Методы и инструменты реализации предметно-ориентированных языков программирования

А. С. Казакова
anastasia.kazakova@oktetlabs.ru

В наше время программные продукты востребованы все более разнообразными предметными областями, и для того чтобы оптимально реализовывать возникающие задачи, необходимы новые языки программирования. Рост производительности вычислительных средств снизил трудоемкость разработки компиляторов и других средств программирования. Сейчас уже встает следующий важный вопрос — как автоматизировать разработку новых языков и соответствующих программных средств (компиляторов, отладчиков и проч.). Цель данной статьи — обзор методов и инструментов, предназначенных для разработки предметно-ориентированных языков программирования (Domain Specific Languages, DSL). Рассматриваются компиляторные подходы, основанные на атрибутных грамматиках и системах переписывания, и методы, основанные на расширении базовых языков — макрорасширения, синтаксические расширения.

Введение

Термин *предметно-ориентированный язык* (Domain Specific Language, DSL) обозначает язык программирования или моделирования, применяющийся для решения конкретного круга задач в терминах, максимально приближенных к данной предметной области [17, 22]. Вот некоторые примеры предметно-ориентирован-

ных языков. Язык SQL, использующийся для работы с базами данных и являющийся одним из самых, наверное, известных и успешных предметно-ориентированных языков. Макроязык пакета Excel, предназначенный для табличных вычислений. Make — язык и утилита для автоматизации сборки ПО главным образом для Unix платформ. За последние пять лет наблюдается всплеск появления новых языков программирования — Ruby, Groovy, Scala, Lua, Nemerle и др. Вместе с возросшими потребностями разработчиков еще одной причиной этого всплеска является значительный прогресс производительности вычислительных средств, что позволяет легко реализовывать компиляторы и сре́ды разработки, просто следуя классическим компиляторным методам, описанным, например, в известной монографии [2], тогда как до недавнего времени такие реализации «в лоб» работали чрезвычайно медленно и использовать на практике их было невозможно. В связи с этим актуальной является задача нового обзора имеющихся в этой области средств с тем, чтобы сделать впоследствии дальнейшие шаги по автоматизации разработки DSL.

Все подходы к разработке DSL можно разделить на две категории: 1) компиляторные подходы, позволяющие реализовывать предметно-ориентированный язык как отдельный и самостоятельный; 2) подходы, в которых новый язык получается расширением базового. В данной статье рассматриваются подходы к реализации предметно-ориентированных языков в соответствии с вышеприведенной классификацией. В начале статьи будут приведены примеры использования DSL-подхода для того, чтобы пояснить, сколь широко он применяется на практике. Технологии разработки компиляторов будут разбиты на две группы — те, которые основаны на атрибутных грамматиках [19, 15], и те, которые базируются на принципе переписывания [4, 14]. Различные атрибутные грамматики и системы переписывания лежат в основе всех традиционных методов разработки компиляторов. В рамках компиляторного подхода будут рассмотрены следующие технологии: основанные на атрибутных грамматиках — Elegant [11], FNC-2 [12], CDL [16]; основанные на системах переписывания — Term Processor Kimwitu [27], TXL [7], ASF+SDF [23, 24]. В рамках подходов, в которых новый язык получается расширением базового, будут рассмотрены метод макрорасширений (на примере реализации системы для тестирования сетевых протоколов) и метод синтаксических расширений (на примере технологии Camlp4 [5, 10]).

1. Предметно-ориентированные языки

При выборе средств для решения какой-либо задачи есть два крайних варианта — либо использовать общий подход и универсальный инструмент, либо применить специализированные средства и инструменты. Общие подходы и универсальные инструменты рассчитаны на решение многих задач, часто — классов задач и потому, как правило, они не являются оптимальными для каждой конкретной задачи, особенно в случае, когда она нестандартна и имеет много индивидуальных особенностей. Специализированные средства и инструменты ориентированы на конкретную узкую область и предоставляют средства для решения задач только этой области. Они оптимальны и эффективны для решения именно этих задач.

Рассматривая языки программирования, можно сказать, что универсальным инструментарием здесь являются универсальные языки, а специализированным — предметно-ориентированные. Нельзя утверждать, что между этими двумя подходами существует четкая граница. Например, такие языки, как COBOL, Fortran, LISP, в свое время появились для решения задач в конкретных узких областях — соответственно, для разработки бизнес-приложений, для реализации математических вычислений и для символьной обработки данных. Но их не принято называть предметно-ориентированными языками, так как к определению DSL, следуя [25], нужно добавить еще компактность самого языка и ограниченность выразительных средств рамками одной конкретной предметной области. И тогда вышеперечисленные языки однозначно попадут в категорию языков общего назначения.

Сама идея языков предметной области не нова. Можно вспомнить «малые» языки и утилиты UNIX: Make, Yacc, Lex, консольные утилиты (awk, sed, dc, bc), разнообразные языки для облегчения работы с данными специального формата (eqn — для математических выражений, tbl — для таблиц, pic — для построения диаграмм) [20]. XML фактически также использует идею DSL: XML-схема задает синтаксис нового языка, а правила преобразования текстов на этом языке определяются в XSLT-спецификациях. Нельзя забывать и про обширную область, связанную с визуальными предметно-ориентированными языками: имеющиеся на данный момент технологии в этой области (Eclipse/GMF, MS DSL TOOLS и др.) позволяют легко создавать спецификацию нового языка и по ней в

полуавтоматическом режиме создавать графический редактор, генераторы кода и проч. [1].

Итак, видно, что предметно-ориентированный подход довольно популярен в области языков разработки ПО. И сейчас потребность в предметно-ориентированных языках программирования постоянно растет, а значит, растет потребность в средствах их реализации.

2. Критерии оценки подходов для разработки предметно-ориентированных языков

Рассматривая подходы к реализации предметно-ориентированных языков программирования, мы будем оценивать их, исходя из следующих критериев.

1. *Семантическая замкнутость.* Подход не соответствует данному критерию, если с его помощью реализуются DSL, в которых присутствует большое количество конструкций, не имеющих семантической трактовки в терминах, соответствующих данной предметной области.
2. *Трудоемкость реализации.* Подход позволяет реализовывать DSL без чрезмерных затрат. В противном случае идея создания и реализации нового DSL может оказаться неоправданной по сравнению с использованием уже готовых универсальных языков.
3. *Трудоемкость внесения изменений в спецификацию и реализацию DSL.* Подход не отвечает данному критерию, если небольшое изменение в предметной области влечет трудоемкую переработку спецификаций DSL и/или созданных с помощью данного подхода средств поддержки DSL. Критерий наиболее актуален для тех DSL, предметные области которых активно развиваются и видоизменяются с течением времени.
4. *Оптимальность реализации.* Очень важно, на сколько подход позволяет создавать оптимальные реализации DSL, так как часто это является одним из основных факторов при принятии решения о создании и реализации нового DSL.
5. *Поддержка оптимизации на уровне предметной области.* Эти оптимизации позволяют зачастую существенно увеличить эффективность кода, так как используют знания об особенностях предметной области.
6. *Сохранение «привязки» к тексту* особенно актуально при разработке DSL посредством расширения базового языка, по-

скольку позволяет реализовать отладку, выдачу сообщениий об ошибках компиляции и проч. в терминах DSL, а не в малоинформативных терминах базового языка.

3. Компиляторные подходы

Существует большое разнообразие методов разработки компиляторов, созданных за более чем полувековую историю этой науки. Эти методы могут быть применены к реализации DSL, хотя в работе [22] доказывается, что процесс реализации DSL принципиально отличается от процесса реализации универсальных языков программирования из-за разных размеров языков, а также из-за стоимости разработки. Как правило, компиляторные подходы используются для реализации тех предметно-ориентированных языков, которые сильно отличаются от существующих. Например, в случае, когда надо лишь добавить несколько конструкций в какой-то уже имеющийся язык программирования, обходятся более простыми средствами.

Две основные парадигмы, использующиеся при разработке компиляторов, — это атрибутные грамматики (attribute grammars) и переписывание (tree/term rewriting). Рассмотрим наиболее зрелые технологии для разработки компиляторов, реализующие эти парадигмы.

3.1. Атрибутные грамматики

Атрибутные грамматики были введены Дональдом Кнутом [15, 19] в конце 60-х как продолжение идеи вычисления семантики целого выражения через семантику его подвыражений. В результате развился подход к построению разнообразных автоматических средств обработки языков на основе их формальных спецификаций с последующей автоматической генерацией компиляторов, отладчиков и т. д.

Атрибутная грамматика — это контекстно-свободная грамматика с атрибутами, которые определяют семантику конструкций языка. Семантика программы определяется набором значений атрибутов корневого узла.

Атрибуты бывают наследуемыми и синтезируемыми. *Наследуемые атрибуты* используются для передачи семантической информации вниз по дереву разбора программы, а *синтезируемые атрибуты* — вверх. Примеры синтезируемых атрибутов — длина списка

и сумма его элементов, а пример наследуемого атрибута — среднее значение всех элементов списка. С теоретической точки зрения наследуемые атрибуты излишни [15], но на практике они применяются довольно часто, особенно при использовании значения атрибута за пределами контекста, в котором он был создан. Наибольшее распространение наследуемые атрибуты получили при реализации многопроходных компиляторов.

Приведем другие примеры атрибутов для описания семантики языка программирования: code (синтезируемый код целевой платформы в виде инструкций в форме «код инструкции», «аргумент»), name (синтезируемое имя константы), value (синтезируемое значение константы или число), envs (синтезируемое окружение, т. е. таблица символов с полями «имя», «значение»), envi (наследуемое окружение) и др. Очевидно, что с помощью подобных атрибутов возможно задать компилятор языка программирования.

Кнут сформулировал следующее условие, гарантирующее разрешимость атрибутной грамматики. Если построить граф зависимостей атрибутов (атрибут «a» в нем имеет дугу в вершину, помеченную атрибутом «b» тогда и только тогда, когда значение атрибута «b» зависит от значения атрибута «a»), то он должен быть ацикличным, т. е. на нем можно определить частичный порядок вычислений. Это условие является достаточно строгим и трудно выполнимым на практике. Поэтому существует много других, более слабых условий, основанных на том факте, что на практике важно не столько выполнить все возможные вычисления, сколько вычислить необходимый набор атрибутов в корне дерева. Существуют различные методы, упрощающие описание вычислений в атрибутных грамматиках (см., например: [19]).

За долгую историю своего развития атрибутные грамматики были реализованы множеством программных средств. Помимо синтезируемых и наследуемых атрибутов, введенных Дональдом Кнутом, появились также атрибуты-коллекции (collections), зависящие от всего дерева разбора, рекурсивные атрибуты (circular attributes) и атрибуты-ссылки (referenced attributes). Различные системы, основанные на атрибутных грамматиках, позволяют использовать самые разные языки реализации, такие как C (Elegant, FNC-2), LISP (FNC-2), ML (FNC-2), Java (Jast-Add), Scala (Kiama).

Прежде чем перейти к описанию инструментальных средств, поддерживающих разработку компиляторов на основе атрибутных грамматик, отметим ряд общих недостатков этого подхода.

1. Компиляторы, автоматически созданные с помощью атрибутных грамматик, зачастую оказываются неэффективными по времени исполнения и по использованию памяти.
2. В атрибутных грамматиках зачастую отсутствуют удобные модульные средства (на самом деле исключения есть, но о них чуть позже). Таким образом, спецификации языков программирования, созданные с помощью атрибутных грамматик, часто оказываются чрезвычайно громоздкими, в частности в них трудно вносить изменения. Этот последний факт является серьезным препятствием при разработке DSL, так как спецификации таких языков склонны часто меняться.
3. Вычисления в атрибутных грамматиках основываются на неизменности входного дерева. Это свойство запрещает какие бы то ни было оптимизации вычислений, требующие преобразования исходного дерева.
4. Вычисления в атрибутных грамматиках являются достаточно строгими — например, нельзя вычислить значение атрибута, если не вычислены все значения атрибутов, от которых он зависит. Чтобы не загромождать такими зависимостями семантику, приходится создавать искусственные конструкции.

3.1.1. Система Elegant

Идея генератора компиляторов, построенного на основе атрибутных грамматик, появилась в исследовательской лаборатории компании Philips в 1987 году. При этом предполагалось совместить в одном подходе атрибутные грамматики и «ленивые» вычисления (lazy evaluations). Первая версия системы Elegant вышла в 1992 году как внутренняя разработка компании Philips. К 1997 году работы по Elegant подошли к завершению и компания выпустила этот продукт как свободно распространяемый. Последний выпуск продукта доступен для большинства популярных платформ: SunOS4/5, Solaris, HP700, Silicon Graphics, Linux, DOS/Windows, IRIX. Подробное описание Elegant можно найти в работе [11].

В системе Elegant существует собственный язык для спецификации компиляторов. Он является императивным и включает в себя объектно-ориентированные и функциональные черты. В языке Elegant есть полиморфизм, функции высших порядков, частично применимые функции, сопоставление по шаблону (pattern matching). Каждый тип в Elegant представляет собой множество (даже

примитивные типы, такие как Int). В языке поддерживается также концепция линейного наследования типов (т. е. запрещается наследоваться от двух и более типов), есть обобщенные функции (generics) и возможность «перегрузки» функций. Спецификации, созданные с помощью этого языка, система Elegant компилирует в ANSI C.

Для описания правил атрибутной грамматики в Elegant используются частичные функции, которые позволяют задавать набор разных правил в зависимости от различных входных значений содержащей их функции. Рассмотрим пример:

```
VarOrCall(in s: Scope, id: Ident out varexpr)
  {Var (in s, id out varexpr : VarExpr)}
VarOrCall(in s: Scope, id: Ident out funcexpr)
  {Call(in s, id out funcexpr : FuncExpr)}
```

Функция `VarOrCall` будет задавать два разных правила в зависимости от того, является ли ее третий параметр переменной (`varexpr`) или вызовом функции (`funcexpr`). Ключевые слова `in` и `out` позволяют помечать наследуемые и синтезируемые атрибуты. В нашем примере атрибуты `in` и `id` являются наследуемыми, а атрибуты `varexpr` и `funcexpr` — синтезируемыми.

Для удобства описания правил в Elegant присутствует конструкция `conditions`, позволяющая при описании функции задавать ограничения на ее входные значения. Таким образом, данное правило применяться только при выполнении описанного условия. Приведем пример:

```
ConstructTerm (level: Int, propc: Bool)
  conditions level = Operators.maxlevel
  {...}
```

В данном случае правило, описываемое функцией `ConstructTerm`, будет выполняться только в том случае, когда значение передаваемого параметра `level` совпадает со значением `Operators.maxlevel`.

Хотя Elegant и представляет собой инструмент для поддержки полного цикла разработки компилятора, но он никак не устраняет указанных выше недостатков атрибутных грамматик.

3.1.2. Система FNC-2

Данная система создана во французском исследовательском институте INRIA. Разработка началась в 1986 году, первый прототип

вышел в 1989 году. После этого система FNC-2 использовалась для разработки нескольких крупных приложений — компилятора для параллельного логического языка Parlog в код абстрактной машины SPM, компилятора с языка ISO-Pascal в P-code, транслятора языка E-LOTOS [12]. Система FNC-2 также использовалась для собственной компиляции в процессе разработки.

При создании системы FNC-2 особенно исследовались возможности повысить эффективность вычислений в атрибутных грамматиках, поскольку главная работа всей системы происходит именно при вычислении атрибутов. Здесь важно задать эффективный порядок вычислений, в связи с чем авторами были предложены три типа вычислителей:

- исчерпывающий (exhaustive);
- инкрементальный (incremental);
- параллельный (parallel).

Исчерпывающий вычислитель является простейшим. Принцип его работы основан на том, что вычислитель минимальное время тратит на выстраивание порядка вычислений, используя порядок, заданный извне. Этот вычислитель может начать свою работу в любом месте дерева атрибутной грамматики. Следующий тип вычислителя, *инкрементальный*, позволяет выполнять семантические функции контроля, назначение которых — определять ситуации, когда один и тот же атрибут вычисляется несколько раз, и избавляться от излишних вычислений. И, наконец, *параллельный вычислитель* предназначается для ускорения процесса вычисления атрибутов за счет параллельной работы. Таким образом, пользователь FNC-2 может построить наиболее эффективное вычисление атрибутов в своей системе, комбинируя имеющиеся типы вычислителей.

Помимо вычислителей FNC-2 содержит в себе и другие функциональные блоки для поддержки процесса реализации компилятора — генераторы лексических/синтаксических анализаторов, генераторы абстрактного атрибутного дерева, средства для модульного описания всей системы и взаимосвязей таких модулей.

Модульность является одним из основных преимуществ системы FNC-2. Авторы системы попытались избавиться от главной проблемы атрибутных грамматик — большого громоздкого дерева, атрибуты в котором надо вычислять сразу везде. FNC-2 позволяет разбить описываемую атрибутную грамматику на модули, для которых вычисления атрибутов могут быть произведены независимо

друг от друга. Таким образом становится возможным производить оптимизации в промежутках между работой разных вычислителей, в разных частях системы, а также модифицировать атрибутное дерево в эти моменты времени.

Для задания атрибутных грамматик в системе FNC-2 используется язык OLGA [12]. Этот язык поддерживает типы-деревья, модульную структуру и является строго типизированным. В OLGA есть полиморфизм для множеств и списков, а также «перегрузка» функций. В языке присутствуют конструкции для задания исключений и сопоставлений по шаблону. Нововведением являются атрибутные классы, позволяющие задавать правила для целой группы атрибутов одновременно, что значительно сокращает размер спецификаций.

В FNC-2 существует несколько генераторов кода — в C, LISP, ML, C/fSDL. Наиболее зрелыми являются генераторы в языки C и ML. Возможность выбора из двух языков, отличающихся используемой парадигмой программирования (императивная, в случае C, и функциональная, в случае ML), — несомненное достоинство системы. Но с ростом предметных областей растет потребность в предметно-ориентированных языках, отличающихся по своим свойствам от большинства имеющихся языков программирования, в частности от C и ML. Поэтому использование только этих двух парадигм не решает проблемы создания разнообразных по своим свойствам DSL.

К сожалению, никаких новых результатов работы над системой уже давно не публикуется.

3.1.3. Система CDL

Если в описании грамматики задействован нетерминал, все вхождения которого преобразуются одинаково, то такой нетерминал называется *аффиксом* (affix), а соответствующая грамматика — *аффиксной*. Compiler Description Language (CDL) [16] является программной системой и языком для разработки компиляторов на основе аффиксных грамматик.

Эта система была впервые представлена в 1971 году Корнелиусом Костером и основывалась на довольно необычной концепции. А именно в CDL отсутствовали базовые примитивы для простейших операций (например, арифметических операций), и эти операции приходится реализовывать самостоятельно. И хотя та-

кой подход предоставляет определенную гибкость, реализация базовых примитивов может потребовать довольно больших усилий, что слишком существенно при разработке DSL.

Версия системы CDL2, появившаяся в 1976 году, несильно отличалась от предыдущей. Единственным важным шагом стало добавление средств создания модулей, что дало возможность использовать CDL2 в больших проектах (например, проект MProlog [18] — индустриальная реализация языка Prolog для широкого набора архитектур). Версия системы CDL3 появилась относительно недавно, в 2004 году, и используется в основном только в академических целях, так как практически не имеет документации. К тому же она выдает неинформативные сообщения об ошибках, что очень сильно затрудняет отладку кода. Однако, в отличие от предыдущих версий, CDL3 имеет реализацию базовых примитивов и систему типов.

Для создания DSL средствами CDL требуется описать грамматику и правила вычисления аффиксов. Правила записываются на языке CDL, который специально ориентирован на конструкции-деревья. В нем существует четыре стандартные операции для работы с такими структурами: объединение (`join`), расщепление (`split`), равенство (`equal`) и присваивание (`assign`). Так, чтобы построить новое дерево из двух других при помощи некоторой операции `OP`, надо всего лишь написать следующее:

```
[EXPR1 OP EXPR2  -> EXPR]
```

Для того чтобы проверить, что дерево собрано из двух других деревьев с помощью операции `OP`, достаточно написать:

```
[EXPR -> EXPR1 OP EXPR2]
```

В правой части правил позволяется использовать код на целевом языке (зачастую это код на языке ассемблера) и таким образом задавать способ вычисления аффиксов.

Одна из главных трудностей использования CDL заключается в том, что язык не поддерживает именования переменных, т. е. переменные одного типа называются с помощью последовательных номеров (например, `EXPR1, EXPR2` и т. д.). И хотя имеется возможность задавать различные определения, подобно тем, которые можно создавать с помощью директивы `#define` в языке C, это является, скорее, «синтаксическим сахаром», а не исчерпывающим решением проблемы. Язык не поддерживает также операции логическо-

го отрицания, и потому поток управления базируется на успехе/-неудаче правил обрабатываемой грамматики (это одно из отличий CDL от языка Prolog, на который он во многом похож). Система CDL является, пожалуй, единственным крупным примером реализации метода аффиксных грамматик для разработки DSL. Но, к сожалению, множество недостатков системы и отсутствие понятной документации для последней версии CDL3 затрудняют ее использование для реализации предметно-ориентированных языков.

3.2. Системы переписывания

Существует большое количество методов и логических формализмов, использующих процедуры последовательной замены частей формул или термов формальной спецификации в соответствии с определенными правилами, которые называются *правилами переписывания* (Rewriting Rules) [4, 14]. Впервые правила переписывания были введены в лямбда-исчислении А. Черчем [6]. Простейшим примером переписывания в графе является замена при оптимизации в дереве программы умножения на два сложением.

Вносить небольшие изменения в DSL, описанный с помощью какой-либо системы переписывания, существенно проще, чем в атрибутные грамматики. В подходах, базирующихся на переписываниях, наиболее острым вопросом является наличие среды для полного цикла реализации DSL. Как правило, таковой не предлагается.

3.2.1. Проект Term Processor Generator Kimwitu

Данный проект является совместным исследовательским проектом нескольких университетов Королевства Нидерландов и ориентирован на создание системы для разработки программ, которые работают с деревьями или термами, в частности компиляторов (будем называть далее эту систему Kimwitu). Первые прототипы системы появились еще в начале 90-х годов прошлого столетия, в 1997 году было представлено первое целостное описание системы [27].

Система Kimwitu принимает на вход абстрактное описание термов с директивами для реализации и описание функций на этих термах, а на выходе выдает код на языке ANSI C, определяющий структуры данных для термов и функции работы с этими термами (создание термов, выделение памяти и т. п.), а также реализацию правил переписывания.

Как и многие другие подобные инструменты, Kimwitu впервые был опробован для генерации собственного кода. В дальнейшем с помощью этой системы разрабатывались другие компиляторы, а также разнообразные инструменты тестирования. Одним из самых известных применений Kimwitu было его использование при реализации языка LOTOS[1] [27]. В этом проекте с помощью Kimwitu были сгенерированы структуры данных и методы ввода/вывода.

Основным средством спецификации в Kimwitu служит алгебра термов, описывающая базовые термы и операции над ними. Внешние и внутренние описания структур в системе одинаковы, потому можно легко разбить описание термов и функций на несколько модулей и работать с ними отдельно, объединяя в нужные моменты результаты работы. Каждый тип терма соответствует какому-то типу в языке C, поэтому при проверке типов Kimwitu полагается на проверку типов в C-компиляторе.

В качестве небольшого примера рассмотрим определение упрощенного алгебраического выражения:

```
expr:   Plus(expr expr)
  |     Minus(expr expr)
  |     Neg(expr)
  |     Zero()
      { float value = 0;} /* атрибут */;
```

Выражение в данном примере определяется рекурсивно, через сложение, вычитание, отрицание и нулевое выражение (Zero). При этом терму `expr` приписывается атрибут `value`, который предназначен для хранения значения выражения и инициализируется нулем. По этой спецификации Kimwitu генерирует следующий C-код:

```
typedef enum {sel_Neg = 1, sel_Minus = 2, sel_Plus = 3,
              sel_Zero = 4
              } kc_enum_operators;
typedef struct kc_tag_expr *expr; struct kc_tag_expr {
        kc_enum_operators prod_sel; /* тип выражения */
        union {
                struct {
                        expr expr_1;
```

[1]Language of Temporal Ordering Specification — язык формальных спецификаций, использующийся для описания телекоммуникационных протоколов в ISO-стандартах, основанных на модели ISO/OSI.

```
                } Neg;
                struct {
                        expr expr_1;
                        expr expr_2;
                } Minus;
                struct {
                        expr expr_1;
                        expr expr_2;
                } Plus;
        } u;
        float value;
};
```

Перечислимый тип `enum kc_num_operators` определяет, какой именно тип выражения представляется (сложение, вычитание или отрицание). Само выражение описывается с помощью структуры `kc_tag_expr`, которая содержит идентификатор типа выражения `prod_sel` и выбор одной из структур для отрицания, сложения или вычитания, каждая из которых, в свою очередь, содержит поля типа `expr`. Ниже представлено одно из правил переписывания для заданного выше терма `expr`:

```
Neg(x) -> <:  Minus(Zero(), x)  >;
```

Это правило определяет, что отрицание <x> — это то же самое, что и выражение «ноль минус x».

Для описания функций над термами в системе имеется возможность использовать в коде сопоставление по шаблону, например:

```
expr sum(exprlist el) {
    expr sub_total = Zero();
    foreach( $e; exprlist el ) {
        Add( x ):       { sub_total = Plus( sub_total, x ); }
        Subtract( x ):  { sub_total = Minus( sub_total, x ); }
    }
    return sub_total;
 }
```

Данная функция вычисляет сумму выражений из списка `el`. В цикле `foreach`, благодаря использованию сопоставления по шаблону, выполняется только та часть, которая удовлетворяет шаблону. То есть для шаблона `Add` выполняется код `sub_total = Plus (sub_total, x)`, а для шаблона `Subtract` выполняется код

`sub_total = Minus (sub_total, x)`. Такой подход избавляет от необходимости описывать множество операторов ветвления с различными условиями.

Разработчики планируют расширить систему, добавив в нее возможность использования полиморфных функций (в основном для работы со списками), а также средства для задания условных правил переписывания и создания пользователем своих атомарных типов. Планируется также реализовать поддержку языков C++ и Java, а также базовых типов коллекций (множеств, стеков, очередей, массивов). Пока же реализация DSL с помощью данной системы затруднительна, так как от разработчика нового языка требуется создать множество функций на языке C, оперирующих сложными структурами данных над термами. При этом возможности языка в существующей версии сравнительно невелики.

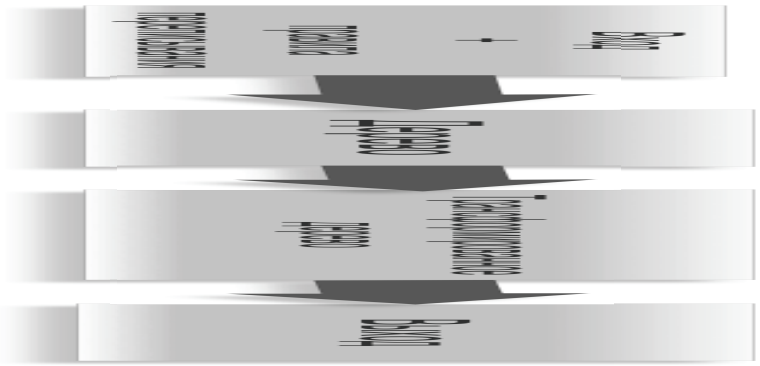
3.2.2. Система TXL

В 1985 году исследователями из университета города Торонто была предложена система (язык и программное средство) TXL, предназначенная для решения различных задач на основе переписывания и основанная на концепции функциональных языков и парадигме продукционных правил (rule-based paradigm) [7]. Любая программа на языке TXL состоит из двух частей:

- описание исходных структур с помощью грамматик в нормальной форме Бэкуса—Наура (BNF),
- набор правил для переписывания термов.

Общая схема работы системы TXL представлена на рисунке. По входной программе строятся древовидные структуры, которые затем преобразуются согласно поданным на вход правилам переписывания, и затем из полученного преобразованного дерева строится выходная спецификация. Таким образом, можно производить «переписывание» из одного языка в другой, т. е. реализовывать компилятор. Правда, для этого придется описать грамматику для входного языка, чтобы преобразовывать входной текст программ на этом языке в представление в виде дерева и далее — из дерева в итоговый исполняемый код, а также правила и функции переписывания.

Система TXL применяется для решения множества задач: это и перевод программ с одних языков программирования на другие, преобразования различных формальных спецификаций, а также структурированных документов, трансформация схем баз данных,



Общая схема работы системы TXL.

распознавание математического рукописного текста, анализ сетевых пакетов и многое другое. Система используется как в научных, так и в коммерческих целях. Следующие компании используют TXL в промышленных целях: Intel, AT&T, Sun Microsystems, Digital Equipment Corp., IBM, Hewlett Packard, Motorola, Ericsson и др. В список учебных заведений, где TXL используется в академических целях, входят многие крупные университеты мира.

В качестве примера рассмотрим, как с помощью языка TXL можно задать расширение языка Pascal оператором присваивания «+=».

```
redefine statement
    ...
  | [coalesced_addition]
end redefine define coalesced_addition
    [reference] += [expression]
end define
```

Для введения нового оператора требуется переопределить конструкцию `statement`, добавив в нее новую альтернативу `coalesced_addition`. Заметим, что «`...`» является конструкцией языка TXL, позволяющей не повторять уже имеющиеся альтернативы для `statement`, а просто сослаться на них. Требуется также определить новую грамматическую конструкцию `coalesced_addition`, что делается с помощью ключевого слова `define`.

Ниже представлено правило переписывания, которое позволяет заменить использование нового оператора присваивания «+=» на обычный оператор присваивания `V := V + (E)`:

```
rule transformCoalescedAssignments
    replace [statement]
        V [reference] += E [expression]
    by
        V := V + (E)
end rule
```

Сам по себе базовый язык TXL очень прост и малофункционален и описать на нем сколько-нибудь сложную систему очень непросто. Однако существуют многочисленные более полнофункциональные диалекты-расширения TXL, например Evolving TXL [8], в котором добавлены следующие возможности:

- правила сопоставления по образцу с возвратом ошибки времени исполнения в случае невыполнения ни одного соответствия (Must Matching Rules);
- правила, выполняющие не сопоставление по образцу или замену, а служащие для размещения блоков служебного кода (например, для определения переменных, описания вспомогательных процедур);
- сильная типизация, предназначенная для того, чтобы задавать уникальный тип, к которому может быть применено то или иное правило;
- выходные параметры для возможности возвращать значения из правил;
- оператор ветвления (известная конструкции if-then-else);
- параметризация правил типом данных;
- модульность.

Ряд других диалектов, улучшающих базовый язык TXL, сейчас находятся в стадии разработки.

3.2.3. Система ASF+SDF

Система ASF+SDF представляет собой среду разработки и набор инструментов для реализации новых языков программирования, для генерации и прототипирования средств поддержки разработки ПО, а также для создания анализаторов кода и проведения

различного рода преобразований над кодом. Первые версии системы появились еще в начале 90-х годов прошлого века. Система продолжает развиваться, последний выпуск состоялся в сентябре 2008 года. В качестве примеров использования системы ASF+SDF для разработки DSL можно указать проект EURIS (посвящен решению вопросов безопасности на железных дорогах, 1995 год) [9], проект Risla (финансы, 1998 год) [26], проект ALMA-0 (поисковая система, 1998 год) [3].

Для работы с системой ASF+SDF имеется среда разработки MetaStudio IDE, которая включает в себя компиляторы для двух основных языков, используемых в системе — ASF и SDF, — генераторы парсеров, библиотеку ATerms (описывает общий вид представления данных в системе) и несколько визуализаторов различных промежуточных представлений.

Syntax Definition Formalism (SDF) является языком для описания контекстно-свободных грамматик. Его отличительной особенностью является возможность модульного описания синтаксиса языков программиования (в частности, в одну SDF-грамматику можно импортировать полностью или частично другую). Ниже представлен пример описания булевского выражения с помощью SDF:

```
module basic/Booleans context-free syntax
   "true"                      -> Boolean
   "false"                     -> Boolean
   lhs:Boolean "|" rhs:Boolean -> Boolean {left}
   lhs:Boolean "&" rhs:Boolean -> Boolean {left}
   "not" "(" Boolean ")"       -> Boolean
   "(" Boolean ")"             -> Boolean
 context-free priorities
   Boolean "&" Boolean -> Boolean >
   Boolean "|" Boolean -> Boolean
```

Булевское выражение задается в виде модуля `Boolean`, который располагается в модуле `basic`. Булевское выражение определяется рекурсовно и является, согласно этому определению, или логической константой `true/false`, или одной из логических операций `|`, `&` или `not`, или операцией взятия подвыражения в скобки. Модификатор `left` указывает, что данный оператор является ассоциативным слева. Раздел `priorities` позволяет задать приоритеты одних операторов над другими. В данном случае указывается, что приоритет

операции `&` выше приоритета операции `|`. Модификаторы `Lhs` и `rhs` являются указаниями на левый и правый аргумент операции. Для них определяется, что они имеют тип `Boolean`.

Algebraic Specification Formalism (ASF) является языком для описания семантики нового языка программирования, грамматика которого уже была определена с помощью SDF. Семантика описывается с помощью правил переписывания.

Общая схема работы системы ASF+SDF выглядит следующим образом.

1. Синтаксический анализ, т. е. разбор грамматики, описанной на SDF, и построение по ней абстрактного синтаксического дерева.
2. Препроцессирование — выполнение преобразований над тем абстрактным представлением, которое было получено на предыдущей стадии (например, связывание переменных в условиях, введение конструкций if-then-else для правил с одинаковыми левыми частями).
3. Генерация кода. На этой стадии для правил переписывания порождается код на языке C. Выбор языка C обусловлен хорошей переносимостью и обширными возможностями по оптимизации получаемого кода.
4. Постпроцессинг — выполнение различных улучшающих преобразований над полученным C-кодом (например, кэширование констант).

Система ASF+SDF является универсальной и предназначена для реализации самых разнообразных предметно-ориентированных языков. И это сказывается на размере спецификаций, которые нужно создать при описании DSL, а также приводит к громоздкости и избыточности получаемого кода. Также недостатком является тот факт, что язык C — это единственный выходной язык системы. В данный момент разработчики системы ASF+SDF работают над системой Rascal [13], предназначенной для реализации предметно-ориентированных языков и базирующейся на тех же идеях, что и система ASF+SDF. Разработчики обещают сделать систему Rascal более простой и удобной.

4. Расширение базового языка

В этом разделе речь пойдет о средствах реализации небольших DSL, которые основываются на универсальных языках и добавляют в них новые возможности и/или ограничивают существующие. Здесь используются разные подходы: самым широко используемым является метод макрорасширений, известен также метод шаблонов C++ (template metaprogramming), относительно недавно появился метод синтаксических расширений.

В рамках данной статьи мы рассмотрим метод макрорасширений, а также метод синтаксических расширений, который авторы статьи считают наиболее перспективным для разработки предметно-ориентированных языков.

4.1. Метод макрорасширений

Во многие языки пограммирования, например C/C++, LISP, встроены макроязыки, которые позволяют прикладным программистам в текстах своих программ осуществлять замену специально оформленных фрагментов программы на другой текст в соответствии с некоторыми заданными шаблонами. Для этого в тексте программы должны быть описаны как сами шаблоны, так и случаи их использования, что выполняется с помощью специальных конструкций макроязыка (так называемых макроконструкций). С формальной точки зрения макроязык позволяет задать систему переписывания, а макропроцессор, который запускается до компилятора, выполняет заданные правила переписывания и таким образом избавляет текст программы от макроконструкций, является универсальным инструментом переписывания, который, однако, используется не разработчиками компиляторов, а обычными прикладными программистами.

Фактически с помощью макроконструкций можно ввести в язык программирования новую синтаксическую конструкцию. Именно эта возможность делает макроязыки удобным и естественным способом задания небольших DSL на основе универсальных языков программирования (разумеется, это справедливо для тех языков, которые содержат в себе макроязыки).

В качестве примера рассмотрим определение конструкции if-then-else в диалекте языка LISP, где такой конструкции нет, а есть только конструкция `cond` — многовариантное условное предложе-

ние. Конструкция `cond` в языке LISP представляет собой список пар условие/результат. При выполнении программы условия просматриваются по очереди до тех пор, пока одно из них не окажется выполненным, и тогда результат из этой пары становится результатом оператора cond. Если ни одно условие не выполнилось, то используется результат из формального условия, обозначаемого с помощью идентификатора `t`. Новую конструкцию назовем `Iff` и определим следующим образом:

```
(defmacro Iff (Test Then Else)
  '(cond
     (,Test ,Then)
     (t     ,Else)) )
```

Эта макроконструкция принимает на вход три аргумента и сначала вычисляет первый (`Test`), который задает логическое условие нашей новой конструкции `Iff`. Если это условие выполнено, то вычисляется второй аргумент (`Then`) и `Iff` возвращает его значение в качестве результата. В противном случае вычисляется третий аргумент (`Else`) и его значение возвращается в качестве результата. Макроязыки широко используются на практике, так как позволяют в рамках того или иного программного проекта создать оптимальный набор языковых абстракций, удобно отражающих специфику данной предметной области.

Рассмотрим пример. В проекте по реализации сетевых протоколов, разрабатываемом на языке C, для удобства создания тестов с помощью макроязыка C был создан специальный DSL. Он позволил скрывать в макроконструкциях большое количество повторяющихся технических деталей и легко создавать многочисленные тесты. Одной из важных составляющих этого DSL являются конструкции для ведения журнала событий тестирования, позволяющие вставлять в разные места тестов вывод сообщений об обнаруженных ошибках и различные предупреждения, варьируя степень подробности и используя разные форматы, с сохранением в журнале соответствующего места в коде программы (имя функции, номер строки). Ниже представлена макрокоманда `ENTRY` для сохранения в журналах записей о точках входа в функцию:

```
#define _LOG_ENTRY(_us, _fs, _args...) \
    TE_LOG(TE_LL_ENTRY_EXIT, TE_LGR_ENTITY, _us,\
           "ENTRY to %s(): " _fs, __FUNCTION__, _args + 0)
```

```
#define TE_LOG_ENTRY(_us, _fs...) \
    do { \
        if (TE_LOG_LEVEL & TE_LL_ENTRY_EXIT) { \
            if (!!(#_fs[0])) { \
                _LOG_ENTRY(_us, _fs); \
            } \
            else { \
                TE_LOG(TE_LL_ENTRY_EXIT, TE_LGR_ENTITY, _us,\
                       "ENTRY to %s()", __FUNCTION__); \
            } \
        } \
    } while (0)
#define ENTRY(_fs...)  TE_LOG_ENTRY(TE_LGR_USER, _fs)
```

При этом если дополнительно в качестве параметра этой команде указать некоторое сообщение, то оно также будет записываться в журнал. При описании макрокоманды `ENTRY` используется другая макрокоманда `TE_LOG`, описывающая общий случай записи информации в журнал событий, а также макрокоманда `__FUNCTION__`, позволяющая определить имя функции, в которую перешел поток исполнения теста.

Теперь для вывода сообщения в журнал в момент входа теста в функцию требуется добавить в код этой функции одну из следующих строк:

```
ENTRY("%s(): value=%s", __FUNCTION__, value);
```

или:

```
ENTRY();
```

В первом случае в журнал будет выведена строка с именем функции, а также значение параметра `value` макрокоманды `ENTRY`. Во втором случае в журнал будет добавлена строка `ENTRY to <имя функции>`.

В данном DSL существует также группа макрокоманд для работы с данными, содержащимися в пакете того или иного протокола, а также макрокоманды, определяющие подготовительные и заключительные действия тестов. Использование именно макрокоманд, а не процедур оказывается более экономичным по времени выполнения, не требуя передачи параметров при работе теста, что важно при выполнении больших тестовых пакетов. Также макроконструкции могут использоваться там, где трудно или невозможно использовать процедуры, так как макроконструкции могут «разрывать»

конструкции языка и позволять оформлять повторно используемые фрагменты текста программ, которые не образуют законченную последовательность операторов.

При всей несложности реализации DSL методом макрорасширений и несмотря на имеющиеся многочисленные примеры таких реализаций, этот метод имеет ряд серьезных недостатков.

1. Отсутствие различных синтаксических проверок для новых конструкций, например проверок соответствия типов параметров.
2. При переводе препроцессором конструкций нового DSL в конструкции базового языка теряется их «привязка» к исходному тексту программы, что затрудняет понимание программистами сообщений об ошибках компиляции и сильно усложняет пошаговую отладку программы. При этом программистам приходится разбираться в том коде, который макроконструкции удобно скрывали при разработке программ.
3. Никто не мешает пользователю ввести свои макрокоманды, возможно переопределив при этом уже имеющиеся, что приводит к серьезным ошибкам в работе программы.
4. Пользователь так реализованного DSL никак не ограничен в возможностях базового языка и, значит, может создавать программы, выходящие за пределы данной предметной области, т. е. у данного подхода отсутствует семантическая замкнутость.

4.2. Метод синтаксических расширений

Метод синтаксических расширений (syntax extensions) основан на идее введения в существующий язык новых конструкций и семантических правил их трансляции в базовый язык. Проекции новых конструкций в базовый язык задаются с помощью правил синтаксических расширений [21].

Метод является следствием следующей идеи Джона Маккарти, предложенной им еще в конце 50-х годов при разработке языка LISP: программу можно представлять в виде списка выражений, который может быть создан, изменен, передан другой программе. Программа, оперирующая другой программой, например ее порождающая, — это и есть идея, которая лежит в основе всех синтаксических расширений. Причем, в отличие от макрорасширений, кото-

рые, в общем-то, тоже порождают текст на языке программирования, здесь при определении новых конструкций нам доступна информация о грамматике базового языка. Мы, например, можем указать, что параметром этой нашей новой конструкции может быть любое выражение базового языка или только переменные целого типа. В макрокомандах нет возможности указывать такую информацию, они ничего «не знают» о том языке, в который они будут раскрыты.

В качестве примера рассмотрим реализацию метода синтаксических расширений CamlP4 для функционального языка Ocaml, разработанную французким институтом INRIA [5]. Система CamlP4 предоставляет специальный язык для задания новых конструкций, расширяющих язык Ocaml. Причем новые конструкции описываются в терминах абстракного синтаксического дерева, на основе механизма цитирования (quotations). Система CamlP4 позволяет также преобразовать заданные расширения в промежуточное представление, которое можно откомпилировать компилятором Ocaml и получить исполняемые модули. Эти модули вместе с компилятором Ocaml будут компилятором нашего нового языка программирования, заданного с помощью данных синтаксических расширений. Процесс компиляции будет работать подобно препроцесорной обработке макрорасширений с последующей компиляцией полученного при этом кода, но в данном случае будут обеспечены синтаксические проверки правильности использования новых конструкций в программах пользователей языка.

Приведем несколько примеров. Первый пример рассматривает добавление оператора композиции двух функций в язык Ocaml:

```
EXTEND
  expr: AFTER "apply"
      [[ f=expr; "o"; g=expr -> <:expr< fun x -> $f$ ($g$ x) >>]];
END;
```

В данном синтаксическом расширении задается, что если в тексте пользовательской программы встретилась конструкция `f o g`, где `f` и `g` являются выражениями языка Ocaml, то эту конструкцию при компиляции нужно заменить на фрагмент абстрактного дерева, определенного в этом правиле после символа `->`. Детали определения поддерева в Camlp4 мы рассмативать здесь не будем.

Еще один пример — введение конструкции цикла repeat-until с помощью уже имеющейся в языке конструкции `while`:

```
EXTEND
  expr: LEVEL "let"
      [[ "repeat"; e1=expr; "until"; e2=expr ->
           <:expr< do { $e1$; while not $e2$ do { $e1$ } >>]];
END;
```

Данное правило определяет, что при компиляции конструкции `repeat; e1; until; e2`, где `e1` и `e2` являются выражениями языка Ocaml, ее нужно заменить на соотвепствующее поддерево.

В сообществе CamlP4 можно найти довольно интересные примеры DSL, созданные для работы с монадами, итеративными вычислениями, а также для реализации элементов ООП для Ocaml и др.

Интересен пример, который приводит Грейдон Хоар [10]. Автор рассказывает о построении компилятора для небольшого подмножества языка Make. Весь код нового компилятора занимает около 400 строк, т. е. существенно меньше, чем при использовании традиционных компиляторных методов. При этом порождаются лексический и синтаксический анализатор, оказываются реализованными связывание переменных, проверка типов, диагностика ошибок и генерация машинного кода.

Однако в методе синтаксических расширений до настоящего времени не решена проблема более строгого семантического контроля в новых конструкциях. Присутствует и проблема эффективности полученного кода компилятора, а также генерируемого итоговым компилятором кода программ, созданных с помощью нового DSL. Кажется необходимым построение некоторых оптимизирующих преобразований для базового языка, чтобы порождаемый код оказался более эффективным. И еще один недостаток данного подхода — это наследование полученным DSL тех принципов и парадигм, которые присущи базовому языку (т. е. в нашем случае Ocaml).

Заключение

В заключение отметим следующее. Предметно-ориентированные языки программирования бывают самыми разнообразными. Некоторые DSL лишь незначительно расширяют базовые универсальные языки и наследуют все их особенности, а также имеющиеся для данного языка инструменты. Для таких DSL используются методы, основанные на расширении базового языка. Но существует

и другая группа DSL, которые представляют собой полностью новые языки. Они самостоятельны и не содержат ничего, выходящего за рамки своей предметной области, они более точно отражают ее понятия и могут действительно рассматриваться как ее спецификации. Для реализации таких DSL следует использовать компиляторные подходы.

Обращаясь к критериям, сформулированным в начале статьи, приведем сравнение рассмотренных инструментов и методов (таблица). Отметим, что подходы, основанные на расширении базового языка, хотя и не могут решить проблему получения принципиально новых DSL, все же имеют свои достоинства: это быстрота разработки и компактность получаемого решения. Существующие реализации данного подхода целесообразно дополнить средствами спецификации семантического контроля. Для этого целесообразно использовать парадигму переписывания. Если сюда еще добавить возможность оптимизаций для повышения эффективности получаемого кода, то окажутся выполненными все критерии, сформулированные в начале статьи. Разработка такого подхода является направлением дальнейшей работы автора данной статьи.

Список литературы

[1] *Кознов Д. В.* Основы визуального моделирования. Интернет-университет информационных технологий; БИНОМ. Лаборатория Знаний, 2008. 246 с.

[2] *Aho A. V., Sethi R., Ullman J. D.* Compilers: Principles, Techniques, and Tools. Addison Wesley, 1986. 796 p.

[3] *Apt K. R., Brunekreef J. J., Partington V. and Schaerf A.* Alma-0: An Imperative Language that Supports Declarative Programming // ACM Transactions on Programming Languages and Systems 20. Vol. 20, N 5. 1998. P. 1014–1066.

[4] *Baader F. and Nipkow T.* Term Rewriting and All That // Cambridge University Press. 1999. 301 p.

[5] Camlp4 Reference Manual, Version 3.07, Institut National de Recherche en Informatique et Automatique. 2002. http://caml.inria.fr/pub/docs/manual-camlp4/index.html

[6] *Church A.* An Unsolvable Problem of Elementary Number Theory // American Journal of Mathematics. Vol. 58. 1936. P. 354–363.

[7] *Cordy J. R.* The TXL Source Transformation Language // Science of Computer Programming. Vol. 61. Issue 3. 2006. P. 190–210.

Сравнение рассмотренных инструментов и методов

	Компиляторные подходы						Расширение базового языка	
	Атрибутные грамматики			Системы переписывания			Макрорасширения	Синтакс. расширения
	Elegant	FNC-2	CDL	Kimwitu	TXL	ASF+SDF		
Семантическая замкнутость	Есть	Есть	Есть	Есть	Есть	Есть	Нет	Нет[a]
Трудоемкость реализации	Большая	Средняя (за счет модульности)	Большая	Большая	Большая	Большая	Малая	Малая
Трудоемкость внесения изменений	Большая	Средняя (за счет модульности)	Большая	Средняя	Средняя	Средняя	Малая	Малая
Оптимальность реализации	Зачастую реализация неоптимальна	Зачастую реализация неоптимальна	Зачастую реализация неоптимальна	Возможно задать оптимизации через переписывания	Возможно задать оптимизации через переписывания	Возможно задать оптимизации через переписывания	Реализация не всегда оптимальна	Реализация не всегда оптимальна
Поддержка оптимизации на уровне предметной области	Возможны[b]	Возможны[b]	Возможны[b]	Возможны[b]	Возможны[b]	Возможны[b]	Нет	Нет
Сохранение «привязки» к тексту	Есть	Есть	Есть	Есть	Есть	Есть	Нет	Нет

Примечания: [a] — возможно реализовать и встроить средства семантического контроля; [b] — поддержку оптимизаций на уровне предметной области можно добавить в компилятор.

[8] *Cordy J. R., Thurston A. D.* Evolving TXL // Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation. 2006. P. 117–126.

[9] *Groote J. F., Koorn J. W. C. and van Vlijmen S. F. M.* The Safety Guaranteeing System at Station Hoorn-Kersenboogaard // Proceedings of the 10th Annual Conference on Computer Assurance. (COMPASS '95). IEEE, Computer Society Press, Los Alamitos, CA. 1995. P. 57–68.

[10] *Hoare G.* One-Day Compilers or How I Learned to Stop Worrying and Love Static Metaprogramming, 2002. http://www.venge.net/graydon/talks/mkc/html/mgp00001.html

[11] *Jansen P., Augusteijn L., Munk H.* An Introduction to Elegant // Philips Research Laboratories. The Netherlands, 1993. 210 p.

[12] *Jourdan M., Parigot D., Julie C., Durin O., Le Bellec C.* Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System // Proceedings of the ACM SIGPLAN'90 Congerence on Programming Languages Design and Implementation. White Plains: New York, 1990. P. 209–222.

[13] *Klint P., Vinju J., Van der Storm T.* Easy Meta-Programming with Rascal // Pre-proceedings of the International Summer School. GTTSE 2009. Braga, Portugal, 2009. P. 185–238.

[14] *Klop J. V., De Vrijer R.* Term Rewriting Systems // Cambridge University Press. 2003. 680 p.

[15] *Knuth D.* Semantics of Context-Free Languages // Theory of Computing Systems. Vol. 2, N 2. 1968. P. 127–145.

[16] *Koster Cornelis H. A.* Compiler Description Language Manual. 2004. http://www.cs.ru.nl/~kees/cdl3/cdl3.pdf

[17] *Mernik M., Heering J., Sloane A.* When and How to Develop Domain-Specific Languages // ACM Computing Surveys. Vol. 37, N 4. 2005. P. 316–344.

[18] *Nguyen L. A.* The Modal Logic Programming System MProlog // J. J. Alferes and J. A. Leite (eds.). Proceedings of JELIA 2004. Springer. LNCS 3229. 2004. P. 266–278.

[19] *Paakki J.* Attribute Grammar Paradigms — A High-Level Methodology in Language Implementation // ACM Computing Surveys. Vol. 27, N 2. 1995. P. 196–255.

[20] *Raymond E. S.* The Art of UNIX Programming // Addison-Wesley. 2003. 550 p.

[21] *Skalski K.* Syntax-Extending and Type-Reflecting Macros in an Object-Oriented Language // Institute of Computer Science. University of Wroclaw. Master Thesis. 2005. 60 p.

[22] *Spinellis D.* Notable Design Patterns for Domain-Specific Languages // Journal of Systems and Software. 56(1). 2001. P. 91–99.

[23] *Van Den Brand M. G. J., Heering J., Klint P., Oliver P.* Compiling Language Definitions: the ASF+SDF Compiler // ACM Transactions on Programming Languages and Systems. Vol. 24, N 4. 2002. P. 334–368.

[24] *Van Den Brand M. G. J., Van Deursen A., Klint P., Klusener S., Van der Meulen E.* Industrial Applications of ASF+SDF / M. Wirsing and M. Nivat (eds.). Algebraic Methodology and Software Technology (AMAST'96). Springer-Verlag. LNCS, Vol. 1101. 1996. P. 9–18.

[25] *Van Deursen A., Klint P., Visser J.* Domain-Specific Languages: An Annotated Bibliography // ACM SIGPLAN Notices. Vol. 35, N 6. 2000. P. 26–36.

[26] *Van Deursen A., Klint A.* Little Languages: Little Maintenance? // J. Softw. Maint. 1998. Vol. 10, N 2. P. 75–92.

[27] *Van Eijk P., Belinfante A., Eertink H., Alblas H.* The Term Processor Generator Kimwitu. CTIT Technical report. Dec. 1996. P. 96–49.