# Создание языка для проверки свойств контекстно-свободных грамматик

Силина Ольга Александровна
Кафедра системного программирования,
математико-механический факультет СПбГУ, 444 группа

Научный руководитель: Андрей Бреслав,
аспирант кафедры высшей математики СПб ИТМО
9 мая 2009

## 1 Введение

Изучение контекстно-свободных грамматик важно для современной информатики. Соответствующие им контекстно-свободные языки находят применение в разнообразных программных продуктах, таких как:

- компиляторы
- средства форматирования исходного кода
- средства статического анализа программ
- синтаксические редакторы
- системы верстки
- программы просмотра форматированного текста
- поисковые системы

и другие.

При работе с контекстно-свободными грамматиками важно уметь проверять различные свойства этих грамматик.

Например, представим себе генератор парсеров. Обычно генератору на вход можно подать не любую грамматику, а только обладающую определенными свойствами (например, LL(k)). Если грамматика неправильная, генератор ничего не выдаст или выдаст ерунду. Конечно, можно менять грамматику до тех пор, пока мы не увидим корректный вывод генератора, но хотелось бы иметь более элегантный метод решения этой проблемы. Прежде всего, хотелось бы выполнять проверку статически, и хотелось бы иметь самостоятельный инструмент, никак не привязанный к тому, для чего мы хотим использовать построенную грамматику. В конце концов, нашей задачей может быть просто построение грамматики без немедленного её использования в каких-либо инструментах, которые займутся за нас проверкой свойств, но и в этом случае мы хотим иметь возможность гарантировать какие-то её свойства. Проверять свойства грамматики вручную можно только для очень маленьких грамматик, но даже и в этом случае велика вероятность ошибиться.

Таким образом, понятна потребность в удобном, самостоятельном инструменте, позволяющем автоматически проверять контекстно-свободные грамматики на соответствие заданному списку свойств. Эта задача и решается в моей курсовой работе.

В настоящее время практически не существует инструментов, специально предназначенных для проверки свойств грамматик. Большая часть имеющихся инструментов решает только узкие специальные задачи.

# 2 Мой подход к решению проблемы

Если мы хотим создать инструмент, который будет использоваться для решения реальных задач, а не пылиться в архивах кафедры, необходимо задуматься об удобстве интерфейса пользователя. В моей работе мне

хотелось достичь следующих целей:

- создание красивой и корректно работающей программы
- удобство ввода и изменения данных
- возможность при необходимости легко дополнить программу новыми свойствами

Наличие специфической области задач (проверка свойств грамматик) навело меня на мысль о создании доменно-специфичного языка. Предметно–ориентированные (или доменно–специфичные, DSL) языки – одна из популярных идей в мире высокоуровневого программирования. Идея проста: для того, чтобы решать часто встречающиеся в какой–то конкретной области задачи, проще создать маленький новый язык, чем писать громоздкие конструкции на языке общего назначения.

Но мало создать язык – в современном мире никто не будет использовать новый язык, если к нему не прилагается удобная IDE (интегрированная среда разработки), с подсветкой синтаксиса, контекстной подсказкой и прочими прекрасными вещами, которые вы привыкли видеть в своем редакторе кода. Разработка такой IDE требует огромного количества усилий и времени, и, казалось бы, является слишком сложной и несоразмерно большой задачей в рамках этой курсовой работы. Однако, решение существует – можно воспользоваться какой-либо из систем метапрограммирования, позволяющих получить программу при меньших затратах времени и усилий, чем если бы программист писал её вручную.

Одно из первых решений, которые приходится принимать при выборе системы метапрограммирования – выбор между текстовым и визуальным синтаксисом. Для описания грамматик программисты испокон веков использовали текстовый синтаксис, который хорошо подходит для этой задачи, поэтому и при создании моего языка я хотела использовать что-то подобное.

В конце концов, я остановилась на использовании системы JetBrains MPS (Meta Programming System).

# 3 JetBrains MPS

Основным преимуществом MPS является возможность расширять языки. Каждый существующий язык имеет строго определенный синтаксис, и это ограничивает его гибкость. Главная проблема расширения языка – текстовое представление кода. Это особенно важно, если мы хотим использовать разные расширения языка, у каждого из которых может быть свой собственный синтаксис. Это приводит к идее нетекстового представления программного кода. Огромный плюс такого подхода в том, что он избавляет нас от необходимости парсить код. В JetBrains MPS код хранится в виде абстрактного синтаксического дерева (Abstract Syntax Tree, AST), которое полностью описывает программный код. В то же время, MPS предлагает эффективный способ писать код так, как будто это текст.

В процессе создания языка на MPS, можно создавать правила для редактирования и отображения кода. Можно так же определять систему типов и ограничений языка, что позволяет MPS верифицировать код на лету и делает программирование на новом языке проще.

Использование доменно-специфичных языков позволяет писать программы на более высоком уровне абстракции, пользуясь понятиями, естественными для данной конкретной области. MPS позволяет создавать новые языки так же, как обычный программист создает классы или методы, и вместе с языком создается и удобный редактор.

# 4 Основные идеи

Для решения поставленной задачи были созданы три языка:

- GrammarLanguage
- GrammarPropertiesLanguage
- GrammarVerificationLanguage

**GrammarLanguage** – вспомогательный язык для описания контекстно-свободных грамматик. Он позволяет описывать грамматики в форме,

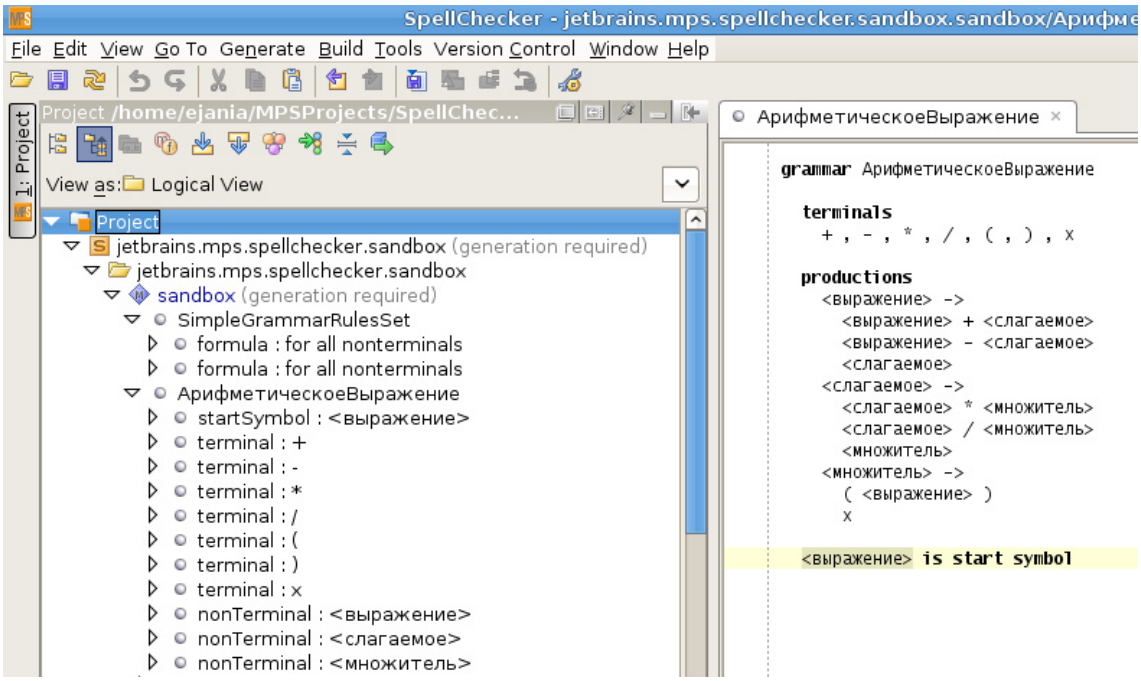


Рис. 1: Пример программы на GrammarLanguage: арифметическое выражение

похожей на стандартную запись грамматики в форме Бэкуса-Науэра (BNF). Этот язык используется для того, чтобы ввести в систему грамматику, свойства которой мы хотим проверить. Этому языку легко найти применение и вне моего проекта – он пригодится в любой программе, имеющей дело с грамматиками.

Грамматика, введенная при помощи этого языка, хранится в программе в виде дерева, удобного для обхода.

**GrammarPropertiesLanguage** – язык, описывающий свойства самой грамматики. Основное выразительное средство языка – оператор "нетерминал выводит строку, соответствующую регулярному выражению". Кроме этого, в языке есть операторы, соответствующие логическим операциям, кванторам и некоторым другие служебные понятиям.

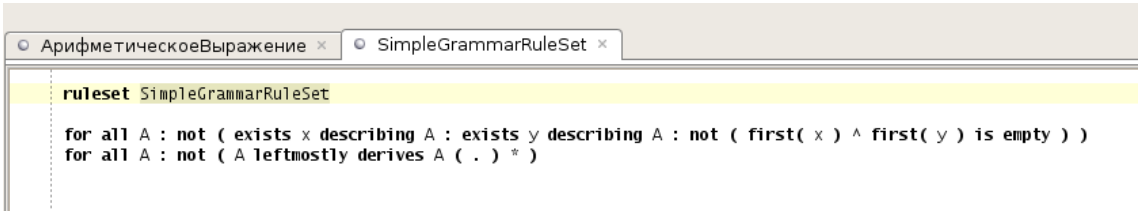


Рис. 2: Пример программы на GrammarPropertiesLanguage: LL(1) и самовставленность

В этих терминах можно описать очень много важных свойств контекстно-свободных грамматик. Приведем несколько примеров свойств, программы для проверки которых можно записать буквально в одну строчку:

- леворекурсивность
- самовставленность
- соответствие нормальной форме Хомского
- соответствие нормальной форме Грейбах
- является ли граммитика автоматной
- является ли грамматика LL(1)-грамматикой

и так далее.

Программа на GrammarPropertiesLanguage представляет собой набор свойств, на соответствие которым нужно проверить грамматику.

Наконец, **GrammarVerificationLanguage** – служебный язык, который объединяет все воедино. Программа, написанная на этом языке, принимает на вход программу на GrammarLanguage и программу на GrammarPropertiesLanguage и выдает результат, говорящий, удовлетворяет ли данная грамматика данному набору правил, и если нет, то почему.

Алгоритм, используемый для выяснения вопроса о том, выводится ли из данного нетерминала данное регулярное выражение, в первом приближении представляет собой поиск в ширину по дереву, получаемому при

раскрытии продукций, с корнем, соответствующим нетерминалу. Этот простой алгоритм оптимизирован путем отсечения веток, заведомо не соответствующих регулярному выражению. Для облегчения такого отсечения проводится предвычисление, позволяющее за O(1) выяснять, может ли данный символ оказаться на первом месте в данной строке после выполнения нескольких (возможно, не всех) продукций.

# 5 Результаты

В результате работы над проектом реализован абстрактный и конкретный синтаксис упомянутых выше языков, а также прототип интерпретатора.

Время работы интерпретатора, к сожалению, пока что оставляет желать лучшего. Оно экспоненциально зависит от длины регулярного выражения, используемого для определения проверяемых свойств грамматики, и линейно – от количества продукций грамматики. Дальнейшее направление исследований по этому вопросу включает в себя улучшение времени работы интерпретатора, а также исследование вопроса о том, можно ли применять заменить основной алгоритм на более эффективный.