобильный инфракрасный передатчик), но существуют и другие системы — например, в Нидерландах используется радиографическая система под названием vetag.

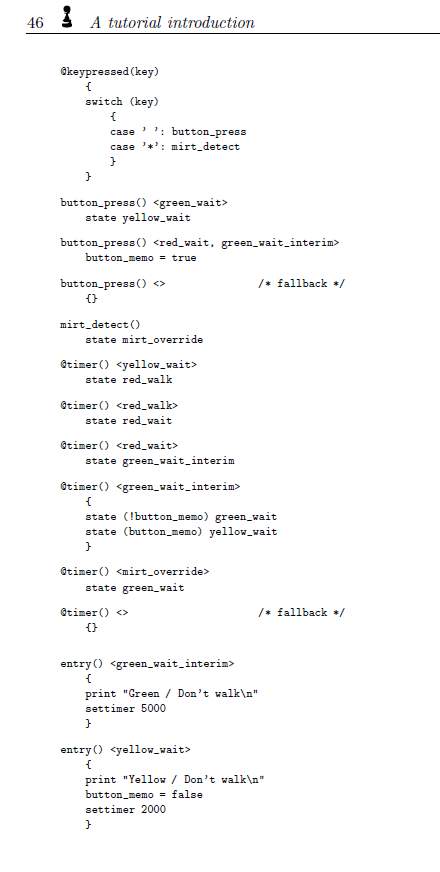
Новая диаграмма состояний для светофора пешеходного перехода имеет еще два состояния, но что более важно: она должна сохранять данные между событиями и обмениваться ими между состояниями. Когда пешеход нажимает кнопку в состоянии red\_wait, мы не хотим ни реагировать на кнопку немедленно (это было наше «правило справедливости»), ни игнорировать кнопку или «забыть». Другими словами, мы переходим в состояние green\_wait\_interim вне зависимости от нажатия кнопки, но запоминаем нажатие для решения, принятого в момент выхода из состояния green\_wait\_interim.

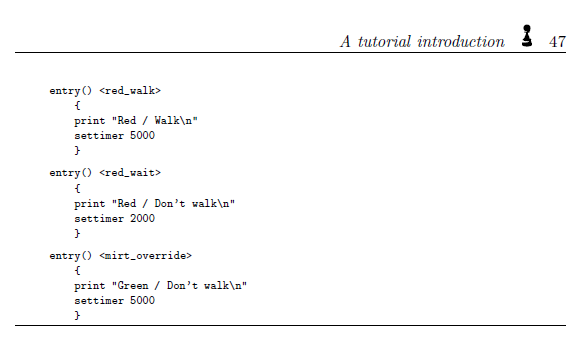


Автоматы превосходно моделируют поток управления в реактивных/интерактивных системах, но поток данных традиционно был их слабым местом. Чтобы понять почему, рассмотрим, что каждое событие обрабатывается функцией отдельно и что локальные переменные в этой функции исчезают, когда функция возвращается. Следовательно, локальные переменные нельзя использовать для передачи данных от одного события к другому. Глобальные переменные, хотя и обеспечивают обходной путь, имеют недостатки: глобальную область действия и «вечный» срок службы. Если переменная используется только в обработчиках событий одного состояния, желательно скрыть ее от других состояний, чтобы защитить от случайного изменения. Аналогично, сокращение срока жизни до состояния, в котором активна переменная, уменьшает объем памяти. «Переменные состояния» представляют собой сочетание переменной области действия и переменной продолжительности жизни, которые привязаны к ряду состояний, а не к функциям или модулям.

pawn обогащает стандартный конечный автомат (или автомат) переменными, объявленными с помощью классификатора состояний. Эти переменные доступны только из перечисленных состояний, и память, которую хранят эти переменные, может быть повторно использована для других целей, пока автомат находится в другом состоянии (отличном от перечисленных). Помимо классификатора состояния, объявление переменной состояния аналогично объявлению глобальной переменной. Объявление переменной button\_memo в следующем листинге иллюстрирует концепцию.







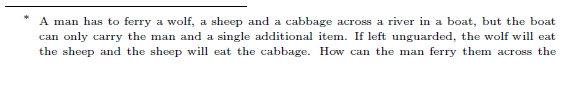
• Завершение государственного программирования

Обычное обозначение, используемое на диаграммах состояний, состоит в том, чтобы обозначать переходы стрелками, а состояния — кружками или прямоугольниками со скругленными углами. Кружок/прямоугольник со скругленными углами дополнительно также упоминает действия функции входа и события, которые обрабатываются внутри, не вызывая перехода. Стрелка перехода содержит название события (или псевдособытия), необязательное условие в квадратных скобках и необязательное действие после косой черты («/»).

Государства вездесущи, даже если мы не всегда признаем их таковыми.

Концепция конечных автоматов традиционно применялась в основном к программам, имитирующим механические устройства и программное обеспечение, реализующее протоколы связи. С появлением оконных систем, управляемых событиями, конечные автоматы теперь также появляются в дизайне графического интерфейса настольных программ. Состояния изобилуют веб-программами, потому что браузер и хост сценариев веб-сайта имеют лишь слабое звено. Тем не менее, конечный автомат в веб-приложениях обычно реализуется по принципу ad-hoc.

Государства также можно узнать в общих проблемах и загадках. В хорошо известной загадке о человеке, который должен переправить капусту, овцу и волка через реку\*, состояния очевидны — хитрость загадки в том, чтобы избежать запретных состояний.



Но теперь, когда мы повсюду обнаруживаем «состояния», мы должны быть осторожны, чтобы не переусердствовать. Например, во второй реализации светофора на пешеходном переходе (см. стр. 45) я использовал переменную (button\_memo) для хранения критерия для решения, принятого позднее. Альтернативной реализацией было бы добавление еще пары состояний для хранения ситуаций «красный-ожидание-и-кнопка-нажата» и «зеленый-ожидание-промежуточный-и-кнопка-нажата». Тогда больше не потребуется никаких переменных, но за счет более сложной диаграммы состояний и реализации. В общем, количество состояний должно быть небольшим.

Хотя автоматы обеспечивают хорошую абстракцию для моделирования реактивных и интерактивных систем, прийти к правильной диаграмме непросто, а иногда просто очень сложно. Слишком часто сначала прорисовывается «сценарий солнечного дня» состояний и событий, а все, что отклоняется от этого пути, затем импровизировано добавляется. Этот подход сопряжен с риском того, что некоторые комбинации событий и состояний будут забыты, и действительно, я столкнулся с двумя диаграммами парсера комментариев (например, на странице 39) от разных авторов книг/журналов, которые были ошибочны таким образом. Вместо этого я советую сосредоточиться на событиях и на ответах на отдельные события. Для каждого состояния следует учитывать каждое событие; не перенаправляйте события через резервную копию общего назначения слишком рьяно.

К сожалению, стало обычной практикой вводить теорию автоматов в приложения, для которых существуют лучшие решения. Одним из часто повторяющихся примеров является автомат, который накапливает стоимость серии монет или «вычисляет» остаток после деления на 3 двоичного числа. Эти приложения, возможно, имели смысл в механических/пневматических конструкциях, где «состояние» — это единственная память, которая есть у автомата, но в программном обеспечении использование переменных и арифметических операций — лучший выбор. Другим типичным примером является сопоставление слов или шаблонов с использованием конечного автомата: каждая следующая введенная буква переключается в новое состояние. Лексические сканеры, такие как те, которые компиляторы и интерпретаторы используют для интерпретации исходного кода, могут использовать такие конечные автоматы для фильтрации «зарезервированных слов». Однако при любом практическом наборе зарезервированных слов такие автоматы становятся громоздкими, и никто не будет их проектировать вручнуюКроме того, нет никаких причин, по которым лексический сканер не может заглянуть в текст вперед или вернуться к ранее установленной отметке — что является одним из критериев выбора реализации состояния в первую очередь и, наконец, таких решений, как попытка Поиски, вероятно, проще спроектировать и реализовать, но они, по крайней мере, такие же быстрые.



• Проверка программы

Должен ли компилятор/интерпретатор не отлавливать все ошибки? Этот риторический вопрос имеет как техническую, так и философскую стороны. Я воздержусь от всех нетехнических аспектов и лишь упомяну, что на практике существует компромисс между «выразительностью» компьютерного языка и «принудительной правильностью» (или «доказуемой правильностью») программ на этом языке. Делать язык очень «строгим» — не решение, если нужно выполнить работу, превышающую размер игрушечной программы. Слишком строгий язык заставляет программиста бороться с языком, в то время как «проблема, которую нужно решить», должна быть настоящей борьбой, а язык должен быть простым средством выражения решения.

Цель языка пешек — предоставить разработчику неформальный и удобный в использовании механизм для проверки того, ведет ли программа так, как предполагалось. Этот механизм называется «утверждениями», и, хотя концепция утверждений предшествует идее «проектирование по контракту», его легче всего объяснить с помощью методологии «дизайн по контракту».

Парадигма «проектирование по контракту» предлагает альтернативный подход к работе с ошибочными условиями. Предпосылка состоит в том, что программист знает стоящую перед ним задачу, условия, в которых должно работать программное обеспечение, и окружающую среду. В такой среде каждая функция задает определенные условия в форме утверждений, которые должны выполняться, прежде чем клиент сможет выполнить функцию. Кроме того, функция также может указывать любые условия, которые остаются истинными после завершения ее работы. Это «контракт» функции.

Название «проектирование по контракту» было придумано Бертраном Мейером, и его принципы восходят к логике предикатов и алгоритмическому анализу.

⋄ Предварительные условия задают допустимые значения входных параметров и атрибутов среды;

⋄ Постусловия определяют выход и (возможно, измененное) окружение;

⋄ Инварианты указывают условия, которые должны выполняться в ключевых точках функции, независимо от пути прохождения функции.

Например, функция, вычисляющая квадратный корень из числа, может указать, что ее входной параметр не должен быть отрицательным. Это предварительное условие. Он также может указать, что его выходной сигнал, возведенный в квадрат, представляет собой входное значение ± 0,01%. Это постусловие; он проверяет правильность работы подпрограммы. Удобный способ вычисления квадратного корня — это «деление пополам». На каждой итерации этот алгоритм дает по крайней мере один дополнительный бит (двоичную цифру) точности. Это инвариант (хотя это может быть инвариант, который трудно проверить).

Предусловия, постусловия и инварианты похожи в том смысле, что все они состоят из теста, а неудачный тест указывает на ошибку в реализации.

В результате вы можете реализовать предусловия, постусловия

и инварианты с одной конструкцией: «утверждение». Для предварительных условий пишите утверждения в самом начале подпрограммы; для инвариантов напишите утверждение, в котором должен выполняться инвариант; для пост-условий напишите утверждение перед каждым оператором «return» или в конце функции.

В pawn инструкция называется assert; это простое утверждение, содержащее тест. Если результат теста «истина», ничего не происходит. Если

Если результат равен «ложь», инструкция утверждения завершает программу с сообщением, содержащим детали утверждения, которое не удалось.

Утверждения — это проверки, которые никогда не должны завершаться ошибкой. Подлинные ошибки, такие как ошибки пользовательского ввода, следует обрабатывать с помощью явных тестов в программе, а не с помощью утверждений. Как правило, выражения, содержащиеся в утверждениях, не должны иметь побочных эффектов: утверждение никогда не должно содержать код, необходимый вашему приложению для корректной работы.

Однако это приводит к тому, что утверждения никогда не срабатывают в программе, свободной от ошибок: они просто делают код толще и медленнее без каких-либо заметных для пользователя преимуществ. Однако это не так уж и плохо. Дополнительным свойством утверждений является то, что вы можете создавать исходный код без утверждений, просто используя флаг или параметр парсера пешки. Идея состоит в том, что вы включаете утверждения во время разработки и создаете «розничную версию» кода без утверждений. Это лучший подход, чем удаление утверждений, потому что все утверждения автоматически «возвращаются» при перекомпиляции программы — например. для поддержания. Если во время сопровождения или даже во время первоначальной разработки вы обнаружите ошибку, которая не была перехвачена утверждением, прежде чем исправлять ошибку, вы должны подумать о том, как утверждение могло перехватить эту ошибку. Затем добавьте это утверждение и проверьте, действительно ли оно улавливает ошибку, прежде чем исправлять ошибку. Делая это, код постепенно станет более прочным и надежным.

• Комментарии к документации

Когда программы становятся больше, документирование программы и функций становится жизненно важным для ее обслуживания, особенно при работе в команде. Языковые инструменты pawn имеют некоторые функции, которые помогут вам документировать код в комментариях. Документирование программы или библиотеки в ее комментариях имеет несколько преимуществ, например: документацию легче поддерживать в актуальном состоянии вместе с программой, она эффективна в том смысле, что программные комментарии теперь удваиваются как документация, а синтаксический анализатор помогает вашим усилиям по документированию в создание описаний синтаксиса и перекрестных ссылок.

Каждый комментарий, который начинается с трех косых черт («///»), за которыми следует пробел, или который начинается с косой черты и двух звездочек («/\*\*»), за которыми следует пробел, является специальным комментарием для документации. Компилятор pawn извлекает комментарии к документации и при необходимости записывает их в файл «отчета». См. документацию по приложению или приложение B, как включить генерацию отчета.

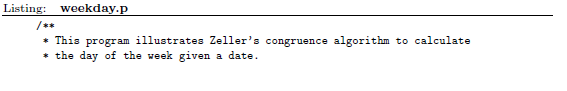
Кроме того, комментарии, начинающиеся с «/\*\*», должны заканчиваться «\*/». Однострочные комментарии документации («///») закрываются в конце строки.

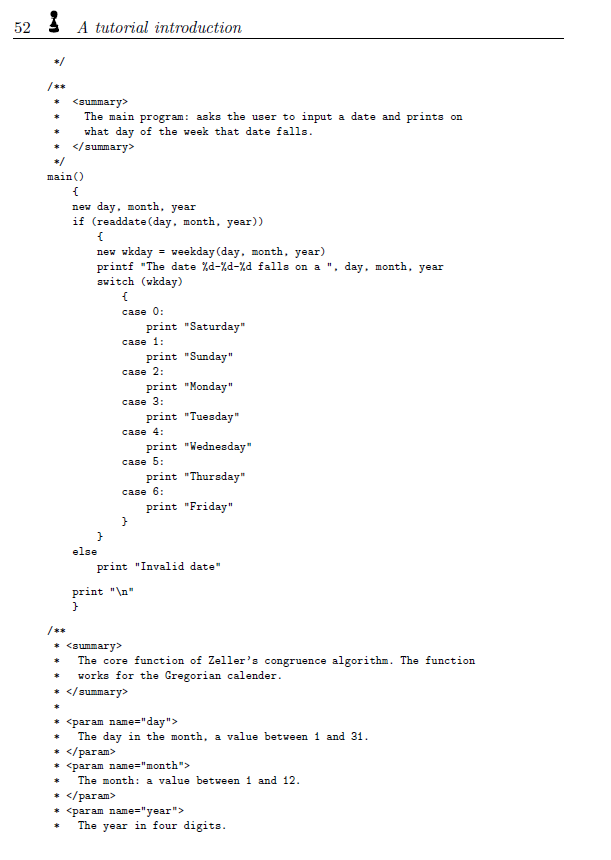
Файл отчета представляет собой XML-файл, который впоследствии можно преобразовать в

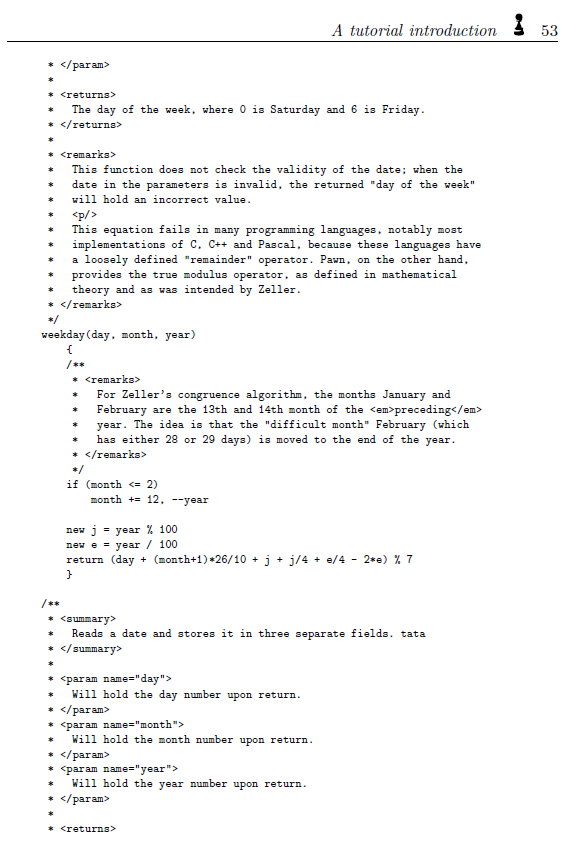
Документация в формате HTML с помощью таблицы стилей XSL/XSLT или с помощью других инструментов для создания печатной документации. Синтаксис файла отчета совместим с таковым в продуктах для разработчиков «.Net», за исключением того, что компилятор Pawn хранит в отчете больше информации, чем просто извлеченные строки документации. Файл отчета содержит ссылку на файл «smalldoc. xsl»

Пример ниже иллюстрирует комментарии к документации в простом скрипте с несколькими функциями. Вы можете писать комментарии к документации для функции над ее объявлением или в ее теле. Все комментарии к документации, которые появляются перед окончанием функции, относятся к функции.

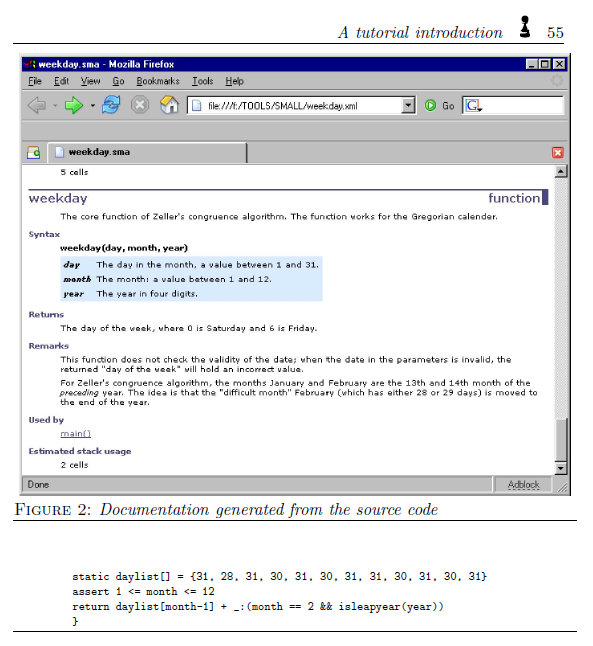
Вы также можете добавлять комментарии к глобальным переменным и глобальным константам — эти комментарии должны появляться над объявлением переменной или константы. На рис. 2 показана часть вывода этого (довольно длинного) примера. Стиль вывода настраивается в таблице каскадных стилей (CSS-файле), связанной с файлом преобразования XSLT.











Формат XML-файла, созданного продуктами разработчика «.Net», задокументирован в документации Microsoft. Парсер пешки создает минимальное описание каждой функции, глобальной переменной или константы, которые используются в проекте, независимо от того, использовали ли вы комментарии к документации для этой функции/переменной/константы. Парсер также генерирует несколько собственных тегов:

атрибут

Атрибуты для функции, такие как «родной» или «стандартный».

автомат

Автомат, которому принадлежит функция (если есть).

зависимость

Имена символов (другие функции, глобальные переменные и/или глобальные константы), которые требуются функции. При желании из зависимостей можно построить дерево вызовов.

параметр

Параметры функции.

Когда вы добавляете описание параметра

в комментарии к документации это описание объединено

с автоматически сгенерированным содержимым для параметра.

параминфо

Теги и массив или справочная информация о параметре.

реферер

Все функции, которые относятся к этому символу; т. е. все функции, которые используют или вызывают эту переменную/функцию. Этой информации достаточно, чтобы служить «перекрестной ссылкой» — дерево «рефереров» является обратным деревом «зависимостей».

stacksize Предполагаемое количество ячеек, которое функция выделит в стеке и куче. Эта оценка использования стека не включает

требования к стеку любых функций, которые «вызываются» из функции, к которой применяется документация. Например, функция readdate задокументирована как занимающая 6 ячеек в стеке, но она также вызывает daysinmonth, который занимает 4 дополнительные ячейки, и, в свою очередь, вызывает isleapyear. Чтобы рассчитать общие требования к стеку для функции readdate, следует рассмотреть дерево вызовов.

В дополнение к локальным переменным и параметрам функции,

компилятор также использует стек для хранения промежуточных результатов в

сложные выражения. Объем стека, необходимый для этих промежуточных результатов, также не включен в этот отчет. В общем случае требуемые накладные расходы на промежуточные результаты не являются кумулятивными (по всем функциям), поэтому было бы неточно добавлять «запас прочности» для каждой функции. Для программы в целом настоятельно рекомендуется запас прочности. См. приложение B (стр. 169) для параметра -v, который может сообщить вам максимальную оценку использования стека на основе дерева вызовов.

название тэга

Тег константы, переменной, результата функции или функции

параметр(ы).

переход

Переходы, которые вызывает функция, и их условия — см. раздел автоматов на стр. 37.

Весь текст комментариев к документации также копируется в каждую функцию, переменную или константу, к которым он прикреплен. Текст комментария к документации копируется без дальнейшей обработки — за одним исключением, см. ниже. Поскольку остальная часть файла отчета находится в формате XML, а наиболее подходящий способ преобразования XML в онлайн-документацию — через процессор XSLT (например, современный браузер), вы можете выполнить любое форматирование комментариев к документации, используя HTML-теги. Обратите внимание, что вам часто потребуется

явно закрывать любые теги HTML; стандарт HTML не требует этого, но процессоры XML/XSLT обычно требуют этого. Инструментарий pawn поставляется с примером файла XSLT (с соответствующей таблицей стилей), который поддерживает следующие теги XML/HTML:

<code> </code> Отформатированный исходный код моноширинным шрифтом; хотя «&», «<» и «>» должны быть набраны как «&amp;», «&lt;» и «&rt;» соответственно.

<example> </example> Текст установлен под темой «Пример».

<param name="..."> </param> Описание параметра с именем параметра, появляющимся внутри открывающего тега (опция «name=»), и описанием параметра, следующим за ним.

<paramref name="..." /> Ссылка на параметр с именем параметра, появляющимся внутри открывающего тега (опция «name=»).

<remarks> </remarks> Текст, установленный под темой «Замечания».

<returns> </returns> Текст установлен в теме «Возвраты».

<seealso> </seealso> Текст, установленный под темой «См. также».

<summary> </summary> Текст устанавливается сразу под заголовком символа.

<section> </section> Устанавливает текст в заголовке. Это следует использовать только в документации, которая не связана с функцией или переменной.

<subsection> </subsection> Устанавливает текст в подзаголовок. Это следует использовать только в документации, которая не связана с функцией или переменной.

Следующие дополнительные теги HTML поддерживаются для форматирования текста общего назначения внутри любого из указанных выше разделов:

<c> </c> Текст набран моноширинным шрифтом.

<em> </em> Текст выделяется выделением, обычно курсивом.

<p> </p> Текст в новом абзаце. Вместо обтекания <p> и </p> вокруг каждого абзаца вставка <p/> в качестве разделителя между двумя абзацами дает тот же эффект.

<para> </para> Альтернатива <p> </p>

<ul> </ul> неупорядоченный (маркированный) список.

<ol> </ol> Упорядоченный (нумерованный) список.

<li> </li> Элемент в упорядоченном или неупорядоченном списке.

Как уже говорилось, есть одно исключение при обработке комментариев к документации: если ваш комментарий к документации содержит тег <param ...> (и соответствующий </param>), синтаксический анализатор ищет параметр и объединяет ваше описание с параметр с автоматически сгенерированным содержимым.

• Предупреждения и ошибки

Большим препятствием, которое я преодолел, является то, как на самом деле скомпилировать

фрагменты кода, представленные в этой главе. Причина в том, что процедура зависит от используемой вами системы: в некоторых приложениях есть командная кнопка или пункт меню «Создать» или «Скомпилировать сценарий», в то время как в других средах вам нужно ввести команду, например «pawncc myscript». в командной строке. Если вы используете стандартный набор инструментов pawn, вы найдете инструкции по использованию компилятора и времени выполнения в сопроводительном буклете «The pawn booklet — Implementor’s Guide». Если вы используете Microsoft Windows, может оказаться наиболее удобным использовать Quincy IDE, которая поставляется с Pawn для написания, запуска и отладки скриптов.

Независимо от различий в запуске компиляции, явление, возникающее в результате запуска компиляции, вероятно, будет очень похожим во всех системах:

⋄ либо компиляция завершается успешно, и создается исполняемая программа, которая может запускаться или не запускаться автоматически после компиляции;

⋄ или компиляция выдает список предупреждений и сообщений об ошибках.

Ошибки случаются, и синтаксический анализатор пешек пытается поймать как можно больше из них. Когда вы проверяете код, на который жалуется парсер пешки, иногда вам может быть довольно сложно понять, почему код ошибочен (или подозрительн). Следующие советы могут помочь:

⋄ Каждый номер ошибки или предупреждения пронумерован. Вы можете найти сообщение об ошибке с этим номером в приложении А вместе с кратким описанием того, что на самом деле означает это сообщение.

⋄ Если синтаксический анализатор выдает список ошибок, первая ошибка в этом списке является истинной ошибкой, но диагностические сообщения под ней могут вообще не быть ошибками.

После того, как синтаксический анализатор закладок увидит ошибку, он попытается обойти ее и завершить компиляцию. Однако спотыкание об ошибке могло сбить с толку синтаксический анализатор пешки, так что последующие допустимые утверждения были неправильно истолкованы и также сообщались как ошибки.

Если сомневаетесь, исправьте первую ошибку и перекомпилируйте.

⋄ Парсер Pawn проверяет только синтаксис (орфографию/грамматику), а не семантику (то есть «смысл») кода. Когда он обнаруживает код, который не соответствует синтаксическим правилам, на самом деле могут быть разные способы изменить код, чтобы он стал «правильным» в синтаксическом смысле этого слова, даже несмотря на то, что многие из этих «исправлений» привели бы к ошибкам. к бессмысленному коду. Однако в результате синтаксический анализатор пешки может испытывать трудности с точным обнаружением ошибки: он не знает, что вы хотели написать. Следовательно, анализатор часто выводит два номера строки, и ошибка находится где-то в диапазоне (между номерами строк).

⋄ Помните, что программа, в которой нет синтаксических ошибок (парсер пешки принимает ее без сообщений об ошибках и предупреждениях), все же может иметь семантические и логические ошибки, которые парсер пешки не может отловить. Инструкция assert (стр. 113) предназначена для того, чтобы помочь вам отловить эти ошибки «времени выполнения».

• В заключение

Если вы знаете язык программирования C, вы увидите много знакомых вам понятий и несколько новых. Если вы не знаете C, темп этого введения, вероятно, был довольно высоким. Независимо от того, являетесь ли вы новичком в C или имеете опыт работы с C, я рекомендую вам внимательно прочитать следующие страницы. Кстати, если вы знаете C или C-подобный язык, вы можете сначала обратиться к главе «Подводные камни» (стр. 134).

Этот буклет пытается быть одновременно и неформальным введением, и (более формальной) спецификацией языка, но, возможно, ни в одном из них не преуспел. Поскольку это также стандартная книга о пешке\*, основное внимание в этой брошюре уделяется точности и полноте, а не простоте понимания.



Двойственный характер этой брошюры проявляется в том порядке, в котором в ней представлены темы. Сначала рассматриваются более крупные концептуальные части языка, переменные и функции. Операторы, операторы и общие правила синтаксиса следуют позже — не то чтобы они менее важны, но их легче выучить, найти или принять как должное.

65  
Данные и декларации

пешка является бестиповым языком. Все элементы данных имеют тип «ячейка», и ячейка может содержать целое число. Размер ячейки (в байтах) зависит от системы — обычно ячейка имеет размер 32 бита.

Ключевое слово new объявляет новую переменную. Для специальных объявлений ключевое слово new заменяется на static, public или stock (см. ниже). Простое объявление переменной создает переменную, которая занимает одну «ячейку» памяти данных. Если она не инициализирована явно, значение новой переменной равно нулю.

Объявление переменной может произойти:

⋄ в любой позиции, где утверждение было бы допустимым — локальные переменные;

⋄ в любой позиции, где допустимо объявление функции (объявления собственных функций) или реализация функции — глобальные переменные;

⋄ в первом выражении инструкции цикла for — также локальные переменные.

Местные объявления

Локальное объявление появляется внутри составного оператора. Доступ к локальной переменной возможен только из составного оператора, и из вложенных составных операторов. Декларация в первом выражение инструкции цикла for также является локальным объявлением.

Глобальные объявления

Глобальное объявление появляется вне функции и глобальной переменной доступна любая функция. Объекты глобальных данных могут быть инициализированы только константными выражениями.

• Объявления переменных состояния

Переменная состояния — это глобальная переменная с добавленным в конце классификатором состояния. Область действия и срок действия переменной ограничены состояниями, перечисленными в классификаторе. Спецификаторы резервного состояния не разрешены для переменных состояния.

Переменные состояния не могут быть инициализированы. В отличие от обычных переменных (которые после объявления равны нулю, если они не инициализированы явно), переменные состояния сохраняют неопределенное значение после объявления и после первого входа в состояние в своем классификаторе. Как правило, для правильной инициализации переменной состояния используются функции входа в состояние.

• Статические локальные объявления

Локальная переменная уничтожается, когда выполнение покидает составной блок, в котором она была создана. Локальные переменные в функции существуют только во время выполнения этой функции. Каждый новый запуск функции создает и инициализирует новые локальные переменные. Когда локальная переменная объявляется с ключевым словом static, а не new, переменная остается в существовании после завершения функции. Это означает, что статические локальные переменные обеспечивают частное постоянное хранилище, доступное только из одной функции (или составного блока). Как и глобальные переменные, статические локальные переменные могут быть инициализированы только константными выражениями.

• Статические глобальные объявления

Статическая глобальная переменная ведет себя так же, как обычная глобальная переменная, за исключением того, что ее область действия ограничена файлом, в котором находится объявление. Чтобы объявить глобальную переменную статической, замените ключевое слово new на static.

• Складские декларации

Глобальная переменная может быть объявлена как «запас». Стандартное объявление — это объявление, которое синтаксический анализатор может удалить или проигнорировать, если окажется, что переменная не используется в программе.

Переменные запаса полезны в сочетании с функциями запаса. Публичные переменные также могут быть объявлены как «общедоступные» — объявление общедоступных переменных как «общедоступных» позволяет вам объявить все общедоступные переменные, которые хост-приложение предоставляет во включаемом файле, причем только те переменные, которые скрипт фактически использует, завершаются. файл P-кода.

• Публичные заявления

Глобальные «простые» переменные (без массивов) могут быть объявлены «общедоступными» двумя способами:

⋄ объявить переменную, используя ключевое слово public вместо new;

⋄ начинайте имя переменной с символа «@».

Публичные переменные ведут себя как глобальные переменные, за исключением того, что хост-программа также может читать и записывать общедоступные переменные. К (нормальной) глобальной переменной могут обращаться только функции в вашем скрипте — хост-программа не знает о них. Таким образом, хост-программа может потребовать, чтобы вы объявили переменную с определенным именем как «общедоступную» для особых целей, таких как номер самой последней ошибки или общее состояние программы.

• Постоянные переменные

Иногда удобно иметь возможность создать переменную, которая инициализируется один раз и не может быть изменена. Такая переменная ведет себя так же, как символическая константа, но все же является переменной.

Чтобы объявить переменную-константу, вставьте ключевое слово const между ключевым словом, с которого начинается объявление переменной (новая, статическая, публичная или стандартная), и именем переменной.

Examples:



Три типичных случая, когда можно использовать постоянную переменную:

⋄ Создать константу «массив»; символьные константы не могут быть проиндексированы.

⋄ Для общедоступной переменной, которая должна быть установлена хост-приложением и только хост-приложением. См. предыдущий раздел для общедоступных переменных.

⋄ Особым случаем является пометка массивов аргументов функций как констант. Аргументы массива всегда передаются по ссылке, объявляя их константными для защиты от непреднамеренных изменений. Обратитесь к странице 72 для примера аргументов константной функции.

• Массивы (одно измерение)

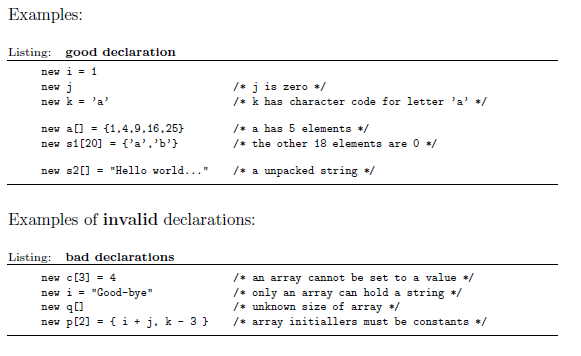
Синтаксис name[constant] объявляет имя массивом «константных» элементов, где каждый элемент представляет собой одну ячейку. Имя является заполнителем имени идентификатора по вашему выбору, а константа представляет собой положительное ненулевое значение; константа может отсутствовать. Если в скобках нет значения, количество элементов устанавливается равным количеству инициализаторов — см. пример ниже.

Диапазон индексов массива «отсчитывается от нуля», что означает, что первый элемент имеет имя [0], а последний элемент имеет имя [константа-1].

• Инициализация

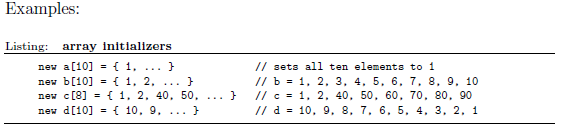
Объекты данных могут быть инициализированы при их объявлении. Инициал объекта глобальных данных должен быть константой. Массивы, глобальные или локальные, также должны быть инициализированы константами.

Неинициализированные данные по умолчанию равны нулю.



• Прогрессивные инициализаторы для массивов

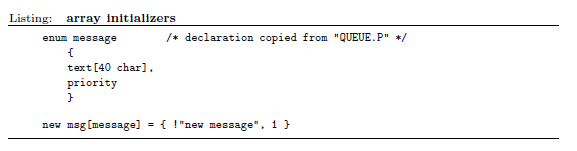
Оператор многоточия продолжает последовательность констант инициализации для массива на основе двух последних инициализированных элементов. Оператор многоточия (три точки или «...») инициализирует массив до объявленного размера.



• инициализация массива и перечисление

Размер массива может быть установлен с помощью константы, представляющей перечисление: примером этого является пример программы «приоритетная очередь» на стр. 19. Когда отдельные поля перечисления имеют размер, связанные элементы массива интерпретируются как подмассивы. , по случаю. Пример такого поведения см. в программе калькулятора rpn на стр. 28.

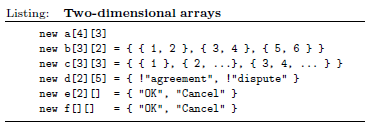
Синтаксис подмассива также применяется к инициализации «нумерованного» массива. Снова обратимся к примеру программы «приоритетная очередь», чтобы инициализировать массив «сообщений» с фиксированными значениями, синтаксис следующий:



Инициализатор состоит из строки (литерального массива) и значения; они входят в поля «текст» и «приоритет» соответственно.

• Многомерные массивы

Многомерные массивы — это массивы, содержащие ссылки на подмассивы. То есть двумерный массив — это «массив одномерных массивов».\* Ниже приведены несколько примеров объявлений двумерных массивов.





Как показывают последние два объявления (переменная «e» и «f»), конечное измерение массива может иметь неуказанную длину, и в этом случае длина каждого подмассива определяется из соответствующего инициализатора. Каждый подмассив может иметь разный размер; в этом конкретном примере «e[1][5]» содержит букву «l» из слова «Отмена», но «e[0][5]» недопустимо, поскольку длина подмассива «e[ 0]» состоит всего из трех ячеек (содержащих буквы «О», «К» и завершающий нуль).

Разница между объявлениями для массивов «e» и «f» заключается в том, что мы позволяем компилятору подсчитывать количество инициализаторов для основного измерения — «sizeof f» равно 2, как и «sizeof e» (см. размер оператора).

• Массивы и оператор sizeof

Оператор sizeof возвращает размер переменной в «элементах». Для

простая (не составная) переменная, результат sizeof всегда равен 1, потому что элемент является ячейкой для простой переменной.

Массив с одним измерением содержит несколько ячеек, и оператор sizeof возвращает это число. Таким образом, приведенный ниже фрагмент кода выведет на дисплей цифру «5», потому что массив «msg» содержит четыре символа (каждый в одной ячейке) плюс завершающий ноль:



В многомерных массивах оператор sizeof может возвращать количество элементов в каждом измерении. Для последнего (второстепенного) измерения элемент снова будет ячейкой, но для основного измерения (размеров) элемент будет подмассивом. В следующем фрагменте кода обратите внимание, что синтаксис sizeof matrix относится к старшему измерению двумерного массива, а синтаксис sizeof matrix[] относится к младшему измерению массива. Значения, которые печатает этот фрагмент, равны 3 и 2 (соответственно для основного и дополнительного измерений):



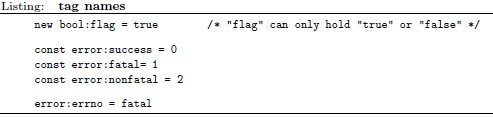
Применение оператора sizeof к многомерным объектам удобно, когда он используется в качестве значения по умолчанию для аргументов функции

• Имена тегов

Тег — это метка, обозначающая цель или значение переменной, константы или результата функции. Теги необязательны, их единственная цель - обеспечить более строгую проверку ошибок операндов в выражениях, аргументов функций и индексов массива во время компиляции.

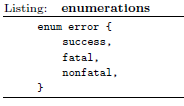
Тег состоит из имени символа, за которым следует двоеточие; он имеет тот же синтаксис, что и метка. Тег предшествует имени символа переменной, константы или функции. В задании может быть отмечена только правая часть знака «=».

Примеры допустимых теговых переменных и определений констант:

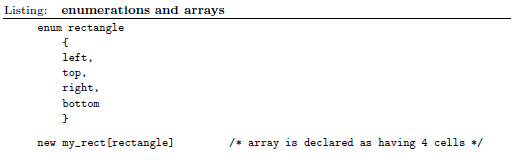


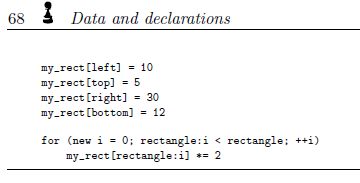
Последовательность констант Success, Fatal и NonFatal может быть более удобно объявлена с помощью инструкции перечисления, как показано ниже.

Приведенная ниже инструкция перечисления создает четыре константы: успех, фатальный, нефатальный и ошибка, все с тегом error.



Типичное использование «помеченных» перечислений в сочетании с массивами. Если каждое поле массива имеет определенную цель, вы можете использовать перечисление с тегами, чтобы объявить размер массива и добавить проверку тегов к использованию массива за один шаг:





После объявления «my\_rect» выше вы можете получить доступ ко второму полю my\_rect с помощью «my\_rect[top]», но произнесение «my\_rect[1]» даст диагностику синтаксического анализатора (предупреждение или сообщение об ошибке). Переопределение тега (или преобразование тега) настраивает функцию, константу или переменную на желаемое имя тега. Цикл for в последних двух строках предыдущего примера изображает это: переменная цикла i — это обычная ячейка без тегов, и ее необходимо преобразовать в прямоугольник тега, прежде чем использовать ее в качестве индекса в массиве my\_rect. Обратите внимание, что конструкция enum создала как константу, так и тег с именем «прямоугольник».

Представленные до сих пор имена тегов начинались со строчной буквы; это «слабые» теги. Имена тегов, начинающиеся с заглавной буквы, являются «сильными» тегами.

Разница между слабыми и сильными тегами заключается в том, что слабые теги могут в некоторых случаях неявно отбрасываться синтаксическим анализатором пешки, так что выражение со слабым тегом становится выражением без тега. Механизм проверки тегов проверяет следующие ситуации:

⋄ Когда выражения с обеих сторон бинарного оператора имеют разные теги или когда одно из выражений помечено, а другое нет, компилятор выдает диагностику «несоответствие тегов». В этой ситуации нет разницы между слабыми и сильными тегами.

⋄ Для оператора присваивания существует особый случай: компилятор выдает диагностику, если переменная в левой части оператора присваивания имеет тег, а выражение в правой части либо имеет другой тег, либо не имеет тега. Однако, если переменная слева от оператора присваивания не имеет тега, она принимает выражение (справа) со слабым тегом. Другими словами, слабый тег отбрасывается в присваивании, когда lvalue не имеет тега.

⋄ Передача аргументов функциям осуществляется по правилу присваивания. Компилятор выдает диагностику, когда формальный параметр (в определении функции) имеет тег, а фактический параметр (в вызове функции) либо не имеет тега, либо имеет другой тег. Однако, если формальный параметр не имеет тега, он также принимает параметр с любым слабым тегом.

⋄ Массив может указывать тег для каждого измерения, см. пример «my\_rect» выше. Индексы массива проверки тегов следуют правилу проверки тегов бинарных операторов: нет разницы между слабыми и сильными тегами.

Функции

Объявление функции указывает имя функции и в круглых скобках ее формальные параметры. Функция также может возвращать значение. Объявление функции должно отображаться на глобальном уровне (т. е. вне любых других функций) и доступно глобально.

Если точка с запятой следует за объявлением функции (а не за оператором), это объявление обозначает предварительное объявление функции.

The return statement sets the function result. For example, function sum

(see below) has as its result the value of both its arguments added together.

The return expression is optional for a function, but one cannot use the

value of a function that does not return a value.



Аргументы функции являются (неявно объявленными) локальными переменными для этой функции. Вызов функции определяет значения аргументов.

Другой пример полного определения функции високосного года (которая возвращает true для високосного года и false для невисокосного года):

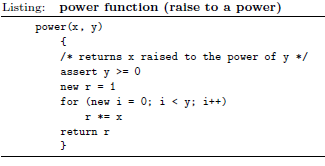


Логические и арифметические операторы, используемые в примере с високосным годом, таковы:

на страницах 109 и 105 соответственно.

Обычно функция содержит объявления локальных переменных и состоит из составного оператора. В следующем примере обратите внимание на оператор assert для

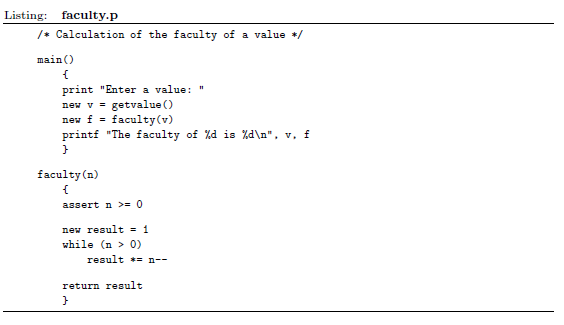
защищаться от отрицательных значений экспоненты.



Функция может содержать несколько операторов возврата — обычно это делается для быстрого выхода из функции при ошибке параметра или когда выясняется, что функции нечего делать. Если функция возвращает массив, все операторы возврата должны указывать массив с одинаковым размером и размерностями.

Аргументы функции (вызов по значению или вызов по ссылке)

Функция «способность» в следующей программе имеет один параметр, который она использует в цикле для вычисления способности этого числа. Что заслуживает внимания, так это то, что функция изменяет свой аргумент.

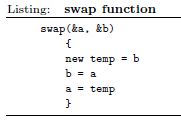


Какое бы (положительное) значение не имело значение «n» на входе в цикл while в

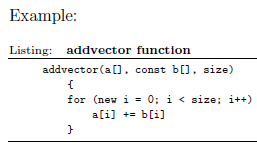
функциональная способность, «n» будет равно нулю в конце цикла. В случае функции факультета параметр передается «по значению», поэтому изменение «n» является локальным для функции факультета. Другими словами, функция main передает «v» в качестве входных данных функции факультета, но после возврата факультета «v» по-прежнему имеет то же значение, что и до вызова функции.

Аргументы, занимающие одну ячейку, можно передавать по значению или по ссылке. По умолчанию используется «передача по значению». Чтобы создать аргумент функции, который передается по ссылке, добавьте к имени аргумента префикс &.

Пример:



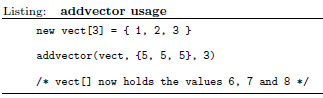
Чтобы передать массив функции, добавьте пару квадратных скобок к имени аргумента. Вы можете дополнительно указать размер массива; это улучшает проверку ошибок синтаксического анализатора.



Массивы всегда передаются по ссылке. В качестве примечания: массив b в приведенном выше примере не изменяется в теле функции. Аргумент функции был объявлен как const, чтобы сделать это явным. Помимо улучшения проверки ошибок, это также позволяет синтаксическому анализатору пешек генерировать более эффективный код.

Чтобы передать массив литералов в функцию, используйте тот же синтаксис, что и для инициализаторов массива: литеральная строка или последовательность индексов массива, заключенная в фигурные скобки (см. стр. 99; многоточие для прогрессивных инициализаторов использовать нельзя). Литеральные массивы могут иметь только одно измерение.

Следующий фрагмент вызывает addvector для добавления пяти к каждому элементу массива «vect»:

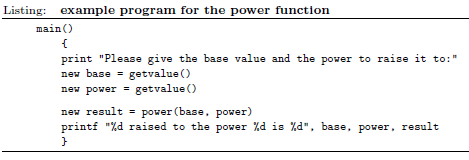


Вызов функции printf со строкой «Hello world\n» в первой вездесущей программе — еще один пример передачи литерального массива в функцию.

• Функции вызова

При вставке имени функции с ее параметрами в оператор или выражение функция будет выполняться в этом операторе/выражении. Оператор, который ссылается на функцию, является «вызывающей», а сама функция в этот момент является «вызываемой»: вызываемой.

Стандартный синтаксис для вызова функции заключается в написании имени функции, за которым следует список всех явно переданных параметров в круглых скобках. Если никакие параметры не передаются или у функции их нет, пара круглых скобок после имени функции все еще присутствует. Например, чтобы опробовать функцию мощности, следующая программа вызывает ее следующим образом:



Функция может дополнительно возвращать значение. Функции суммы, високосного года и мощности возвращают значение, а функция подкачки — нет. Даже если функция возвращает значение, вызывающая сторона может его проигнорировать.

Для ситуации, когда вызывающая сторона игнорирует возвращаемое функцией значение, существует альтернативный синтаксис для вызова функции, который также иллюстрируется предыдущим примером программы, вызывающей функцию мощности. Круглые скобки вокруг всех аргументов функции необязательны, если вызывающий объект не использует возвращаемое значение. В последнем утверждении пример программы читает



скорее, чем



который делает то же самое. Синтаксис без круглых скобок вокруг списка параметров называется синтаксисом «вызова процедуры». Вы можете использовать его, только если:

⋄ вызывающая сторона не присваивает результат функции переменной и не использует его в выражении или, например, в качестве «проверочного выражения» оператора if;

⋄ первый параметр не начинается с открывающей скобки

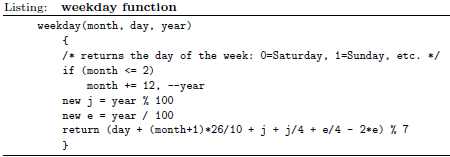
⋄ первый параметр находится в той же строке, что и имя функции, если вы не используете именованные параметры (см. следующий раздел).

Как вы можете заметить, синтаксис вызова процедуры применяется к случаям, когда вызов функции ведет себя скорее как оператор, как в вызовах print и printf в предыдущем примере. Синтаксис нацелен на то, чтобы такие операторы выглядели менее загадочными и более удобными для чтения, но не на то, чтобы использование синтаксиса было необязательным.

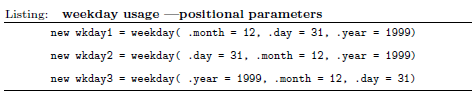
В качестве примечания: все круглые скобки в примере программы, представленном в этом разделе, обязательны: возвращаемые значения вызовов getvalue хранятся в двух переменных, поэтому за именем функции должна следовать пара пустых круглых скобок. Функция getvalue имеет необязательные параметры, но в этом примере программы они не передаются.

• Именованные параметры в сравнении с позиционными параметрами

В предыдущих примерах порядок параметров вызова функции был важен, потому что каждый параметр копируется в аргумент функции с той же последовательной позицией. Например, с помощью функции weekday (использующей алгоритм конгруэнтности Целлера), определенной, как показано ниже, вы должны вызвать weekday(12,31,1999), чтобы получить день недели последнего дня предыдущего столетия.



Форматы даты различаются в зависимости от культуры и страны. В то время как формат месяц/день/год является обычным в Соединенных Штатах Америки, европейские страны часто используют формат день/месяц/год, а технические публикации иногда стандартизируют формат год/месяц/день (ISO/IEC 8824). Другими словами, ни один порядок аргументов в функции дня недели не является «логичным» или «обычным». В таком случае альтернативным способом передачи параметров в функцию является использование «именованных параметров», как в следующих примерах (три вызова функций эквивалентны):



Для именованных параметров точка (".") предшествует имени аргумента функции. Аргумент функции может быть установлен в любое выражение, допустимое для аргумента. Знак равенства («=») в случае именованного параметра не указывает на присваивание; скорее он связывает выражение, следующее за знаком равенства, с одним из аргументов функции.

Можно смешивать позиционные параметры и именованные параметры в вызове функции с ограничением, что все позиционные параметры должны предшествовать любым именованным параметрам.

• Значения аргументов функции по умолчанию

Аргумент функции может иметь значение по умолчанию. Значение по умолчанию для

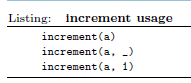
аргумент функции должен быть константой. Чтобы указать значение по умолчанию, добавьте знак равенства («=») и значение к имени аргумента.

Когда вызов функции указывает заполнитель аргумента вместо действительного аргумента, применяется значение по умолчанию. Заполнитель аргумента — это символ подчеркивания («\_»). Заполнитель аргумента действителен только для аргументов функции, имеющих значение по умолчанию.

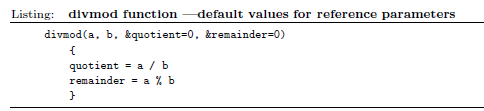
Заполнители крайних правых аргументов могут быть просто удалены из списка аргументов функции. Например, если приращение функции определено как:



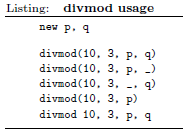
все следующие вызовы функций эквивалентны:



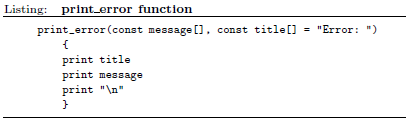
Значения аргументов по умолчанию для переданных по ссылке аргументов полезны, чтобы сделать входной аргумент необязательным. Например, если функция divmod предназначена для возврата как частного, так и остатка от операции деления через свои аргументы, значения по умолчанию делают эти аргументы необязательными:



С предыдущим определением функции divmod все следующие вызовы функций теперь допустимы:

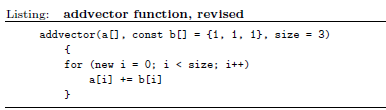


Аргументы по умолчанию для аргументов массива часто удобны для установки строки по умолчанию или приглашения для функции, которая получает строковый аргумент. Например:



В следующем примере поля одного массива добавляются в другой массив, а

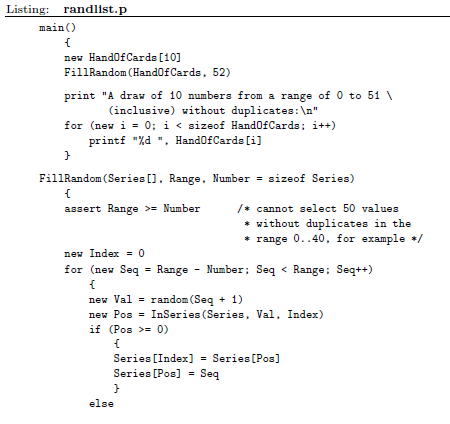
default увеличивает первые три элемента целевого массива на единицу:

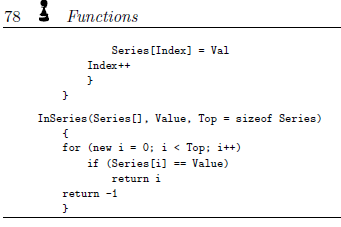


• оператор sizeof и аргументы функции по умолчанию

Значение аргумента функции по умолчанию должно быть константой, и его значение определяется в точке объявления функции. Использование оператора sizeof для установки значения аргумента функции по умолчанию является особым случаем: вычисление значения выражения sizeof откладывается до момента вызова функции и принимает размер фактического аргумента, а не его размер. формального аргумента. Когда функция используется в программе несколько раз с разными аргументами, результат выражения «sizeof» потенциально может быть разным при каждом вызове, что означает, что «значение по умолчанию» аргумента функции может измениться.

Ниже приведен пример программы, которая рисует десять случайных чисел в диапазоне от 0 до 51 без дубликатов. Примером приложения для розыгрыша случайных чисел без дубликатов могут служить карточные игры — эти десять чисел могут представлять карты для двух «рук» в игре в покер. Достоинства алгоритма, используемого в этой программе, изобретенного Робертом У. Флойдом, заключаются в том, что он эффективен и беспристрастен — при условии, что генератор псевдослучайных чисел также непредвзят.





Функция main объявляет массив HandOfCards размером в десять ячеек, а затем вызывает функцию FillRandom с целью получения десяти положительных случайных чисел меньше 52. Обратите внимание, однако, что единственные два параметра, которые main передает в вызов FillRandom, — это массив HandOf-Cards, где должны храниться случайные числа, и верхняя граница «52». Количество случайных чисел для рисования («10») неявно передается в FillRandom.

Определение функции FillRandom ниже main указывает для ее третьего параметра «Number = sizeof Series», где «Series» относится к первому параметру функции. Из-за особого случая «значения sizeof по умолчанию» значением по умолчанию аргумента Number является не размер формального аргумента Series, а размер фактического аргумента в точке вызова функции: HandOfCards.

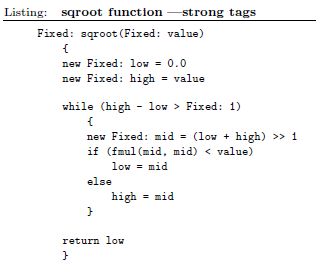
Обратите внимание, что внутри функции FillRandom, запрашивая «размер» функции

аргумент Series будет (по-прежнему) оцениваться как нуль, потому что массив Series объявлен с неопределенной длиной (см. стр. 110 о поведении sizeof).

Использование sizeof в качестве значения по умолчанию для аргумента функции является особым случаем. Если бы формальный параметр Series был объявлен с явным размером, как в Series[10], было бы излишним добавлять числовой аргумент с размером массива фактического аргумента, потому что синтаксический анализатор в этом случае заставлял бы формальные и фактические аргументы иметь размер и габариты.

• Аргументы с именами тегов

Тег необязательно предшествует аргументу функции. Использование тегов улучшает проверку скрипта на ошибки во время компиляции и служит «неявной документацией» функции. Например, функция, вычисляющая квадратный корень из входного значения с точностью до фиксированной точки, может потребовать, чтобы входной параметр был значением с фиксированной точкой и чтобы результат также был с фиксированной точкой.  
Приведенная ниже функция использует модуль расширения с фиксированной точкой и алгоритм аппроксимации, известный как «деление пополам», для вычисления квадратного корня. Обратите внимание на использование переопределений тегов для числовых литералов и результатов выражений.

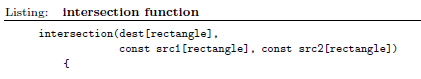


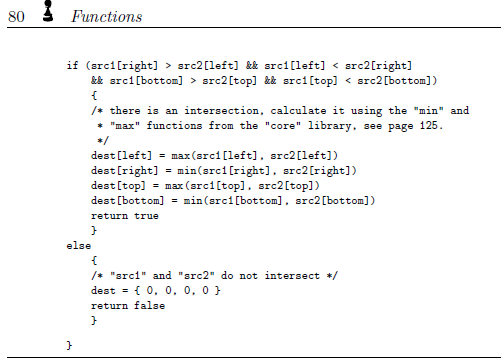
С приведенным выше определением синтаксический анализатор пешки выдает диагностику, если кто-то вызывает функцию sqroot с параметром с тегом, отличным от «Fixed:», или когда он пытается сохранить результат функции в переменной с «non-Fixed:» ярлык.

Алгоритм деления пополам связан с бинарным поиском в том смысле, что он

непрерывно делит пополам интервал, в котором должен лежать результат. Алгоритм «последовательной замены», такой как Ньютон-Рафсон, учитывающий наклон кривой функции, быстрее достигает точных результатов, но ценой того, что критерий остановки определить сложнее. Современные алгоритмы вычисления квадратных корней сочетают алгоритмы деления пополам и алгоритмы Ньютона-Рафсона.

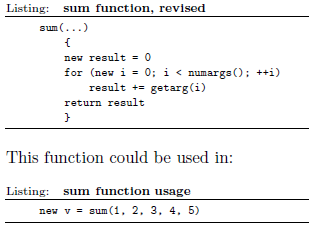
В случае массива индексы массива также могут быть помечены. Например, функция, создающая пересечение двух прямоугольников, может быть записана так:





• Переменные аргументы

Функция, которая принимает переменное количество аргументов, использует оператор «многоточие» («...») в заголовке функции для обозначения позиции первого переменного аргумента. Функция может получить доступ к аргументам с помощью предопределенных функций numargs, getarg и setarg (см. стр. 125). Функция sum возвращает сумму всех своих параметров. Он использует список параметров переменной длины.



Тег может предшествовать многоточию, чтобы гарантировать, что все последующие параметры имеют один и тот же тег, но в противном случае проверка ошибок с переменным списком аргументов не выполняется, и поэтому эту функцию следует использовать с осторожностью. Функции getarg и setarg предполагают, что аргумент передается «по ссылке». При использовании getarg для обычных параметров функции (вместо переменных аргументов) следует быть осторожным с этим, так как ни компилятор,

ни абстрактная машина не может этого проверить. Фактические параметры, которые передаются как часть «списка переменных аргументов», всегда передаются по ссылке.

• Правила принуждения

Если аргумент функции, согласно определению функции (или ее объявлению),

является «параметром значения», вызывающая сторона может передать функцию в качестве параметра:

⋄ значение, которое передается по значению;

⋄ ссылка, разыменованное значение которой передается;

⋄ (индексированный) элемент массива, который является значением.

Если аргумент функции является ссылкой, вызывающая сторона может перейти к функции:

⋄ значение, адрес которого передается;

⋄ ссылка, которая передается по значению, потому что она имеет тип, ожидаемый функцией;

⋄ (индексированный) элемент массива, который является значением.

Если аргумент функции является массивом, вызывающая сторона может перейти к функции:

⋄ массив той же размерности, у которого передается начальный адрес;

⋄ (индексированный) элемент массива, в этом случае передается адрес элемента.

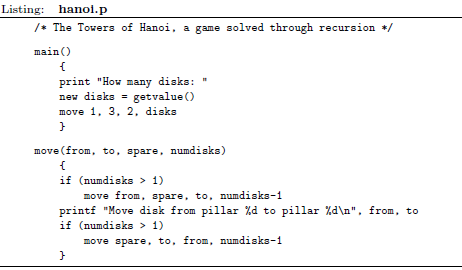
• Рекурсия

Ранее в этой главе в примере функции факультета использовался простой цикл. Пример функции, которая вычисляла число из ряда Фибоначчи, также использовала цикл и дополнительную переменную, чтобы добиться цели. Эти две функции являются наиболее популярными подпрограммами для иллюстрации рекурсивных функций, поэтому, реализуя их как итерационные процедуры, вы можете подумать, что pawn не поддерживает рекурсию.

Что ж, пешка поддерживает рекурсию, но вычисление способностей и чисел Фибоначчи является хорошим примером того, когда не следует использовать рекурсию.

Факультет легче понять с циклом, чем с рекурсией. Решение чисел Фибоначчи с помощью рекурсии действительно упрощает задачу, но ценой того, что оно крайне неэффективно: рекурсивное число Фибоначчи вычисляет одни и те же значения снова и снова.

Программа ниже представляет собой реализацию знаменитой игры «Ханойские башни» в рекурсивной функции:



• Форвардные объявления

Для стандартных функций текущая «эталонная реализация» компилятора pawn не требует объявления функций перед их первым использованием.\*

Пользовательские операторы — это специальные функции, и в отличие от стандартных функций они должны быть объявлены перед использованием. Во многих случаях удобно поместить реализацию определяемого пользователем оператора во включаемый файл, чтобы реализация и объявление предшествовали любому вызову/вызову. Однако иногда может потребоваться (или удобно) сначала объявить определяемый пользователем оператор и реализовать его в другом месте. В частности, этот метод используется для реализации «запрещенных» определяемых пользователем операторов.

Чтобы создать форвардное объявление, добавьте перед именем функции и списком ее параметров ключевое слово forward. Для совместимости с ранними версиями pawn и для сходства с C/C++ альтернативным способом прямого объявления функции является ввод заголовка функции и завершение его точкой с запятой (которая следует за закрывающей скобкой списка параметров).

Полное определение функции с непустым телом реализовано в другом месте исходного файла (за исключением запрещенных пользовательских операторов).

Классификаторы состояния игнорируются в предварительных объявлениях.

• Классификаторы состояний

Все функции, кроме нативных, могут дополнительно иметь атрибут состояния.

Он состоит из списка имен состояний (и автоматов) в угловых скобках за заголовком функции. Имена разделяются запятыми. Когда состояние является частью автомата не по умолчанию, имя автомата и разделитель двоеточия должны предшествовать состоянию; например, «parser:slash» обозначает косую черту состояния автоматного синтаксического анализатора.

Если функция имеет состояния, в исходном коде должно быть несколько «реализаций» функции. Все функции должны иметь одинаковый заголовок функции (за исключением списка классификаторов состояний).

В качестве специального синтаксиса, когда между угловыми скобками нет имен, функция связывается со всеми состояниями, которые не относятся к другим реализациям функции. Функция, которая обрабатывает «все состояния, не обработанные где-либо еще», — это так называемая резервная функция.

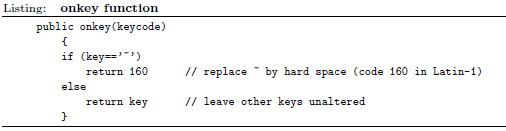
• Государственные функции, основная функция

Автономная программа должна иметь функцию main. Эта функция является отправной точкой программы. Функция main может не иметь аргументов.

В библиотеке функций не обязательно должна быть главная функция, но она должна быть.

либо основная функция, либо хотя бы одна публичная функция. Функция main является основной точкой входа в скомпилированную программу; публичные функции являются альтернативными точками входа в программу. Виртуальная машина может начать выполнение с одной из общедоступных функций. Библиотека функций может иметь основную функцию для выполнения однократной инициализации при запуске.

Чтобы сделать функцию общедоступной, добавьте перед именем функции ключевое слово public. Например, текстовый редактор может вызвать общедоступную функцию «onkey» для каждой клавиши, которую набрал пользователь, чтобы пользователь мог изменять (или отклонять) нажатия клавиш. Приведенная ниже функция onkey заменит каждый символ «~» (код 126 в наборе символов ISO Latin-1) кодом «жесткого пробела» в таблице символов ANSI:



Функции, имя которых начинается с символа «@», также являются общедоступными. Таким образом, альтернативный способ написать публичную функцию onkey:



Символ «@» при использовании становится частью имени функции; то есть в последнем примере функция называется «@onkey». Хост-приложение выбирает имена общедоступных функций, которые может реализовать сценарий.

Аргументы публичной функции могут не иметь значений по умолчанию. Общедоступная функция связывает хост-приложение со сценарием пешки. Следовательно, аргументы, передаваемые в общедоступную функцию, исходят из хост-приложения, и хост-приложение не может знать, какие «значения по умолчанию» автор сценария подставил для аргументов функции, поэтому синтаксический анализатор пешки помечает использование значений по умолчанию для аргументов общедоступных функций. работает как ошибка. Проблема значений по умолчанию в аргументах общедоступных функций возникает только в том случае, если вы хотите вызывать общедоступные функции из самого скрипта.

• Статические функции

Когда перед именем функции стоит ключевое слово static, область действия функции ограничивается файлом, в котором находится функция.

Статический атрибут можно комбинировать с атрибутом «запас».

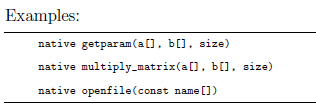
• Стандартные функции «Стандартная» функция — это функция, которую синтаксический анализатор должен «вставить» в программу, когда она используется, и которую он может просто «удалить» из программы (без предупреждения), когда она не используется. Стандартные функции позволяют компилятору или интерпретатору оптимизировать объем памяти и размер файла (скомпилированной) программы-заложника: любая стандартная функция, на которую нет ссылки, полностью пропускается, как если бы она отсутствовала в исходном файле.

Таким образом, стандартные функции обычно используются для создания набора «библиотечных» функций. Набор функций общего назначения, помеченных как «стандартные», может быть помещен в отдельный включаемый файл, который затем включается в любой скрипт ломбарда. Только библиотечные функции, которые фактически используются, «связываются». Чтобы объявить стандартную функцию, добавьте перед именем функции ключевое слово stock. Публичные функции и нативные функции не могут быть объявлены «стандартными». Когда стандартная функция вызывает другие функции, обычно рекомендуется объявлять и эти другие функции как «стандартные», за исключением нативных функций. Точно так же любые глобальные переменные, используемые функцией запаса

в большинстве случаев следует также определить как «запас». Удаление неиспользуемых (стандартных) функций может вызвать цепную реакцию, при которой доступ к другим функциям и глобальным переменным также будет закрыт. Затем эти функции также удаляются, тем самым продолжая цепную реакцию до тех пор, пока не останутся только те функции, которые прямо или косвенно используются.

• Собственные функции

Программа-пешеход может вызывать специфичные для приложения функции через «собственную функцию». Нативная функция должна быть объявлена в программе-заложнике с помощью прототипа функции. Перед именем функции должно стоять ключевое слово native.



Имена «getparam», «multiply\_matrix» и «openfile» являются входными.

внутренние имена нативных функций; это имена, под которыми функции известны в ломбардной программе. При желании вы также можете установить внешнее имя для собственной функции, которое является именем функции, которое знает «хост-приложение». Для этого добавьте к прототипу функции знак равенства, за которым следует внешнее имя. Например:





Когда нативная функция возвращает массив, размеры и размер массива должны быть явно объявлены. Спецификация массива находится между именем функции и списком параметров. Например:



Если не указано явно, внешнее имя равно внутреннему имени собственной функции. Одним из типичных способов использования явных внешних имен является установка символического имени для определяемого пользователем оператора, который реализован как встроенная функция.

См. «Руководство по реализации» для реализации собственных функций на C/C++ (на стороне «хост-приложения»).

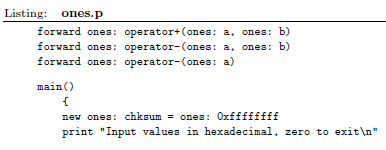
Собственные функции могут не иметь спецификаторов состояния.

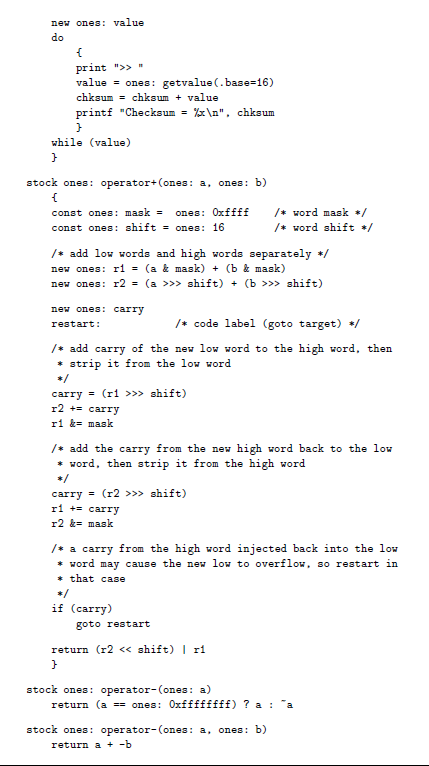
Пользовательские операторы

Единственный тип данных пешки — это «ячейка», обычно 32-битное число или бит.

шаблон. Значение значения в ячейке зависит от конкретного приложения — оно не всегда должно быть целым числом со знаком. пешка позволяет придать ячейке «значение» с помощью механизма «метки».

На основе тегов pawn также позволяет переопределять операторы для ячеек с определенной целью. В приведенном ниже примере определяется тег «единицы» и оператор для сложения двух значений «единицы» (в примере также реализованы операторы для вычитания и отрицания). Пример был вдохновлен алгоритмом контрольной суммы нескольких протоколов в наборе протоколов TCP/IP: он имитирует арифметику дополнения, добавляя бит переноса арифметического переполнения обратно к младшему значащему биту значения.





Примечательной строкой в ​​примере является строка «chksum = chksum + value» в цикле в функции main. Поскольку обе переменные chksum и value имеют тэговые единицы, оператор «+» относится к определяемому пользователем оператору (вместо стандартного оператора «+»). Определяемые пользователем операторы служат просто для удобства записи. Тот же эффект достигается явным вызовом функций. Определение оператора аналогично определению функции, с той разницей, что имя оператора состоит из ключевого слова «оператор» и символа самого оператора. В приведенном выше примере переопределены как унарный «-», так и бинарный «-». Операторная функция для бинарного оператора должна иметь два аргумента, одна для унарного оператора должна иметь один аргумент. Обратите внимание, что бинарный оператор «-» складывает два значения вместе после инвертирования знака второго операнда. Таким образом, оператор вычитания относится как к определяемому пользователем «отрицанию»,

(унарный «-») и операторы сложения.

Переопределенный оператор должен придерживаться следующих ограничений:

⋄ Пользовательский оператор должен быть объявлен перед использованием (в отличие от «обычных» функций): либо поместите реализацию пользовательского оператора над функциями, которые его используют, либо добавьте предварительное объявление в верхней части файл.

⋄ Переопределять можно только следующие операторы: +, -, \*, /, %, ++, --, ==,

!=, <, >, <=, >=, ! и =. То есть множества арифметических и реляционных

операторы могут быть перегружены, но побитовые операторы и логические

операторы не могут. = и ! операторы - это частный случай.

Вы не можете изобретать новые операторы; например, вы не можете определить оператор «#».

⋄ Уровень приоритета и ассоциативность операторов, а также их

«арность» остаются такими, как они определены. Вы не можете сделать унарный оператор «+», так как

пример.

⋄ Тег возврата операторов отношения и «!» оператор должен быть «bool:».

⋄ Тег возврата арифметических операторов выбирается по вашему выбору, но вы не можете переопределить оператор, который идентичен другому оператору, за исключением его тега возврата. Например, вы не можете сделать оба



(Оператор присваивания является исключением из этого правила.)

⋄ pawn уже определяет операторы для работы с непомеченными ячейками, вы не можете переопределить операторы только с аргументами без тегов.

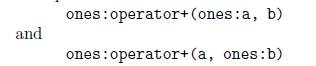
⋄ Аргументы операторной функции не должны быть массивами, передаваемыми по значению. Вы не можете заставить оператор работать с массивами.

В приведенном выше примере оба аргумента бинарных операторов имеют один и тот же тег. Это не требуется; вы можете, например, определить бинарный оператор «+», который добавляет целочисленное значение к числу «единицы:».

Напомню, что операция синтаксического анализатора пешки заключается в поиске тега (ов) операнда (ов), над которым работает оператор, и в поиске того, существует ли пользовательский оператор для комбинации оператора и тега ( с). Однако синтаксический анализатор распознает особые ситуации и предоставляет следующие возможности:

⋄ Синтаксический анализатор распознает такие операторы, как «+=», как последовательность «+» и «=», и будет вызывать пользовательский оператор «+», если он доступен, и/или пользовательский оператор «=». В примере программы строка «chksum = chksum + value» могла быть сокращена до «chksum += value».

⋄ Синтаксический анализатор распознает коммутативные операторы («+», «\*», «==» и «!=») и меняет местами операнды коммутативного оператора, если это соответствует заданному пользователем оператору. Например, обычно нет необходимости реализовывать оба



(реализация обеих функций допустима и полезна в случае, если определяемый пользователем оператор не должен быть коммутативным).

⋄ Префиксные и постфиксные операторы обрабатываются автоматически. Вам нужно определить только один пользовательский оператор для операторов «++» и «--» для тега.

⋄ Парсер вызывает «!» оператор неявно в случае теста без

явное сравнение. Например, в утверждении «if (var)…»

когда «var» имеет тег «ones:», определяемый пользователем оператор «!» будет называться

для вар. «!» Таким образом, оператор удваивается как оператор «теста на ноль». (В

арифметике с дополнением до единицы, битовые шаблоны «все единицы» и «все нули» представляют ноль.)

⋄ Определяемый пользователем оператор присваивания неявно вызывается для аргумента функции, который передается «по значению», когда имена тегов формального и фактического аргументов совпадают с именами тегов в левой и правой частях оператора. Другими словами, синтаксический анализатор пешки имитирует, что «передача по значению» происходит посредством присваивания. Пользовательский оператор не вызывается для аргументов функции, которые передаются «по ссылке».

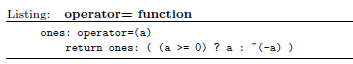
⋄ Если вы хотите запретить операцию, вы можете «объявить вперед» оператор, даже не определяя его (см. стр. 82). Это пометит ошибку при вызове определяемого пользователем оператора. Например, чтобы запретить оператор «%» (остаток после деления) над значениями с плавающей запятой, можно добавить строку:



Определяемые пользователем операторы могут быть опционально объявлены «стандартными» или «нативными». В случае встроенной операторной функции определение должно включать внешнее имя. Например (когда на стороне хоста нативная функция вызывается float\_add):

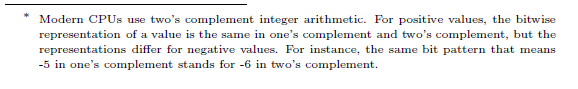


Пользовательский оператор присваивания — это особый случай, потому что это оператор с побочным эффектом. Хотя оператор имеет вид бинарного оператора, его «результатом выражения» является значение справа — оператор присваивания был бы «нулевым» оператором, если бы не его побочный эффект. В pawn определяемый пользователем оператор присваивания объявляется как:



Определяемый пользователем оператор «=» в этом определении выглядит как унарный оператор, но, тем не менее, это особый случай. В отличие от других операторов, тег возвращаемого значения для определяемого пользователем оператора важен: синтаксический анализатор пешки использует теги аргумента и возвращаемого значения, чтобы найти соответствующий определяемый пользователем оператор.

Вышеприведенный пример функции представляет собой типичное приложение для определяемого пользователем оператора присваивания: для автоматического приведения/преобразования непомеченного значения в помеченное значение и для необязательного изменения представления значения в памяти в процессе. В частности, оператор «новые:A = -5» вызывает выполнение определяемого пользователем оператора, а для константы -5 оператор вернет «~(- -5)», или ~5, или −6.\*



• Арифметика с плавающей запятой и фиксированной запятой

pawn имеет встроенную поддержку только целочисленной арифметики (Z-домен или «целые числа», как положительные, так и отрицательные). Поддержка арифметики с плавающей запятой или арифметики с фиксированной запятой должна быть реализована через (собственные) функции. Таким образом, пользовательские операторы обеспечивают более естественную запись выражений с числами с фиксированной или плавающей запятой.

Парсер пешки поддерживает литеральные значения с дробной частью, которые он называет «рациональными числами». Поддержка рациональных литералов должна быть включена явно с помощью #pragma. #pragma указывает, как должны храниться рациональные числа — с плавающей запятой или с фиксированной точкой. Для рациональных значений с фиксированной точкой #pragma также указывает точность в десятичных дробях. Два примера для #pragma:



Поскольку значение с фиксированной запятой все равно должно умещаться в ячейке, количество десятичных знаков напрямую влияет на диапазон значения с фиксированной запятой. Для значения с фиксированной запятой с 3 десятичными знаками диапазон будет равен −2, 147, 482 . . .+ 2, 147, 482.

Формат рационального числа может быть указан только один раз для всей пешечной программы. В реализации обычно выбирают либо поддержку с плавающей запятой, либо поддержку с фиксированной запятой. Как указано выше, для фактической реализации арифметики с плавающей или фиксированной точкой pawn требует помощи (собственных) функций и определяемых пользователем операторов. Хорошее место для размещения #pragma для поддержки рациональных чисел было бы во включаемом файле, который также определяет функции и операторы.

Включаемый файл † для арифметики с фиксированной точкой содержит такие определения, как:



Определяемые пользователем операторы для умножения и деления двух чисел с фиксированной запятой напрямую связаны с родными функциями fmul и fdiv. Таким образом, основное приложение должно предоставлять эти собственные функции.

Другой собственный пользовательский оператор удобен для автоматического преобразования целого числа в фиксированную точку, если он назначен переменной с тегом «Fixed:»:

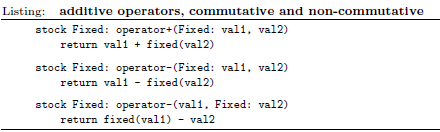




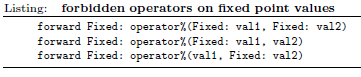
С этим определением вы можете сказать «new Fixed: fract = 3», и значение будет преобразовано в 3.000, когда оно будет сохранено в переменной fract. В качестве

как описано в разделе, посвященном пользовательским операторам, оператор присваивания также выполняется для аргументов функции, которые передаются по значению. В выражении «new Fixed: root = sqroot(16)» (см. реализацию функции sqroot на стр. 79) для аргумента 16 вызывается пользовательский оператор присваивания.

Для сложения двух значений с фиксированной точкой достаточно стандартного оператора «+», то же самое относится и к вычитанию. Добавление нормального (целого) числа к числу с фиксированной запятой отличается: нормальное значение должно быть масштабировано перед его добавлением. Следовательно, включаемый файл также реализует операторы для этой цели:



Оператор «+» является коммутативным, поэтому одна реализация обрабатывает оба случая. Для оператора «-» оба случая должны быть реализованы отдельно. Наконец, включаемый файл запрещает использование оператора остатка («%») для значений с фиксированной точкой: остаток применим только к целочисленным делениям:



Из-за наличия (прямого) объявления оператора синтаксический анализатор пешки попытается использовать определяемый пользователем оператор, а не оператор по умолчанию «%». Не реализовав оператор, синтаксический анализатор впоследствии выдаст сообщение об ошибке.

Препроцессор

Первым этапом компиляции исходного файла пешки в исполняемый P-код является «предварительная обработка»: текстовый фильтр общего назначения, который изменяет/очищает текст перед его подачей в синтаксический анализатор. На этапе предварительной обработки удаляются комментарии, удаляются «условно скомпилированные» блоки, обрабатываются директивы компилятора и выполняются операции поиска и замены в тексте исходного файла. Директивы компилятора кратко изложены на стр. 118, а подстановка текста («найти-и-заменить») является темой этой главы.

Препроцессор — это процесс, который вызывается для всех строк исходного кода сразу после их чтения. Во время замены текста проверка синтаксиса не выполняется. В то время как препроцессор допускает мощные трюки на языке пешек, с его помощью также легко выстрелить себе в ногу.

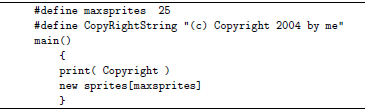
В этой главе я несколько раз буду ссылаться на язык C/C++.

потому что препроцессор pawn подобен тому, что используется в C/C++. При этом препроцессор pawn несовместим с препроцессором C/C++.

Директива #define определяет макросы препроцессора. Простые макросы:



Затем в скрипте пешки вы можете использовать их так же, как и константы. Например:



Кстати, для этих простых макросов есть эквивалентные конструкции pawn:



Преимущество этих объявлений констант состоит в лучшей проверке ошибок и возможности создавать теговые константы. Синтаксис строковой константы — это переменная массива, объявленная как «const», так и «stock». Атрибут const запрещает любое изменение строки, а атрибут stock делает объявление «исчезающим», если на него никогда не ссылаются.

Макрос подстановки может принимать до 10 параметров. Типичное использование макросов с параметрами — имитация крошечных функций:



Если вы знаете C/C++, вы узнаете привычку заключать каждый аргумент и все выражение подстановки в круглые скобки.

Если вы используете вышеуказанный макрос в скрипте следующим образом:



препроцессор переводит его в:



что приводит к тому, что «a», возможно, увеличивается дважды. Это одна из ловушек, в которые вы можете попасть при использовании макросов подстановки (эта конкретная проблема хорошо известна программистам на C/C++). Поэтому может быть хорошей идеей использовать соглашение об именах, чтобы отличать макросы от функций. В C/C++ принято писать макросы препроцессора в верхнем регистре. Чтобы показать, почему заключать аргументы макроса в круглые скобки — хорошая идея, рассмотрим макрос:



Этот макрос делит первый аргумент на второй аргумент, но с округлением в большую сторону до ближайшего целого числа (оператор деления «/» округляет в меньшую сторону). Если вы используете его следующим образом:



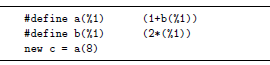
вторая строка расширяется до «новое b = (8 + a - 2 - 1) / a - 2», что, учитывая уровни приоритета пешечных операторов, приводит к тому, что «b» устанавливается равным нулю (если «a» 5). Глядя на вызов макроса, вы могли бы ожидать, что восемь делится на три ("a - 2"), округленное вверх, следовательно, "b" будет установлено равным 3. Изменение макроса для заключения каждого параметра в круглые скобки решает проблему. По аналогичным причинам также рекомендуется заключать полный текст замены в круглые скобки. Ниже приведен соответствующий измененный макрос ceil\_div:



Сопоставление с образцом более тонкое, чем сопоставление строк, которые выглядят как вызовы функций. Шаблон соответствует тексту буквально, но принимает произвольный текст, если шаблон указывает параметр. Вы можете создавать такие узоры, как:



Когда раскрытие макроса содержит текст, совпадающий с другими макросами, раскрытие выполняется во время вызова, а не во время определения. Таким образом, код:



будет оцениваться как «новый c = (1+(2\*(8)))», даже если макрос «b» был

не определено во время определения «а».

Сопоставление с образцом ограничено следующими правилами:

⋄ В шаблоне может не быть пробелов. Если вы должны соответствовать

пробел, вам нужно использовать «\32;» последовательность выхода. Текст замены,

с другой стороны, может содержать символы пробела. Из-за совпадения

правила шаблона макроса (поясняются ниже), соответствующие символу пробела

требуется редко.

⋄ Как видно из предыдущей строки, управляющие последовательности могут появляться в

шаблон (хотя они не очень полезны, за исключением, возможно, совпадения с буквальным символом «%»).

⋄ Шаблон не может заканчиваться параметром; шаблон типа «set:%1=%2» недопустим. Если вы хотите соответствовать концу оператора, вы можете добавить точку с запятой в конце шаблона. Если точка с запятой необязательна в конце каждого оператора, точка с запятой также будет соответствовать новой строке в исходном коде.

⋄ Шаблон должен начинаться с буквы, символа подчеркивания или символа «@». Первая часть шаблона, состоящая из буквенно-цифровых символов (плюс «\_» и/«@»), является «именем» или «префиксом» макроса. Для определенного оператора и директивы #undef вы указываете префикс макроса.

⋄ При сопоставлении с образцом препроцессор игнорирует пробелы между небуквенно-цифровыми символами и пробелы между буквенно-цифровыми символами и небуквенно-цифровыми, за одним исключением: между двумя одинаковыми символами пробелы не игнорируются. Следовательно: шаблон abc(+-) соответствует «abc ( + - )»

шаблон abc(--) соответствует «abc (--)», но не соответствует «abc(--)»

⋄ Имеется до 10 параметров, обозначаемых «%» и одной цифрой (от 1 до 9 и 0). Порядок параметров в шаблоне не важен.

⋄ Символ #define является директивой парсера. Как и все директивы синтаксического анализатора, определение шаблона должно умещаться на одной строке. Вы можете обойти это с помощью «\» в конце строки. Текст для соответствия также должен умещаться на одной строке.

Обратите внимание, что при наличии (параметризованных) макросов строки исходного кода могут быть не такими, какими кажутся: то, что выглядит как доступ к массиву, может быть «предварительно обработано» для вызова функции, и наоборот.

Ведущее приложение, в которое встроен парсер пешки, может предоставить возможность проверки результата подстановки текста с помощью макросов. Если вы используете стандартный набор инструментов pawn, вы найдете инструкции по использованию компилятора и среды выполнения в сопроводительном буклете «The pawn booklet — Implementor’s Guide».

Общий синтаксис

Формат

Идентификаторы, числа и токены разделяются пробелами, символами табуляции, возвратами каретки и «потоками форм». Последовательности из одного или нескольких таких разделителей называются пробелами.

Необязательные точки с запятой

Точки с запятой (для завершения оператора) необязательны, если они стоят в конце строки. Точки с запятой необходимы для разделения нескольких операторов

на одной линии. Выражение может по-прежнему переноситься на несколько строк,

но постфиксные операторы (++, -- и char) должны стоять на одном и том же

строка в качестве их операнда.

Комментарии

Текст между токенами /\* и \*/ (оба токена могут быть одновременно

строки или на разных строках) и текст позади // (до конца

строка) является программным комментарием. Синтаксический анализатор рассматривает комментарий как

белое пространство. Комментарии не могут быть вложенными.

Комментарий, начинающийся с «/\*\*» (две звездочки и пробел).

за второй звездочкой) и оканчивается на «\*/» — это документация

комментарий. Комментарий, начинающийся с «///» (три слэша и

пробел за третьей косой чертой) также является комментарием к документации. Синтаксический анализатор может особым образом обрабатывать комментарии к документации; например, он может создать из него интерактивную справку.

Идентификаторы

Имена переменных, функций и констант. Идентификаторы состоят из

персонажи а. . . з, А. . . З, 0. . . 9, \_ или @; первый символ может

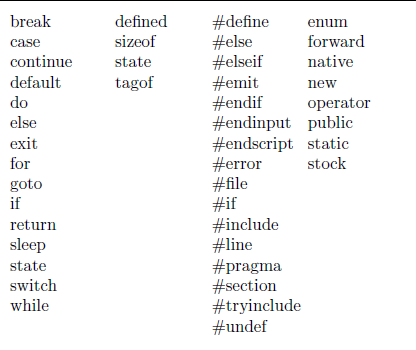
не быть цифрой. Символы @ и \_ сами по себе недействительны

идентификаторы, т. е. «\_Up» является допустимым идентификатором, а «\_» — нет.

пешка чувствительна к регистру. Синтаксический анализатор может обрезать идентификатор после максимальной длины. Количество значащих символов определяется реализацией, но не менее 16 символов.

Reserved words (keywords)





Помимо зарезервированных слов, pawn также имеет несколько предопределенных констант, вы не можете использовать имена символов предопределенных констант для имен переменных или функций.

Constants (literals)

Integer numeric constants

Binary 0b, за которым следует последовательность цифр 0 и 1.

Decimal последовательность цифр от 0 до 9.

Hexadecimal 0x, за которым следует ряд цифр от 0 до 9 и

буквы от а до ф.

Во всех основных числах знак подчеркивания может использоваться для разделения групп (шестнадцатеричных цифр. Символы подчеркивания между цифрами игнорируются.

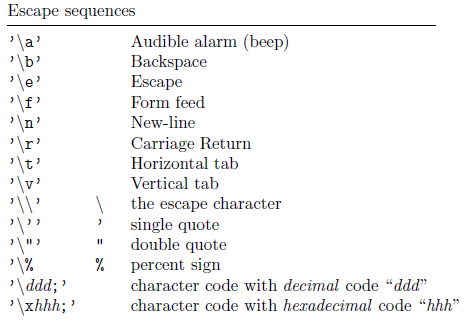
Rational number constants

Рациональное число — это число с дробной частью. Рациональное число начинается с одной или нескольких цифр, содержит десятичную точку и имеет по крайней мере одну цифру после десятичной точки. Например, «12,0» и «0,75» — допустимые рациональные числа. При желании к рациональному числу может быть добавлен показатель степени; обозначение степени — это буква «e» (нижний регистр), за которой следует целочисленная числовая константа со знаком. Например, «3.12e4» — допустимое рациональное число с показателем степени.

Поддержка рациональных чисел должна быть включена с помощью директивы #pragma rational. В зависимости от опций, установленных этой директивой, рациональное число представляет число с плавающей или фиксированной точкой.

Character constants

Один символ ASCII, заключенный в одинарные кавычки, является символьной константой (например: «a», «7», «$»). Символьные константы считаются числовыми константами.



Точка с запятой после \ddd; и \xhhh; коды необязательны. Его цель состоит в том, чтобы дать escape-последовательности явный символ завершения, когда он используется в строковой константе.

Обратная косая черта («\») — это символ «экранирования» по умолчанию. При желании вы можете установить другой escape-символ с помощью директивы #pragma ctrlchar (стр. 121).

String constants

Предполагается, что строковые константы представляют собой массивы с размером, достаточным для хранения всех символов плюс завершающий '\0'. Каждая строка хранится в уникальной позиции в памяти; нет устранения повторяющихся строк.

Неупакованная строка представляет собой серию из нуля или более символов ASCII, заключенных в двойные кавычки. Каждый элемент массива содержит один символ. Неупакованная строка может содержать символы в многобайтовом наборе символов, таком как Unicode или UCS-4.



Литерал упакованной строки следует синтаксису неупакованной строки, но символ «!» предшествует первой двойной кавычке.



В случае упакованной строки синтаксический анализатор упаковывает в ячейку столько символов, сколько поместится. Символ не адресуется как отдельная единица, вместо этого каждый элемент массива содержит несколько символов. Первый символ в «пакете» занимает старшие биты элемента массива. В средах, в которых слова памяти хранятся со старшим байтом по младшему адресу (формат Big Endian или Motorola), отдельные символы хранятся в ячейках памяти в том же порядке, что и в строке. Упакованная строка заканчивается нулевым символом, и строка дополняется (нулевыми байтами) до нескольких ячеек.

Упакованная строка может содержать символы только из однобайтового набора символов, такого как ascii или один из расширенных наборов ascii из нормы ISO 8859.

Escape-последовательности могут использоваться внутри строк. См. раздел о символьных константах (стр. 99) для получения списка escape-последовательностей.

Существует альтернативный синтаксис для «простых строк». В простой строке каждый символ воспринимается как есть, а управляющие последовательности не распознаются. Обычные строки удобны для хранения имен файлов/ресурсов, особенно в случае, когда escape-символ также используется в качестве специального символа операционной системой или хост-приложением.

Синтаксис простой строки — это escape-символ, за которым следует строка в двойных кавычках. Обратная косая черта («\») — это символ «экранирования» по умолчанию. Вы не можете вводить escape-последовательности в простую строку: все символы будут восприняты буквально.



В приведенном выше примере вхождения «\a» и «\n» указывают не на escape-последовательности, а на буквальные пары символов «\» и «a», а также «\» и «n».

Упакованная простая строка имеет как «!» и escape-символ перед открывающей двойной кавычкой. Обе строки ниже представляют собой упакованные простые строки:



Array constants

Последовательность числовых констант между фигурными скобками является константой массива. Константы массива можно использовать для инициализации переменных массива с помощью (см. стр. 64) и их можно передавать в качестве аргументов функции (см. стр. 71).

Symbolic constants

Исходный файл объявляет символические константы с помощью инструкций const и enum. Ключевое слово const объявляет одну константу, а перечисление определяет список — обычно — последовательных констант с одинаковым именем тега.



Создает символическую константу со значением константного выражения справа от оператора присваивания.

Константу можно использовать в любом месте, где допустимо буквальное число (например: в выражениях, в объявлениях массивов и в директивах типа «#if» и «#assert»).



Инструкция enum создает серию констант с увеличением

ценности. Список констант представляет собой серию идентификаторов, разделенных запятыми. Если не отменено, первая константа списка перечислений имеет значение 0, а каждая последующая константа имеет значение предыдущей плюс 1.

И значение константы, и значение приращения можно установить, добавив значение к идентификатору константы. Чтобы установить значение, используйте в списке констант. 

Чтобы установить приращение, используйте:



Значение приращения сбрасывается на 1 после каждого объявления символа константы в списке констант.

Если для константы должны быть установлены и приращение, и значение, то приращение (нотация «[...]») должно предшествовать значению (нотация «=»).

Символы в постоянном списке могут иметь явный тег, который должен предшествовать имени символа.

Маркер имени, который следует за ключевым словом enum, является необязательным. Если оно включено и имена символов не имеют явного тега, это имя используется в качестве имени тега для каждого символа в постоянном списке. Кроме того, команда enum создает дополнительную константу с именем для имени константы и имени тега. Значение последней константы равно значению последнего символа в списке констант плюс приращение

значение этой последней константы.

Поля перечисления, имеющие тег, могут переопределять другие поля в других перечислениях при условии, что оба перечисления имеют имя тега. Когда поле перечисления используется в выражении, и особенно когда оно используется в качестве индекса массива, используемая константа выбирается на основе ожидаемого тега.

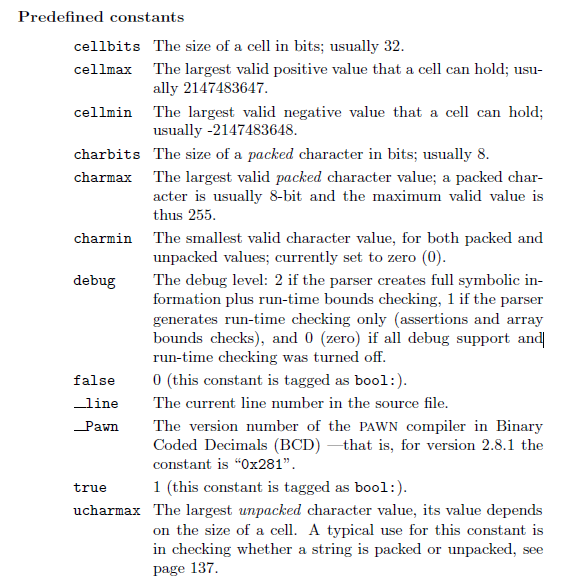
Маркер приращения, который следует за необязательным токеном имени, также является необязательным. Если он включен, он указывает другое правило постинкремента. По умолчанию перечисление увеличивает значение каждой последующей константы на 1, но вы можете указать другое правило с синтаксисом «(операторная константа)», где оператор должен быть +=, \*= или <<=. Оператор += создает аддитивное приращение, \*= и <<= создают мультипликативное приращение. Константа может быть буквальным значением или символьной константой. Правило приращения должно быть заключено в круглые скобки.

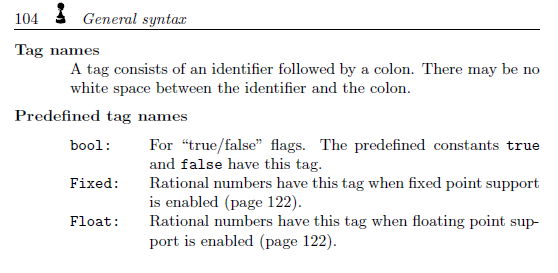
Если правило приращения не указано, круглые скобки также могут быть опущены.

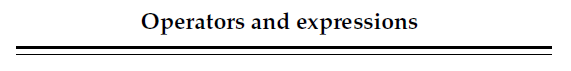
Символическая константа, определенная локально, действительна во всем блоке. Локальная символьная константа не может иметь то же имя, что и переменная (локальная или глобальная), функция или другая константа (локальная или

Глобальный).

Predefined constants







• Условные обозначения

Работа некоторых операторов зависит от конкретных типов операндов.

Поэтому операнды обозначаются так:

любое выражение;

v любое выражение, которому может быть присвоено значение (выражения «lvalue»);

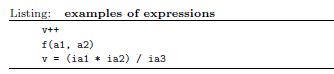
массив;

функция;

s символ — переменная, константа или функция.

• Выражения

Выражение состоит из одного или нескольких операндов с оператором. Операнд может быть переменной, константой или другим выражением. Выражение, за которым следует точка с запятой, является оператором.



• Арифметика

+ е1 + е2

Приводит к добавлению e1 и e2.

- е1 - е2

Результат вычитания e1 и e2.

-е

Результатом является арифметическое отрицание a (дополнение до двух).

\* е1 \* е2

Результат умножения e1 и e2.

/ е1 / е2

Результат деления e1 на e2. Результат усекается

до ближайшего целого значения, меньшего или равного

частное. И отрицательные, и положительные значения округляются в меньшую сторону,

т. е. в сторону −∞.

% е1 % е2

Получается остаток от деления e1 на e2. Знак

остатка следует за знаком e2. Целочисленное деление и

остальные имеют евклидово свойство: D = q\*d + r, где

q = D/d и r = D%d.

++ в++

увеличивает v на 1; результат, если выражение является значением

v до того, как он будет увеличен.

++v

увеличивает v на 1; результат, если выражение является значением

v после его увеличения.

-- в--

уменьшает v на 1; результат, если выражение является значением

v до того, как он будет уменьшен.

--v

уменьшает v на 1; результат, если выражение является значением

v после его уменьшения.

Примечания: унарный + не определен в пешке.

Операторы ++ и -- изменяют операнд. Операнд

должно быть lvalue.

• Битовые манипуляции

~ ~е

приводит к дополнению e к единице.

>> е1 >> е2

приводит к арифметическому сдвигу вправо от e1 на e2 бит.

Операция сдвига является знаковой: копируется крайний левый бит e1.

на свободные биты в результате.

>>> е1 >>> е2

приводит к логическому сдвигу вправо от e1 на e2 бит.

операция сдвига беззнаковая: свободные биты результата

заполнены нулями.

<< е1 << е2

приводит к смещению значения e1 влево на e2 битов; в

крайние правые биты устанавливаются равными нулю. Нет различия между

арифметический и логический сдвиг влево

и е1 и е2

приводит к побитовому логическому «и» e1 и e2.

| е1 | е2

приводит к побитовому логическому «или» e1 и e2.

^ е1 ^ е2

приводит к побитовому «исключающему или» e1 и e2.

• **Assignment**

Результатом выражения присваивания является значение левого операнда после присваивания. Левый операнд не может быть помечен.

= v = е

присваивает значение e переменной v.

= v = а

присваивает массив a переменной v; v должен быть массивом с

того же размера и размеров, что и а; a может быть строкой или литералом

множество.

Примечание: следующие операторы сочетают присваивание с арифметикой.

или побитовая операция; результат выражения

значение левого операнда после арифметического или побитового

операция.

+= в += е

увеличивает v на e.

-= v -= е

уменьшает v на e

\*= в \*= е

умножает v на e

/= v /= е

делит v на e.

%= v %= е

присваивает остаток от деления v на e на v.

>>= v >>= е

сдвигает v арифметически вправо на e бит.

>>>= в >>>= е

сдвигает v логически вправо на e бит.

<<= v <<= е

сдвигает v влево на e бит.

&= v &= е

применяет побитовое «и» к v и e и присваивает результат v.

|= v |= е

применяет побитовое «или» к v и e и присваивает результат v.

^= в ^= е

применяет побитовое «исключающее или» к v и e и присваивает

результат к v.

• **Relational**

Логическое «ложь» представлено целочисленным значением 0; логическое «истина» представлено любым значением, отличным от 0. Результатами значений реляционных выражений являются либо 0, либо 1, а их тег установлен на «bool:»

== е1 == е2

приводит к логическому «истине», если e1 равно e2.

!= е1 != е2

приводит к логическому «истине», если e1 отличается от e2.

Примечание: следующие операторы могут быть «сцеплены», как в выражении

«e1 <= e2 <= e3», с семантикой, что результат равен «1»

если все отдельные сравнения выполняются, и «0» в противном случае.

< е1 < е2

приводит к логическому «истине», если e1 меньше, чем e2.

<= е1 <= е2

приводит к логическому «истине», если e1 меньше или равно e2.

> е1 > е2

приводит к логическому «истине», если e1 больше, чем e2.

>= е1 >= е2

приводит к логическому «истине», если e1 больше или равно e2.

• **Boolean**

Логическое «ложь» представлено целочисленным значением 0; логическое «истина»

представлен любым значением, отличным от 0. Значение результатов логических выражений

либо 0, либо 1, а их тег установлен на «bool».

! !е

приводит к логическому «истине», если e было логически «ложным».

|| е1 || е2

приводит к логическому «истине», если либо e1, либо e2 (или оба)

логически «истинно». Выражение e2 оценивается только в том случае, если e1 равно

логически "ложно".

&& е1 && е2

приводит к логическому «истине», если и e1, и e2 логически

"истинный". Выражение e2 оценивается только в том случае, если e1 логически

"истинный".

• **Miscellaneous**

[ ] а[е]

индекс массива: результаты в ячейку e из массива a.

{ } а{е}

индекс массива: приводит к символу e из «упакованного» массива a.

( ) f(e1,e2,...eN)

приводит к значению, возвращаемому функцией f. Функция

вызывается с аргументами e1, e2, . . . эн. Порядок оценки

аргументов не определено (реализация может

выбрать оценку аргументов функции в обратном порядке).

? : е1 ? е2 : е3

приводит либо к e2, либо к e3, в зависимости от значения e1.

условное выражение — это составное выражение с двумя

часть оператора, «?» а также ":". Выражение e2 оценивается, если e1

логически «истинно», e3 оценивается, если e1 логически «ложно».

: тэг: e

переопределение тега; значение выражения e не меняется,

но его тег меняется. См. стр. 67 для получения дополнительной информации.

, е1, е2

приводит к e2, e1 вычисляется перед e2. Если используется в функции

списки аргументов или условное выражение, запятая

должны быть заключены в круглые скобки.



приводит к значению 1, если символ определен. Символ может быть константой (стр. 98), глобальной или локальной переменной. Тег этого выражения — bool:.



приводит к размеру в «элементах» указанной переменной. Для простых переменных и для массивов с одним измерением элементом является ячейка. Для многомерных массивов результатом является количество элементов массива в этом измерении — добавьте [] к имени массива, чтобы указать меньшее/меньшее измерение. Если размер переменной неизвестен, результат равен нулю. При использовании в значении по умолчанию для аргумента функции выражение вычисляется в момент вызова функции, а не

в определении функции.



где s — имя состояния, перед которым может быть указано имя автомата, этот оператор приводит к значению 1, если автомат находится в указанном состоянии, и к 0 в противном случае. Тег этого выражения — bool:.



приводит к уникальному числу, представляющему тег переменной, константу, результат функции или метку тега. При использовании в значении по умолчанию для аргумента функции выражение вычисляется в точке вызова функции, а не в определении функции.

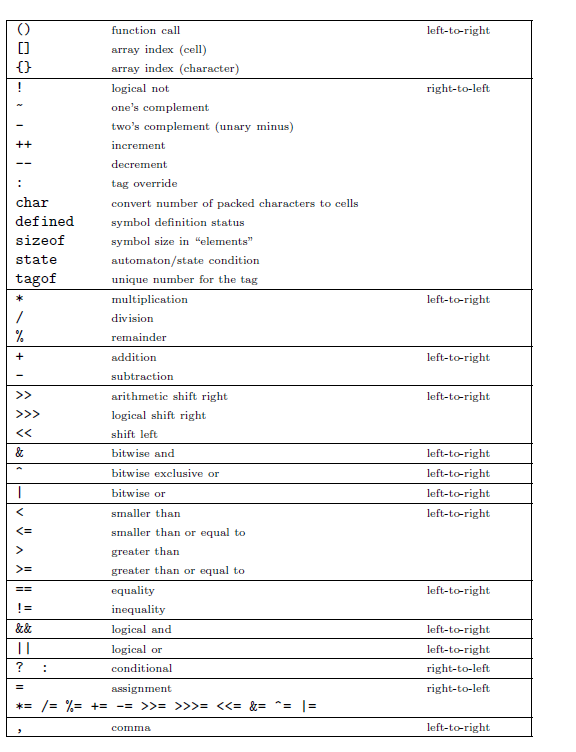


возвращает количество ячеек, необходимых для хранения упакованного массива из e символов.

• **Operator precedence**

В таблице ниже сгруппированы операторы с одинаковым приоритетом, начиная с группы операторов с наивысшим приоритетом в верхней части таблицы.

Если порядок вычисления выражений явно не установлен скобками, он определяется правилами ассоциации. Например: a\*b/c эквивалентно (a\*b)/c из-за связи слева направо, а a=b=c эквивалентно a=(b=c).





Оператор может занимать одну или несколько строк, тогда как одна строка может содержать два или более операторов.

Операторы потока управления (if, if-else, for, while, do-while и switch) могут быть вложенными.

Statement label

Метка состоит из идентификатора, за которым следует двоеточие («:»). Метка — это «цель перехода» оператора goto.

Каждому оператору может предшествовать метка. После метки должен быть оператор; допускается пустой оператор.

Областью действия метки является функция, в которой она объявлена (поэтому оператор goto не может перейти от текущей функции к другой функции).

Compound statement

Составной оператор — это последовательность из нуля или более операторов, заключенных в фигурные скобки ({ и }). Последняя фигурная скобка (}) не должна быть

после точки с запятой. Любой оператор может быть заменен составным оператором. Составной оператор также называется блоком. Составной оператор с нулевыми операторами является особым случаем и называется «пустым оператором».

Expression statement

Любое выражение становится оператором, когда к нему добавляется точка с запятой (";"). Выражение также становится оператором, когда только

пробел следует за ним в строке, и выражение не может быть расширено на следующую строку.

Empty statement

Пустой оператор не выполняет никакой операции и состоит из составного блока с нулевыми операторами; то есть он состоит из токенов «{}». Пустые операторы используются в операторах потока управления, если нет действия (например, while (!iskey()) {}) или при определении метки непосредственно перед закрывающей фигурной скобкой составного оператора. Пустой оператор не заканчивается точкой с запятой.

assert expression

Прерывает программу с ошибкой времени выполнения, если выражение логически оценивается как «ложь».

Break

Завершает и выходит из наименьшего объемлющего оператора do, for или while из любой точки внутри цикла, отличной от логического конца. Оператор break перемещает управление программой к следующему оператору вне цикла.

Continue

Завершает текущую итерацию наименьшего включающего оператора do, for или while и перемещает управление программой в часть цикла, связанную с условием. Если оператор цикла является оператором for, управление переходит к третьему выражению в операторе for (и затем ко второму выражению).

do statement while ( expression )

Выполняет оператор до того, как часть условия (предложение while) будет оценена. Оператор повторяется до тех пор, пока условие логически «истинно». Оператор выполняется как минимум один раз.

exit expression

Прервите программу. Выражение является необязательным, но оно должно начинаться с той же строки, что и оператор выхода, если он присутствует. Инструкция выхода возвращает значение выражения (плюс тег выражения) хост-приложению или ноль, если выражение выхода отсутствует. Значение и назначение кодов выхода определяется реализацией.

for ( expression 1 ; expression 2 ; expression 3 ) statement

All three expressions are optional.

выражение 1

Вычисляется только один раз и перед входом в цикл. Это выражение может использоваться для инициализации переменной. Это выражение также может содержать объявление переменной с использованием нового синтаксиса. Переменная, объявленная в этом выражении, существует только в цикле for.

В этом поле нельзя совмещать и выражение (используя уже существующие переменные) и объявление новых переменных — либо все переменные в этом поле уже должны существовать, либо все они должны быть созданы в этом поле.

выражение 2 Вычисляется перед каждой итерацией цикла и завершает цикл, если выражение приводит к логическому «ложному результату». Если он опущен, результат выражения 2 считается логически «истинным». выражение 3 Вычисляется после каждого выполнения оператора. Программное управление переходит от выражения 3 к выражению 2 для следующей (условной) итерации цикла. Оператор for( ; ; ) эквивалентен оператору while (true).

goto label

Переносит управление программой (безоговорочно) на оператор, следующий за указанной меткой. Метка должна находиться внутри той же функции, что и инструкция goto (оператор goto не может выходить за пределы функции).

if ( expression ) statement 1 else statement 2

Выполняет оператор 1, если выражение приводит к логическому «истине». Предложение else оператора if является необязательным. Если выражение приводит к логическому «ложному» результату и существует предложение else, выполняется оператор, связанный с предложением else (оператор 2).

Когда операторы if являются вложенными и присутствуют предложения else, данное else связывается с ближайшим предшествующим оператором if в т ом же блоке.

return expression

Завершает текущую функцию и перемещает управление программой на оператор, следующий за вызывающим оператором. Значение выражения возвращается как результат функции. Выражение может быть переменной массива или литеральным массивом.

Выражение является необязательным, но оно должно начинаться с той же строки, что и оператор return, если он присутствует. Если отсутствует, значение функции равно нулю.

sleep expression

Прервите программу, но оставьте ее в состоянии перезапуска. Выражение является необязательным. Если включена инструкция sleep, она возвращает значение выражения (плюс тег выражения) хост-приложению. Значение и назначение кодов/тегов выхода определяется реализацией; как правило, приложение использует инструкцию сна, чтобы обеспечить облегченную многозадачность нескольких параллельных программ-закладок или для реализации «скрытых» функций.

state ( expression ) automaton :name

Изменяет текущее состояние в указанном автомате. Выражение в круглых скобках является необязательным; если он отсутствует, скобки также должны быть опущены. Имя автомата также не является обязательным при изменении состояния автомата по умолчанию, анонимного; если имя автомата отсутствует, то двоеточие (":") также должно быть опущено.

Ниже приведены два примера безусловного изменения состояния. Первый для автомата по умолчанию:



а второй для конкретного автомата:



Часто изменение состояния зависит от параметров события или состояния автомата в целом. Поскольку условные состояния меняются очень часто, условие может быть в самой инструкции состояния. \* Условие следует за ключевым словом состояние в круглых скобках. Состояние изменится только в том случае, если условие логически «истинно».

Команда состояния вызывает неявный вызов функции входа для указанного состояния, если такая функция входа существует.

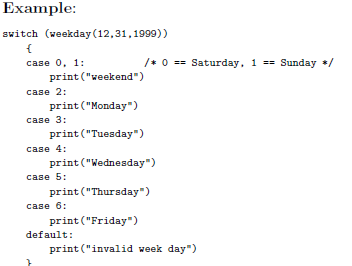


Передает управление различным операторам в теле переключателя в зависимости от значения выражения переключателя. Тело оператора switch представляет собой составной оператор, который содержит ряд «предложений case».

Каждое «предложение case» начинается с ключевого слова case, за которым следует список констант и один оператор. Список констант представляет собой серию выражений, разделенных запятыми, каждое из которых оценивается как постоянное значение. Список констант заканчивается двоеточием. Чтобы указать «диапазон» в списке констант, разделите нижнюю и верхнюю границы диапазона двойной точкой («..»). Пример диапазона: «case 1..9:».

Оператор switch перемещает управление на «предложение case», если значение одного из выражений в списке констант равно результату выражения switch.

«Предложение по умолчанию» состоит из ключевого слова по умолчанию и двоеточия. Предложение по умолчанию является необязательным, но если оно включено, оно должно быть последним предложением в теле переключателя. Оператор switch перемещает управление в «предложение по умолчанию» и выполняется, если ни одно из предложений case не соответствует результату выражения.



while ( expression ) statement

Вычисляет выражение и выполняет инструкцию, если результат выражения логически «истинен». После выполнения оператора управление программой снова возвращается к выражению. Таким образом, оператор выполняется, пока выражение истинно. 

Все директивы должны стоять первыми в строке (им может предшествовать пробел, но не какие-либо другие символы). Все директивы начинаются с символа #, а полная инструкция не может занимать более одной строки.’

#assert constant expression  
Выдает ошибку времени компиляции, если предоставленное константное выражение оценивается как ноль. Директива #assert наиболее полезна для защиты от определенных реализацией конструкций, от которых может зависеть программа, таких как размер ячейки в битах или количество упакованных символов в ячейке.

#define pattern replacement

Определяет макрос замены текста. Шаблон сопоставляется со всеми строками, считанными из исходных файлов; соответствующие разделы заменяются заменяющими текстами. Шаблон и замещающие тексты могут содержать параметры, обозначаемые от «%0» до «%9». Подробности и примеры замены текста см. на стр. 93.

#emit opcode, parameters

Директива #emit служит встроенным ассемблером. В настоящее время он используется только для тестирования абстрактной машины.

#endinput & #endscript

Закрывает текущий файл и тем самым игнорирует весь текст после директивы #endinput. Директива #endscript является синонимом #endinput.

#error

message: сигнализирует об «ошибке пользователя» указанным сообщением. Пользовательские ошибки являются фатальными ошибками и служат той же цели, что и директива #assert.

#file name

Изменяет имя текущего файла. Эта директива неявно используется текстовым препроцессором; обычно нет необходимости явно задавать имя файла.

#if constant expression, #elseif, #else, #endif

Части программы могут анализироваться или игнорироваться в зависимости от определенных условий. Парсер пешки (компилятор или интерпретатор) генерирует код только для тех частей, для которых условие истинно.

За директивой #if должно следовать константное выражение. Чтобы проверить, определена ли переменная или константа, используйте определенный оператор.

За исходной директивой #if может следовать ноль или более директив #elseif. Эти блоки пропускаются, если какой-либо из предыдущих блоков #if или #elseif был проанализирован (т. е. не пропущен). Как и в случае директивы #if, выражение-константа должно следовать за выражением #elseif.

#else заставляет синтаксический анализатор пропускать все строки до #endif, если предыдущая #if или любая из предыдущих директив #elseif были «истинными», и анализирует эти строки, если все предыдущие блоки были пропущены. Директиву #else можно опустить; если он присутствует, может быть только один #else, связанный с каждым #if.

Директива #endif завершает часть программы, которая анализируется условно. Условные директивы могут быть вложенными, и каждая директива #if должна заканчиваться директивой #endif.

#include filename or <filename>

Вставляет содержимое указанного файла в текущую позицию в текущем файле. Имя файла в угловых скобках («<» и «>») относится к системному файлу; анализатор пешки (компилятор или интерпретатор) будет искать такие файлы только в предварительно заданном списке каталогов, а не в «текущем» каталоге. Имена файлов, которые не заключены в кавычки или заключены в двойные кавычки, являются обычными включаемыми файлами, для которых синтаксический анализатор пешки сначала будет искать в правильном каталоге.

Парсер пешки сначала пытается открыть файл с указанным именем. Если это не удается, он пытается добавить расширения «.inc», «.p» и «.pawn» к имени файла (в указанном порядке). Предлагаемое расширение по умолчанию для включаемых файлов — «.inc».

Когда файл может быть успешно открыт, директива #include

определяет константу с именем «\_inc\_» плюс базовое имя

файл (имя файла без пути и расширения) и значение 1. Если константа уже существует, директива #include пропускает открытие и включение файла, тем самым предотвращая двойное включение. Чтобы принудительно выполнить двойное включение, удалите определение константы с помощью директивы #undef перед вторым включением файла.

#line number

Текущий номер строки (в текущем файле). Эта директива неявно используется текстовым препроцессором; обычно нет необходимости явно указывать номер строки

#pragma extra information

«Прагма» — это хук для парсера, чтобы указать дополнительные настройки, такие как уровни предупреждений или дополнительные возможности. Общие #pragma:

#pragma align

Выравнивает следующее объявление по смещению, установленному с помощью параметра компилятора выравнивания. Некоторые (собственные) функции могут работать лучше с параметрами, которые передаются по ссылке, когда они находятся на границах 8, 16 или даже 32 байтов. Требования к выравниванию зависят от хост-приложений.

Помещение строки #pragma align перед объявлением глобальной или статической переменной выравнивает эту переменную по границе, заданной параметром компилятора. Эта #pragma выравнивает только ту переменную, которая следует непосредственно за #pragma. Выравнивание последующих переменных зависит от размера и выравнивания предшествующих переменных. Например, если глобальная переменная массива из 2 ячеек выровнена по 16-байтовой границе, а ячейка занимает 4 байта, следующая глобальная переменная располагается на 8 байт дальше.

Помещение строки #pragma align перед объявлением функции приведет к выравниванию кадра стека этой функции по границе, указанной ранее, в результате чего первая локальная, не «статическая» переменная будет выровнена по этой границе. Выравнивание последующих переменных зависит от размера и выравнивания предшествующих переменных. На практике, чтобы выровнять локальную нестатическую переменную, вы должны выровнять кадр стека функции и объявить эту переменную перед любыми другими переменными.

#pragma amxlimit value

Устанавливает максимальный размер в байтах, до которого может вырасти скомпилированный скрипт. Эта прагма полезна для (встроенных) сред, в которых максимальный размер скрипта привязан к жесткому верхнему пределу.

Если нет настройки объема оперативной памяти для данных и стека (см. прагму amxram), это относится к общим требованиям к памяти; если объем оперативной памяти задан явно, это значение дает только объем памяти, необходимый для кода и статических данных.

#pragma amxram value

Задает максимальные требования к памяти в байтах для данных и стека, которые может иметь скомпилированный сценарий. Это значение полезно для (встроенных) сред, в которых максимальный размер данных сценария ограничен жестким верхним пределом. Особенно в случае, когда скрипт ломбарда запускается из ПЗУ, необходимо установить размеры разделов кода и данных.

#pragma codepage name/value

Парсер пешки может преобразовывать символы в символьных константах и в неупакованных строках в «широкие» символы Unicode/UCS-4. Эта #pragma указывает кодовую страницу, которая должна использоваться для перевода. См. раздел «Интернационализация» на стр. 139 для получения подробной информации и необходимых дополнительных ресурсов для перевода кодовой страницы.

#pragma compress value  
Парсер пешки может записывать сгенерированный P-код в компактной или простой («некомпактной») кодировке. Значение по умолчанию зависит от конфигурации парсера (и, возможно, пользовательских настроек). Эта #pragma позволяет сценаристу переопределить значение по умолчанию и принудительно использовать компактную кодировку (когда значение не равно нулю) или принудительно использовать простую кодировку (когда значение равно нулю). Особенно полезно отключение компактного кодирования (принудительное использование простого кодирования), потому что синтаксический анализатор пешки может быть не в состоянии скомпилировать конкретный сценарий в режиме «компактного кодирования».

#pragma ctrlchar character

Определяет символ, используемый для обозначения начала «escape-последовательности». По умолчанию управляющим символом является «\».



Вы можете указать новое значение управляющего символа как символьную константу (в одинарных кавычках) или как десятичное или шестнадцатеричное значение. Когда вы опускаете значение нового управляющего символа, синтаксический анализатор возвращается к управляющему символу по умолчанию.

#pragma deprecated value

Последующий символ помечен как «устаревший». Если сценарий использует его, анализатор выдает предупреждение.

#pragma dynamic value

Задает размер в ячейках блока памяти для динамических данных (стека и кучи) равным значению, указанному выражением.

Размер блока динамических данных по умолчанию определяется реализацией. Реализация также может увеличивать блок по мере необходимости (подробности см. в документации по основной программе или в «Руководстве по реализации»).

#pragma library name

Задает имя (динамически связанного) модуля расширения, который содержит необходимые собственные функции. Эта #pragma должна отображаться над собственными объявлениями функций, которые являются частью модуля расширения.

Параметр имени библиотеки может отсутствовать, и в этом случае любые последующие объявления собственных функций не будут связаны с каким-либо модулем расширения.

Область действия этой #pragma начинается со строки, в которой она появляется, до конца файла, в котором она появляется. При обычном использовании строка библиотеки #pragma появляется в верхней части включаемого файла, в котором объявляются собственные функции для модуля расширения, а область «ссылки» библиотеки заканчивается в конце этого включаемого файла.

#pragma pack value

Если значение равно нулю, упакованные литеральные строки начинаются с «!» (восклицательный знак + двойная кавычка), а неупакованные литеральные строки - только с двойной кавычки («"»), как описано в этом руководстве на стр. 99. Если значение не- нуля, синтаксис упакованных и неупакованных литеральных строк меняется местами: литеральные строки, начинающиеся с двойной кавычки, упаковываются, а литеральные строки, начинающиеся с «!», распаковываются.

#pragma rational tagname(value)

Включает поддержку рациональных чисел. Имя тега — это имя тега, которым будут обладать рациональные числа; обычно выбирают имена «Float:» или «Fixed:». Наличие значения в круглых скобках после тэга необязательно: если оно опущено, рациональное число сохраняется как значение «с плавающей запятой» в соответствии с нормой IEEE 754; если оно присутствует, рациональное число представляет собой число с фиксированной точностью («масштабированное целое число») с указанным количеством десятичных знаков.

#pragma semicolon value

Если значение равно нулю, точка с запятой не требуется для завершения оператора, если этот оператор является последним в строке. Точки с запятой по-прежнему необходимы для разделения нескольких операторов в одной строке.

Когда точки с запятой необязательны (по умолчанию), постфиксный оператор (один из «++», «--» и «char») не может быть первым токеном в строке, поскольку они будут интерпретироваться как префиксные операторы.

#pragma tabsize value

Количество символов между двумя последовательными позициями табуляции. Значение по умолчанию — 8. Возможно, вам потребуется установить размер табуляции, чтобы избежать предупреждения 217 (свободный отступ), если исходный код имеет отступ попеременно с пробелами и символами табуляции. В качестве альтернативы, установив для #pragma «tabsize» нулевое значение, синтаксический анализатор больше не будет выдавать предупреждение 217.

#pragma unused symbol,. . .

Отмечает именованный символ как «используемый». Как правило, пешка

Отмечает именованный символ как «используемый». Обычно синтаксический анализатор пешек предупреждает о неиспользуемых переменных и неиспользуемых локальных константах. В большинстве ситуаций эти переменные и константы избыточны, и их лучше убрать для ясности кода. Однако, особенно в случае локальных констант, может быть лучше (или необходимо) сохранить определения констант. Затем эта #pragma позволяет пометить символ (переменную или константу) как «используемый» и избежать предупреждения синтаксического анализатора.

#pragma должна стоять после объявления символа, но не обязательно сразу после объявления.

Несколько имен символов могут отображаться в одной прагме #pragma; символы должны быть разделены запятыми.

#section name

Начинает новый раздел для сгенерированного кода. Любые переменные и функции, которые объявлены как «статические», видны только в том разделе, к которому они принадлежат. По умолчанию каждый исходный файл представляет собой отдельный раздел, и в каждом файле есть только один раздел.

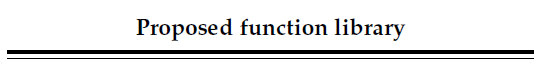
С помощью директивы #section вы можете создать несколько разделов в исходном файле. Имя раздела необязательно, если оно не задано, то в качестве имени раздела используется уникальный идентификатор исходного файла. Любой объявленный раздел автоматически заканчивается в конце файла.

#tryinclude filename or <filename>

Эта директива ведет себя так же, как директива #include, но она не выдает ошибку, когда файл для включения не существует, т. е. пытается включить, но не работает при ошибке.

#undef name

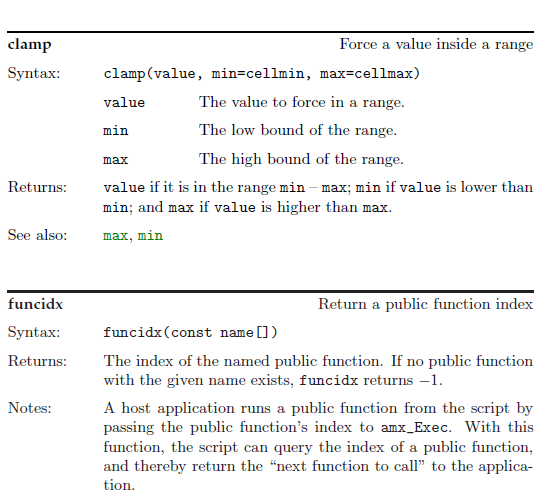
Удаляет макрос подстановки текста или числовую константу, объявленную с помощью const. Параметр «имя» должен быть «префиксом» макроса — буквенно-цифровой частью макроса. Подробности и примеры замены текста см. на стр. 93.

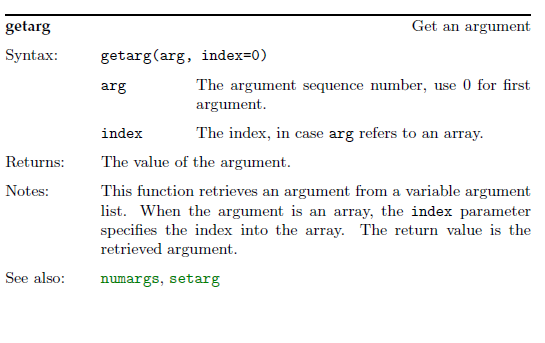


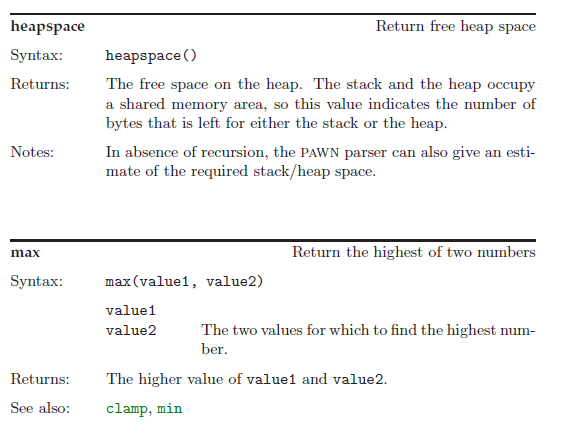
Поскольку pawn предназначен как язык расширения приложения, большинство функций, доступных программам pawn, будут специфичны для основного приложения. Тем не менее, небольшой набор функций может оказаться полезным во многих средах.

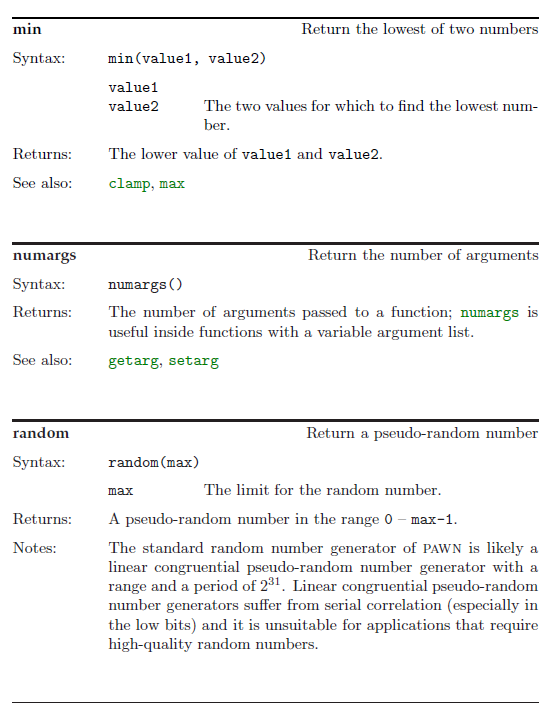
• Основные функции

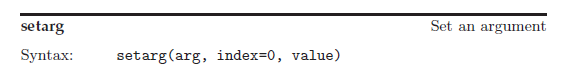
«Основной» модуль состоит из набора функций, поддерживающих сам язык. Несколько функций необходимы для извлечения аргументов из списка переменных аргументов (см. стр. 80).

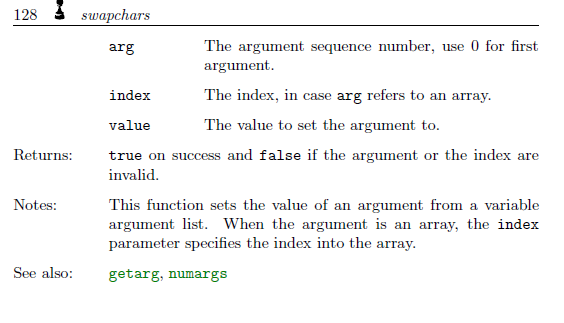


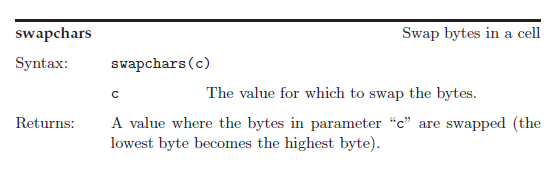


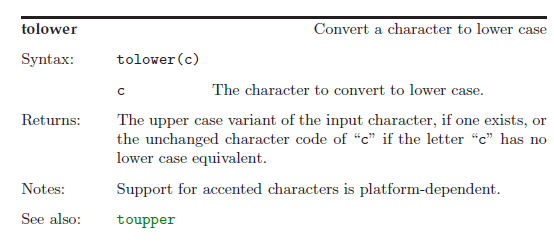


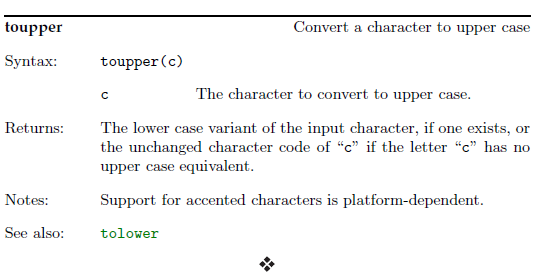












Свойства — это имена или значения общего назначения. Подпрограммы списка свойств поддерживают список этих пар имя/значение, который является общим для всех абстрактных машин. Таким образом, список свойств является способом обмена информацией между параллельными абстрактными машинами.

Все функции «обслуживания собственности» имеют необязательный параметр «id». Вы можете использовать этот параметр, чтобы указать, какой абстрактной машине принадлежит свойство. (Хост-приложение, которое поддерживает параллельные абстрактные машины, обычно предоставляет каждой абстрактной машине уникальный идентификатор.) При запросе

(или удаление) свойства, значение идентификатора, которое вы передаете, сопоставляется со значениями идентификатора списка.

Свойство идентифицируется его «абстрактным идентификатором машины» плюс либо имя, либо значение. Интерфейс на основе имени позволяет вам прикрепить значение (например, дескриптор объекта) к имени по вашему выбору. Интерфейс на основе значений позволяет присоединять строку к числу. Разница между ними в основном заключается в ключе поиска и выходном параметре.

Все функции обслуживания свойств имеют параметры «имя» и «значение». Должна быть заполнена только одна из этой пары. Когда вы задаете значение, функция getproperty сохраняет результат в строковом аргументе, а функция setproperty считывает строку для сохранения из строкового аргумента.

Количество свойств, которые вы можете добавить, ограничено только доступной памятью.

getproperty(id=0, const name[]=“”, value=cellmin, string[]=“”)

Возвращает значение свойства при передаче имени; заполняет строковый аргумент при передаче значения. Строка имени может быть как упакованной, так и неупакованной строкой. Если свойство не существует, эта функция возвращает ноль.

setproperty(id=0, const name[]=“”, value=cellmin, const string[]=“”)

Добавьте новое свойство или измените существующее свойство.

deleteproperty(id=0, const name[]=“”, value=cellmin)

Возвращает значение свойства и впоследствии удаляет его. Если свойство не существует, функция возвращает ноль.

existproperty(id=0, const name[]=“”, value=cellmin)

Возвращает true, если свойство существует, и false в противном случае.

• **Console functions**

Для целей тестирования часто удобны консольные функции, которые считывают пользовательский ввод и выводят строки в прокручиваемом окне или на стандартном дисплее терминала. Не все типы и реализации терминалов могут реализовать все функции — особенно функции, которые очищают экран, устанавливают цвета переднего плана и фона и управляют положением курсора, требуют расширенного управления терминалом.

getchar(echo=true)

Прочитайте один символ с клавиатуры и верните его. Функция может дополнительно отображать символ в окне консоли.

getstring(string[], size=sizeof string, bool

pack=false): прочитать строку с клавиатуры. Функция getstring прекращает чтение, когда либо нажата клавиша ввода, либо достигнута максимальная длина. Максимальная длина указана в ячейках (не в символах) и включает завершающий нулевой символ. Функция может читать как упакованные, так и неупакованные строки; при чтении упакованной строки функция может прочитать больше символов, чем указано в параметре размера, поскольку каждая ячейка содержит несколько символов. Возвращаемое значение — это количество прочитанных символов, исключая завершающий нулевой символ.

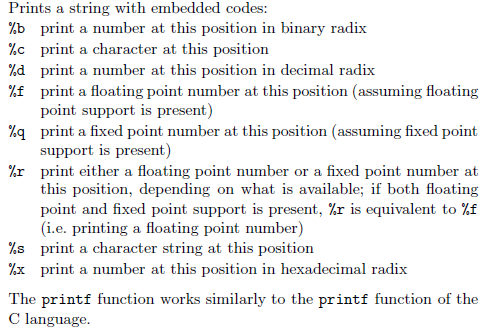
getvalue(base=10, end=‘\r’, ...)

Прочитайте значение (число со знаком) с клавиатуры. Функция getvalue позволяет вам считывать числовую систему счисления от 2 до 36 (базовый параметр) с десятичной системой счисления по умолчанию. По умолчанию ввод завершается, когда пользователь набирает клавишу ввода, но могут быть выбраны одна или несколько различных клавиш (конечный параметр и последующие). В списке завершающих ключей положительное число (например, '\r') отображает ключ и завершает ввод, а отрицательное число завершает ввод без отображения завершающего ключа.

print(const string[], foreground=-1, background=-1)

Выводит на консоль простую строку. Цвета переднего плана и фона могут быть установлены опционально (но учтите, что терминал или хост-приложение могут не поддерживать цвета). Смотрите setattr ниже для списка цветов.

printf(const format[], ...)



clrscr()

Очищает консоль и устанавливает курсор в верхний левый угол.

clreol()

Очищает строку, на которой находится курсор, от позиции курсора до правого края консоли. Эта функция не перемещает курсор.

gotoxy(x=1, y=1)

Устанавливает позицию курсора на консоли. Верхний левый угол находится в (1,1).

setattr(foreground=-1, background=-1)

Устанавливает цвета переднего плана и фона для текста, записываемого на консоль. Когда любой из двух параметров отрицателен (или отсутствует), соответствующая настройка цвета не будет изменена. Значение цвета должно находиться в диапазоне от нуля до семи в соответствии с Escape-последовательностями ANSI, ISO 6429. Предопределенные константы для цветов: черный (0), красный (1), зеленый (2), желтый (3), синий ( 4), пурпурный (5), голубой (6) и белый (7).

• **Date/time functions**

Функции для получения и установки текущей даты и времени, а также «таймер событий» с миллисекундным разрешением описаны в примечаниях по применению под названием «Библиотека функций времени», которые доступны отдельно.

• **File input/output**

Функции для работы с текстовыми и двоичными файлами с прямой поддержкой текстовых файлов UTF-8 описаны в примечаниях по применению, озаглавленных «Библиотека поддержки файлового ввода-вывода», которые доступны отдельно.

• **Fixed point arithmetic**

Модуль десятичной арифметики с фиксированной точкой для пешки описан в примечаниях по применению, озаглавленных «Библиотека поддержки фиксированной точки», которые доступны отдельно.

• **Floating point arithmetic**

Арифметический модуль с плавающей запятой для pawn описан в примечаниях по применению под названием «Библиотека поддержки операций с плавающей запятой», которые доступны отдельно.

• **Process and library call interface**

Функции для запуска и управления внешними приложениями, а также функции для использования библиотек DLL общего назначения или общих библиотек описаны в примечаниях по применению, озаглавленных «Управление процессами и интерфейс внешних функций», которые доступны отдельно.

• **String manipulation**

Общий набор функций манипулирования строками, работающих как с упакованными, так и с неупакованными строками, описан в примечаниях по применению, озаглавленных «Библиотека манипуляций со строками», которые доступны отдельно.



⋄ pawn не имеет механизма типизации C. pawn — это «только целочисленная» разновидность C; здесь нет структур или объединений, а поддержка с плавающей запятой должна быть реализована с помощью определяемых пользователем операторов и с помощью встроенных функций.

⋄ Принятый синтаксис для рациональных чисел более строгий, чем для значений с плавающей запятой в C. Значения, такие как «0,5» и «6». в C допустимы, а в пешке надо писать «0,5» и «6,0» соответственно. В C десятичная точка необязательна, если включена экспонента, поэтому можно написать «2E8»; пешка не принимает заглавную букву «Е» (используйте строчную букву «е») и требует десятичной точки: например. «2.0e8». См. стр. 98 для получения дополнительной информации.

⋄ пешка не предоставляет «указателей». Для передачи аргументов функции по ссылке pawn предоставляет «ссылочный» аргумент (стр. 71). Аргумент «заполнитель» заменяет некоторые варианты использования указателя NULL (стр. 75).

⋄ Числа могут иметь шестнадцатеричную, десятичную или двоичную систему счисления. Восьмеричная система счисления не поддерживается. См. «Константы» на стр. 98. Шестнадцатеричные числа должны начинаться с «0x» (строчный «x»), префикс «0X» недействителен.

⋄ Управляющие последовательности («\n», «\t» и т. д.) такие же, за исключением «\ddd», где «ddd» представляет собой три десятичных цифры вместо восьмеричных цифр, которые используются в C/C++. Обратная косая черта («\») может быть заменена другим символом; см. #pragma ctrlchar на стр. 121 — примечательно, что в предыдущих версиях pawn использовалась каретка («^») в качестве escape-символа.

⋄ Случаи в операторе switch не являются «проваленными». Только одна инструкция может (и должна) следовать за каждой меткой case. Чтобы выполнить несколько инструкций, вы должны использовать составной оператор. Предложение по умолчанию оператора switch должно быть последним предложением оператора switch. Подробнее на стр. 116. В C/C++ переключатель — это «условный переход», аналогичный вычисляемым меткам Фортрана. В пешке переключатель — это структурированное «если».

⋄ Оператор break прерывает только циклы. В C/C++ оператор break также завершает оператор case в операторе switch. Операторы switch в pawn реализованы по-разному (см. стр. 116). ⋄ pawn поддерживает «назначение массива» с тем ограничением, что оба массива должны иметь одинаковый размер. Например, если «a» и «b» являются массивами с 6 ячейками, выражение «a = b» допустимо. Помимо литеральных строк, pawn также поддерживает литеральные массивы, что позволяет использовать выражение «a = {0,1,2,3,4,5}» (где «a» — это переменная массива с 6 элементами).

⋄ char — это оператор, а не тип. См. стр. 111 и советы на стр. 137.

⋄defined — это оператор, а не директива препроцессора. Определенный оператор в pawn работает с константами (с const и enum), глобальными переменными, локальными переменными и функциями.

⋄ Оператор sizeof возвращает размер переменной в «элементах», а не в «байтах». Элемент может быть ячейкой или подмассивом. Подробности см. на стр. 110.

⋄ Пустая инструкция — это пустой составной блок, а не точка с запятой (стр. 113). Эта модификация позволяет избежать частой ошибки.

⋄ Директивы компилятора отличаются от команд препроцессора C. Примечательно, что директива #define несовместима с директивой C/C++, а #ifdef и #ifndef заменены более общей директивой #if (см. «Директивы» на стр. 118). Чтобы создать числовые константы, см. также стр. 101; для создания строковых констант см. также стр. 93.

⋄ Подстановки текста (макросы препроцессора; см. директиву #define) не сопоставляются между строками. То есть текст, который вы хотите сопоставить и заменить макросом #define, должен располагаться в одной строке. Определение макроса #define также должно располагаться в одной строке.

⋄ Направление усечения для оператора «/» всегда направлено к меньшему значению, где -2 меньше -1. Оператор «%» всегда дает положительный результат, независимо от знаков операндов. См. стр. 105.

⋄ Нет унарного оператора «+», который в любом случае является оператором «без операции».

⋄ Три побитовых оператора имеют другой приоритет, чем в C. Уровни приоритета «&», «^» и | операторы выше, чем реляционные операторы (Деннис Ритчи объяснил, что эти операторы получили свои низкие уровни приоритета в C, потому что ранние компиляторы C еще не имели логических операторов «&&» и ||, поэтому вместо них использовались побитовые «&» и | ).

⋄ Ключевое слово «extern» не существует в пешке; текущая реализация компилятора не имеет «фазы компоновки». Чтобы создать программу из нескольких исходных файлов, добавьте все исходные файлы в командную строку компилятора или создайте один основной файл сценария проекта, который #include включает все остальные исходные файлы. Компилятор pawn может оптимизировать функции и глобальные переменные, которые вы не используете. Подробности см. на стр. 62 и 84.

⋄ В большинстве случаев предварительное объявление функций (т. е. прототипов) не требуется. pawn — это двухпроходный компилятор, он увидит все функции на первом проходе и использует их на втором проходе. Однако пользовательские операторы должны быть объявлены перед использованием. Упреждающие объявления, если они предоставлены, должны точно совпадать с определением функции, имена параметров не могут быть опущены в прототипе или отличаться от определения функции. pawn заботится об именах параметров в прототипах из-за возможности «именованных параметров». Один использует прототипы для вызова предварительно объявленных функций. Делая это с именованными параметрами, компилятор должен уже знать имена параметров (и их положение в списке параметров). В результате имена параметров в прототипе должны совпадать с именами в определении.



• **Working with characters and strings**

Строки могут быть упакованы или распакованы. В упакованном формате каждая ячейка обычно содержит четыре символа (в обычных реализациях ячейка является 32-битной, а символ — 8-битной). В этой конфигурации первый символ в «пакете» из четырех является старшим байтом ячейки, а четвертый символ находится в младшем байте каждой ячейки. Строка должна храниться в массиве. Для неупакованной строки массив должен быть достаточно большим, чтобы вместить все символы строки плюс завершающую нулевую ячейку. То есть в приведенном ниже примере переменная usstring определена как имеющая пять ячеек, чего как раз достаточно, чтобы содержать строку, которой она инициализируется:



В упакованной строке каждая ячейка содержит несколько символов, а строка заканчивается нулевым символом. Оператор char помогает объявить размер массива, чтобы он содержал необходимое количество символов. В приведенном ниже примере будет выделено достаточно ячеек для хранения пяти упакованных символов. В типичной реализации в массиве будет две ячейки.



Другими словами, операторы char делят свой левый операнд на количество байтов, которое помещается в ячейку, и округляют его в большую сторону. Опять же, в типичной реализации это означает деление на четыре и округление в большую сторону.

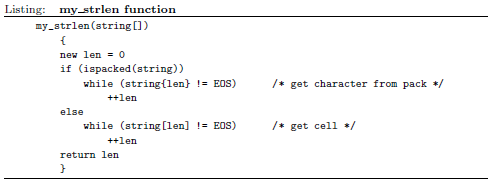
Вы можете создавать подпрограммы, которые работают со строками как в упакованном, так и в неупакованном формате. Чтобы узнать, упакована строка или нет, посмотрите на первую ячейку строки. Если его значение отрицательное или превышает максимально возможное значение неупакованного символа, строка является упакованной строкой. В противном случае это неупакованная строка.

Фрагмент кода ниже возвращает true, если входная строка упакована, и false в противном случае:

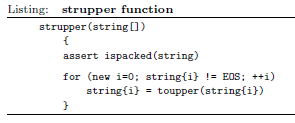


Неупакованная строка заканчивается полной нулевой ячейкой. Конец упакованной строки помечается только нулевым символом. Поскольку в 32-битной ячейке может быть до четырех символов, этот нулевой символ может находиться в любой из четырех позиций в «пакете». Оператор {} извлекает символ из ячейки массива. По сути, используется оператор индекса ячейки («[ ]») для неупакованных строк и оператор индекса символа («{ }») для работы с упакованными строками.

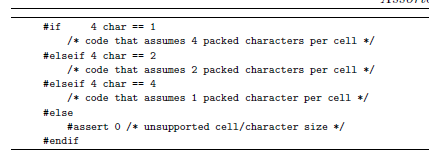
Например, подпрограмма, которая возвращает длину в символах любой строки (упакованной или неупакованной):



Если вы заставляете функции работать исключительно с упакованными или неупакованными строками, рекомендуется добавить утверждение для принудительного выполнения этого условия:

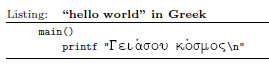


Хотя в предыдущих абзацах мы предполагали, что ячейка имеет ширину 32 бита, а символ — 8 бит, на это не следует полагаться. Размер ячейки определяется реализацией; максимальное и минимальное значения находятся в предопределенных константах cellmax и cellmin. Существуют аналогичные предопределенные константы для символов. Однако можно с уверенностью предположить, что и размер символа в байтах, и размер ячейки в байтах являются степенью двойки. Оператор char позволяет определить, сколько упакованных символов помещается в ячейку. Например:



• **Internationalization**

Примеры программирования в этом руководстве используют английский язык для всего вывода (подсказки, сообщения и т. д.) и набор латинских символов. Это не обязательно так; можно, например, изменить первую программу «hello world» на странице 3 так:



pawn имеет базовую поддержку нелатинских алфавитов, но принимает только нелатинские символы в строках и символьных константах. Язык пешек требует, чтобы все ключевые слова и символы (имена функций, переменных, тегов и других элементов) были закодированы в наборе символов ascii.

Для языков, требуемый набор символов которых относительно невелик, распространенным решением является использование 8-битного расширенного набора символов ascii (набор символов ascii является 7-битным и содержит 128 символов). Верхние 128 кодов расширенного набора содержат глифы, характерные для языка. Для западноевропейских языков хорошо известен набор символов «Latin-1», который стандартизирован как ISO 8859-1 — этот же набор также имеет название «кодовая страница 1252», по крайней мере, для Microsoft Windows. Кодовые страницы были определены. для многих языков; например, ISO 8859-2 («Latin-2») содержит глифы, используемые в Центральной и Восточной Европе, а ISO 8859-7 содержит греческий алфавит в верхней половине расширенного набора ascii.

К сожалению, выбор кодовой страницы может сбивать с толку, поскольку производители операционных систем обычно создают свои собственные кодовые страницы независимо от того, что уже существует. В результате для большинства наборов символов существует несколько несовместимых кодовых страниц. Например, кодовая страница 1253 для MicrosoftWindows также кодирует греческий алфавит, но несовместима с ISO 8859-7. При написании текстов на греческом языке теперь важно проверять, какая кодировка используется, потому что многие приложения Microsoft Windows поддерживают обе.



Когда набор символов для языка превышает 256 глифов, кодовой страницы недостаточно. Традиционно метод кодовой страницы был расширен за счет резервирования специальных кодов «сдвига» в базовом наборе символов, которые переключаются на новый набор глифов. Затем следующий символ указывает на конкретный глиф. По сути, глиф теперь идентифицируется 2-байтовым индексом. С другой стороны, некоторые символы (особенно 7-битный набор ascii) по-прежнему могут обозначаться одним байтом. Стандарт «Shift-JIS» для японского набора символов является примером кодирования переменной длины.

Кодовые страницы становятся проблематичными при обмене документами или данными с людьми в регионах, где используется другая кодовая страница, или при использовании разных языков в одном и том же документе. Кодовые страницы, в которых используются символы «сдвига», еще больше усложняют дело, поскольку обработка текста теперь должна учитывать, что символ может занимать либо один, либо два байта. Просмотр строки справа налево может даже стать невозможным, так как байт может указывать либо на глиф из базового набора («несмещенный»), либо на глиф из сдвинутого набора — в последнем случае предшествующий байт указывает сдвиг установлен, но значение предшествующего символа зависит от предшествующего символа.

Стандарт ISO/IEC 10646 «Универсальный набор символов» (UCS) имеет амбициозную цель — в конечном итоге включить все символы, используемые во всех письменных языках мира, с использованием 31-битного набора символов. Это решает обе проблемы, связанные с кодовыми страницами и «сдвинутыми» наборами символов. Однако организация ISO/IEC не смогла вовремя разработать стандарт, и поэтому консорциум, состоящий в основном из американских производителей программного обеспечения, начал параллельную работу над упрощенным 16-битным набором символов под названием «Юникод». Обоснование Unicode заключалось в том, что он будет кодировать абстрактные символы, а не глифы, и что, следовательно, 65 536 будет достаточно. † Однако на практике Unicode действительно кодирует глифы, и вскоре после его появления стало очевидно, что 65 536 кодовых точек не будут достаточно. Чтобы противостоять этому, более поздние версии Unicode были расширены несколькими «плоскостями» и специальными кодами, которые выбирают плоскость. Комбинация селектора плоскости и указателя кода внутри этой плоскости называется «суррогатной парой». Первые 65 536 кодовых точек находятся в «базовой многоязычной плоскости» (BMP), и символы в этом наборе не нуждаются в селекторе плоскости.



По сути, введение суррогатных пар в стандарт Unicode эквивалентно кодам сдвига более ранних наборов символов и несет в себе некоторые проблемы, которые Unicode должен был решить. Кодирование UCS-4 по ISO/IEC 10646 не имеет/не требует суррогатных пар.

Поддержка Unicode/UCS-4 в (хост) приложениях и операционных системах появилась двумя разными способами: либо внутреннее представление символов является многобайтовым (обычно 16-битным или 2-байтовым), либо приложение хранит строки внутри в формате UTF-8, и эти строки преобразуются в правильные глифы только при их отображении или печати. Последние версии MicrosoftWindows используют Unicode внутри; Операционная система Plan-9 впервые использовала кодировку UTF-8, которая сейчас широко используется в Unix/Linux. Преимущество кодировки UTF-8 как внутреннего представления заключается в том, что физически это 8-битная кодировка и, следовательно, совместима почти со всеми существующими базами данных, форматами файлов и библиотеками. Это устраняет необходимость в двойных точках входа для функций, принимающих строковые параметры, как в случае с Microsoft Windows, где многие функции существуют в версиях «A»nsi и «W»ide. Недостатком UTF-8 является то, что это кодировка переменной длины, и поэтому многие строковые операции в памяти неуклюжи (и неэффективны). Тем не менее, с появлением суррогатных пар Unicode теперь также стал кодировкой переменной длины.

Язык пешек требует, чтобы его ключевые слова и имена символов были в ascii, и он допускает символы, отличные от ascii, в строках. Есть пять способов, которыми хост-приложение может поддерживать символы, отличные от ascii, в строках и символьных литералах:

1 Поддержка кодовых страниц: в этой стратегии вся сложность выбора правильных глифов и шрифтов делегируется хост-приложению. Поддержка кодовых страниц основана на файлах сопоставления кодовых страниц с файловым форматом «таблиц перекрестного сопоставления», распространяемых консорциумом Unicode.

2 Поддержите Unicode или UCS-4 и позвольте компилятору Pawn конвертировать сценарии, написанные с использованием кодовой страницы, в «широкие» символы: для этой стратегии вам необходимо установить кодовую страницу #pragma или использовать эквивалентную опцию компилятора. Компилятор будет корректно переводить только символы в неупакованных строках.

3 Поддержите Unicode или UCS-4 и позвольте компилятору pawn конвертировать сценарии, закодированные в UTF-8, в «расширенные» символы: когда исходный файл для компилятора pawn находится в кодировке UTF-8, компилятор расширяет символы до Unicode/UCS-4 в нераспакованных строках. 4. Поддержка внутренней кодировки UTF-8 (в хост-приложении) и запись исходного файла также в UTF-8: теперь все строки должны быть упакованы, чтобы компилятор не преобразовывал их.

Как видите, для большинства стратегий интернационализации хост-приложение должно поддерживать Unicode или UCS-4. Кстати, компилятор pawn не генерирует суррогатные пары Unicode. Если необходимы символы за пределами BMP, а хост-приложение (или операционная система) не поддерживает полную кодировку UCS 4, хост-приложение должно разделить 32-битную символьную ячейку, предоставленную компилятором пешки, на суррогатную пару.

Компилятор pawn принимает исходный файл как текстовый файл в кодировке UTF-8 — см. стр. 169. Когда исходный файл имеет кодировку UTF-8, «расширенные» символы в неупакованной строке сохраняются как многобайтовые Unicode/UCS-4. персонажи; широкие символы в упакованной строке остаются в кодировке UTF-8. Для записи исходных файлов в кодировке UTF-8 нужен, конечно же, редактор (программистский), поддерживающий UTF-8. Перевод кодовой страницы не применяется к файлам в кодировке UTF-8.

Для случайного символа Unicode в литеральной строке альтернативой является использование управляющей последовательности. Поскольку таблицы символов Unicode обычно документируются с помощью шестнадцатеричных индексов глифов, \xhhh; последовательность, вероятно, является более удобной спецификацией случайного символа Unicode. Например, управляющая последовательность «\x2209» означает символ «6∈».

Интернационализация — это гораздо больше, чем просто базовая поддержка расширенных наборов символов, таких как форматирование полей даты и времени, порядок чтения (слева направо или справа налево) и перевод системных сообщений, зависящий от локали. Инструментарий Pawn делегирует эти проблемы хост-приложению.

• **Working with tags**

Система имен тегов была изобретена, чтобы добавить механизм «проверки использования» в pawn. Тег обозначает «назначение» значения или переменной, и компилятор пешки выдает диагностическое сообщение, когда тег выражения не соответствует требуемому тегу для контекста выражения.

Многие современные компьютерные языки предлагают типы переменных, где тип определяет структуру памяти и назначение переменной. Затем язык программирования проверяет эквивалентность типов; язык паскаль очень строго проверяет равенство типов, в то время как язык программирования Си более щадящий. Язык пешек не имеет типов: все переменные имеют размер и расположение ячейки, хотя битовые представления в ячейке могут зависеть от назначения переменной. В итоге:

⋄ тип определяет структуру памяти и диапазон переменных и результатов функции

⋄ тэг обозначает назначение переменных, констант и результатов функций

Теги в залоге в основном необязательны. Программа, которая была «укреплена» именами тегов в объявлениях переменных и констант, будет функционировать одинаково, когда все имена тегов будут удалены. Одно исключение составляют определяемые пользователем операторы: компилятор пешки использует теги операндов для выбора между любыми определяемыми пользователем операторами и стандартным оператором. Фрагмент ниже объявляет три переменные и выполняет три назначения, два из которых выдают диагностическое сообщение «несоответствие тегов»:



Первое присваивание вызывает диагностику «несоответствие тегов», поскольку оно присваивает переменную с тегом «оранжевый» переменной с тегом «яблоко». Второе присваивание помещает немаркированное значение x в тегированную переменную, что снова вызывает диагностику. Когда нетегированная переменная находится слева от оператора присваивания, как в третьем присваивании, предупреждения или сообщения об ошибке нет. Поскольку переменная x не имеет тега, она может принимать значение любого слабого тега.

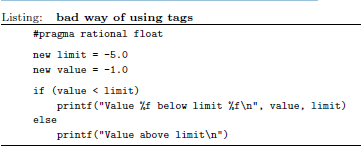
Тот же механизм применяется к передаче переменных или выражений функциям в качестве операндов функций — пример см. на стр. 78. Короче говоря, когда функция ожидает определенное имя тега в аргументе, вы должны передать этой функции выражение/переменную с соответствующим тегом; но если функция ожидает немаркированный аргумент, вы можете передать аргументы с любым слабым тегом.

Иногда необходимо временно изменить тег выражения. Например, с объявлениями предыдущего фрагмента кода, если вы хотите сравнить яблоки с апельсинами (недавние исследования показывают, что сравнение яблок с апельсинами не так абсурдно, как принято считать), вы можете использовать:



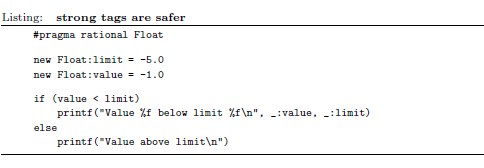
Тестовое выражение оператора if (в скобках) сравнивает переменную valencia с переменной elstar. Чтобы избежать диагностики «несоответствие тегов», он помещает тег override apple: на valencia — после этого выражения слева и справа от оператора > имеют одно и то же имя тега: «apple:». Вторая строка, назначение elstar для valencia, переопределяет имя тега elstar или Orange: перед назначением. В назначении вы не можете переопределить имя тега назначения; т. е. левая рука оператора =. Написание «apple:valencia = elstar» является ошибкой. В назначении valencia является «lvalue», и вы не можете переопределить имя тега lvalue.

Как показано ранее, когда левая часть присваивания содержит непомеченную переменную, выражение в правой части может иметь любое слабое имя тега. При использовании в качестве lvalue непомеченная переменная совместима со всеми именами слабых тегов. Или, скорее, слабый тег молча отбрасывается, когда он присваивается непомеченной переменной или когда он передается функции, которая ожидает непомеченный аргумент. Когда имя тега указывает на битовый шаблон ячейки, удаление слабого тега может скрыть ошибки. Например, в приведенном ниже фрагменте есть ошибка, которая не очевидна сразу:



Через «#pragma рациональность» все рациональные числа получают имя тега «плавающее», и эти числа кодируются в 4-байтовом формате IEEE 754. Фрагмент объявляет две переменные, limit и value, обе из которых не имеют тегов (это ошибка). Хотя литеральные значения -5.0 и -1.0 неявно помечены тегом float:, этот слабый тег автоматически отбрасывается, когда значения присваиваются пределу и значению немаркированных символов. Теперь оператор if сравнивает значение с пределом как целые числа, используя встроенный стандартный оператор < (определяемый пользователем оператор был бы более подходящим для сравнения двух значений, закодированных по стандарту IEEE 754). При запуске этот фрагмент кода сообщает нам, что «Значение -1.000000 ниже предела -5.000000» — что, конечно, неверно.

Чтобы такие тонкие ошибки не остались незамеченными, следует использовать сильные теги. Строгий тег — это просто имя тега, начинающееся с заглавной буквы, например Float: вместо float:. Сильный тег никогда не удаляется автоматически, но его можно явно переопределить. Ниже приведен модифицированный фрагмент кода с предлагаемыми адаптациями:



Если забыть имя тега Float: в объявлении переменных limit или value, немедленно выдается диагностика «несоответствие тегов», потому что литеральные значения -5.0 и -1.0 теперь имеют строгое имя тега.

printf — это функция общего назначения, которая может печатать строки и значения в различных форматах. Чтобы быть универсальной, printf принимает аргументы с любым слабым именем тега, будь то яблоко:, апельсин: или что-то еще. Функция printf делает это, принимая аргументы без тегов. Слабые теги отбрасываются, когда ожидается аргумент без тегов. Однако сильные теги никогда не удаляются, и в приведенном выше фрагменте (в котором используется исходное определение printf) мне нужно было поместить переопределение пустого тега «\_:» перед значением переменных и ограничением в первом вызове printf.

Существует альтернатива выражениям отмены тегов со строгими именами тегов в функциях общего назначения: скорректируйте определение функции так, чтобы оно принимало как все слабые теги, так и выборочный набор имен сильных тегов. Язык пешек поддерживает несколько имен тегов для каждого аргумента функции. Исходное определение printf (из файла console.inc):

При добавлении как тега Float:, так и пустого тега перед многоточием («...»), printf будет принимать аргументы с именем тега Float:, аргументы без имени тега и аргументы со слабым именем тега. Чтобы указать несколько имен тегов, заключите все имена тегов без последнего двоеточия между фигурными скобками и запятой, разделяющей имена тегов (см. пример ниже). Необходимо добавить пустую спецификацию тега в список имен тегов, потому что в противном случае printf будет принимать только аргументы с именем тега Float:. Ниже приведено новое определение функции printf:



Множественные теги позволяют вам написать одну функцию, которая принимает ячейки с точно указанным подмножеством тегов (сильных и/или слабых). В то время как аргумент функции может принимать фактические аргументы с разными тегами, переменная может иметь только один тег, а формальный аргумент функции — это локальная переменная в теле функции. При наличии множественных тегов аргумент формальной функции принимает тег, указанный первым.

Иногда вам может понадобиться проверить, какой тег имеет фактический аргумент функции, когда аргумент принимает теги во множественном числе. Проверка тега формального аргумента (в теле функции) бесполезна, потому что он всегда будет иметь первый тег в списке тегов в объявлении аргумента функции. Вы можете проверить тег фактического аргумента, добавив в функцию дополнительный аргумент и установив его значение по умолчанию равным «tagof» рассматриваемого аргумента. Подобно оператору sizeof, оператор tagof имеет особое значение, когда он применяется к значению по умолчанию аргумента функции: выражение оценивается в точке вызова функции, а не в определении функции. Это означает, что «значение по умолчанию» аргумента функции — это фактический тег параметра, переданного в функцию.

Внутри тела функции вы можете сравнить тег с известными тегами, опять же, используя оператор tagof.

• **Concatenating lines**

pawn — язык свободного формата, но директивы парсера должны располагаться в одной строке. Строки также не могут занимать несколько строк. Когда это неудобно, вы можете использовать символ обратной косой черты («\») в конце строки, чтобы «склеить» эту строку со следующей строкой.

Например:



Вы также используете символ конкатенации, чтобы разрезать длинные литеральные строки на несколько строк. Обратите внимание, что «\» поглощает все конечные пробелы, которые идут после него, и начальные пробелы в следующей строке. В приведенном ниже примере выводится «Hello world» с одним пробелом между двумя словами (поскольку между «Hello» и обратной косой чертой есть пробел):



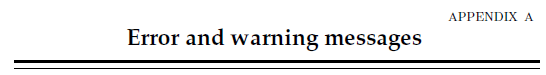
• **A program that generates its own source code**

Странным, слегка академическим критерием количественной оценки «выразительности» языка программирования является размер наименьшей программы, которая при выполнении повторно генерирует свой собственный исходный код. Смысл этого критерия в том, что чем короче самогенерирующаяся программа, тем более гибким и выразительным должен быть язык. Программы такого рода были созданы для многих языков программирования — иногда удивительно малых, как для языков, имеющих встроенные рефлексивные возможности.

Самогенерирующиеся программы называются «кинами» в честь философа Уилларда Ван Ормана Куайна, который писал самогенерирующиеся фразы на естественном языке. Творчество Ван Ормана Куайна стало широко известно благодаря книгам Дугласа Хофштадтера «Гёдель, Эшер, Бах» и «Метамагические темы».

В приведенном ниже примере фигурирует пешка; он создан по образцу знаменитого квина «C» (существует множество вариаций). В 77 символах это одна из самых маленьких версий для класса императивных языков программирования, а размер можно уменьшить до 73 символов, удалив четыре пробела, которые были оставлены для удобства чтения.





Пропустил, есть в pawno Book

Многие приложения, в которые встроен язык сценариев pawn, используют автономный компилятор, который поставляется с набором инструментов pawn. Компилятор pawn — это утилита командной строки, что означает, что вы должны запускать его из «консольного окна», терминала/оболочки или «окна DOS» (в зависимости от того, как ваша операционная система его называет).

• **Usage**

Предполагая, что компилятор pawn для командной строки называется «pawncc» (Unix/Linux) или «pawncc.exe» (DOS/Windows), синтаксис командной строки следующий:



Имя входного файла — любое допустимое имя файла. Если расширение не указано, предполагается «.pawn» или «.p». Компилятор создает выходной файл с тем же именем, что и входной файл, и расширением «.amx».

После перехода в каталог с примерами программ команда:



должен скомпилировать самый первый пример «hello world» (стр. 3). Должен, потому что команда подразумевает, что:

⋄ операционная система может найти программу «pawncc» — возможно, вам потребуется добавить ее в путь поиска;

⋄ компилятор pawn может определить свое собственное местоположение в файловой системе, чтобы он мог найти включаемые файлы — некоторые операционные системы не поддерживают это и требуют, чтобы вы использовали параметр -i (см. ниже).

• Входной файл

Входной файл для компилятора pawn, файл «исходного кода» для скрипта/программы, должен быть текстовым файлом. Все зарезервированные слова и все имена символов (имена переменных, функций, символических констант, тегов и т. д.) должны использовать набор символов ascii. Литеральные строки, т. е. текст в кавычках, могут быть в расширенном формате ascii, например, один из наборов, стандартизированных в норме ISO 8859 — ISO 8859-1 — это хорошо известный набор «Latin 1».

Компилятор pawn также поддерживает текстовые файлы в кодировке UTF-8, что практично в среде, основанной на Unicode или UCS-4. Компилятор pawn распознает символы в кодировке UTF-8 только внутри неупакованных строк и символьных констант. Компилятор строго интерпретирует правила синтаксиса для файлов UTF-8; несоответствующие файлы UTF-8 не распознаются. Входной файл может иметь, но не требует подписи «Знак порядка байтов»; компилятор распознает формат UTF-8 на основе содержимого файла.

• **Options**

Параметры начинаются с тире («-») или, в Microsoft Windows и DOS, с косой черты («/»). Другими словами, все платформы принимают параметр, записанный как «-a» (о назначении этого параметра см. ниже), а платформы DOS/Windows принимают «/a» в качестве альтернативного способа написания «-a». Все варианты должны быть разделены хотя бы одним пробелом.

Многие параметры принимают значение, которое иногда является обязательным. Значение может быть отделено от буквы опции двоеточием или знаком равенства («:» и «=» соответственно), либо значение может быть приклеено к букве опции. Таким образом, три эквивалентных параметра для установки уровня отладки равным двум:

⋄ -d2

⋄ -d:2

⋄ -d=2

Варианты:

-a Ассемблер: создать текстовый файл с псевдоассемблером

код для пешечной абстрактной машины вместо двоичного кода.

-C+/- Компактная кодировка бинарного файла, уменьшающая размер

выходной файл обычно меньше половины исходного размера. Использовать

-C+, чтобы включить его, и -C-, чтобы вернуться к «обычной» кодировке.

опция -C (без суффикса + или -) переключает текущую настройку.

-cname Кодовая страница: установить кодовую страницу для перевода исходного файла из расширенного ASCII в Unicode/UCS-4. По умолчанию нет перевода. Параметр имени может указывать полный путь к «файлу сопоставления» или просто идентификатор кодовой страницы — в последнем случае компилятор добавляет к идентификатору префикс с буквами «cp», добавляет расширение «.txt» и загружает сопоставление файл из системного каталога.

-Dpath

Каталог: «активный» каталог, в котором компилятор должен искать свои входные файлы и сохранять выходные файлы. Этот параметр поддерживается не на каждой платформе. Чтобы проверить, поддерживает ли компилятор pawn эту опцию, запустите компилятор без какой-либо опции или имени файла в командной строке. Затем компилятор перечислит синтаксис использования и все доступные параметры в алфавитном порядке. Если переключатель -D отсутствует, опция недоступна.

-dlevel

Уровень отладки: 0 = нет, 1 = только проверка границ и утверждения, 2 = полная символьная информация, 3 = полная символьная информация и отключенные оптимизации (то же, что и комбинация -d2 и -O0).

Когда уровень отладки равен 2 или 3, компилятор pawn также печатает предполагаемое количество пространства в стеке/куче, необходимое для программы.

-efilename

Файл ошибок: задайте имя файла, в который компилятор должен записывать любые предупреждения и сообщения об ошибках; когда установлено, нет вывода на экран.

-Hvalue

«HWND» (только версия для Microsoft Windows): компилятор может дополнительно отправить сообщение указанному дескриптору окна после завершения генерации P-кода. Хост-приложения, которые вызывают компилятор пешки, могут дождаться прихода этого сообщения или сообщить пользователю о завершении компиляции.

Номер сообщения, отправляемого окну, создается с помощью функции RegisterWindowMessage Microsoft Windows SDK с использованием имени «PawnNotify». wParam сообщения содержит код возврата компилятора: 0 = успех, 1 = предупреждения, 2 = ошибки (плюс, возможно, предупреждения), 3 = компиляция прервана пользователем.

-ipathname

Включить путь: укажите путь, по которому компилятор может найти включаемые файлы. Этот параметр может появляться несколько раз в командной строке, чтобы вы могли установить несколько путей включения.

-l

Листинг: выполнять только шаги чтения и предварительной обработки файла; например, для проверки эффектов раскрытия макросов и условно скомпилированных/пропущенных разделов.

-Olevel

Уровень оптимизации: 0 = нет оптимизации; 1 = только оптимизации, совместимые с JIT (JIT = компилятор «Just In Time», высокопроизводительная абстрактная машина); 2 = полная оптимизация.

-ofilename

Выходной файл: задайте имя и путь к двоичному выходному файлу.

-pfilename

Файл префикса: имя «файла префикса», это файл, который анализируется перед входным файлом (как своего рода неявное «включаемый файл»). При использовании этот параметр переопределяет включаемый файл по умолчанию «default.inc». Параметр -p сам по себе (без имени файла) отключает обработку любого неявного включаемого файла.

-rfilename Отчет: включите создание отчета и при необходимости задайте имя файла, в который будет записана извлеченная документация и отчет о перекрестных ссылках.

Отчет в формате «XML». Параметр имени файла является необязательным; если не указано, файл отчета имеет то же имя, что и входной файл, с расширением «.XML».

-Svalue

Размер стека: размер стека и кучи в ячейках.

-svalue

Skip count: количество строк, которые нужно пропустить во входном файле перед началом компиляции; например, чтобы пропустить «заголовок» в исходном файле, который не имеет допустимого синтаксиса пешки.

-tvalue

размер табуляции: количество пробелов, используемых для символа табуляции. При нулевом значении (т. е. параметр -t0) компилятор больше не будет выдавать предупреждение 217 (слабый отступ).

-vvalue

Verbose: отображать информационные сообщения во время компиляции. Значение может быть 0 (ноль) для «тихой» компиляции, 1 (единица) для обычного вывода и 2 для отчета об использовании кода/данных/стека.

-wvalue+/-

Управление предупреждением: номер предупреждения после «-w» включается или отключается в зависимости от того, стоит ли за номером «+» или «-». Когда «+» или «-» отсутствует, статус предупреждения переключается. Например, -w225- отключает предупреждение о «недоступном коде», -w225+ включает его, а -w225 переключает между включенным/отключенным.

Только предупреждения могут быть отключены (ошибки и фатальные ошибки не могут быть отключены). По умолчанию все предупреждения включены.

-Xvalue

Ограничение для абстрактной машины: максимальные требования к памяти, которые может иметь скомпилированный скрипт, в байтах. Это значение полезно для (встроенных) сред, в которых максимальный размер скрипта ограничен жестким верхним пределом. Если нет настройки объема оперативной памяти для данных и стека, это относится к общим требованиям к памяти; если объем оперативной памяти задан явно, это значение дает только объем памяти, необходимый для кода и статических данных.

-XDvalue

Предел оперативной памяти для абстрактной машины: максимальные требования к памяти для данных и стека, которые может иметь скомпилированный скрипт, в байтах. Это значение полезно для (встроенных) сред, в которых максимальный размер данных сценария ограничен жестким верхним пределом. Особенно в случае, когда скрипт pawn запускается из ПЗУ, необходимо установить размеры разделов кода и данных.

-\

Управляющие символы начинаются с «\» (для сходства с C, C++ и Java).

-^

Управляющие символы начинаются с «ˆ» (для совместимости с более ранними версиями pawn).

-;+/-

С -;+ каждый оператор должен заканчиваться точкой с запятой; с -;- точка с запятой необязательна для завершения оператора, если оператор является последним в строке. Опция -; (без суффикса + или -) переключает текущую настройку.

sym=value

определить константу «sym» с заданным (числовым) значением, значение необязательно;

@filename

прочитать (дополнительные) параметры из указанного «файла ответов».

• **Response file**

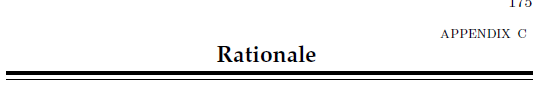
Для поддержки операционных систем с ограниченной длиной командной строки (например, Microsoft DOS) компилятор pawn поддерживает «файлы ответов». Файл ответов — это текстовый файл, содержащий параметры, которые в противном случае вы вводили бы в командной строке. С командой:



компилятор pawn компилирует файл «prog.pawn», используя параметры, перечисленные в файле ответов «opts.txt».

• **Configuration file**

На платформах, которые его поддерживают (в настоящее время Microsoft DOS, Microsoft Windows и Linux), компилятор считывает параметры в «файле конфигурации» при запуске. Файл конфигурации должен иметь имя «pawn.cfg» и должен находиться в том же каталоге, что и исполняемая программа компилятора. В некотором смысле файл конфигурации является неявным файлом ответов. Параметры, указанные в командной строке, могут иметь приоритет над параметрами в файле конфигурации.



Первым вопросом при представлении нового компьютерного языка должен быть: зачем вообще нужен новый язык?

Действительно, я просмотрел несколько существующих языков, прежде чем разработал свой собственный. Многие маленькие языки предназначались для написания сценариев командной оболочки (TCL, Perl, Python). Другие языки не были разработаны как языки расширений, и их встраивание возлагалось исключительно на основное приложение. Поскольку изначально я пытался использовать Java в качестве языка расширения (а не создавать свой собственный, как я сделал сейчас), различия между pawn и Java иллюстрируют почти взаимные цели разработки обоих языков. Например, Java продвигает распределенные вычисления, когда «пакеты» находятся на разных машинах, pawn спроектирован так, чтобы скомпилированные апплеты можно было легко хранить в составном файле вместе с другими данными. Кроме того, Java спроектирован так, чтобы быть архитектурно-нейтральным и независимым от приложений, а обратная пешка предназначена для тесной связи с приложением; нативные функции в какой-то степени табуированы в Java (по крайней мере, она считается «нечистой»), тогда как нативные функции — это «повод быть» для pawn. С точки зрения pawn предполагаемое использование Java перевернуто: нативные функции рассматриваются как вспомогательная библиотека, которую использует приложение — на Java; в pawn нативные функции являются частью «приложения», а сама программа pawn представляет собой набор вспомогательных функций, которые использует приложение.

Язык для написания сценариев приложений и устройств: pawn is target

в качестве языка расширения, предназначенного для написания макросов или подпрограмм для конкретных приложений. pawn не является подходящим языком для реализации бизнес-приложений или операционных систем. pawn спроектирован таким образом, чтобы его можно было легко интегрировать и встраивать в другие системы/приложения. Он также предназначен для работы в средах с ограниченными ресурсами, например, на небольших микроконтроллерах.

В качестве языка расширения программы-ломбарды обычно манипулируют объектами основного приложения. В системе анимации сценарии закладок имеют дело со спрайтами, событиями и временными интервалами; в коммуникационном приложении скрипты пешки обрабатывают пакеты и соединения.

предположим, что хост-приложение или устройство

делает (подмножество) своих ресурсов и функций доступными через функции, дескрипторы, волшебные файлы cookie. . . подобно тому, как современная операционная система предоставляет интерфейс для процессов, написанных на C/C++ — например, Win32 API («работает везде») или «glibc» GNU/Linux.. Для этого пешка

имеет простой и эффективный интерфейс к «родным» функциям хост-приложения.

Сценарий пешки манипулирует объектами данных в хост-приложении.

через вызовы функций, но он не может напрямую обращаться к данным хост-приложения.

Первыми и главными критериями пешечного языка были скорость исполнения и надежность. Надежность в том смысле, что программа-ломбард не должна вызывать сбой приложения или инструмента, в который она встроена, — по крайней мере, не так просто. Хотя это значительно ограничивает возможности языка, у него есть два преимущества:

⋄ поставщик приложения может быть уверен, что его приложение не выйдет из строя из-за пользовательских дополнений или макросов,

⋄ пользователь может свободно экспериментировать с языком без (или с небольшим) риском повреждения файлов приложения.

Скорость важна: программы-пешеходы, вероятно, будут работать на абстрактной машине, а абстрактные машины, как известно, работают медленно. Мне нужно было создать язык с низкими накладными расходами и язык, на котором можно написать быструю абстрактную машину. Скорость также должна быть надежной в том смысле, что скрипт пешки не должен замедляться со временем или иметь случайные сбои в работе. Следовательно, pawn исключает любой необходимый «фоновый процесс», такой как сборка мусора, а ядро ​​абстрактной машины не выделяет неявным образом какие-либо ресурсы системы или приложения во время работы. То есть pawn не выделяет память и не открывает файлы, не без помощи нативной функции, которую скрипт вызывает явно. Как сказал Деннис Ритчи, по замыслу язык C ограничивается средствами, которые могут быть относительно эффективно отображены непосредственно в машинные инструкции. То же самое верно и для pawn, и это также частичное объяснение того, почему pawn так похож на C. Несмотря на то, что pawn работает на абстрактной машине, цель состоит в том, чтобы эта абстрактная машина была маленькой и быстрой. pawn используется в крошечных встроенных системах с размером оперативной памяти 32 КБ или меньше, а также в высокопроизводительных играх, которым требуется каждый цикл процессора для их графического движка и игрового процесса. В обеих средах трудно проглотить поддержку тяжеловесных сценариев. Краткий анализ показал, что логика декодирования инструкций для абстрактной машины быстро станет узким местом в производительности абстрактной машины. Для простоты декодирования каждый код операции должен иметь одинаковый размер (за исключением операндов), а код операции должен полностью определять

Краткий анализ показал, что логика декодирования инструкций для абстрактной машины быстро станет узким местом в производительности абстрактной машины. Для простоты декодирования каждый код операции должен иметь одинаковый размер (за исключением операндов), и код операции должен полностью определять инструкцию (включая методы адресации, размер операндов и т. д.). Это означало, что для каждой операции над переменной абстрактной машине требовался отдельный код операции для каждой комбинации типа переменной, класса хранения и метода доступа (прямого или разыменованного). Даже для трех типов (int,

char и unsigned int), два класса хранения (глобальный и локальный) и три метода доступа (прямой, косвенный или индексированный), в общей сложности требуется 18 кодов операций (3\*2\*3), чтобы просто получить значение переменной.

В то же время, чтобы абстрактная машина оставалась небольшой и управляемой, я поставил цель примерно в 100 инструкций. \* Имея 18 кодов операций для загрузки переменной в регистр, еще 18 для сохранения регистра в переменную, еще 18 для получения адрес переменной и т.д. . Я быстро превысил установленный мной лимит в сотню кодов операций.

Языки bob и rexx вдохновили меня на разработку бестипового языка.

Это сэкономило мне много кодов операций. В то же время язык уже нельзя было назвать «подмножеством C». Я менял язык. Почему бы тогда не пойти еще дальше в изменении языка? Здесь в игру вступили еще несколько рекомендаций по дизайну:

⋄ дайте программисту инструмент общего назначения, а не специальное решение

⋄ избегайте подверженных ошибкам языковых конструкций; поощрять проверку ошибок

⋄ быть прагматичным

Инструмент общего назначения: pawn предназначен как язык расширения,

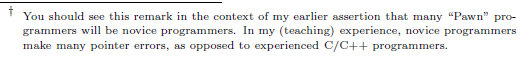
без точного указания, в чем именно она будет заключаться. Как правило, приложение или инструмент, использующий pawn в качестве своего языка расширения, предоставляет множество оптимизированных подпрограмм или команд для работы с собственными объектами, будь то текст, записи базы данных или анимированные спрайты. Язык расширения существует, чтобы позволить пользователю делать то, что разработчик приложения забыл или решил не включать. Вместо того, чтобы предоставлять исчерпывающую библиотеку функций для сортировки данных, сопоставления регулярных выражений или рисования кубических сплайнов Безье, pawn должен предоставлять средства (общего назначения) для использования, расширения и объединения конкретных («собственных») функций, которые приложение может использовать. обеспечивает.

pawn не имеет обширной стандартной библиотеки. В Pawn намеренно отсутствуют такие функции, как указатели, динамическое выделение памяти, прямой доступ к операционной системе или оборудованию, которые необходимы для сохранения конкурентоспособности в области приложений общего назначения или системного программирования. Вы не можете создавать связанные списки или динамические древовидные структуры данных в pawn, и вы также не можете получить доступ к какой-либо памяти за пределами абстрактной машины. Это не означает, что программа-ломбард никогда не сможет использовать динамические отсортированные таблицы символов или изменить параметр в операционной системе; это можно сделать, но для этого необходимо вызвать «собственную» функцию, которую приложение предоставляет абстрактной машине.

Другими словами, если приложение решит реализовать хорошо известные функции peek и poke (из BASIC) в абстрактной машине, программа-заложник может получить доступ к любому байту в памяти, насколько это позволяет операционная система. Точно так же приложение может предоставлять встроенные функции, которые вставляют, удаляют или ищут символы в таблице и позволяют выполнять над ними несколько операций. Предлагаемые базовые функции getproperty и setproperty являются примером встроенных функций, которые создают связанный список в фоновом режиме.

Поощряйте проверку ошибок. Как вы могли заметить, один из главных критериев разработки языка C — «доверие программисту» — отсутствует в моем списке критериев разработки. Пользователи скриптовых языков могут не быть опытными программистами; и даже если они будут, пешка, вероятно, не будет их основным языком. Большинство программистов-пешеходов будут продолжать изучать язык по ходу дела и даже спустя годы не станут экспертами. Следовательно, достаточная причина для замены подверженных ошибкам элементов языка C (указателей) более безопасными, хотя и менее общими, конструкциями (ссылками).† Ссылки скопированы из C++. Они не что иное, как замаскированные указатели, но они ограничены различными, в основном полезными, способами. Обратитесь к книге по C++, чтобы найти больше обоснований для ссылок.

Мне грустно, что многие, даже современные языки программирования имеют так мало встроенных или простых в использовании средств подтверждения того, что программы работают так, как задумал программист. Я имею в виду не теоретическую правильность (что слишком дорого для чего-то большего, чем игрушечные программы), а практичные, простые в использовании механизмы проверки в помощь программисту. pawn предоставляет утверждения как времени компиляции, так и времени выполнения для использования в предусловиях, постусловиях и инвариантах.



Механизм типизации, используемый в большинстве языков программирования, также является автоматическим «ловцом» целого класса ошибок. В силу того, что язык не имеет типов, в pawn не было таких возможностей проверки ошибок. Это явно было недостатком, и я создал механизм «меток» как эквивалент для проверки передачи параметров функции, индексации массива и других операций.

Качество инструментов: компилятора и абстрактной машины также оказывает большое влияние на надежность кода — на любом языке. Хотя это очень слабо связано с дизайном языка, я решил создать инструменты таким образом, чтобы они способствовали проверке ошибок. Система предупреждения Pawn выходит за рамки простого сообщения о том, что синтаксический анализатор не может интерпретировать данные в соответствии с грамматикой языка. В некоторых случаях компилятор запускает проверки, совершенно не связанные с генерацией кода и реализованные специально для выявления возможных ошибок. Аналогичным образом, «ловушка отладчика» встроена прямо в абстрактную машину, а не является надстройкой, реализованной задним числом.

Будьте прагматичны: на мой взгляд, парадигма объектно-ориентированного программирования не полностью оправдала возложенные на нее надежды. С одной стороны, ООП решает многие задачи проще и чище благодаря добавленному уровню абстракции. С другой стороны, современные объектно-ориентированные языки заставляют вас бороться с языком. Борьба должна быть с реализацией функционала под конкретную задачу, а не с языком, используемым для реализации. Объектно-ориентированные языки привлекательны в основном из-за обширных библиотек классов, с которыми они поставляются, но опора на стандартную библиотеку противоречит одной из целей разработки Pawn. Объектно-ориентированное программирование не является решением для неопытного программиста, не терпящего искусственной сложности. Критерий «быть прагматичным» — это напоминание о том, что нужно искать решения, а не элегантность.

• **Practical design criteria**

Тот факт, что пешка так похожа на C, не может быть совпадением, и это не так. pawn начинался как диалект C и остался им, потому что C имеет проверенный послужной список. Изменения по сравнению с C были в основном рождены по необходимости после того, как были удалены черты C, которые я не хотел использовать в языке сценариев: никаких указателей и никакой системы «типизации».

pawn, будучи бестиповым языком, нуждался в других средствах для объявления переменных. В процессе модификации я также отказался от требования языка C о том, что все переменные должны быть объявлены в начале составного оператора. pawn немного больше похож на C++ в этом отношении.

Функции языка C могут передавать «выходные значения» через аргументы-указатели. Стандартная функция scanf, например, сохраняет значения или строки, считанные с консоли, в свои аргументы. Вы можете разработать функцию на C так, чтобы она при необходимости возвращала значение через аргумент-указатель; если вызывающая функция не заботится о возвращаемом значении, она передает NULL в качестве значения указателя. Стандартная функция strtol является примером функции, которая делает это. Этот метод часто избавляет вас от объявления и передачи фиктивных переменных. pawn заменяет указатели ссылками, но ссылки не могут быть NULL. Таким образом, pawn нуждался в другой технике для «отбрасывания» значений, которые функция возвращает через ссылки. Ее решение заключается в использовании «заполнителя аргумента», который записывается как символ подчеркивания (« »); Программисты, работающие на Прологе, узнают в нем аналогичную функцию этого языка. Заполнитель аргумента резервирует временный объект анонимных данных («ячейка» или массив ячеек), который автоматически уничтожается после вызова функции.

Временная ячейка для заполнителя аргумента по-прежнему должна иметь значение, потому что функция может видеть ссылочные параметры как входные/выходные. Следовательно, функция должна указывать для каждого аргумента, переданного по ссылке, какое значение он будет иметь при входе, когда вызывающая программа передает заполнитель вместо фактического аргумента. Кроме того, я также добавил значения по умолчанию для аргументов, которые «передаются по значению». Возможность необязательного удаления справа всех аргументов со значениями по умолчанию была скопирована из C++. Говоря о BCPL и B, Деннис Ритчи сказал, что C был изобретен отчасти для обеспечения правдоподобного способа работы со строками символов, когда вы начинаете с языка, ориентированного на слова. pawn предоставляет два варианта работы со строками: упакованными и неупакованными строками. В неупакованной строке каждый символ помещается в ячейку. Накладные расходы для типичной 32-битной реализации велики: один символ занимает четыре байта. Упакованные строки хранят до четырех символов в одной ячейке за счет того, что их значительно сложнее обрабатывать, если вы можете получить доступ только к полным ячейкам. Современные реализации BCPL предоставляют два метода индексации массива: один для получения слова из массива, а другой — для получения символа из массива. pawn копирует эту концепцию, хотя синтаксис отличается от синтаксиса BCPL. Функция упакованной строки также привела к появлению нового оператора char.

Приложениям Unicode часто приходится иметь дело с двумя наборами символов: 8-битным для устаревших форматов файлов и стандартизированных форматов передачи (как многие интернет-протоколы) и 16-битным набором символов Unicode (или 31-битным набором символов UCS-4). . Хотя у компилятора pawn есть опция, которая делает символы 16-битными (поэтому только два символа помещаются в 32-битную ячейку), обычно более удобно хранить однобайтовые строки символов в упакованных строках, а многобайтовые строки — в неупакованных строках. Это превращает слабость пешки — необходимость отличать упакованные строки от неупакованных строк — в силу: пешка может довольно легко провести это различие. И вместо двух реализаций для каждой функции, работающей со строками (версия ascii и версия Unicode — взгляните на Win32 API или даже на стандартную библиотеку C), pawn позволяет функциям легко обрабатывать как упакованные, так и неупакованные строки. Несмотря на упомянутые выше изменения, а также те, что описаны в главе «Подводные камни: отличия от C» (стр. 134), я старался держать пешку ближе к C. И последний момент, не связанный с дизайном языка, но, тем не менее, важный: лицензия: pawn распространяется по либеральной лицензии, позволяющей вам использовать и/или адаптировать код с минимальными ограничениями — см. приложение D.