Федеральное агентство по образованию РФ
Государственное образовательное учреждение высшего профессионального образования
Уральский государственный университет им. А.М.Горького

Математико-механический факультет
Кафедра алгебры и дискретной математики

О способах и технологиях расширения языков программирования

Специальность 010109 — «Дискретная математика и математическая кибернетика»

Квалификационная работа на степень магистра математики студента гр. МГКН-2
Мелентьева Артема Алексеевича

научный руководитель:
ассистент кафедры алгебры и дискретной математики, к.ф.-м.н.
Клепинин Александр Владимирович

«Допущен к защите»

____

«___» ______________ 2009 г.

Екатеринбург

2009

Реферат

Мелентьев А.А. “О способах и технологиях расширения языков программирования”, квалификационная работа на степень магистра математики: стр. 46, рис. 4, библ. 27. назв.

Ключевые слова: расширяемый язык программирования, расширение языка, расширяемый компилятор.

В работе произведено сравнение и классификация различных средств для расширения как синтаксических, так и семантических возможностей некоторых языков программирования. Также приведены способы создания таких средств. Полученные данные были использованы автором для создания XWQL — расширения языка запросов, которое успешно используется в проекте XWiki.

Оглавление

1 Введение . . . 4

1.1 Точки расширения компилятора . . . 7

1.2 Определения . . . 10

2 Обзор области . . . 11

2.1 История . . . 11

2.2 Современное состояние . . . 12

3 Классификация . . . 15

3.1 Семантические расширения . . . 15

3.2 Синтаксические расширения . . . 20

3.3 Расширяемые компиляторы . . . 21

3.4 Расширяемые языки программирования . . . 24

4 Средства создания расширений . . . 29

4.1 Среда JetBrains MPS . . . 29

4.2 Среда Textual Modeling Framework(TMF) Xtext . . . 30

4.3 Среда Eclipse IDE Meta-Tooling Platform . . . 32

4.4 Генераторы парсеров . . . 33

4.5 Адаптивные грамматики . . . 41

4.6 Создание расширения языка . . . 42

4.7 Создание расширяемого компилятора . . . 43

4.8 Дизайн IDE для расширяемого компилятора . . . 43

4.9 Создание расширяемого языка . . . 43

5 Заключение . . . 44

1 Введение

В процессе разработки ПО, как правило, используются универсальные языки программирования, возможностей которых зачастую не хватает для многих областей. Естественным образом появляется задача подгонки возможностей языка под конкретную предметную область. Отсюда возникает задача исследования как общих подходов к расширению языков, так и расширений конкретных языков программирования.

Для начала определимся что же такое расширение языка программирования. До сих пор четкого определения такого общего понятия, как ни странно, нет. В данной работе расширением языка будем считать программные средства для увеличения выразительных и/или функциональных возможностей этого языка. В качестве примера приведем язык X10 (см.[14]) — расширение языка Java для высокопроизводительных параллельных вычислений:

```
finish {
  // запускаем параллельный поток вычисления суммы нечетных элементов
  async for (int i = 1 ; i < n ; i += 2 ) oddSum += A[i];
  for (int i = 0 ; i < n ; i += 2 ) evenSum += A[i];
} // ждем завершения всех параллельных потоков
System.out.println("oddSum = " + oddSum + " ; evenSum = " + evenSum);
```

X10 добавляет в язык Java некоторые выразительные средства (в примере это finish{} и async for()) для реализации параллельных вычислений. Помимо этого X10 предоставляет высокопроизводительные библиотеки для реализации параллелизма (отличные от стандартных в Java, написанные на C), которые используются в программах на X10. Таким образом язык X10 добавляет и выразительные, и функциональные возможности в язык Java.

В некоторых источниках в качестве расширений неких языков рассматриваются и специальные библиотеки, используемые в программах на этих языках. В данной работе такие расширения не рассматриваются, т.к. они реализуются внутриязыковыми средствами языка (с использованием процедурной, функциональной, объектно-ориентированной или др. парадигмами программирования). Исключение составляют только так называемые расширяемые языки программирования, которые могут изменять свой синтаксис с помощью специальных выразительных средств языка. Т.е. их внутриязыковые средства способны изменять грамматику языка.

История расширений языков начинается в 60-ых годах с работ о макроподстановках и компиляторах компиляторов. Область активно развивалась, и конечной целью представля-

лось создание действительно расширяемого языка программирования. В то же время был представлен проект расширяемого языка программирования Simula. Но его расширяемость воплотить так и не удалось из-за возникших сложностей. Интерес к области стал снижаться в 70-ых годах. Одним из объяснений этого является чрезмерная сложность создания расширений. Активный интерес к области вновь появляется в начале 21 века. Из за всеобщей компьютеризации многим областям стали необходимы специальные языки программирования, а развитие методов программирования позволило создавать сложные расширяемые языки. Подробней про историю расширяемых языков можно прочитать в разделе 2.1.

В настоящее время область расширяемых языков очень большая и активно развивающаяся, но она слабо структурирована. Существует множество расширений различных языков. Некоторые из них заброшены, а некоторые до сих пор успешно используются. Существует несколько компиляторов, которые заявляют о своей расширяемости. Естественным образом возникает вопрос: какой из них более расширяем и требует меньше знаний и усилий для расширения? Также существует огромное количество инструментов для создания компиляторов. И также возникает вопрос: какие из них лучше подходят для создания именно расширяемых компиляторов или для одиночных расширений языка?

Перед автором была поставлена задача: исследовать наработки в области расширений языков, классифицировать и структурировать различные технологии, проекты и способы, используемые в расширяемых языках и в одиночных расширениях языков. Задача была успешно решена, а именно, в данной работе произведен обзор исследованных расширений, произведена попытка их классификации и структуризации по общим признакам. Более того, приведен возможный план создания расширяемого компилятора на основе изученных технологий. Полученные навыки автор применил для создания расширения XWiki Query Language языка JPQL, про которое написано в разделе 4.4.

По-настоящему расширяемые языки даже в настоящее время очень трудно реализуемы. Помимо обеспечения расширяемости, трудности возникают с совместностью расширений и их отладкой. Также некоторые опасаются, что расширяемые языки затруднят взаимодействие программистов, так как будут поощрять создание индивидуальных расширений “под себя”, что каждый будет писать на своем диалекте языка, несовместимом с диалектом другого.

Существуют проекты, как например SQLj (см. [11]), которые являются одиночными расширениями базового языка и не позволяют дальнейшие расширения (по крайней мере внешне). Такие расширения в работе также исследуются.

В развивающихся языках тоже можно проследить расширения. Так, например, в язык

C# в каждой версии добавляется множество новых синтаксических конструкций. Но их сложно оценивать с точки зрения создания расширений, так как это внутренние расширения, сильно связанные с языком. В большинстве языков программирования нет системы создания расширений, и для их создания требуется глубокое знание устройства компилятора.

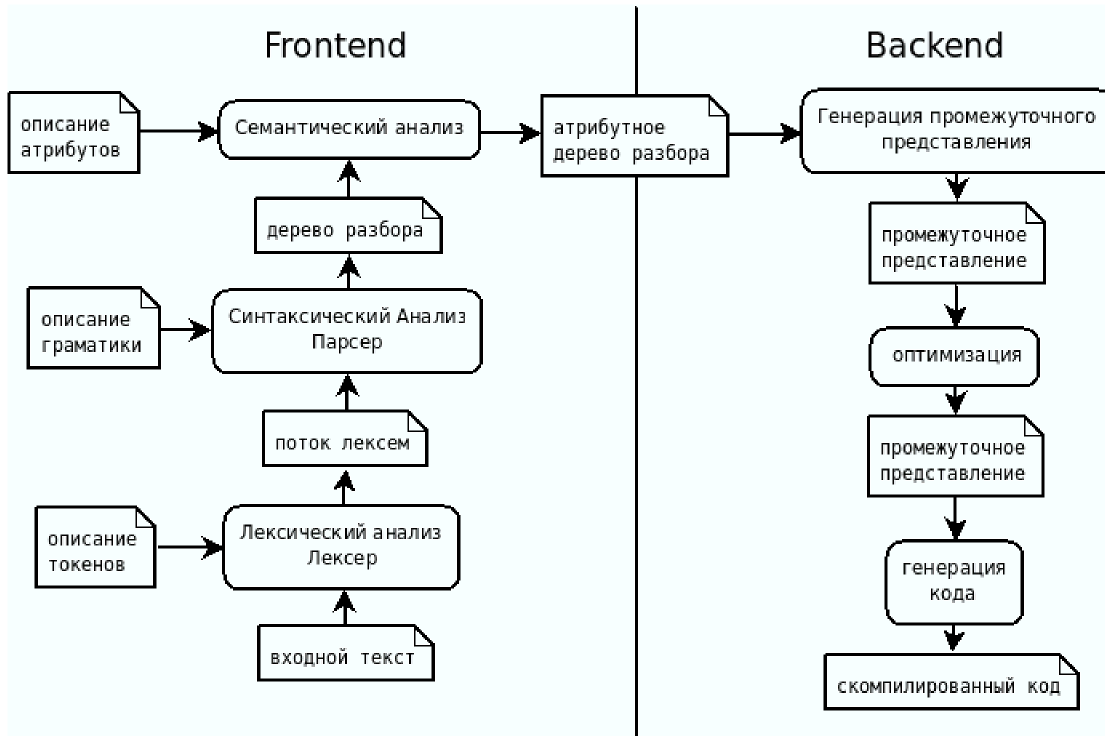
В отличие от C#, язык Java более консервативен в развитии и синтаксически меньше. Java специально создавался синтаксически минимальным для облегчения взаимодействия программистов. Отчасти благодаря этому, Java сейчас является наиболее популярным языком программирования (см. [6]). Вследствие большого сообщества и большей необходимости расширений, у языка Java гораздо больше внешних расширений и проводится больше исследований в этой области чем у C#. Поэтому в данной работе в основном исследуются расширения языка Java и языков, основанных на платформе Java. Автор также не скрывает большого опыта работы с ним и личного предпочтения.

Для создания расширений с нуля используется множество инструментов, облегчающих их создание: генераторы лексеров, генераторы парсеров, компиляторы компиляторов, различные вспомогательные библиотеки, средства связи со средой разработки и прочее. Все это будет рассмотрено в главе 4.

Работа устроена следующим образом. Во введении далее идет обзор типичной структуры компилятора, его возможных точек расширения и список определений, используемых в работе. За ним идет история развития расширений а также их современное состояние. В главе 3 произведена попытка обзора и классификации текущих проектов области. Наконец, в главе 4 содержится обзор инструментов для создания расширений языков. Почти все приведенные в работе программные проекты доступны в исходных кодах (для изучения и использования). Закрытых проектов в данной области очень мало.

1.1 Точки расширения компилятора

Для того, чтобы расширить компилятор, сначала необходимо понять его структуру. Рассмотрим примерную типовую структуру компилятора:



Здесь листки обозначают данные, а закругленные прямоугольники — процессы.

Процесс компиляции обычно состоит из следующих этапов:

1. Лексический анализ. Лексер. На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем, то есть пар (идентификатор лексемы, последовательность символов). Лексемы также часто называют токенами. Модуль, отвечающий за лексический анализ, называется лексером или токенайзером. Не все компиляторы имеют выделенный лексический анализатор, поскольку он может являться незаметной частью синтаксического анализатора.

2. Синтаксический (грамматический) анализ. Парсер. Последовательность лексем преобразуется в дерево разбора в соответствии с грамматикой языка. Модуль, отвечающий за синтаксический анализ, называется парсер. Дерево разбора тут понимается в общем смысле. Обычно в синтаксическом анализе используется синтаксическое дерево (Abstract Syntax Tree, AST), которое отличается от дерева разбора отсутствием несущественных грамматических элементов (например: запятые, скобки, ключевые слова; их можно восстановить по типу и структуре узлов AST).

3. Семантический анализ. Дерево разбора обрабатывается с целью установления его семантики (смысла). Например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т.д. Результатом обычно является расширенное дерево разбора. Часто семантический анализ делается с помощью атрибутной грамматики, и в результате получается расширенное атрибутами дерево разбора.

4. Генерация промежуточного представления. По данным семантического анализа создается представление (например, в форме какого нибудь специального высокоуровнего ассемблера) удобное для дальнейшей оптимизации и генерации кода.

5. Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может производиться на разных уровнях и этапах: например, над промежуточным кодом или над конечным машинным кодом.

6. Генерация кода. Из промежуточного представления порождается код на целевом (обычно машинном) языке.

В конкретных реализациях компиляторов эти этапы могут быть разделены или совмещены в том или ином виде.

Три первых этапа (также, возможно, с включением последующей генерации промежуточного представления) часто называют фронтендом (frontend) компилятора. А три последних этапа — бэкендом (backend). (Фронтенд и бэкенд — это общие понятия, отражают соответственно начальное и конечное состояния процесса. Фронтенд отвечает за получение ввода (входной информации) в любых формах от пользователя и обработку полученной информации в ту форму, которую бэкенд способен использовать. Фронтенд — это интерфейс между пользователем и бэкендом) Причем возможно наличие нескольких взаимозаменяемых фронтендов и бэкендов. Например, в известном наборе компиляторов Gnu Compiller Collections фронтендами являются распознаватели языков программирования C, C++, Pascal, Java, Ada и др. А бэкендами — оптимизаторы и генераторы кода под соответствующие архитектуру процессора (x86, arm, mips, и др.) и операционную систему.

Итак, что же можно расширить в данной схеме? Почти все, если иметь доступ к соответствующим частям компилятора. Традиционные компиляторы этот доступ вовне не предоставляют. И даже если бы и предоставили, то в больших компиляторах он был бы чрезвычайно сложен. Поэтому в большинстве случаев внешние расширения создают путем внедрения

в начало этой схемы своего препроцессора или генератора кода из своего языка на исходный язык компилятора.

Но, допустим, у нас есть удобный доступ ко всем структурам компилятора. Тогда появляются возможности расширения на каждом из уровней:

1. Лексер. Расширением лексера можно ввести новые лексемы, которые могут быть затем использованы в синтаксическом анализаторе. Также можно сменить оформление программы. Например, в компиляторе Nemerle опциональным расширением лексера достигается поведение отступов как в Python ("лесенка" отступов задает структуру программы, т.е. расставляет блоки фигурных скобок).

2. Парсер. Наиболее интересное место для расширений. Здесь можно добавить новые синтаксические конструкции или изменить старые. Например именно на уровне парсера работают макросы в языке Nemerle. Т.е. можно считать макросы Nemerle внутренним расширением парсера.

3. Семантический анализатор. Тут наделяется смысл новым синтаксическим конструкциям. Можно, например, свести новые конструкции к старым, чтобы не заниматься генерацией их промежуточного представления. Также можно добавить какую-нибудь новую семантику существующим синтаксическим конструкциям. Примером расширения семантического анализатора может служить расширение NotNullTypes (см [5]) компилятора JastAddJ которое проверяет правильность присваиваний с точки зрения допустимости переменных быть неопределенными. JastAddJ будет рассмотрен подробнее в разделе 3.3.

4. Бэкенд. Тут возможны дополнительные оптимизационные проходы, генерация дополнительного (например, отладочного) кода.

Проблема тут в том, что почти любая такая модификация компилятора требует его пересборки. Например, малейшее изменение грамматики требует полной пересборки парсера. Таким образом, пользовательские расширения компилятора сильно затруднены. Но расширяемые компиляторы, про которые будет написано в разделе 3.3, пытаются преодолеть эти проблемы различными способами.

1.2 Определения

Приведем некоторые определения, используемые в области:

Язык предметной области (Domain Specific Language, DSL) — это специально разработанный язык для решения определённого круга задач, в отличие от языков программирования общего назначения. Как правило сильно повышает продуктивность работы в этой области, в отличие от универсальных языков. Существуют внутренние и внешние DSL.

Внутренний DSL (embedded или internal DSL) — это DSL внутри другого языка. В языке создается своего рода мини язык через синтаксические возможности базового языка. Таким образом, программа на внутреннем DSL является и программой на базовом языке. В языках семейства Lisp внутренние DSL наиболее легко реализуемы.

Внешний DSL — это DSL, не связанный с базовым языком, в отличие от внутреннего DSL. Он требует отдельного парсера для распознавания. В качестве примера можно привести файлы описания сборки (makefile).

Языково-ориентированное программирование (ЯОП, Language oriented programming, LOP) — стиль программирования, в котором, в отличие от решения задач на универсальных языках, программист сначала создает один или несколько DSL для задачи, и потом решает задачу на этих языках. Понятие ЯОП пришло из языка Lisp, где из-за синтаксических возможностей широко практикуется создание внутренних DSL. Про языки семейства Lisp подробнее написано в разделе 3.4.

Компилятор компиляторов (Compiler compiler) — инструмент для создания парсера, интерпретатора или компилятора из формального описания языка.

Генератор парсеров (Parser Generator) — наиболее ранняя и широко используемая форма компилятора компиляторов. Создает парсер по описанию грамматики.

Дерево разбора (Parse Tree, Concrete Syntax Tree(CST)) — корневое, помеченное дерево, представляющее синтаксическую структуру исходного текста согласно какой-либо формальной грамматике. Внутренние узлы дерева помечены нетерминалами грамматики, а листья дерева — терминалами.

Абстрактное синтаксическое дерево (Abstract Syntax Tree(AST) или просто синтаксическое дерево) — специальная разновидность дерева разбора, используемая в компиляторах. Отличается от дерева разбора отсутствием необязательных для семантики узлов (например, запятые, скобки, ключевые слова) и некоторыми другими структурными отличиями.

2 Обзор области

2.1 История

Область расширяемых языков программирования впервые появилась в 60-ых годах (см [24]). Первая научная работа — "Макросы для высокоуровневых языков программирования" принадлежит Дугласу Маклрою (M. Douglas Mcllroy's) [23]. Другие ранние описания принципа расширяемости приводятся в статье Брукера и Мориса про компилятор компиляторов [26]. Пик движения отмечен двумя академическими симпозиумами в 1969 [15] и 1971 [16]. Обзорная статья по данной области была написана в 1975 году Томасом Стандишом [25], после чего интерес к области начал пропадать. Но интерес к области вновь резко возрастает в 21-ом веке (см [27]).

Характеристика ранних подходов

В 60-ые-70-ые годы, расширяемый язык программирования рассматривался как совокупность из простого базового языка и метаязыка, способного изменять базовый язык. Расширенный язык получается модификациями базового языка с помощью метаязыка. Большинство техник расширений языка в это время были основаны на макроподстановках. Тем не менее модификации грамматик также исследовались, и в результате появились адаптивные грамматики, способные изменять свой синтаксис прямо во время выполнения. Про них подробнее будет рассказано в разделе 4.5. Сообщество языка Lisp оставалось отделенным от сообщества исследователей расширяемых языков видимо потому, что, как исследователь Гаррисон заметил в [22]:

> "любой язык программирования, в котором программа и данные существенно неразличимы, может считаться расширяемым языком. ...это легко можно увидеть из того факта, что Lisp использовался как расширяемый язык многие годы"

На конференции 1969 года был представлен Simula как расширяемый язык программирования (что показывает заинтересованность многих исследователей в расширяемых языках), но в силу возникших трудностей его расширяемость так и не была реализована.

Стандиш в [25] описал три класса расширений языков, которые он назвал перефраз (paraphrase), ортофраз (orthophrase) и метафраз (metaphrase).

- Перефраз добавляет новые возможности, сводя их к уже имеющимся. В качестве примера можно привести макроподстановки, которые в конечном счете разворачиваются в конструкции базового языка.

- Ортофраз добавляет возможности, невыразимые в базовом языке, такие как, например, добавление системы ввода-вывода к языку, не имеющему оной. Таким образом, ортофраз-расширение должно быть написано на другом языке. Из современных понятий близким аналогом является понятие подключаемого дополнения (plug-in).

- Метафраз модифицирует интерпретацию существующих элементов языка, и, таким образом, выражения разбираются по-новому. Оно соответствует современному понятию рефлексии.

Закат ранних подходов

Стандиш в [25] аргументировал неудачу движения сложностью программирования последующих расширений. Обычный программист может создать некое расширение вокруг базового языка, но если второе расширение будет создано вокруг первого, то программисту нужно быть близко знакомым и с базовым языком, и с первым расширением. Третье расширение будет требовать еще больше знаний и т.д. Следует заметить, что отделение программиста от низкоуровневых деталей, — это цель подхода, предполагающего введение абстракции. Этот подход вытеснил исследования расширяемости в то время.

2.2 Современное состояние

В современном понимании система, поддерживающая расширяемое программирование, предоставляет все нижеперечисленные возможности (см.[12],[27]):

Расширяемый синтаксис

Расширяемый синтаксис предполагает, что язык не должен быть фиксированным или статичным. Он должен предоставлять возможность добавлять новые синтаксические конструк-

ции. Допустима неизменность некоторых фундаментальных и внутренних свойств языка, но система не должна полностью от них зависеть.

Расширяемый компилятор

В расширяемом программировании компилятор не является монолитной программой, которая преобразует исходный код в бинарный исполняемый формат. Компилятор сам должен быть расширяем через набор дополнений, которые помогают в преобразовании исходного языка во что угодно. Например, расширяемый компилятор будет генерировать объектный код, документацию к коду, переформатированный исходный код или любую другую желаемую информацию. Архитектура компилятора должна предоставлять интерфейсы для доступа внутрь процесса компиляции и предоставлять возможность использовать альтернативные задачи обработки на каждом шаге процесса компиляции.

Для простой задачи трансляции исходного кода в что-либо, способное запуститься на компьютере, расширяемый компилятор должен:

- Использовать модульную (plug-in) или компонентную архитектуру для почти каждого шага компиляции.
- Определить язык или вариант языка, который будет компилироваться, и найти подходящий компонент для распознавания и проверки этого языка.
- Проверять семантическую правильность путем запуска соответствующего компонента.
- Иметь возможность выбора из нескольких генераторов кода для разных процессоров, операционных систем, виртуальных машин и пр.
- Предоставить методы для генерации ошибок и их расширений.
- Предоставить возможность добавлять новые узлы, значения и ребра в синтаксическое дерево.
- Предоставить возможность трансляции и трансформации синтаксического дерева или поддерева через внешние компоненты.
- Обеспечить поток информации между внутренними и внешними компонентами

Расширяемая среда исполнения

Среда исполнения расширяемой системы программирования должна позволять языкам расширять множество допустимых операций. Например, если система использует интерпретатор байт-кода, то она должна иметь возможность определения новых команд и соответствующих им представлений в байт-коде. Как и в расширяемом синтаксисе допустимо наличие небольшого набора неизменных фундаментальных внутренних инструкций. Тем не менее, должно быть возможно перегрузить (overload) эти операторы так, чтобы появились новые или дополнительные возможности.

Отделение содержания от формы

Расширяемые системы программирования должны рассматривать программы как данные для обработки. Эти программы должны быть полностью лишены любой информации о форматировании. Система должна преобразовать программу в форму, наиболее подходящую для просмотра, редактирования или генерации кода.

Поддержка отладки создаваемых языков

Расширяемая система программирования должна поддерживать отладку программ, используя конструкции языка, на котором она записана, вне зависимости от количества расширений языка и трансформаций программы. Отладчик должен позволять отображать данные в форме, подходящей для языка. Например, если язык поддерживает структуры данных для бизнес процессов или потоков выполнения, то его отладчик должен выводить эти структуры в виде диаграмм Ишикавы [10] (Ishikawa, fishbone, cause-and-effect diagrams) или в другой форме, предоставляемой неким специальным компонентом.

Такое современное определение расширяемой системы программирования требует очень многого. Получился своего рода идеал. Истинно расширяемых систем в таком понимании на данный момент нет. Наиболее близким к данному определению является, наверное, XLR: Extensible Language and Runtime, про который подробнее написано в 3.4. В данной работе будет рассматриваться более широкий класс расширений, в которых не все эти требования выполнены.

3 Классификация

Расширения компилятора можно условно подразделить на синтаксические и семантические. Синтаксические расширения добавляют новые синтаксические конструкции в язык. Например, перечисления (enum), обобщенные типы (generics) в Java 5. Семантические расширения, в отличие от синтаксических, просто меняют поведение кода без введения новых синтаксических конструкций. Примером семантических расширений является аспектно-ориентированное программирование (АОП), которое будет рассмотрено в разделе 3.1.

3.1 Семантические расширения

Семантические расширения меняют поведение программы каким-либо внешним образом. Например, через препроцессор исходного или байт-кода. Рассмотрим представителей данного класса:

Метапрограммирование

Метапрограммирование — область ориентированная на создание программ, которые создают другие программы как результат своей работы (либо, в частном случае, изменяющие или дополняющие себя во время выполнения).

Метапрограммирование разделяется на 2 направления: на стадии компиляции (генерация кода) и на стадии выполнения (самомодифицирующийся код). Первое направление позволяет получить программу при меньших затратах времени и усилий, чем если бы программист писал её вручную. Второе — расширяет возможности программиста.

Метапрограммирование применяется как метод во многих областях.

Аспектно-ориентированное программирование

Аспектно-ориентированное программирование (АОП) — это методика программирования в рамках классовой парадигмы, основанная на понятии аспекта, т.е. блока кода, инкапсулирующего сквозное поведение в составе классов и повторно используемых модулей.

Аспектно-ориентированное программирование выросло из осознания того, что в типовых программах на объектно-ориентированных языках часто представлено поведение, которое не вмещается естественно в один или даже в несколько тесно связанных программных модулей. Пионеры аспектного подхода ввели термин «пересечение» (crosscutting, сквозная функциональность) для обозначения поведения кода, при котором пересекаются ответственности

разработчиков программных модулей. В объектно-ориентированном программировании, например, единицей модульности является класс, а «секущее» свойство охватывает несколько классов. Часто пересечение встречается при организации журналирования приложений, контекстно зависимой обработке ошибок, оптимизации выполнения программ, а также в шаблонах проектирования.

АОП дополняет объектно-ориентированное программирование, обогащая его другим типом модульности, который позволяет локализовать код реализации логики пересечения в одном модуле. Такие модули обозначаются термином аспекты, от аспектно-ориентированного программирования. Аспекты в системе могут изменяться, вставляться, удаляться на этапе компиляции и, более того, повторно использоваться.

Все языки АОП предоставляют способы для выделения сквозной функциональности в отдельную сущность. Различие между ними заключается в удобстве, безопасности и области применения средств, которые они предоставляют. Наиболее популярный на данный момент язык АОП — AspectJ. Используемые в нем понятия распространились на большинство языков АОП.

Рассмотрим AspectJ на примере аспекта для журналирования событий. Допустим у нас есть множество методов doSomething, в которых необходимо журналировать моменты входа и выхода, например:

```
public void doGet(JspImplicitObjects theObjects) throws ServletException
{
  logger.entry("doGet(...)");
  JspTestController controller = new JspTestController();
  controller.handleRequest(theObjects);
  logger.exit("doGet");
}
```

Вручную писать журналирование для каждого метода требует существенных усилий, и может привести к трудно выявляемым ошибкам, если программист немного ошибется в строке журналирования. Но эта задача легко решается введением сквозной функциональности через следующий аспект:

```
public aspect AutoLog
{
  pointcut publicMethods() : execution(public * my.package..*(..));
```

```
  pointcut logObjectCalls() : execution(* Logger.*(..));
  pointcut loggableCalls() : publicMethods() && ! logObjectCalls();
  before() : loggableCalls() {
    Logger.entry(thisJoinPoint.getSignature().toString());
  }
  after() : loggableCalls() {
    Logger.exit(thisJoinPoint.getSignature().toString());
  }
}
```

Проанализируем этот пример и посмотрим, какие действия осуществляет аспект. Первое, на что нужно обратить внимание — это объявление аспекта. Оно подобно объявлению класса и, так же как класс, определяет тип Java. Кроме того, аспект содержит конструкции pointcut и advice.

Прежде всего, рассмотрим, что представляет собой соединения (join point). Точки соединения — это однозначно определенные точки при выполнении программы. Так, под точками соединения в AspectJ подразумеваются: вызовы методов, точки обращения к членам класса, исполнение блоков обработчиков исключений и т. д. Точки соединения могут, в свою очередь, содержать другие точки соединения. Например, результат вызова метода может передаваться каким-то другим методам. А срез (pointcut) является языковой конструкцией, которая отбирает множество точек соединения на основании определенного критерия. В приведенном выше примере первый срез под именем publicMethods выбирает исполнения всех public методов в пакете my.package. Подобно int, который является базовым типом Java, execution является базовым срезом. Он выбирает исполнения методов, соответствующих сигнатуре, заданной в скобках. Для сигнатур допустимо включение символов шаблонов: в приведенном примере оба среза содержат несколько таких символов. Второй срез с именем logObjectCalls выбирает все исполнения методов в классе Logger. Третий срез loggableCalls объединяет два предыдущих, используя булевы операции && и !, что означает выбор всех public методов из my.package за исключением таковых в классе Logger (регистрация методов в классе Logger привела бы в результате к бесконечной рекурсии).

Теперь, после того, как в аспекте определены срезы, нужно использовать конструкцию advice (совет), чтобы выполнить текущую регистрацию. Совет — это фрагмент кода, выполняющийся до, после или в составе точки соединения. Совет определяется для pointcut, что представляет собой нечто наподобие указания «выполнить этот код после каждого вы-

зова метода, который надо регистрировать». В итоге «след» вызова метода (с применением аспекта) в log-файле выглядит примерно следующим образом:

```
entering method: void test.Logging.main(String[])
entering method: void test.Logging.foo()
exiting method: void test.Logging.foo()
exiting method: void test.Logging.main(String[])
```

В то время как pointcut и advice позволяют влиять на динамику выполнения программы, introduction (внедрение) предоставляет аспектам возможность модифицировать статическую структуру программы. Используя внедрение, аспекты могут добавлять новые методы и переменные в классы, объявлять класс как реализацию интерфейса, устанавливать или отменять проверку исключений.

У AspectJ существует хорошая инструментальная поддержка: среды программирования Eclipse, NetBeans и Idea имеют расширения для удобной работы с AspectJ. AspectJ легко применять и в системах сборки ant, maven.

Также интересно мнение сторонников языка Лисп об AspectJ:

> “В Лиспе, если охота аспектно-ориентированного программирования, нужно лишь настругать немного макросов, и готово. В Java, нужен Грегор Кичалес, создающий новую фирму, и месяцы и годы попыток заставить всё работать.” (Петер Норвиг)

АОП можно рассматривать как вариант метапрограммирования при компиляции. Обычно аспекты описываются расширенным базовым языком, поэтому такое АОП можно рассматривать и как синтаксическое расширение. Но существуют и реализации АОП обходящиеся только средствами языка (например, с помощью аннотаций Java 5).

Макроподстановки

Макрос, макроподстановка (от англ. macros, мн.ч. от macro) — программный объект, при обработке «развертывающийся» в последовательность действий или команд. Таким образом, макросы можно отнести к метапрограммированию на стадии компиляции.

Макросы по отношению к языку можно разделить на внешние и внутренние. Внешние макросы не связаны с языком и не имеют доступа к контексту применения. Пример внешних макросов — препроцессор языка C, который реализован внешней утилитой (cpp, The C PreProcessor) и никак не связан с языком.

Внутренние макросы, в отличие внешних, имеют доступ к контексту применения, и поэтому обладают гораздо большей выразительной способностью. Как следствие, внутренние макросы тесно связаны с языком. При применении они имеют доступ к дереву разбора и могут изменять его.

Наиболее мощными внутренними макросами обладают языки семейства Lisp (Common Lisp, Scheme). Языки Nemerle и Boo также обладают мощной встроенной системой макросов, с помощью которых можно даже добавлять новые синтаксические конструкции в язык. Для языка Java существует система внутренних макросов Java Syntactic Extender, которая будет рассмотрена в разделе 3.2.

Системы трансформации программ

Техники трансформации программ используются во многих сферах: от генерации программ, оптимизации и рефакторинга до обратной разработки и генерации документации. Данная область является, наверное, самой старой. Многие теории, инструменты и приложения в этой области разработаны более 30 лет назад. В связи большим возрастом области в ней сформировались некоторые стандарты. В первую очередь это SDF [7] — syntax definition formalism, формат файлов для описания грамматик контекстно-свободных языков. Форматом SDF пользуется большинство систем трансформации.

Кроме того проект SDF предоставляет стандартный генератор парсеров класса SGLR (Scanerless Generalized Left-to-right Rightmost derivation parser). SGLR иногда называют параллельным парсером. Это расширение LR парсера для поддержки недетерминированных и неоднозначных грамматик. Алгоритмическая сложность разбора имеет порядок $O(n^3)$. При отсутствии неопределенностей — $O(n)$. Стоит заметить, что большинство парсеров, используемых в компиляторах имеют линейную сложность, поэтому кубическая сложность SGLR выглядит плохо. Но SGLR ценен тем, что может распознавать недетерминированные и неоднозначные грамматики. Его соперниками здесь являются LL(*) грамматики (требуют специальных указаний для разрешения конфликтов) и адаптивные грамматики (очень сложны). Большой класс распознаваемых языков позволяет распознавать сложные языки и создавать большие расширения языков.

Рассмотрим некоторые системы трансформации:

- Stratego/XT — это язык трансформации Stratego и набор инструментов XT для построения систем трансформации. Язык Stratego основан на парадигме стратегической перезаписи термов (strategic term rewriting). Для описания грамматики используется

стандартный формат SDF. На основе Stratego/XT создан фронтенд JavaFront, распознающий язык Java.

- ASF+SDF Meta Environment — это среда и набор инструментов для интерактивного анализа и трансформаций. ASF+SDF комбинирует описанный выше Syntax Definition Formalism и Algebraic Specification Formalism (ASF), а также некоторые другие технологии. ASF используется для перезаписи термов. В отличие от Stratego/XT отличается ориентацией на среду программирования (MetaStudio IDE) и визуализацию процессов.

- TermWare — это система обработки термов, предназначенная для встраивания в Java приложения. TermWare не использует SDF и вообще не содержит своего парсера. Сам TermWare предназначен только для обработки термов. На основе TermWare и JavaCC создан фронтенд JavaChecker, распознающий язык Java и позволяющий проверять некоторые семантические ошибки, используя правила TermWare. TermWare — проект украинской фирмы Градсофт [8].

3.2 Синтаксические расширения

Как уже было сказано, синтаксическое расширение добавляет в язык новые синтаксические конструкции и связывает с ними какую-либо семантику. Рассмотрим подробнее представителей этого класса.

Система Java Syntactic Extender

Java Syntactic Extender (JSE) — расширение языка Java, позволяющее добавлять в язык новые синтаксические конструкции. Оно основано на системе макросов Dylan с учетом специфики языка Java. Расширения могут быть реализованы непосредственно на Java без каких-либо ограничений. JSE — это препроцессор языка Java, который разворачивает макросы.

JSE использует парсер языка Java с макросами, основанный на генераторе парсеров ANTLR. Парсер строит специальное дерево разбора, оптимизированное для макроподстановок, которое в JSE называется скелетным синтаксическим деревом (Skeleton Syntax Tree, SST). В дальнейшем макросы явно или неявно манипулируют именно SST.

Сами макросы представляют собой обычные классы Java, которые, имея доступ к SST, разворачивают вызов макроса во что угодно. После применения всех макросов SST преобразуется обратно в текст на языке Java для последующей обработки стандартным компилятором.

Недостатки:

- не учитывается область видимости, возможен конфликт имен;
- SST предоставляет мало контекстной информации;
- проект заброшен. последняя версия 0.20 вышла в 2003 году.

Для успешности расширения языка нужна еще и его инструментальная поддержка. Наиболее существенна поддержка среды программирования, которой у JSE не было. Наверное это и явилось причиной низкого интереса к этому проекту. Тем не менее проект ценен тем, что является единственной системой внутренних макросов для языка Java и позволяет с их помощью легко добавлять новые синтаксические конструкции в язык.

3.3 Расширяемые компиляторы

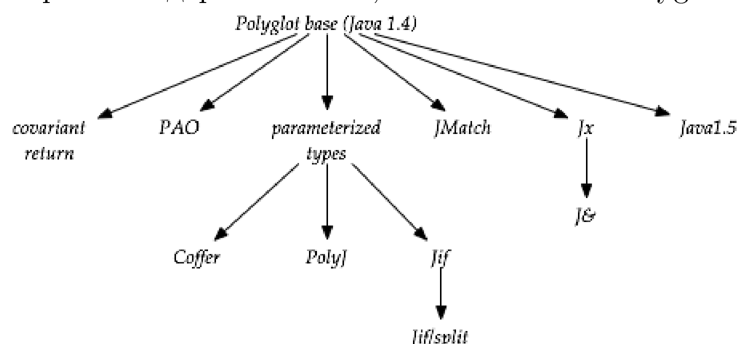
В данном разделе приведены компиляторы, которые заявляют о своей расширяемости. Такие компиляторы позволяют создавать расширения, изменяющие процесс компиляции.

Компилятор Polyglot

Polyglot — расширяемый текстовый преобразователь Java программ (транслятор из Java в Java). Polyglot состоит из базового компилятора jlc и его расширений. Компилятор jlc — полноценный Java фронтенд. Он разбирает исходный код и проводит его семантический анализ. Компилятор генерирует Java код. Таким образом, базовый компилятор не производит трансляции в байт-код, в отличие от стандартного компилятора Java.

Языковые расширения реализуются поверх базового компилятора путем расширения конкретного и абстрактного синтаксиса, создания новых типов и определения новых трансформаций кода. В результате получается абстрактное синтаксическое дерево, которое преобразуется в код на Java и компилируется стандартным компилятором javac.

Фрагмент дерева языков, основанных на Polyglot:



В Polyglot используется Polyglot Parser Generator (PPG) — собственный генератор парсеров на основе JavaCUP (см [2]). В PPG для поддержки синтаксических расширений были добавлены возможности модификации грамматики. Пример переопределения грамматики языка Java на PPG:

```
include "polyglot/parse/java12.cup"
...
drop { relational_expression ::=
    relational_expression:a INSTANCEOF reference_type:b; }
extend relational_expression ::=
    relational_expression:a INSTANCEOF type:b
    {: RESULT = parser.nf.Instanceof(parser.pos(a), a, b); :}
    ;
```

Пример убирает ограничение языка Java на использование только ссылочных типов в качестве аргумента оператора *instanceof*. Теперь возможно применение также и примитивных типов. Пример взят из расширения "примитивы как объекты" (primitives as objects, PAO) языка Java 1.4.

Polyglot — пример наиболее популярного подхода к созданию расширений языка: сведение конструкций нового языка к конструкциям на базовом языке. Но, в отличие от других подобных проектов, Polyglot смог собрать вокруг себя большое сообщество пользователей, что привело к появлению многих расширений языка Java на его основе. В последнее время активность проекта упала, и некоторые проекты (например Soot, Aspect Bench Compiller(abc)) переходят на использование более перспективного проекта JastAdd.

Система JastAdd. Компилятор JastAddJ

Фактически JastAdd — это специальное расширение языка AspectJ для создания трансляторов и семантических анализаторов. JastAdd позволяет сделать из обычного синтаксического дерева (AST) атрибутную грамматику (а точнее, Rewritable Circular Reference Attributed Grammar — циклическую (допустимы циклические ссылки), ссылочную, атрибутную грамматику с возможностью перезаписи узлов, ReCRAG) средствами АОП. JastAdd добавляет атрибуты, их расчет и различные действия над атрибутной грамматикой прямо в классы AST. В результате получаются расширенные классы AST с атрибутами. Расчет атрибутов можно писать как в декларативном (рекомендуется для расширяемости), так и в импера-

тивном стиле. JastAdd не содержит генератора парсера и требует для работы использования любого внешнего парсера и AST.

JastAddJ — расширяемый компилятор Java на основе JastAdd. Компилятор примечателен тем, что почти целиком полагается на атрибутную грамматику. У JastAddJ этапы семантического анализа, генерации промежуточного представления, оптимизации и генерации кода выполняет атрибутная грамматика, предоставляемая системой JastAdd. В итоге JastAddJ состоит почти целиком из определения и расчетов атрибутов грамматики. JastAddJ в качестве генератора парсеров использует Beaver — малоизвестный генератор парсеров, который, как и LPG, эмулирует множественное наследование просто путем включения всех файлов в результирующую грамматику.

Пример использования атрибутов для вывода переформатированного кода (prettyprinting):

```
// для всех выражений определяем атрибут pp()
syn String Expression.pp()
// Литералы выводятся просто как значения
eq Literal.pp() = value();
// для отображения бинарных выражений, формируем строку:
eq BinaryExpression.pp() = left().pp() + op() + right().pp();
// определяем атрибут op() для бинарных выражений:
syn BinaryExpression.op();
eq AddExpression.op() = "+";
eq SubExpression.op() = "-";
```

Проект JastAddJ состоит из:

1. Java1.4 frontend — разбирает Java 1.4 код, создает AST и производит статико-семантический анализ (соответствие типов и т.д.).

2. Java1.4 backend — расширение Java1.4 frontend. Генерирует стандартный байт-код для JVM.

3. Java1.5 frontend — расширение Java1.4 frontend. Добавляет синтаксические расширения Java 1.5 (generics, annotations, и пр.).

4. Java1.5 backend — расширение Java1.5 frontend и Java1.4 backend. Генерирует байт-код.

Компилятор JastAddJ проходит большинство тестов тестового комплекта Jacks, больше чем другие компиляторы, включая javac, ejc, polyglot jlc, jikes. При этом JastAddJ медленнее

javac всего примерно в 3 раза (что неплохо, т.к. JastAddJ предоставляет больше функций) и меньше всех по объему исходного кода [19].

Атрибутная грамматика и декларативный стиль расчета атрибутов способствуют легкой расширяемости компилятора и обуславливают его малый размер. Но компилятор JastAddJ, как и Polyglot, не предоставляет внешних интерфейсов расширения. Таким образом, для создания расширений все равно необходимо разбираться во всем внутреннем устройстве компилятора.

В отличие от Polyglot, сообщество JastAdd довольно мало, и количество расширений не так велико. Несмотря на это, JastAdd успешно используется в некоторых проектах в качестве экспериментальной замены Polyglot. Стоит заметить, что JastAdd абсолютно не совместим с Polyglot. У них разные принципы и способы работы, но цель одна — создание расширяемых компиляторов.

3.4 Расширяемые языки программирования

В данном разделе приведены расширяемые языки программирования. От расширяемых компиляторов расширяемые языки отличаются возможностью изменения синтаксиса на уровне языка с помощью специальных синтаксических конструкций.

Семейство языков Лисп

Лисп (Lisp, LISt Processing language — язык обработки списков) — семейство языков программирования, программы и данные в которых представляются системами линейных списков символов. Лисп является вторым в истории (после Фортрана) высокоуровневым языком программирования, который используется по сей день. Создатель языка, Джон Маккарти, занимался исследованиями в области искусственного интеллекта, и созданный им язык по сию пору является одним из основных средств моделирования различных аспектов ИИ.

Традиционный Лисп имеет динамическую систему типов. Язык является функциональным, но многие поздние версии обладают также чертами императивности. К тому же, имея полноценные средства символьной обработки, становится возможным реализовать объектно-ориентированность. Примером такой реализации является платформа CLOS — Common Lisp Object System.

Язык Лисп, наряду с языком Ada, прошел процесс фундаментальной стандартизации для использования в военном деле и промышленности, в результате чего появился стандарт Common Lisp. Его реализации существуют для большинства платформ.

Лисп имеет жесткий базовый синтаксис скобочных выражений. Остальное расширяемо функциями, макросами, лямбда-исчислением и концепцией кода программы как данных.

Пример определения макроса на Lisp:

```
(defmacro nif (expr pos zero neg)
  `(case (truncate (signum ,expr))
     (1 ,pos)
     (0 ,zero)
     (-1 ,neg)))
```

Если мы далее в программе запишем $(nif\ x\ 'p\ 'z\ 'n)$ то макрос nif развернется в:

```
(case (truncate (signum x))
  (1 'p)
  (0 'z)
  (-1 'n))
```

Т.е. в зависимости от знака выражения x будут выполнятся выражения $'p$, $'z$ или $'n$.

Язык Boo

Boo — объектно ориентированный статически типизируемый язык для платформы .NET. У языка Boo схожий с Python синтаксис. Также Boo специально ориентирован на расширяемость языка и компилятора. Как и в Nemerle, язык Boo позволяет определять синтаксические макросы, рекурсивно разворачивающиеся в базовые конструкции языка путем манипуляций с синтаксическим деревом. Макросы могут быть записаны просто в виде объекта языка, как в Java Syntactic Extender, или с помощью специальных синтаксических конструкций, как в Nemerle. Когда при синтаксическом разборе встречается неизвестная синтаксическая структура, парсер Boo проверяет, является ли она макросом, и если является, вызывает соответствующие методы у класса макроса.

Рассмотрим пример определения и использования макроса на языке Boo. В качестве примера возьмем макрос “with” (из языка VisualBasic), который позволяет писать вместо такого длинного кода:

```
fooInstanceWithReallyLongName.f1 = 100
fooInstanceWithReallyLongName.f2 = "abc"
fooInstanceWithReallyLongName.DoSomething()
```

такой компактный код:

```
with fooInstanceWithReallyLongName:
    .f1 = 100
    .f2 = "abc"
    .DoSomething()
```

Определение макроса “with” на Boo

```
macro with(target, body as Expression*):
 for expression in body:
  match expression:
   case BinaryExpression(Left: mre = MemberReferenceExpression(Target: \
      OmittedExpression())):
    mre.Target = target
    yield
```

Язык Boo, как и Nemerle имеет ограниченную интеграцию со средами программирования MS Visual Studio 2008, SharpDevelop и MonoDevelop.

Язык Nemerle

Nemerle — это гибридный язык высокого уровня для платформы .NET со статической типизацией, органично сочетающий в себе возможности функционального и объектно-ориентированного программирования. Синтаксически похож на языки C#, ML, OCaml, Haskell. Главная особенность — это очень мощная система метапрограммирования.

Ряд языковых средств кардинальным образом отличает Nemerle от C#, Java, C++. Это макросы и замыкания, причём в виде, более характерном для Lisp или других функциональных языков, нежели для C++. Система макросов позволяет описывать на Nemerle новые синтаксические конструкции и использовать их наравне со встроенными. В действительности, большинство директивных управляющих конструкций, в том числе операторы if, when, циклы всех видов реализованы в виде макросов стандартной библиотеки Nemerle.

Макросы представляют собой описание синтаксиса и действий. В действиях возможно применение квази-цитирования (преобразование кода в цитате в AST для добавления в синтаксическое дерево). Макросы компилируются предварительно в библиотеки .NET, и в дальнейшем используются при разборе исходного кода с макросами. Также как и в Boo, макросы

применяются только на подходящих, но неизвестных базовому парсеру, синтаксических конструкциях. Но в отличие от Boo, макросы в Nemerle требуют отдельной компиляции перед использованием. Также невозможно использовать макрос сразу после определения, в одном файле.

Макрос с квазицитированием на Nemerle

```
macro ReverseFor (i, begin, body)
syntax ("ford", "(", i, ";", begin, ")", body)
{
  <[ for ($i = $begin; $i >= 0; $i--) $body ]>
}
```

В примере определяется макрос “ford”, означающий обратный цикл от $begin до 0.

Система макросов Nemerle считается одной из самых мощных. Многие синтаксические возможности языка на самом деле являются макросами. С помощью его макросов можно, например, реализовать внутренние языки предметной области, частичные вычисления, некоторые возможности аспектно-ориентированного программирования и прочее.

Nemerle имеет опциональную возможность использовать отступы как структуру программы (identation based syntax), подобно тому как это сделано в языке Python. Т.е. вместо использования скобок для обозначения блоков, можно использовать символ табуляции для их оформления (т.е. так называемая “лесенка” задает структуру, а не наоборот). Это можно рассматривать как пример расширения лексера в Nemerle.

Язык Boo довольно близок по возможностям к Nemerle. Из различий можно выделить бо́льшую концентрацию языка Nemerle на макросы и функциональное программирование, а также его научную базу (по языку Nemerle было написано много научных работ). Но язык Boo обладает большим сообществом пользователей и, судя по последним выпускам, более активно развивается в последнее время.

Язык Scala

Scala — это, подобно Nemerle, гибридный язык программирования, спроектированный кратким и типобезопасным для простого и быстрого программирования. В нем органично сочетаются возможности функционального и объектно ориентированного программирования. Основной целью разработки был язык, обладающий хорошей поддержкой компонентного ПО.

Говорят, что Scala — расширяемый язык. На самом деле синтаксис Scala просто настолько широк, что позволяет легко создавать внутренние языки предметной области на основе Scala. Этому способствуют closures (анонимные функции, также известные как замыкания), обилие “синтаксического сахара” (то есть синтаксических сокращений, например: “a+b” вместо “a.+(b)”) и функциональная природа языка.

Система XLR: Extensible Language and Runtime

Система XLR состоит из расширяемого языка XL и среды его исполнения. В рамках языка был введен стиль концептного программирования. Концептом называется какая-либо сущность задачи, которая важна для программы. Концептное программирование исходит из простой идеи: код должен отражать концепты в приложении. Программирование в этом контексте является искусством трансформации концептов в код. Другими словами, программирование отображает концепты из определенного пространства проблем в определенное пространство кода. Концептное программирование акцентирует внимание на том, как программисты переносят концепты в их представление. Это отличается от других методологий, которые акцентируют внимание на наборе технологий или практик, часто полученных эмпирически из опыта. Таким образом, язык XL пытается решать проблемы наиболее прозрачно и лаконично с минимальным синтаксическим шумом (лишними синтаксическими конструкциями, не отображающими концепт). Впрочем, понятия концепт в языке XL нет, поэтому концептное программирование можно условно считать просто красивыми словами.

XLR почти соответствует современному определению расширяемой системы программирования, описанной в разделе 2.2. XLR включает язык с расширяемым синтаксисом XL, модульный компилятор, виртуальную машину. Но так как XLR не предоставляет среды разработки, то про поддержку отладки говорить не приходится.

Изначально XLR была написана на C++, но потом была переписана на самой себе (bootstraping). Проект XLR довольно интересен, но малоактивен. Если бы проект был написан на Java и использовал JVM, то его сообщество было бы гораздо больше.

Приведем пример вычисления факториала на XL:

```
function Factorial(N : integer) return number written N! is
    if N = 0 then
        return 1
    else
        return N * (N-1)!
```

В примере вводится синтаксическая конструкция $N!$, которая вызывает функцию $Factorial(N)$ для вычисления факториала от N.

4 Средства создания расширений

Для создания компиляторов используется множество вспомогательных инструментов. В данном разделе приведены наиболее подходящие из них для реализации именно расширяемых компиляторов и языков.

4.1 Среда JetBrains MPS

Meta Programming System — среда разработки DSL и их интеграции в IDE Idea. Позволяет с легкостью создать свой язык, редакторы и генераторы к нему. Отличается от остальных инструментов тем, что хранит исходные файлы языка не в простом текстовом формате, а в виде синтаксического дерева (в XML). Таким образом лексический, синтаксический анализ и сериализаторы становятся не нужны, а также снимаются многие проблемы синтаксической расширяемости, но пользователь становится сильно зависимым от редактора.

Редактирование языка ведется путем создания типов и структуры синтаксического дерева (структурного концепта в терминологии MPS). Далее для каждого нового типа создается описание редактора, которое определяет то, как будет выглядеть часть редактора языка для данного типа синтаксического дерева. Делается это путем расстановки полей ввода и различных подписей. Редактор редакторов — одна из сильнейших сторон MPS. Пользоваться им достаточно просто.

Генератор представляет собой шаблон кода, который применяется к языку. В результате может получится, например, интерпретатор языка на Java. Редактор генераторов позволяет легко создавать сложные шаблоны с подстановками, итерациями и пр.

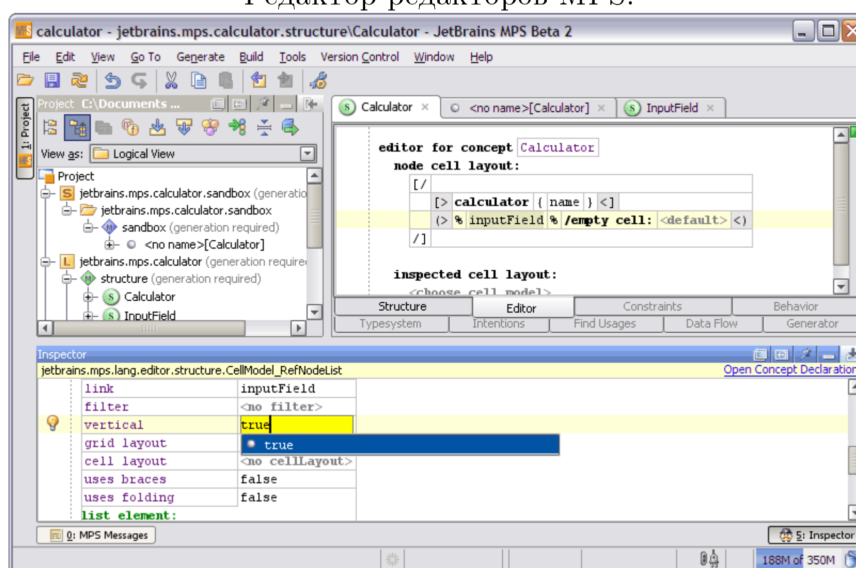
MPS поддерживает множественное наследование языков. При этом просто происходит объединение всех типов синтаксического дерева всех языков. Благодаря хранению исходных файлов в виде синтаксического дерева, снимается проблема множественного наследования, имеющаяся в других инструментах.

В MPS реализовано множество языков, в том числе сама Java, которая применяется в редакторе генераторов. Пользоваться MPS новичкам очень просто. Но для хорошо знакомых с языком, MPS может показаться слишком ограничивающим, т.к. не разрешает свободное редактирование и форматирование кода (ввод возможен только в поля ввода). Таким об-

разом, MPS предоставляет свободу изменения грамматики языка, взамен на отказ от его лексической части. Но для многих программистов невозможность форматирования кода как им нравится является неким психологическим препятствием. Тем не менее это препятствие преодолимо (в качестве примера смежной проблемы можно привести инструменты проверки стиля кода, которые широко используются при разработке ПО в достаточно больших группах).

Большая часть исходного кода MPS открыта под лицензией Apache 2.0. Закрытой является используемая в MPS часть IDE Idea, которая является коммерческим продуктом. На данный момент доступна версия 1.0beta2.

Редактор редакторов MPS:



4.2 Среда Textual Modeling Framework(TMF) Xtext

Xtext — аналог MPS для IDE Eclipse. Является набором инструментов для создания внешних DSL и редактора к ним на основе Eclipse. Основано на таких технологиях проекта Eclipse как Eclipse Modeling Framework(EMF), Graphical Modeling Framework(GMF), Model To Text (M2T), Eclipse Modeling Framework Technology (EMFT). На текущий момент (версия 0.7.0RC4) проект находится на стадии разработки (Eclipse incubation). Планируется интеграция в следующую версию Eclipse 3.5 Galileo.

Грамматика одновременно с моделью задается с помощью компактного и выразительного языка, в форме, похожей на сильно расширенную EBNF. Поддерживается наследование (не множественное) грамматик.

Xtext по грамматике языка создает:

- инкрементальный парсер и лексер, основанные на ANTLR (в разработке собственный генератор парсеров packrat);
- мета модель, основанную на Ecore;
- сериализатор, используемый для преобразования мета модели обратно в текстовую форму, с сохранением исходного форматирования;
- интеграцию языка в Eclipse IDE:
  - подсветка синтаксиса;
  - навигация (F3, и т.п.);
  - автодополнение кода (ctrl-space);
  - outline view;
  - шаблоны кода.

В качестве языка для генераторов кода, в Xtext используется Xpand (статически-типизируемый шаблонный язык).

Пример описания небольшого языка определения структуры сущностей в Xtext:

```
Model:
  (types+=Type)*;
Type:
  DataType | Entity;
DataType:
  "datatype" name=ID;
Entity:
  "entity" name=ID "{"
    (features+=Feature)*
  "}";
Feature:
  type=[Type|ID] name=ID;
```

По мнению автора Xtext хорошо подходит для создания небольших (в грамматическом смысле) языков предметной области, но не для создания полноценных языков программирования. Во первых Xtext позиционируется для построения внешних DSL, а внешние DSL

обычно являются небольшими. Во вторых, используемый в Xtext генератор парсеров не поддерживает синтаксических и семантических предикатов, которые требуются в больших и сложных языках для разрешения неоднозначностей в грамматике. И в третьих при создании языка задействуется очень много технологий, генерируется внушительная модель языка, которая используется при распознавании. Причем эта модель содержит много ненужной информации, что в результате приводит к чрезмерному потреблению памяти. Впрочем это важно только для действительно больших языков. Также Xtext не поддерживает множественное наследование, что не позволяет объединить несколько независимых расширений одного языка. Тем не менее, эти недостатки являются преодолимыми в техническом плане, и учитывая большую активность проекта, возможно разрешатся в будущем.

4.3 Среда Eclipse IDE Meta-Tooling Platform

Eclipse IMP — это набор инструментов для создания среды разработки для произвольных языков. Eclipse — это отличная среда для разработки на Java, но для других языков программирования сделать подобную среду очень сложно. Цель Eclipse IMP — упростить создание среды разработки на основе Eclipse для произвольных языков.

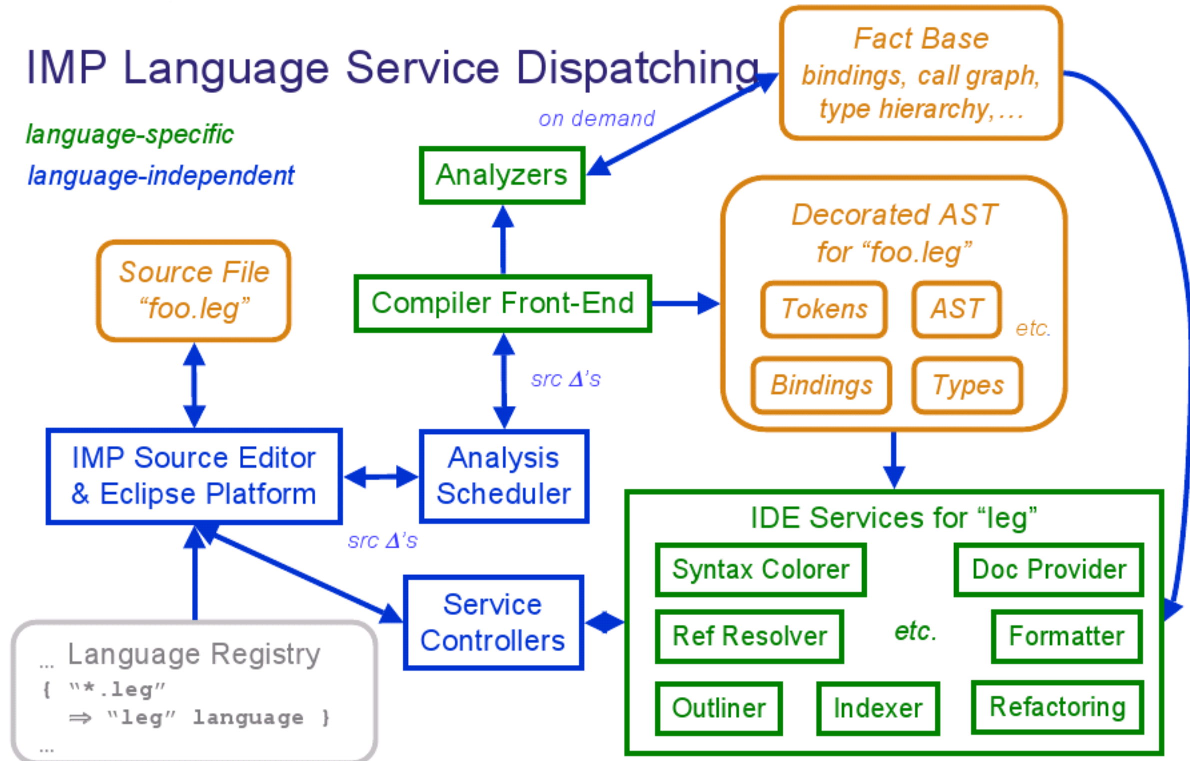
В отличие от Xtext, который ориентирован на небольшие языки предметной области, IMP ориентирован на полноценные языки программирования. На основе IMP были разработана среды для языков X10 и XJ, которые являются расширениями языка Java.

Фактически IMP — это просто набор инструментов для создания сервисов к IDE для разрабатываемого языка. В IMP интегрирован генератор парсеров LPG (ранее известный как JikesPG), но возможно использование и других генераторов или самописных парсеров. LPG может частично эмулировать множественное наследование путем включения нескольких грамматик в одну.

Если компилятор генерирует Java файлы, то IMP сможет даже их отлаживать благодаря стандарту JSR-45 (см. [4]). Для этого нужно в Java-файлы вставлять комментарии вида “//#line” для соответствия строк в исходном файле и сгенерированном по нему Java-файле.

IMP, LPG, X10, XJ — проекты компании IBM.

Схема взаимодействия частей IMP:



Проект IMP можно рассматривать как аналог Xtext, но адаптированный для больших языков программирования. IMP требует больше усилий для создания языков, но предлагает и больше возможностей. Кроме того IMP может сильно помочь и в реализации произвольных компиляторов на основе LPG или SDF.

4.4 Генераторы парсеров

При разработке расширения существующего языка, или создании нового, неизбежно придется столкнуться с генераторами парсеров. Ведь в процессе разработки компилятора разработчик обычно не пишет парсер языка сам, а пользуется генератором парсеров, который генерирует за него парсер на основе формального описания грамматики. Приведем примеры наиболее интересных генераторов парсеров.

Семейство YACC

распознаваемый класс языков: LALR(1)

целевые ЯП: yacc: C; bison: C, C++, и др.

особенности: стандартизован в POSIX P1003.2. Требует внешнего лексера: lex, flex.

Yacc — стандартный генератор парсеров в Unix-системах. Название является сокращением от «Yet Another Compiler Compiler» («всего лишь ещё один генератор компиляторов»). Yacc генерирует парсер на основе аналитической грамматики, описанной в нотации BNF. На выходе yacc выдаётся код парсера на языке программирования C.

Yacc был разработан Стивеном Джонсоном (Stephen C. Johnson) в компании AT&T для операционной системы Unix. Позже были написаны совместимые версии программы, такие как Berkeley Yacc, GNU Bison, MKS Yacc и Abraxas Yacc (обновлённый вариант AT&T-версии с открытым исходным кодом также вошёл в проект OpenSolaris от Sun). Каждый вариант предлагал незначительные улучшения и дополнительные возможности по сравнению с оригиналом, но концепция осталась той же. Yacc также был переписан на других языках, включая Ratfor, EFL, ML, Ada, Java, C# и Limbo.

Парсер, генерируемый с помощью Yacc, требует использования лексического анализатора. В качестве него в большинстве случаев используется Lex либо Flex. Стандарт IEEE POSIX P1003.2 определяет как функциональность, так и требования для Lex и Yacc.

Пример калькулятора на Yacc

```
..
line   : exp ';' '\n'       {printf ("result is %d\n", $1);}
        ;
exp    : term               {$$ = $1;}
        | exp '+' term      {$$ = $1 + $3;}
        | exp '-' term      {$$ = $1 - $3;}
        ;
term   : factor             {$$ = $1;}
        | term '*' factor   {$$ = $1 * $3;}
        | term '/' factor   {$$ = $1 / $3;}
        ;
factor : number             {$$ = $1;}
        | '(' exp ')'       {$$ = $2;}
        ;
number : digit              {$$ = $1;}
        | number digit      {$$ = $1*10 + $2;}
..
```

Как видно из примера, описание грамматики для Yacc состоит из правил вывода и некоторых действий, выполняемых при обнаружении данного вывода. Yacc генерирует C-файл с парсером, который выполняет указанные действия. Таким образом сгенерированный парсер НЕ строит дерева разбора. Его нужно строить самостоятельно в действиях описания грамматики. А это довольно долгое и скучное занятие.

Yacc приведен в обзоре как наиболее стандартный генератор парсеров. Его синтаксис описания грамматики и способ работы используют многие другие генераторы парсеров.

ANTLR

распознаваемый класс языков: LL(*) (шире чем LALR(1))

целевые ЯП: Java, C++, C#, Python, Ruby

особенности: наиболее мощный по возможностям. Требует специальную библиотеку для работы сгенерированного парсера (runtime dependency).

ANTLR — буквально Another Tool For Language Recognition (Ещё Одно Средство Распознавания Языков) — генератор парсеров, позволяющий автоматически создавать программу-парсер (как и лексический анализатор) на одном из целевых языков программирования (C++, Java, C#, Python, Ruby) по описанию LL(*)-грамматики на языке, близком к EBNF. Позволяет конструировать компиляторы, интерпретаторы, трансляторы с различных формальных языков. Предоставляет удобные средства для восстановления после ошибок и сообщения о них. ANTLR — продолжение PCCTS (Purdue Compiler Construction Tool Set), который был разработан в 1989 г. Является наиболее популярным генератором парсеров, по крайней мере на языке Java.

Основоположником проекта и его главным вдохновителем является профессор Теренс Парр (Terence Parr) из Университета Сан-Франциско. ANTLR — проект с открытым исходным кодом; версия 3.1 распространяется по лицензии BSD. Проект в настоящее время активно развивается.

Создатели ANTLR утверждают, что многие преимущества при определении действий для правил являются следствием того, что ANTLR осуществляет LL разбор, то есть использует разбор сверху вниз, в отличие от Yacc, Bison и других, которые используют разбор снизу вверх (LR). LL(*) разбор проще в понимании и диагностике ошибок чем LR(1). К тому же класс языков, распознаваемых LL(*) разбором, шире чем LR(1). Кроме того, ANTLR выгодно отличается от других наличием визуальной среды разработки ANTLRWorks, позволяющей

удобно создавать и отлаживать грамматики: это многооконный редактор, поддерживающий подсветку синтаксиса, автодополнение, визуальное отображение грамматик, строящееся в реальном времени по мере ввода, отладчик, инструменты для рефакторинга и т. д. ANTLR, также как и yacc, выполняет указанные действия для правил вывода.

простейшая программа на ANTLR

```
grammar T;
//нетерминальные символы:
msg : 'name' ID ';'
{
System.out.println("Hello, "+$ID.text+"!");
} ;
//терминальные символы:
ID: 'a'..'z' + ; //произвольное ( но >=1) количество букв
// пропускаемые символы:
WS: (' ' |'\n' |'\r' )+ {$channel=HIDDEN;} ;  пробел, перенос строки, табуляция
```

Программа распознает ввод типа “name <yourname>;” и выводит “Hello, <yourname>!”.

ANTLR: Tree Parser

ANTLR, как и Yacc, не строит дерево разбора автоматически. Но у него есть инструмент Tree Parser для его построения.

Если в описании грамматики не указывать действий, то на выходе парсера получится просто список токенов. TreeParser позволяет указать в описании грамматики правила для построения дерева из этих токенов, а также оперировать этим деревом. Например, заменить набор токенов узлом дерева, которое содержит эти токены в качестве детей.

Таким образом, дерево разбора в ANTLR — это просто дерево из токенов. Оно не типизировано (значение токена всегда текст), и не структурировано (нет информации о структуре узлов дерева: количестве детей, их типе).

ANTLR: Наследование грамматик

У ANTLR начиная с версии 3.1 есть уникальная в своем роде (за исключением LPG, в котором есть эмуляция наследования) возможность наследования грамматик. Эта возможность позволяет расширять грамматики, дописывая новые правила вывода и заменяя старые. В

результате получается производный от базового класс парсера. Причем для наследования не обязателен даже исходный код базового парсера. Наследование работает на бинарном уровне, добавляя и перекрывая методы в производном классе парсера.

JavaCC

распознаваемый класс языков: LL(*)

целевые ЯП: Java

особенности: расширения JJTree и JTB для автоматического построения AST.

JavaCC — второй по популярности генератор парсеров на языке Java. Также как и ANTLR он реализует LL(*) разбор, выполняет действия, описанные в грамматике, но возможностей у него меньше. JavaCC выделяется наличием двух расширений: JJTree и JTB. Эти расширения генерируют классы AST по описанию языка, что значительно облегчает создание парсера.

JJTree транслирует грамматику без действий в грамматику с действиями построения AST. Также генерирует классы AST и шаблон посетитель (Visitor, TreeWalker) для удобного обхода дерева. Генерируемые классы AST не типизируемы (отсутствует информация о типе вложенных узлов)

JTB действует также как и JJTree, но генерирует гораздо более удобные типизированные классы AST. Но и их качество все равно хуже, чем у классов AST от SableCC, про который написано в следующем разделе.

SableCC

распознаваемый класс языков: LALR(1)

целевые ЯП: Java. Через расширения: C++, C#, Python, O'Caml

особенности: генерирует строго типизированное AST и расширенный шаблон “посетитель”.

SableCC — генератор парсеров и лексеров, который по описанию грамматики языка генерирует удобную объектно-ориентированную библиотеку для распознавания и анализа языка. В частности, генерируемая библиотека включает строго типизированное синтаксическое дерево и классы для его обхода (tree walkers). SableCC также сохраняет четкую границу между генерируемым кодом, и кодом, написанным программистом, что ведет к уменьшению длительности цикла разработки. Из недостатков можно отметить слабую диагностику ошибок, присущую большинству LR парсеров.

SableCC был выбран автором для создания специального языка XWiki Query Language (XWQL) [9] для проекта XWiki.org. Проект XWiki совмещает wiki-систему и своего рода среду разработки для пользователей. XWiki позволяет пользователям писать скрипты на некоторых языках, используя модель данных XWiki, которая включает в себя документы, классы, объекты. Для запросов к базе данных используется Hibernate Query Language (HQL) — SQL-подобный язык запросов, в котором достаточно трудно выразима модель данных XWiki. В качестве примера приведем запрос поиска всех статей (документов с объектом класса XWiki.ArticleClass) определенной категории:

```
select doc from XWikiDocument as doc, BaseObject as obj,
 DBStringListProperty as prop join prop.list list
where obj.name=doc.fullName and obj.className='XWiki.ArticleClass'
 and obj.name<>'XWiki.ArticleClassTemplate'  and obj.id=prop.id.id
 and prop.id.name='category' and list='${category}'
order by doc.creationDate desc
```

Этот же запрос на XWQL:

```
where doc.fullName <> 'XWiki.ArticleClassTemplate'
and :category member of doc.object(XWiki.ArticleClass).category
```

Удобный и простой язык запросов очень важен для XWiki, т.к. позволяет простым пользователям писать достаточно сложные запросы в своих скриптах, не имея знаний о внутренней структуре системы хранения XWiki. Также язык важен для планируемого перехода XWiki на другую систему хранения, т.к. позволяет не менять язык запросов, а просто написать транслятор в язык запросов новой системы хранения. При проектировании языка это требование также было учтено.

XWQL является расширением языка JPQL [3] с добавлением новых синтаксических конструкций, выражающих модель данных XWiki. Язык транслируется в HQL и выполняется обычным образом в системе хранения XWiki. Также можно отметить, что почти все существующие запросы на языке HQL (конкретнее это те, которые удовлетворяют грамматике JPQL, т.к. HQL это расширенный JPQL) выполняются без изменений и на XWQL.

В процессе поиска решения были исследованы парсеры проектов Hibernate (язык HQL), EclipseLink (JPQL), JPOX (JPQL), OpenJPA(JPQL) и другие. Используемый в XWiki проект Hibernate не позволял расширить или заменить грамматику HQL. Все рассмотренные парсеры были построены на генераторах парсеров ANTLR или JavaCC и содержали запутанную

грамматику вместе с действиями построения синтаксического дерева. Исследуемые парсеры было почти невозможно повторно использовать, т.к. в них находились очень специфичные для проекта части, которые очень трудно убрать из описания грамматики без нарушения целостности. Поэтому было решено написать парсер XWQL с нуля.

SableCC хорошо подошел для реализации XWQL. Он сгенерировал множество полезных инструментов, которые в других генераторах парсеров пришлось бы писать вручную. Побочным результатом стала грамматика JPQL, записанная в терминах SableCC, которую можно повторно использовать в других проектах, поскольку она не содержит никаких специфичных для XWQL частей (благодаря принципу отделения генерируемого и ручного кода), в отличие от многих других грамматик на ANTLR и JavaCC.

LALR parser generator (LPG)

распознаваемый класс языков: LALR(k)

целевые ЯП: Java, C, C++

особенности: генерирует строго типизированное AST и расширенный шаблон “посетитель”; наследование грамматик; возможность поиска с возвратом (backtracking)

LPG ранее известен как JikesPG. Используется в экспериментальном высокопроизводительном компиляторе Jikes, некоторых проектах Eclipse и расширениях языка Java — X10 и XJ. Имеет интеграцию с IDE Eclipse в проекте Eclipse IMP, описанном в разделе 4.3. По возможностям превосходит SableCC, но менее известен, и по нему имеется очень мало документации. Допускает действия в описании грамматики. LPG может эмулировать множественное наследование путем простого включения описаний наследуемых грамматик в целевую грамматику. В отличие от большинства парсеров, распознающих класс языков LR, LPG может похвастаться хорошей диагностикой ошибок (см [17]).

Комбинаторы парсеров (Parser combinators)

В функциональном программировании популярным подходом к построению рекурсивных синтаксических анализаторов является моделирование парсеров как функций и определение функций высшего порядка (комбинаторов), которые снабжены грамматическими конструкциями, такими как упорядочение, выбор и повторение. Такие парсеры образуют примеры монад, алгебраических структур из математики, которые доказали полезность при исследовании большого числа вычислительных задач.

Во многих функциональных языках есть специальные библиотеки для построения такого рода парсеров. Описание грамматики в этом случае представляет собой программу на языке программирования, состоящую из определений правил вывода с помощью комбинирования специальных функций библиотеки. Таким образом комбинаторы парсеров отличаются от генераторов парсеров отсутствием внешнего языка для определения грамматики. В большинстве библиотек комбинаторов парсеров используется различные варианты рекурсивного нисходящего парсера (см.[13]), которые не требуют предварительной генерации большого количества вспомогательных данных. Поэтому комбинаторы парсеров не требуют отдельной генерации парсера, в отличие от генераторов парсеров, рассмотренных выше.

Приведем пример распознавания простого DSL используя parser combinators. DSL выглядит примерно так (см.[20]):

```
(buy 100 IBM shares at max USD 45,
 sell 50 CISCO shares at min USD 25,
 buy 100 Google shares at max USD 800
 ) for trading_account "SSS1234")
```

Описание DSL на языке Scala с помощью parser combinators:

```
import scala.util.parsing.combinator.syntactical._
object OrderDSL extends StandardTokenParsers {
  lexical.delimiters ++= List("(", ")", ",")
  lexical.reserved += ("buy", "sell", "shares", "at",
    "max", "min", "for", "trading", "account")
  def instr: Parser[ClientOrder] =
    trans ~ account_spec ^^ { case t ~ a => new ClientOrder(scala2JavaList(t), a) }
  def trans: Parser[List[LineItem]] =
    "(" ~> repsep(trans_spec, ",") <~ ")" ^^ { (ts: List[LineItem]) => ts }
  def trans_spec: Parser[LineItem] =
    buy_sell ~ buy_sell_instr ^^ { case bs ~ bsi
      => new LineItem(bsi._1._2, bsi._1._1, bs, bsi._2) }
  def account_spec =
    "for" ~> "trading" ~> "account" ~> stringLit ^^ {case s => s}
  def buy_sell: Parser[ClientOrder.BuySell] =
    ("buy" | "sell") ^^ { case "buy" => ClientOrder.BuySell.BUY
```

```
                          case "sell" => ClientOrder.BuySell.SELL }
  def buy_sell_instr: Parser[((Int, String), Int)] =
    security_spec ~ price_spec ^^ { case s ~ p => (s, p) }
  def security_spec: Parser[(Int, String)] =
    numericLit ~ ident ~ "shares" ^^ { case n ~ a ~ "shares" => (n.toInt, a) }
  def price_spec: Parser[Int] =
    "at" ~ ("min" | "max") ~ numericLit ^^ { case "at" ~ s ~ n => n.toInt }
}
```

Как видно, в классе просто описаны грамматика DSL в форме, похожей на стандартную EBNF, и действия, выполняемые при достижении соответствующих правил вывода. Для описания грамматики используются операции следования (~) и альтернативы (|). Для описания действий — оператор ^^ и собственно функция действия. Все операторы являются функциями из библиотеки комбинаторов парсеров (scala.util.parsing.combinator). В описанных действиях строится синтаксическое дерево для распознаваемого языка.

4.5 Адаптивные грамматики

Адаптивной грамматикой называют формальную грамматику, которая предоставляет механизмы для добавления, удаления и изменения своих правил вывода.

Адаптивные грамматики делятся на императивные, декларативные и гибридные.

- Императивные (или глобальные) адаптивные грамматики модифицируют правила вывода на основании некоего глобального состояния, зависящего от времени (номера шага, time) в распознавании.

- Декларативные (или локальные) адаптивные грамматики модифицируют правила вывода на основании текущей позиции в грамматике (пространстве, space).

- Гибридные (пространственно-временные) адаптивные грамматики объединяют возможности императивных и декларативных. Они могут менять правила вывода как в соответствии со временем, так и в соответствии с пространством.

Обзор адаптивных грамматик можно найти в [1].

Теория адаптивных грамматик начала активно развиваться в 90-ых годах. Адаптивные грамматики способны распознать контекстно-зависимые языки. Существуют очень мало парсеров на основе адаптивных грамматик, например PAISLEI (см. [21]). Также существует про-

блема алгоритмической сложности таких парсеров, т.к. в общем виде их алгоритмическая сложность нелинейна. Но существуют и такие адаптивные грамматики, которые распознаются линейно (см. [21]).

Адаптивные грамматики ценны для расширений языков программирования, т.к. позволяют распознавать больший класс языков, учитывая контекстно-зависимую информацию. С их помощью становятся возможными довольно сложные синтаксические расширения.

4.6 Создание расширения языка

На основании изученных данных приведем небольшую схему действий для единичного расширения какого-либо языка.

- Если язык является расширяемым (см. раздел 3.4), то просто используем его выразительные возможности для создания нашего расширения. Если выразительных способностей не хватает, или требуется изменить неизменяемый базовый синтаксис, то переходим к следующему пункту.

- Если компилятор языка расширяем (см. раздел 3.3), пытаемся расширить нужные нам части в компиляторе. Например, для добавление новых синтаксических конструкций расширяем грамматическую часть компилятора. Если компилятор такого не позволяет, то переходим к следующему пункту.

- Если компилятор языка нерасширяем, можно попытаться расширить его и без модульной системы, используемой в предыдущем пункте. Это может требовать глубоких знаний компилятора. Для расширения синтаксиса, необходимо исправить описание грамматики языка для используемого генератора парсеров. Для добавления семантики нужно найти и использовать точки входа в семантический анализатор. Таким способом создаются внутренние расширения языков. В качестве примера можно привести статью о том, как изменить язык Java в компиляторе OpenJDK[18].

- Последний способ расширения. Используется когда все остальные невозможны или слишком сложны. Создаем внешний транслятор расширенного языка в базовый. Базовые синтаксические конструкции оставляем без изменений, а новые расшифровываем, используя базовые. После трансляции передаем результат базовому компилятору. Благодаря акцентированию внимания только на новых синтаксических конструкциях, транслятор, как правило, получается достаточно небольшим. В качестве генератора

парсеров имеет смысл использовать те, которые автоматически генерируют AST и инструменты для его обхода (например, SableCC, LPG), т.к. это значительно сокращает время разработки. Примером применения такого способа расширения являются проекты JSE, Polyglot и XWiki Query Language.

4.7 Создание расширяемого компилятора

Для создания расширяемого компилятора прежде всего важна модульная структура, удобная для написания расширений. Внутренние части компилятора должны быть четко описаны и документированы. Для расширения синтаксиса имеет смысл применять генераторы парсеров, поддерживающие наследование (например, LPG, ANTLR). На этапе синтаксического анализа компилятор собирает все синтаксические расширения и строит итоговую грамматику. Для расширения семантики удобно пользоваться атрибутными грамматиками (например, с использованием JastAdd).

4.8 Дизайн IDE для расширяемого компилятора

На основе проекта Eclipse IMP, описанного в разделе 4.3, можно спроектировать дизайн среды программирования для модульных компиляторов. Отдельный модуль компилятора будет представлен плагином к Eclipse, в котором содержится грамматика расширения языка. Другие модули могут ей пользоваться при наследовании. У модуля также могут быть сервисы для разных частей IDE (подсветка синтаксиса, outline, автодополнения и пр). Пользователь в среде программирования выбирает модули компилятора. После этого система генерирует итоговый компилятор и сервисы среды разработки к нему.

4.9 Создание расширяемого языка

При создании расширяемых языков основной проблемой является обеспечение возможности изменения синтаксиса прямо во время работы компилятора. Существующие расширяемые языки существенно ограничивают изменение синтаксиса. Наиболее популярными подходами является использование некоего гибридного парсера, который при ошибке распознавания базового синтаксиса пытается подобрать подходящее под неё расширение, либо допускающего только определенные расширения в определенных местах базовой грамматики. Например второй подход используется в языках Nemerle и Boo, где использование синтаксических макросов в некоторых местах программы недопустимо.

5 Заключение

В данной работе произведен обзор расширений языков программирования, расширяемых компиляторов, языков и сред программирования. Произведена классификация и структуризация полученных знаний по общим признакам. Также исследованы различные средства для создания расширений. Полученные навыки автор применил для создания XWiki Query Language[9] — расширения языка запросов JPQL, которое успешно используется в проекте XWiki.

Список литературы

[1] Adaptive grammars. [Электрон. ресурс]. Режим доступа: http://en.wikipedia.org/wiki/Adaptive_grammar.

[2] CUP: LALR parser generator for Java. [Электрон. ресурс]. Режим доступа: http://www2.cs.tum.edu/projects/cup/.

[3] JSR-317: Java Persistance API 2.0. [Электрон. ресурс]. Режим доступа: http://jcp.org/en/jsr/detail?id=317.

[4] JSR-45: Debugging support for other languages. [Электрон. ресурс]. Режим доступа: http://jcp.org/en/jsr/detail?id=45.

[5] Non-null types for java. [Электрон. ресурс]. Режим доступа: http://jastadd.org/jastadd-tutorial-examples/non-null-types-for-java.

[6] Programming community index. [Электрон. ресурс]. Режим доступа: http://www.tiobe.com/index.php/content/paperinfo/tpci/.

[7] Syntax Defenition Formalism. [Электрон. ресурс]. Режим доступа: http://ru.wikipedia.org/wiki/Syntax_Definition_Formalism.

[8] Termware. [Электрон. ресурс]. Режим доступа: http://www.gradsoft.ua/products/termware_rus.html.

[9] XWiki Query Language. [Электрон. ресурс]. Режим доступа, документация: http://dev.xwiki.org/xwiki/bin/Design/XWikiQueryLanguage исходный код: http://svn.xwiki.org/svnroot/xwiki/platform/core/trunk/xwiki-query/.

[10] Диаграммы Ишикава (Ishikawa), рыбьей кости (fishbone), причинно-следственные (cause-and-effect). [Электрон. ресурс]. Режим доступа: http://en.wikipedia.org/wiki/Ishikawa_diagram.

[11] Расширение SQLj. [Электрон. ресурс]. Режим доступа: http://ru.wikipedia.org/wiki/SQLJ.

[12] Расширяемое программирование. [Электрон. ресурс]. Режим доступа: http://en.wikipedia.org/wiki/Extensible_programming.

[13] Рекурсивный нисходящий парсер. [Электрон. ресурс]. Режим доступа: http://ru.wikipedia.org/wiki/Рекурсивный_нисходящий_парсер.

[14] Язык программирования X10. [Электрон. ресурс]. Режим доступа: http://x10-lang.org/.

[15] *Christensen, C., and C. J. Shaw (Eds.), Proceedings of the Extensible Languages Symposium*, August 1969.

[16] *Schuman, S.A., ed., Proceedings of the International Symposium on Extensible Languages*, December 1971.

[17] Philippe Charles. *A practical method for constructing efficient LALR(K) parsers with automatic error recovery*. PhD thesis, New York, NY, USA, 1991.

[18] Joseph D. Darcy. So you want to change the Java Programming Language... [Электрон. ресурс]. Режим доступа: http://blogs.sun.com/darcy/entry/so_you_want_to_change.

[19] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.

[20] Debasish Ghosh. External DSLs made easy with Scala Parser Combinators. [Электрон. ресурс]. Режим доступа: http://debasishg.blogspot.com/2008/04/external-dsls-made-easy-with-scala.html.

[21] Quinn Tyler Jackson. An introduction to PAISLEI.

[22] Harrison M.C. Panel on the concept of extensibility. 1960.

[23] M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Commun. ACM*, 3(4):214–220, 1960.

[24] Jean E. Sammet. *Programming Languages: History and Fundamentals*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1969.

[25] Thomas A. Standish. Extensibility in programming language design. *SIGPLAN Not.*, 10(7):18–21, July 1975.

[26] Robert G. Trout. A compiler—compiler system. In *Proceedings of the 1967 22nd national conference*, pages 317–322, New York, NY, USA, 1967. ACM.

[27] Gregory V. Wilson. Extensible programming for the 21st century. *ACM Queue*, 2(9):48–57, 2004.