Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Метапрограммирование в .NET. Интерпретация Common Lisp.

Курсовая работа студента 445 группы

Омельчука Александра Олеговича

Научный руководитель, В. С. Полозов

ст. преподаватель

Санкт-Петербург

2011

Оглавление

1. Введение

Данная работа посвящена вопросу совместимости платформы .NET и языка программирования Common Lisp. Предложена частичная реализация ядра Common Lisp для .NET, написанная на F#, которая включает в себя

- большинство специальных операторов Common Lisp, связанных с организацией потока управления
- реализацию функции EVAL
- вызов локальных функций и макросов с параметрами &OPTIONAL, &REST, &AUX
- поддержку глобального, динамического и статического окружений
- простейшую консоль REPL

2. Обзор

Язык LISP фактически является первым языком программирования высокого уровня и сохраняет свою теоретическую и прикладную значимость на протяжении более чем пятидесяти лет. Принципы функционального программирования, впервые воплощенные в Lisp, оказали огромное влияние на большинство последующих языков программирования, как академических, так и коммерческих. Простота синтаксиса в сочетании с прочным математическим фундаментом обеспечивает чрезвычайную гибкость языка и позволяет быстро создавать чрезвычайно сложные структуры. Отличительной чертой Lisp с самого начала был единый формат представления данных и кода, который предоставляет программисту широкое поле для модификации языка и создания новых domain-specific languages.

Несмотря на то, что язык оказал влияние на множество популярных сейчас языков и технологий программирования, сам Lisp до сих пор остается в тени других языков, по крайней мере, в сфере коммерческого программирования. Однако, нам кажется, что это не только не отменяет актуальности языка, но и подчеркивает большой потенциал для его развития и расширения области применения.

Платформа .NET, будучи language-independent, предлагает уникальные возможности для создания на ее базе новых языков и стимулирует их воплощение. Мы считаем, что .NET вполне пригодна для базирования как традиционных императивных языков (VB.Net, C#), так и языков с более выраженной функциональной составляющей. Самым ярким примером к этому утверждению является не так давно реализованный для .NET мультипарадигменный язык F#. Кроме того, с появлением DLR – Dynamic Language Runtime – стала возможной реализация динамических языков на этой платформе. Так увидели свет проекты IronPython, IronScheme и другие.

Несмотря на возможность работать с элементами языка в F#, например, через quotations, концепция «код как данные» слабо применяется в этом языке. В частности, сильно ограничен набор лексемм, доступных для расширения языка, а из отсутствия макросов следует, что «развертывание» новых конструкций будет осуществляться путем комбинации или применения функций, во время выполнения программы. Таким образом, при легковесности и

удобстве F#, этот язык все же менее подходит для подхода, называемым метапрограммированием, чем языки Lisp-семейства.

Иллюстрацией этих тезисов служит наш проект по реализации Lisp-подобного языка на платформе .NET. Были приняты за основу следующие ключевые моменты:

- Концептуально Lisp является интерпретируемым языком. Команды одна за другой подаются на вход интерпретатору, который, в свою очередь, выдает соответствующие результаты. Основным элементом языка является применение функции. Такие операторы, как присваивание или условный оператор также записываются в функциональной форме. Рекурсия, как правило, предпочитается циклам. (Knott, 1997)

  В то же время программы на Lisp нельзя назвать неэффективными. Типичный коммерческий компилятор этого языка производит машинный код, сравнимый по скорости с другими языками (при непосредственном указании типов данных). В нашем случае, производится компиляция в IL – промежуточный язык .NET, исполняющийся в виртуальной машине. Режим интерпретации также возможен благодаря такой возможности .NET как Compiler as Service.
- Lisp принципиально не делает различия между кодом (операциями) и данными (операндами). Ключевым структурным элементом Lisp являются S-выражения – нетипизированные списки, используемые как для хранения данных, так и для интерпретации в качестве команды. В последнем случае первый элемент списка считается именем функции, а остальные – ее аргументами. Отличием этой структуры от канонического списка является то, что завершаться он может не только ссылкой на пустой список, но и ссылкой на любой объект, не являющийся списком (такой нестандартный список называется dotted list). По этой причине в данной работе применяется собственная реализация S-выражений.
- Разные диалекты Lisp используют разные механизмы разрешения (resolving) символов. Поскольку данная реализация следует спецификации Common Lisp, здесь используются отдельные пространства имен для переменных и функций. Один и тот же символ, будучи употребленным в разных контекстах, может восприниматься как имя переменной или функции.

- В Common Lisp используются как динамические (dynamic), так и статические (lexical) области видимости с возможностью выбора между ними. В то время как первые предоставляют гибкость в поведении кода и удобны для общения с кодом, скомпилированным «на лету», работа со вторыми более эффективна, так как сводится к простому обращению к стэку по смещению, известному в момент компиляции. Наша реализация обеспечивает оба подхода, при этом уделяя внимание гибкости первого и эффективной работе второго.
- Макросы, используемые в Lisp, являются одной из причин, побудившей нас к созданию этой реализации. Как часть ядра языка, они позволяют создавать собственные синтаксические конструкции и расширять язык под нужды программиста, не жертвуя при этом производительностью программы.
- Автоматическое управление памятью и сборка мусора впервые появились в Lisp (Graham, 2001). В его современных реализациях используется сборка мусора с поколениями (Levine & Pitman). Тот же метод применен и в .NET, поэтому заботы об управлении памятью целиком переадресованы целевому фреймворку.

3. Обоснование и постановка задачи

За пятьдесят с лишним лет существования Lisp появилось большое количество его диалектов. Наиболее распространенными на сегодняшний день являются Common Lisp и Scheme.

Разработанный в MIT, язык Scheme долгое время использовался для обучения студентов основам программирования. Упор в нем сделан на минимализм в конструкциях и интенсивном использовании хвостовой рекурсии. Вследствие этого в языке не используются статические области видимости переменных. Предпочтение в Scheme отдано функциональному стилю, и в спецификации языка не заявлено объектно-оринтированных особенностей (Seibel, 2005), (Kelsey, Clinger, & Rees, 1998).

Существует несколько реализаций Scheme под .NET, наиболее известной из которых является IronScheme.

Common Lisp, напротив, более широко применяется в коммерческой сфере. Это мультипарадигменный язык: программы на Common Lisp сочетают процедурный, функциональный и объектно-ориентированный стили; он определяется своим станартом ANSI, в который включена спецификация объектной системы (Pitman, 1996-2005). Поскольку этот язык существует с 1986 года, для Common Lisp написано большое количество библиотек. Поскольку Common Lisp больше Scheme, он мощнее, но более сложен в реализации.

К сожалению, спецификация объектной системы Common Lisp делает его не вполне совместимым с CLR (.NET Common Language Runtime). В частности, обработка средой .NET чисел с плавающей точкой не согласуется с ANSI-стандартом CL. Кроме того, проблемой (но не такой существенной, как может показаться на первый взгляд) является, что .NET не поддерживает множественного наследования. По этой причине не написано реализаций Common Lisp под .NET (хотя такие существуют для Java).

Тем не менее, преимущества Common Lisp вкупе с неоспоримыми достоинствами платформы .NET, описанными выше, побудили нас к созданию реализации Lisp-подобного языка, как можно более совместимого с Common Lisp. Реализуемый язык предполагается поддерживать в непосредственной близости к стандарту ANSI-CL: Предполагается, что

различное поведение чисел с плавающей точкой не станет препятствием для использования подавляющего большинства библиотек CL из .NET и наоборот. Это позволит, в частности, переиспользовать написанные на Lisp элементы части других свободных реализаций, таких как SBCL, на поздних этапах работы.

Такая задача, как реализация крупного языка программирования, достаточно трудоемка, поэтому в рамках данной работы мы будем рассматривать подзадачу: реализация функции EVAL и потока управления программы на Lisp.

Ядро Lisp-подобного языка можно заключить в десятке основных операторов, но подобный минимализм накладывает серьезные ограничения на производительность и гибкость реализации. Нас же интересует тесная (насколько это возможно) интеграция нашего языка с технологиями платформы .NET, например, ее объектной системой. Это, в частности, будет отличать нашу реализацию Lisp от существующих, где имплементация объектно-ориентированной парадигмы является результатом расширения языка.

Для написания реализации был выбран язык F#, как наиболее лаконичный, гибкий и выразительный язык, доступный для платформы .NET. Тем не менее, поскольку некоторые возможности DLR удобнее использовать из C#, небольшой вспомогательный проект написан на этом языке.

4. Подробности реализации

4.1. Разбор S-выражений

Символом (symbol) в Common Lisp называется не являющийся числом набор цифр, латинских букв (регистр которых неважен и по умолчанию переводится в верхний), и знаков из следующего набора:

```
+ - * / @ $ % ^ & _ = < > ~ .
```

Символы используются для именования каких либо сущностей Lisp: переменных, функций и макросов, типов данных. Специальные константные символы T и NIL являются именами булевских значений True и False.

Структурной единицей Lisp является S-выражение (S-expression, от symbolic expression). S-выражением считается литерал (атомарное выражение) или список S-выражений. Примерами атомарных выражений являются числа, строки и символы. Обращаясь к форме Бэкуса-Наура, имеем следующее:

```
SExpr     ::= Atom | "(" ")" | "(" SExprList ")"
SExprList ::= SExpr | SExpr SExprList
Atom      ::= Number | String | Symbol
```

Предполагая, что Number, String и Symbol и скобочные символы – лексеммы, получаем предельно короткую LL(1)-грамматику. За начальный нетерминал принимаем SExprList. Пустой список эквивалентен сиволу NIL.

Полученную грамматику мы распознаем и транслируем во внутреннее представление с помощью инструментов fslex/fsyacc. Поскольку S-выражением является любое значение в Lisp и при этом S-выражения хранят в себе информацию о типе хранимого значения, логично использовать для их хранения запакованные значения .NET (boxed values). Иными словами, будем использовать тип-псевдоним:

```
type SExpr = obj
```

В спецификации Common Lisp указано, что лексический разбор S-выражений выполняет особая сущность: Reader. В своей работе он использует кастомизируемую таблицу чтения

(readtable), которую можно изменять в процессе работы программы, что позволяет расширять синтаксис языка в «горячем» режиме. Подобная функциональность не требуется для полноценной работы ядра языка, поэтому поддержку readtables планируется добавить позже с использованием существующей кодовой базы.

4.2. Функция EVAL

Любое S-выражение в Lisp вычислимо. Это значит, что мы можем применить к произвольному выражению функцию EVAL и (при отсутствии ошибки) получить результат. Принимая во внимание тот факт, что программа на Lisp является набором S-выражений, которые при этом можно рассматривать как список, и, как следствие, как одно S-выражение, получаем, что функция EVAL в сущности является ядром интерпретатора Lisp.

Доступное для выполнения выражение также называют формой.

В нашем случае интерпретация тесно сопряжена с компиляцией: выражение на входе транслируется в промежуточный код платформы .NET (для краткости мы далее будем называеть его «байт-код»), который затем исполняется виртуальной машиной. Поэтому, говоря о функции EVAL как об абстрактном интерпретаторе, мы будем помнить, что на самом деле происходит трансляция в байт-код; часть действий при этом выполняется в момент трансляции, а часть осуществляется байт-кодом во время его выполнения.

Вычисление выполняется по следующим правилам:

- Атомарные выражения, не являющиеся символами, и некоторые специальные символы (например, T и NIL) дают при вычислении самих себя
- Символы трактуются как имена переменных. При вычислении символа происходит поиск связывания этого символа со значением. Если оно найдено, возвращается это значение, иначе ошибка.
- Вычисление списка представляет собой применение функции. Первым в списке должен быть символ; он принимается за имя вызываемой функции, остальные члены списка являются аргументами. Происходит поиск функции по имени; если он прошел успешно и не была возвращена ошибка, дальнейшие действия зависят от типа найденной функции:

- Собственно *функции* выполняются активно (eagerly). Значения аргументов по очереди вычисляются функцией EVAL, и результаты передаются функции в качестве параметров.
- *Специальные выражения* не требуют вычисления значений аргументов. Аргументы просто передаются им на вход. Существует ограниченый набор специальных операторов.
- Поведение *макросов* комбинирует два вышеописанных случая. Аргументы передаются им на вход без предварительного вычисления, затем результат, который зачастую является сложным списком, исполняется с помощью функции EVAL. Здесь стоит явно выделить, что вычисление макроса происходит во время компиляции S-выражения, а результата – во время исполнения функции EVAL.

Для трансляции выражений в байт-код используются Expression trees (пространство имен System.Linq.Expressions), появившиеся в версии .NET 4.0. Реализация EVAL транслирует S-выражение в дерево, которое затем проходит компиляцию в байт-код (с оптимизацией при желании). Результатом является делегат к функции от текущего динамического окружения (работа окружений описана в п. 4.4), который и исполняется.

Платформа .NET предоставляет программисту возможность написания собственного компилятора Expression trees в байт-код. Тем не менее, существующего компилятора оказалось достаточно для написания EVAL. Указанная возможность может оказаться полезной на этапе оптимизации компилятора.

Вызов функции EVAL доступен как внутренний метод Lisp runtime, так и в качестве стандартной функции. Эта функция принимает на вход S-выражение, которое затем исполняет в текущем динамическом и пустом статическом окружении. Эта простая идея придает языку богатые возможности благодаря стуктурной унификации кода и данных и легкости порожения кода на Lisp.

4.3. REPL

Вопреки ожиданиям, REPL это не еще одна функция. REPL является сокращением от Read-Evaluate-Print Loop – способа быстрой разработки программ, используемого при

написании программ на Lisp. Гибкость языка позволяет вносить изменения в программу на Lisp, не выходя из нее – и поэтому делает возможным весь процесс написания программы без ее остановки, с помощью цикла «прочесть-вычислить-вывести».

Не слишком удивительно, что REPL используется и в .NET. Разработка на F# позволяет использование REPL (в консольном приложении fsi.exe или с помощью его оболочки F# interactive в одном из окон Visual Studio). Интерактивный цикл вычисления доступен для языка C# в Mono – реализации .NET под ОС семейства Unix. Впрочем, вычисление выражений в REPL для C# и F# никак не влияет на работу написанных на них и запущенных приложений.

В спецификации ANSI-CL упомянут ряд возможностей, которыми должен обладать интерпретатор Common Lisp. Например, он должен поддерживать набор глобальных переменных, в которых хранятся последние три запроса к интерпретатору и их результаты, вычисляемую в данный момент форму и т. д. Настраивается также и вывод данных: подобно тому как за чтение данных отвечает особая сущность Reader, вывод данных поручен еще одной кастомизируемой сущности, называемой Printer.

Поскольку наша задача заключается в реализации потока управления языка, а для проверки корректности промежуточных результатов используется рецессионное тестирование, особой нужды в мощном интерактивном интерпретаторе мы не испытывали. Вся работа реализованной версии сводится к чтению-выводу данных и перехватыванию необработанных ошибок.

4.4. Области видимости

Под окружением понимается абстракнтый объект, содержащий информацию о состоянии программы: связывания переменных, точках выхода и т.п. Разнотипные окружения – одна из важных особенностей диалекта Common Lisp. Всего спецификацией определяется три типа окружений:

1. *Глобальное.* Здесь содержится информация, имеющая неограниченную область видимости: связывания глобальных динамических переменных, функций и макросов, константы, специальные операторы, информация о типах и т. д.

2. *Динамическое.* Содержит связывания, срок жизни которых ограничен временем выполнения формы, неявно создавшей динамическое окружение. Помимо прочего, содержит информацию о связывании локальных динамически переменных, активных catch-тегах и точках выхода, созданных специальным оператором unwind-protect.

   Поскольку unwind-protect реализован нами через конструкцию try … finally и использует механизм исключений, его точки выхода фактически не хранятся в объекте, представляющем собой динамическое окружение, а создаются и обрабатываются средствами .NET runtime.
3. *Статическое.* Здесь хранится информация о связываниях, доступных семантическому анализу. К ним относятся связывания статических переменных, функциях и макросах, а также метках, используемых в специальных операторах BLOCK и TAGBODY. Срок его жизни также ограничен временем исполнения установившей окружение формы. Экземпляры класса, представляющего статическое окружение, используются транслятором в процессе преобразования S-выражения в байт-код. Во время выполнения байт-кода лексическое окружение существует лишь в виде абстракции.

Связывания переменных и функций разделены по разным пространствам имен (фактически – по разным хэш-таблицам в окружениях). В каком пространстве имен проводить поиск символа, зависит от контекста, в котором он встречен. Так, например, в выражении (+ + 1) первый знак «плюс» трактуется как имя функции, которая, если не была переопределена, возвращает сумму всех своих численных аргументов, а второй – как имя стандартной переменной, в которой хранится результат последнего выражения, возвращенный интерпретатором. Наше выражение вернет этот результат, увеличенный на 1 (или ошибку, если последний результат не был числом).

Будучи употребляемыми в том же контексте, что и функции, макросы хранятся с ними в одном пространстве имен.

В зафиксированном контексте любой символ может иметь лишь одно активное связывание. В случае, если создано несколько связываний, активным считается установленное позже всех, при этом статические связывания имеют приоритет над динамическими.

Разрешение значения переменной или функции производится с использованием активного связывания.

Реализация механизма активных связываний возможна двумя способами. Первый заключается в создании стека связываний: поиск нужного символа производится начиная с самого верхнего элемента стека, с последовательным спуском вниз, если символ не найден. Второй способ – содержать одну хэш-таблицу значений, в которой помимо данных о связывании хранить информацию о окружении, которому принадлежит связывание. Поиск символа в таком случае сводится к единичному обращению к хэш-таблице, но эту таблицу нужно подерживать, каждый раз добавляя или удаляя новые данные при смене окружения. Нами был выбран первый способ, как более наглядный.

Для обобщения механизма доступа к динамическим переменным введен интерфейс IDynamicEnvironment, который реализуют классы, представляющие глобальное и динамическое окружения.

4.5. Вызов функций

При объявлении функции указывается, какие аргументы она принимает. Любая функция в Lisp возвращает значение (возможно, больше одного). Список аргументов, указанный в объявлении функции, называется лямбда-списком (lambda list).

Первоначальная идея вызова функции довольно проста: список переданных функции аргументов сопоставляется с лямбда-списком, и устанавливаются связывания. Если количество переданных аргументов не совпадает с количеством заявленных, транслятор возвращает ошибку во время семантического анализа.

Для удобства программиста Lisp поддерживает опциональные, именованные и неявные (auxiliary) параметры функций. Для опциональных параметров можно указать значение по умолчанию. Кроме того, бывает полезно узнать, было ли значение опциональной переменной непосредственно передано в функцию, или же было взято значение по умолчанию. Для этого можно указать имя переменной, в которой будет храниться T или NIL в зависимости от ситуации.

Неявные аргументы фактически не являются аргументами функции: в лямбда-списке указывается имя параметра и выражение для его инициализации (оно может зависеть от остальных параметров функции). При вызове функции происходит инициализация этих параметров, как если бы они были указаны в начале тела функции.

Как и во многих других языках программирования, в Lisp можно определить параметр метода, принимающий переменное количество аргументов. Эти аргументы будут переданы в виде Lisp-списка. В аналогичный параметр в .NET аргументы предаются в виде массива, поэтому он не используется в данной реализации.

Обилие возможностей затрудяет техническую реализацию вызова функции. С учетом опциональных и неявных параметров, лямбда-списки представляют собой не обычный список аргументов, а дерево, в котором разные типы переменных отделяются друг от друга специальными символами-разделителями &OPTIONAL, &KEY, &REST, &AUX. При разборе объявления функции происходит чтение лямбда-списка с помощью конечного автомата. Результатом разбора является структура LambdaList, в которой хранится информация об аргументах функции и логика сопоставления этих аргументов с данным набором параметров.

Еще одну техническую трудность составляет эффективная реализация вызовов функций. Для функций, имеющих статическую область видимости, сверку параметров, переданных функии, с лямбда-списком возможно, и поэтому желательно осуществлять во время трансляции в байт-код. Если же функция получена в результате разрешения динамического связывания, то есть во время работы программы, то и сопоставление аргументов с лямбда-списком можно осуществить лишь динамически. Одни и те же действия при сверке должны осуществляться над разными типами данных при разных сценариях, поэтому в реализации этой функциональности используется довольно обобощенный код.

При вызове макросов возможно применять те же типы аргументов, что и для функций, с некоторыми расширениями. В текущей версии реализации объект LambdaLisp переиспользуется для хранения информации о параметрах как функций, так и макросов.

5. Дальнейшее развитие

Common Lisp – большой язык программирования, поэтому его реализация на новой платформе – масштабная задача, несозразмерная по объему с данной работой.

По части потока выполнения нереализованным остался механизм продолжений (restarts), объединяющий в себе лучшие черты исключений и событий.

Предстоит также выполнить работу по реализации CLOS поверх объектной системы .NET. Хотя они имеют много различий (статическая и динамическая типизация, множественное и одиночное наследование), примеры успешной реализации языка Python для .NET и Java показывают, что подобная задача является решаемой.

Интерпретатор, реализованный в данной работе, можно модифицировать и превратиьт в компилятор. Важная часть работы уже сделана: на базе S-выражений генерируются Expression trees, которые компилируются в байт-код средствами .NET. От будущего компилятора требуется обернуть этот код в классы и создать сборку.

Целью проекта остается возможность использования существующих библиотек, написанных на Common Lisp, из других языков платформы .NET.

6. Заключение

В рамках данной работы продемонстрировано ядро Lisp-подобного языка, реализущее многие особенности Common Lisp, благодаря которым он так популярен:

- функция EVAL – ключевая точка интерпретации языка;
- поддержка макросов;
- глобальное, динамические и статические окружения;
- продвинутый механизм вызова функций.

Хотя идеи метапрограммирования возникли еще в 50-60е годы прошлого века, аппаратные ограничения того времени не позволяли пользоваться ими в полной мере. Актуализация Lisp как не только источника вдохновения для других языков, но и самостоятельного мощного инструмента, происходит в настоящее время. Традиция императивных языков до сих пор преобладает в коммерческом программировании, но можно наблюдать, как и эти языки постепенно трансформируются в мультипарадигменные: функциональные черты приобретают такие языки как C++, C# и Java; на базе классических коммерческих программных платформ, .NET и Java, реализуются легковесные гибкие языки F#, Nemerle, Python, Clojure.

Мы надеемся, что рассматриваемая реализация Lisp-подобного языка, yaLisp.NET, вырастет в используемый язык, на котором станет возможно разрабатывать портируемые программы, или же использовать его для быстрого создания и развития програмных прототипов. Были использованы новые возможности платформы .NET: Dynamic Runtime, Expression trees, язык F# (интересный факт: возможности F# позволили реализовать на нем интерпретатор языка для обработки списков, не используя ни одного цикла).

Проект выпущен в свободный доступ по адресу http://yalispdotnet.codeplex.com под лицензией Apache 2.0.

7. Источники

Graham, P. (Декабрь 2001 г.). *What Made Lisp Different.* Получено из http://www.paulgraham.com/diff.html

Kelsey, R., Clinger, W., & Rees, J. (9 Сентябрь 1998 г.). Revised5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation, 11*.

Knott, G. D. (18 Февраль 1997 г.). *Interpreting LISP.* Получено из http://www.civilized.com/files/lispbook.pdf

Levine, N., & Pitman, K. M. (б.д.). *Common Lisp - Myths and Legends.* Получено из http://www.lispworks.com/products/myths_and_legends.html

Pitman, K. (1996-2005). *Common Lisp HyperSpec (TM)*. Получено из http://www.lispworks.com/documentation/HyperSpec/Front/index.htm

Seibel, P. (2005). *Practical Common Lisp.* New York: Apress.