

Расширяемый язык программирования

Лебединская Н. А., СПбГУ, Санкт-Петербург n.lebedinskaya@spbu.ru,
Лебединский Д. М., СПбГУ, Санкт-Петербург d.lebedinsky@spbu.ru,
Смирнов А. А., ВА МТО имени генерала армии А. В. Хрулева, Санкт-Петербург
smirnov89119750282@yandex.ru

Аннотация

Краткое изложение вопросов, связанных с разработкой нового расширяемого языка программирования, в части структур данных, модели вычислений, организации таблиц, содержащих, в том числе, реализации команд и преобразования типов, и некоторых других.

Введение

Данный проект посвящен разработке нового расширяемого языка программирования. В связи с этим представляется разумным создание для этой цели двух языков: низкого уровня, о котором в основном и пойдет здесь речь, и высокого уровня, которому будут посвящены следующие работы. Цель работы состоит в том, чтобы изложить ряд основных идей и проектных решений, принятых при разработке языка низкого уровня.

Сама по себе идея создания расширяемого языка программирования определенно не нова. Существуют очень многие проекты, имеющие похожие цели. Упомянем здесь только некоторые из них; данный список определенно не может претендовать на полноту изложения данного вопроса.

Для начала упомянем поздне-советскую разработку, язык УТОПИСТ [1]. В книге, излагающей его описание, сказано, что он позволяет добавлять в систему новые понятия; однако, под понятиями там понимаются наборы соотношений, описывающие определенные физические явления, и позволяющие по имеющимся данным (значениям некоторых из переменных) восстанавливать значения остальных. Таким образом, речь идет о комбинации такого рода соотношений, позволяющей многоступенчато вычислять неизвестные значения параметров для «составных» явлений. Такой подход определенно сильно отличается от нашего, в котором под понятиями, добавляемыми в систему, имеются ввиду понятия собственно программирования, например новые парадигмы, т. е., скажем, добавление объектно-ориентированности, которая изначально в системе реализована не была.

Далее, упомянем систему katahdin [2]. Она действительно позволяет менять грамматику языка на ходу, и это уже гораздо ближе к тому, что происходит здесь, однако, и с этой системой имеется определенные отличия. Как

сказано в документе, ее описывающем, там используются PEG-грамматики и `packrat`-парсер. С одной стороны, такая реализация имеет то неоспоримое достоинство, что все работает очень быстро, но со стороны богатства выразительности, увы, здесь имеются определенные ограничения. Как вскользь замечено в руководстве, такой подход реализует, в частности, синтаксис формата частично. На наш взгляд, это выглядит не очень многообещающе.

Существует также язык XL [3]. В нем, однако, насколько нам известно, для расширяемости реализована перегрузка выражений. Конечно, необходимость чего-то большего в ближайшем будущем представляется нам маловероятной, но с нашей точки зрения, это серьезное ограничение.

Следующая система, которая будет здесь упомянута, — это язык `seed7` [4]. В его документации написано, что в самом языке нет ключевых слов — все они могут быть определены как часть стандартной библиотеки в порядке расширения языка. Однако, и здесь есть небольшие проблемы — там нет автоматического преобразования типов, и для вычисления произведения целого числа на вещественное целый сомножитель должен быть явно преобразован к вещественному типу. Ввиду привычки к программированию на традиционных языках вроде C или C++, где такое преобразование происходит прозрачно, такая необходимость несколько раздражает.

Наконец, упомянем здесь диссертацию Bravenboer'a [5], в которой также описана система расширяемого языка. В качестве базового языка для расширения выбран Java, а инструментом для расширения служат возможности подключения дополнительных модулей, обеспечивающих понимание системой расширенного синтаксиса (и перевода фрагментов, его использующих, на базовый язык, что задает семантику таких фрагментов). Однако, насколько мы поняли, такое подключение выполняется глобально, на уровне всей программы, и запретить использование расширенного синтаксиса в определенных местах программы очень непросто, если вообще возможно.

Языки низкого и высокого уровня

Как уже было сказано в начале введения, для построения расширяемого языка программирования представляется полезным иметь на самом деле два языка — низкого уровня (как можно проще, с намеренно упрощенным синтаксисом и семантикой, но с реализацией средств расширения), который собственно и подлежит расширению, и язык высокого уровня, который является его расширением и может расширяться далее.

В качестве языка низкого уровня был выбран императивный язык, данные и программы которого выглядят как у LISP'a (с тем отличием, что в соответ-

ствующем дереве атомы, т. е. строки, стоят в каждой вершине дерева, а не только в листьях). Также отличие данной структуры от LISP'a состоит в том, что, кроме строки данных и указателей на сына и правого брата, в каждом узле стоит указатель назад (на левого брата, если он есть, на отца, если нет, и на переменную, содержащую данное дерево, если речь идет о корневом узле; разница между корневым и прочими узлами опознается по специальному флагу, также имеющемуся в каждом узле) и рекурсивный разделяемый мьютекс, предназначенный для организации транзакционной семантики.

Если возникает необходимость реализации дополнительной семантики, например, указания таких дополнительных свойств объекта, как тип, или внутри списка дополнительного указания ключей элементов, превращая список в словарь, используется механизм частных случаев: определенные частные случаи списка закрепляются для этих целей, но при этом сохраняется возможность трактовки такого рода частных случаев непосредственно как они написаны, для чего они должны встречаться внутри других частных случаев такого же вида.

Например, мы можем для указания дополнительных свойств объекта зафиксировать списки, первым элементом которых является строка «свойства», но при этом в таком списке есть данные с ключом «объект», представляющие собой тот объект, свойства которого добавляются, и содержимое этих данных трактуется, как оно написано, даже если это список, первым элементом которых является строка «свойства».

Важным элементом языка низкого уровня является так называемая таблица. Это ассоциативный массив, индексами и данными в котором являются объекты, т. е. деревья со строками в вершинах. При этом смысл данного соответствия определяется типом ключа, например, ситуации вызова функции может соответствовать реализация данной функции, а имени переменной — ее адрес. Такой подход хорош тем, что позволяет иметь все необходимые для разбора и выполнения программы таблицы в рамках одной структуры, и, более того, впоследствии, в качестве расширения, добавлять в систему таблицы, о которых на момент проектирования данной системы вовсе ничего известно не было.

Важным элементом таблицы являются наборы реализаций. Они представляют собой все те же деревья, некоторые части которых помечены особым образом и поэтому воспринимаются как реализации определенных процедур, представленных либо указаниями функций из динамических библиотек, либо списками команд. При этом важно, что сначала запускается самая левая реализация, которая затем может вызывать другие реализации той же процедуры из этого дерева. Такой механизм хорош тем, что позволяет решить относительно просто две проблемы.

Первая из них состоит в обеспечении возможности добавлять новые реализации «поверх» старых (чтобы вызов новой реализации включал в себя, в том числе, функциональность старых), а затем удалять добавленные таким образом реализации в произвольном порядке (при этом при удалении реализации из середины вызов удаленной реализации через обращение к предыдущей реализации из следующей за удаленной просто переключается на ту, которая стояла до удаленной). Вторая проблема состоит в том, чтобы можно было добавлять новые реализации той же функции, расширяющие ее область определения (если выполняемая реализация понимает свою недостаточность для выполнения необходимых действий, она может вызвать следующую, и так до тех пор, пока не будет вызвана последняя, играющая роль реализации по умолчанию).

Таблица, вообще говоря, не монолитна, а состоит из блоков, организованных в список, которые можно добавлять или убирать. Такая организация позволяет таблице иметь «локальные» части, что очень полезно для поддержки механизма локальных понятий для считывания и выполнения.

Модель вычислений

Единственный регистр виртуальной машины, используемой для выполнения программы на языке низкого уровня, хранит указатель на активационную запись работающей в данный момент процедуры. Вся остальная необходимая для ее работы информация, включающая указатели на выполняемый блок и текущую команду в нем, стек данных для вычислений, а также указатели на активационные записи вызвавших ее процедур и вызванных ею, приостановленных, но не законченных, хранятся в этой активационной записи.

В качестве начального механизма синтаксического разбора используется обобщенный GLL-парсер. В нем правыми частями правил грамматики выступают процедуры, код которых линеаризован, т. е. преобразован к такому виду, в котором выражения состоят из вызова одной операции, и нет вложенных управляющих структур (они реализуются через механизм безусловного и условного переходов).

На момент запуска программы, набор понятий, которые могут быть считаны, ограничивается тем самым расширением LISP, о котором шла речь в предыдущем разделе, с той лишь разницей, что поддерживается многостадийное программирование, позволяющее выполнять только что считанные списки команд (они определяются метками, т. е. теми строками, которые стоят в корневых узлах соответствующих деревьев), и если при считывании какого-либо изначально имевшегося понятия возникает ошибка, запускается

определенная для каждого типа ошибки функция, реализация которой берется из таблицы. Это позволяет эффективно бороться с бесконечной рекурсией: если для считывания и выполнения программы достаточно стандартных средств, она просто выполняется, а если нет — запускаются механизмы, отвечающие за расширения.

Одно из неоспоримых достоинств используемой структуры данных состоит в возможности строить большие структуры, содержащие другие подструктуры как свои части, поэтому вполне достаточно для любых функций иметь сигнатуру с одним параметром такого типа по значению и результатом такого же типа. Однако для удобства допускаются функции с большим числом аргументов. Большинство функций, как встроенных, так и реализованных, имеют в качестве параметра указатель на активационную запись, которая строится непосредственно перед первым запуском данной функции и определяет состояние ее выполнения (как, впрочем, и состояние выполнения функций, вызванных из данной, если такие были), так что если функция встроенная, она должна начинать свое выполнение с восстановления состояния, соответствующего содержимому активационной записи. Такой подход превращает функции в сопрограммы. Дополнительные аргументы при передаче их в качестве параметров помещаются в стек данных, имеющийся в активационной записи.

Важным также понятием является понятие типа. В этом проекте тип — это просто одно из дополнительных свойств, некий ярлык, навешиваемый на объект, для правильного выбора реализаций функций обработки данного объекта. Двойственным по отношению к понятию типа является понятие роли — дополнительном свойстве контекста, в который помещается объект. Информация об автоматическом преобразовании типов и ролей, а также соответствия между теми и другими хранится в таблице.

Такие преобразования, в частности, удобны для указания приоритетов операций; такое их использование, впрочем, показывает, что не для всякой роли годится всякое преобразование типов. Например, выражение может быть преобразовано к его значению при помощи вычисления, но если сумма сначала приводится к ее значению, а затем полученное значение используется в роли сомножителя, то получается обход правила о том, что сумма не может быть сомножителем (это правило означает, что приоритет сложения меньше, чем умножения).

Команды

Встроенная команда выглядит как определенного типа список, содержащий имя команды, отвечающее за то, что она должна делать, и параметры, набор которых меняется в зависимости от имени.

Встроены арифметические, логические команды, команды сравнения, простые структурные команды базовой обработки списков, простейшие управляющие команды.

Список встроенных команд может быть вычислен при помощи соответствующей функции, написанной на C++ и помещенной в стандартную библиотеку.

Наряду со встроенными, возможны и реализованные команды, как функции из динамических библиотек или списки команд, помещенные в соответствующий раздел таблицы и вызываемые оттуда, если при выполнении команды как встроенной возникают ошибки (или тип или имя нестандартные, или аргументы неподходящие). Также, возможны списки команд, выполняемые другими функциями, информация о чем должна храниться в дополнительных свойствах соответствующих списков.

Допустимыми также являются команды, "размножающие" потоки выполнения. Вызов таких команд приводит к тому, что вместо обычных объектов из процедур, использующих такие команды, возвращаются "суперсписки" результатов. Для реализации такого поведения можно в активационной записи процедур иметь как суперсписок уже накопленных результатов, так и суперсписок уже порожденных, но еще не довычисленных активационных записей.

Указатели

Существует два варианта указателей: традиционные (адреса в памяти) и человеко-читаемые (состоят из последовательности шагов, на каждом из которых происходит уточнение указываемого объекта, как правило, в пределах предыдущего). Наличие у каждого узла дерева указателя назад позволяет легко и быстро переходить от одной формы указателя к другой. Впрочем, шаги указателя могут иметь и менее тривиальный смысл, например, такой, как операция разыменования (указываемый до этого шага объект сам воспринимается как указатель, и мы переходим к тому объекту, на который он указывает). Также, как и в случае с разбором программы, если шаг указателя стандартный (т. е. разновидность шага указателя встроенная и необходимая обработка его данных тоже), его переход обслуживается встроенной функцией, но если во время этого перехода встречаются ошибки, запускается со-

ответствующая функция из таблицы.

Ссылки

Среди специальных видов списков, имеющих в данной системе, есть еще и ссылки. Встроенными вариантами ссылок являются непосредственные, прямые и косвенные, с очевидным смыслом. Однако, допустимы и пользовательские варианты, для которых в таблице определяются процедуры чтения и записи. Такие ссылки могут определять как положение объектов, на которые они ссылаются, так и дополнительную структуру.

В обычной активационной записи выполняемой процедуры, как правило, имеется стек данных, с которым работает большинство встроенных команд, в частности, команды чтения и записи по ссылкам. Такой подход позволяет разделить одну операцию присваивания на две операции чтения и записи, для которых и определяются пользовательские процедуры по-отдельности.

Список литературы

- [1] Тыгу Э. Х. Концептуальное программирование. — М.: Наука, 1984.
- [2] Christopher G. S. A Programming Language Where the Syntax and Semantics Are Mutable at Runtime. — Bristol: University of Bristol, 2007.
- [3] Lambert M Surhone (ed.), Miriam T Timpledon (ed.), Susan F Marseken (ed.) XL (Programming Language): Programming Language, Concept Programming, Imperative Programming, Derivative, Computer Programming. — Beau Bassin, Mauritius: Betascript Publishing, 2010.
- [4] Mertes T. Entwurf einer erweiterbaren höheren Programmiersprache. — Vienna: Vienna University of Technology, 1984.
- [5] Bravenboer M. Exercises in Free Syntax Syntax Definition, Parsing, and Assimilation of Language Conglomerates. — Utrecht: Universiteit Utrecht, 2008.