Анализ синтаксического разбора текста с помощью парсер-комбинаторов

С.В. Чубейко, А.Н. Цуриков, В.С. Палагута

Ростовский государственный университет путей сообщения, Ростов-на-Дону

Аннотация: В статье исследуются идеи использования стратегий синтаксического анализа для оптимизации производительности парсер-комбинаторов. Рассмотрено в качестве примера служебные данные синтаксического анализа PetitParser, а так же работа компилятора синтаксического анализатора. Так же подробно описана оптимизация и анализ производительности синтаксических анализаторов. На различных примерах оценена их эффективность и приведен сравнительный анализ.
Ключевые слова: оптимизация, синтаксический разбор, анализатор, парсер-комбинатор, анализ производительности.

Введение

В связи с постоянным увеличением информации в интернете, осуществления поиска информации посредством поисковых систем не охватывает всех потребностей пользователей [1]. Все более распространен синтаксический анализ, при этом используются синтаксические анализаторы, то есть программы, выполняющие данный разбор. Существует довольно много реализаций различных анализаторов [2], однако при появлении динамики и необходимости построения более сложных синтактических анализаторов используются парсер-комбинаторы. Парсер-комбинаторы используются для построения синтаксических анализаторов по правилам некоторой грамматики, используя возможности функциональных языков программирования.

Парсер-комбинаторы – универсальны и имеют гибкий подход к синтаксическому анализу. Они следуют структуре лежащей в основе грамматики, являются модульными, хорошо структурированными, легко распознают большое количество языков, включая контекстно-зависимые. Однако эти преимущества создают накладные расходы так как, используется

тот же мощный алгоритм синтаксического анализа для распознавания даже простых языков [3].

Существуют методы для достижения линейной асимптотической эффективности парсер-комбинаторов, однако, необходимо учитывать постоянный множитель. Множитель можно опустить в некоторой степени, но для этого требуются передовые методы метапрограммирования.

Принцип работы парсер-комбинатора

Рассмотрим пример служебных данных синтаксического анализа PetitParser [4]. PetitParser - это комбинатор синтаксического анализатора, который использует синтаксический анализ пакетов, синтаксический анализ без сканирования, а также разбор выражений грамматик (PEG). Грамматика разбора синтаксического анализа макета - это кортеж $G = (N; \Sigma; R; e_s)$, где N – конечное множество нетерминалов, Σ – является конечным набором терминальных символов, R – конечный набор правил, e_s – начальное выражение. Для формализации семантики грамматики $G = (N; \Sigma; R; e_s)$ мы расширяем стандартную семантику PEG следующим образом: вход представляет собой тройку $(e; x; S_{in})$ (выражение, вход, стек), вывод – это тройка $(o; y; S_{out})$ (выход, суффикс, новый стек). Где e – выражение синтаксического анализа, $x \in \Sigma^*$ – входная строка, подлежащая распознаванию, S – стек с отступом уровней [5].

Рассмотрим работу PetitParser, используя пример с грамматикой, описывающей простое программирование языка, как показано на Рис 1.

Программа, соответствующая этой грамматике, состоит из непустой последовательности классов. Класс начинается с classToken, за которым следует idToken и тело. Правило classToken является ключевым словом «class», за которым должен следовать пробел, который не используется. Это достигается за счет использования предиката и последующего #space, который ожидает пробел или табулятор. Идентификаторы начинаются с

буквы, за которой следует любое количество букв или цифр. Ключевое слово и идентификаторы класса преобразуются в экземпляры Token, которые содержат информацию о начальной и конечной позиции и строковом значение.

На Рис.1 представлен пример грамматики, определенный в упрощенной версии DSL PetitParser [6]. Domain-specific language (DSL) - компьютерный язык, специализированный для какой-либо конкретной области применения. Он отличается от языков общего назначения (GPL), которые широко применяются во многих областях.

Основная задача компилятора синтаксического анализатора состоит в создании высокопроизводительного анализатора из парсер-комбинаторов, сохраняя при этом все преимущества. Анализируя данную грамматику PEG, компилятор синтаксического анализатора выбирает наиболее подходящую стратегию синтаксического анализа для каждого из своих правил и создает перекрытие синтаксического анализатора, где каждый метод представляет собой правило грамматики с выбранной стратегией, реализованная в теле метода.

```
letterOrDigit← #letter / #digit
identifier   ← #letter letterOrDifit*
idToken      ←  identifier token
classToken   ← ('class' &#space) token
class        ←  classToken idToken body
          map:  [ : classToken : idToken : body  |
                   classNode new
                      name: idToken value;
                      body: body
                ]
methodToken ← ('method' &#space) token
method           ← methodToken idToken body
               map:  [ : methodToken: idToken : body  |
                        MethodNode new
                          name: idToken value;
                          body: body
                       ]
body                ← #indent
                         (class / mathod)*
                       #dedent
program         ← class+
```

Рис. 1 – Пример грамматики, определенный в упрощенной версии DSL PetitParser

На Рис. 2 показан рабочий процесс синтаксического анализатора с помощью компилятора. Во-первых, используется гибкость и легкость выражения комбинаторов по языку для определения грамматики (этап прототипа). Затем синтаксический анализатор анализирует грамматику, чтобы выбрать наиболее подходящие стратегии синтаксического анализа, строит анализатор «сверху вниз» (этап компиляции) и позволяет эксперту дополнительно модифицировать скомпилированный синтаксический анализатор для дальнейшего повышения производительности (этап ручной настройки) [7].

В итоге полученный анализатор может быть развернут как обычный класс и использоваться для синтаксического анализа с максимальной производительностью (этап развертывания) [8].

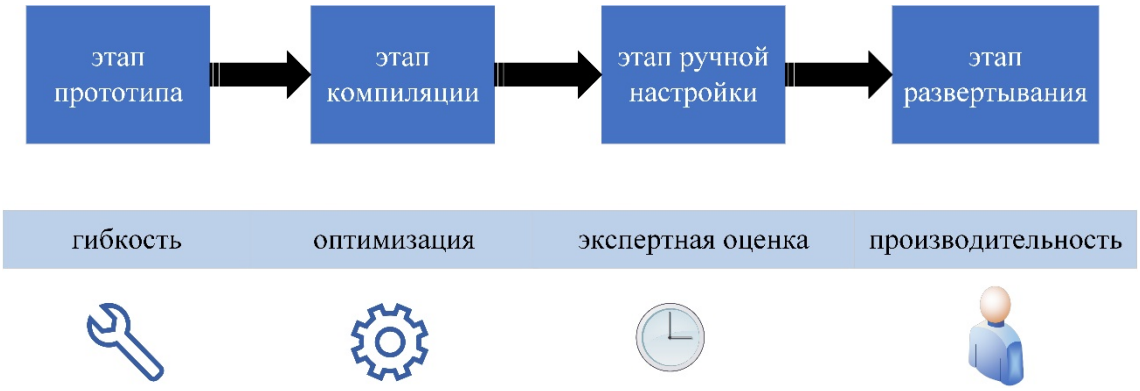

Рис. 2 – Фазы синтаксического анализатора

Эффективность синтаксических анализаторов

Рассмотрим эффективность скомпилированных синтаксических анализаторов по сравнению с простым PetitParser, а также эффективность синтаксических анализаторов Smalltalk, доступных в языке программирования Pharo.

Компилятор PetitParser применяет методы компилятора синтаксического анализатора и выводит класс Smalltalk [9], который выступает в качестве анализатора «сверху вниз», эквивалентного комбинатору ввода. Компилятор PetitParser доступен для Pharo и Smalltalk/X. Он уже используется в двух средах: язык для среды программирования Live Robot и язык разметки Pillar.

Компилятор PetitParser охватывает более трех тысяч модульных тестов. Кроме того, мы проверили компилятор синтаксического анализатора, взяв несколько существующих комбинаторов PetitParser и сравнив их результаты с результатами, полученными их эквивалентным скомпилированным вариантом. В частности, мы подтвердили результаты из четырех синтаксических анализаторов: синтаксический анализатор Java, Smalltalk, синтаксический анализатор Ruby и Python.

Измеряем производительность по следующим критериям:

1.Выражения являются эталоном, измеряющим производительность арифметических выражений. Вход состоит из выражений с операторами (,), *, + и целыми числами. Скобки должны быть сбалансированы и рассмотрены приоритеты операторов.

Грамматика не находится в детерминированной форме, т. е. она использует неограниченные возможности поиска и обратного отслеживания. Синтаксический анализатор содержит восемь правил.

2. Smalltalk является эталоном, измеряющим производительность синтаксического анализатора Smalltalk. Вход состоит из исходного кода на языке программирования Pharo. Синтаксический анализатор содержит приблизительно восемьдесят правил.

3. Java является эталоном, измеряющим производительность синтаксического анализатора Java, предоставляемого сообществом платформы анализа Moose. Вход состоит из стандартных библиотек

библиотеки JDK 6. Синтаксический анализатор содержит около двухсот правил.

4. Ruby является эталоном, измеряющим производительность синтаксического анализатора Ruby. Вход состоит из нескольких проектов GitHub Ruby. Синтаксический анализатор содержит около сорока правил. Синтаксический анализатор не завершен. Он использует отступы, ограниченные «большие пространства» и специальную версию изолированного синтаксического анализа для извлечения модулей, классов, методов, вызовов методов и их приемников.

5. Python является эталоном, измеряющим производительность отступающего паресера Python. Вход состоит из нескольких проектов Python с открытым исходным кодом. Синтаксический анализатор содержит около сорока правил. Синтаксический анализатор не завершен. Он использует изоляцию для извлечения структурных элементов и пропускает остальное. Структурными элементами, которые извлекаются, являются: классы, методы, if, while, for и with операторы.

6. JSON является эталоном, измеряющим производительность стандартного синтаксического анализатора JSON. Вход состоит из случайно сгенерированных JSON-файлов. Синтаксический анализатор содержит приблизительно двадцать правил.

Представленные тесты охватывают множество грамматик от небольших до сложных, варьирующихся от восьми до двухсот грамматических правил. Они также охватывают стандартные грамматики (Java, Smalltalk, JSON), изолированные грамматики (Python, Ruby), контекстно-свободные грамматики (Java, Smalltalk, JSON) и контекстно-зависимые (Python, Ruby).

Проводя сравнительный анализ, выполняем каждый тест десять раз, используя версию Pharo VM для Linux [10]. Все синтаксические анализаторы и входы инициализируются заранее, а затем мы измеряем время для

синтаксического анализа. Размер входного файла настроен таким образом, что даже самый быстрый тест работает как минимум на секунду. Сообщаем об ускорении (соотношение между лучшим временем оригинального PetitParser, служащим базовым и лучшим временем его скомпилированной версии) и временем на символ.

Результаты. Ускорение скомпилированной версии по сравнению с исходной версией показано на Рис. 3.

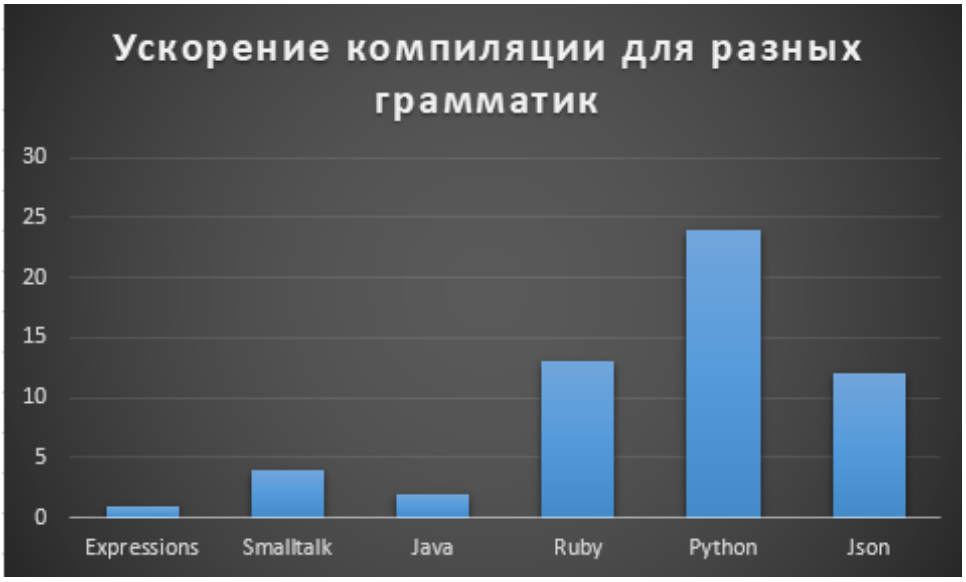

Рис. 3 – Ускорение компиляции для разных грамматик

Анализатор выражений показывает двойное ускорение. Мы приписываем этот результат тому факту, что синтаксический анализатор выражений выполняет неоднократное обратное отслеживание, а простота базовой грамматики не позволяет много оптимизировать. Анализаторы Smalltalk и Java демонстрируют ускорение чуть ниже четырех раз, более лучшие показатели у анализаторов Python, Ruby и JSON – десятикратное ускорение. Для наглядности они представлены на Рис. 4.

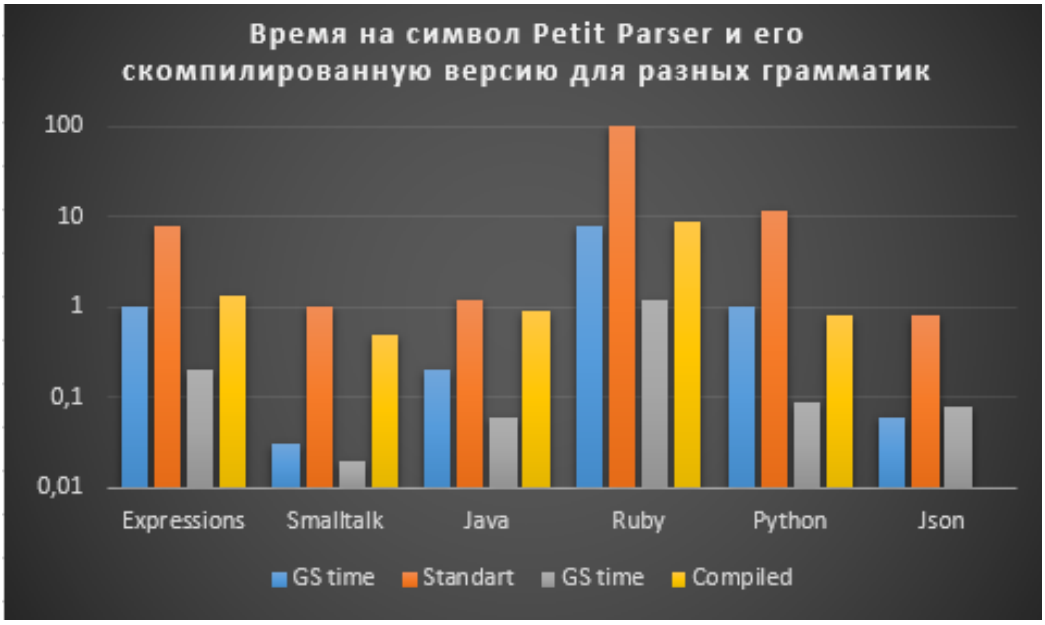

Рис. 4 – Время на символ анализатора PetitParser и его скомпилированную версию для разных грамматик

Анализаторы Python и Ruby чувствительны к контексту, а их исходные версии очень медленные. Это связано с накладными расходами на копирование стека отступов, который в значительной степени удаляется парсер-компилятором. Сам анализатор JSON является относительно быстрым, но его производительность может быть значительно улучшена.

Производительность синтаксических анализаторов

Чтобы понять, как конкретные стратегии синтаксического анализа влияют на общую производительность, разделим стратегии разбора на три основные группы.

1. регулярные (RE);
2. контекстно-свободные (CF);
3. контекстно-зависимые (CS).

Включаем различные стратегии синтаксического анализа во время компиляции. Из-за ограничений компилятора PetitParser некоторые стратегии синтаксического анализа не могут применяться без преобразования синтаксического анализатора в контекстно-зависимые выражения

синтаксического анализа (PE). Поэтому контекстно-зависимые выражения синтаксического анализа всегда включаются.

Мы используем контуры, приведенные на Рис. 5.

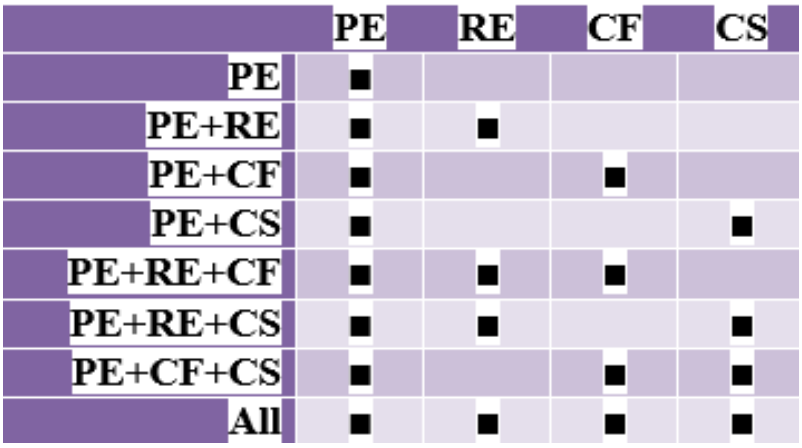

Рис. 5 – Контуры грамматик

Например, конфигурация PE преобразует комбинаторы PetitParser в контекстно-зависимые выражения синтаксического анализа, выполняя разворот цикла и конфигурация PE + RE применяет все преобразования в контекстно-зависимые выражения синтаксического анализа (PE).

Ниже мы кратко изложим контуры:

PE+RE. Если используются регулярные выражения синтаксического анализа, то в противном случае используются специализации и стратегии распознавания.

PE+CF. Используются только основанные на символах детерминированные варианты и ограничения, поскольку методы RE не применяются, и поэтому токены не могут быть обнаружены.

PE + CS. Разбор выражений анализируется с помощью push-pop-анализа. Функции push и pop используются для добавления и удаления элементов с конца массива. Для контекстно-зависимых выражений синтаксического анализа используются стандартные выражения и для контекстно-зависимых выражений используются контекстно-зависимые

выражения. Для неизвестных выражений синтаксического анализатора используются комбинаторы.

PE + RE + CF. Если используются регулярные выражения синтаксического анализа, то в противном случае используются специализации и стратегии распознавания. Используются только детерминированные варианты или ограничения.

PE + RE + CS. Если используются регулярные выражения синтаксического анализа, то используются другие специализации и стратегии распознавания. Для контекстно-зависимых выражений синтаксического анализа используются стандартные выражения и для контекстно-свободных выражений используются контекстно-зависимые объекты. Для неизвестных выражений парсинга используются комбинаторы.

PE + CF + CS. Используются только основанные на символах детерминированные варианты и ограничения, поскольку методы RE не применяются, и поэтому токены не могут быть обнаружены. Для контекстно-зависимых выражений синтаксического анализа используются стандартные «хранители», позволяющие зафиксировать и сохранить внутреннее состояние объекта. Для контекстно-свободных выражений разбора используются контекстно-зависимые объекты. Для неизвестных выражений парсинга используются комбинаторы.

All. Применяет все возможные оптимизации.

Ускорение конкретной конфигурации показано на Рис. 6. Графики иллюстрируют, насколько конкретная конфигурация влияет на общую производительность синтаксических анализаторов.

Различные стратегии оказывают сильное влияние на синтаксические анализаторы. Стратегии регулярных выражений (PE + RE), например, хорошо оптимизируют синтаксический анализатор JSON и Smalltalk, но они хуже, чем автономные выражения синтаксического анализа (PE) в случае

выражений. Это вызвано сканером, который обеспечивает лучшую производительность в сочетании с контекстно-свободными оптимизациями.

Контекстно-свободные стратегии (PE + CF) - это синтаксические анализаторы Python и Java. Контекстно-зависимые стратегии (PE + CS) существенно влияют на контекстно-зависимые синтаксические анализаторы, т.е. Анализаторы Ruby и Python. В других случаях контекстно-зависимые стратегии немного улучшают производительность из-за более эффективной мемоизации [11], т.е. сохранении результатов выполнения функций для предотвращения повторных вычислений: «хранители» являются целыми числами, а не объектами.

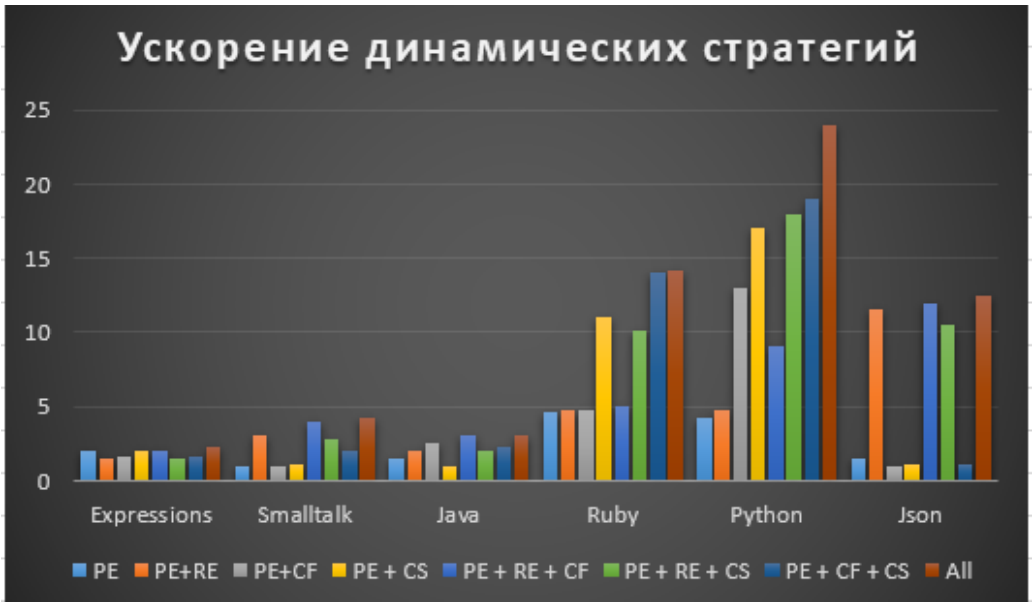

Рис. 6 – Сравнение ускорения для разных конфигураций со стандартным (простым) анализатором PetitParser

Сочетание стратегий приносит еще лучшие результаты. Регулярные и контекстно-бесплатные стратегии (PE + RE + CF) почти достигают максимальной производительности контекстно-свободных грамматик (Expressions, Smalltalk и Java), тогда как регулярные и контекстно-зависимые

(RE + CS) стратегии почти достигают максимальной производительности контекстно-зависимых грамматик (Ruby, Python).

Заключение

В этой работе мы рассмотрели идею использования адаптируемых стратегий синтаксического анализа для оптимизации производительности парсер-комбинаторов. В качестве доказательства концепции мы представили компилятор синтаксического анализатора - опережающий оптимизатор источника для PetitParser. Ускорение синтаксических анализаторов, создаваемых диапазонами компилятора синтаксического анализатора от двух до четырех для контекстно-свободных грамматик и от десяти до двадцати для контекстно-зависимых грамматик. Компилятор синтаксического анализатора не налагает никаких ограничений на основные грамматики и сохраняет преимущества и гибкость парсер-комбинаторов. Основываясь на нашем примере компилятор синтаксического анализатора обеспечивает лучшую производительность, чем синтаксический анализ, управляемый таблицами, и хуже, чем ручной оптимизированный анализатор, который использует распознавание.

Работа выполнена при поддержке РФФИ, проекты 18-08-00549 А, 17-07-00620 А.

Литература

1. Розин М.Д., Свечкарев В.П. Анализ принципов формирования общедоступного интернет пространства в сфере инженерной деятельности // Инженерный вестник Дона, 2018 №1. URL: ivdon.ru/ru/magazine/archive/n1y2018/4750

2. Харламов А.А., Ермоленко Т.В., Дорохина Г.В. Сравнительный анализ организации систем синтаксических парсеров // Инженерный вестник Дона, 2013, №4. URL: ivdon.ru/ru/magazine/archive/n4y2013/2015

3. Redziejowski R. R. Applying classical concepts to parsing expression grammar. Fundamentica Informaticae, 93(1-3), pp. 325–336, 2009.

4. Kurs J., Larcheveque G., Renggli L., Bergel A., Cassou D., Ducasse S., and Laval J. In Deep Into Pharo, page 36. Square Bracket Associates, Sept. 2013.

5. Kursa Jan, Vrany Jan, Ghafari Mohammad, Lungu Mircea, Nierstrasz Oscar. Science of Computer Programming. Volume 161, 1 September 2018, Pages 57-88, DOI:10.1016/j.scico.2017.12.001

6. Fabio Mascarenhas R. I., Medeiros Sergio CoRR, abs/1304.3177, pp. 235-250, 2013.

7. Kurs J., Lungu M., and Nierstrasz O. In Proceedings of International Workshop on Smalltalk Technologies. (IWST 2014), pp.127-133, 2014.

8. Moonen L. Proceedings Eighth Working Conference on Reverse Engineering (WCRE 2001), pages 13-22. IEEE Computer Society, Oct. 2001.

9. Kurs J., Vrany J., Ghafari M., Lungu M., and Nierstrasz O. In Proceedings of International Workshop on Smalltalk Technologies (IWST 2016), pages 1:1-1:13, 2016.

10. Black A., Ducasse S., Nierstrasz O., Pollet D., Cassou D., and Denker M. Square Bracket Associates, 2009, 333p.

11. Frost R. A. and Szydlowski B. Memoizing purely functional top-down backtracking language processors. Science of Computer Programming, 27(3), pp. 263-288, Nov. 1996.

References

1. Rozin M.D., Svechkarev V.P. Inženernyj vestnik Dona (Rus), 2018, №1. URL: ivdon.ru/ru/magazine/archive/n1y2018/4750

2. Harlamov A.A., Ermolenko T.V., Dorohina G.V. Inženernyj vestnik Dona (Rus), 2013, №4. URL:.ru/ru/magazine/archive/n4y2013/2015

3. Redziejowski R. R. Fundamentica Informaticae, 93(1-3), pp. 325–336, 2009.

4. Kurs J., Larcheveque G., Renggli L., Bergel A., Cassou D., Ducasse S.,

and Laval J. PetitParser: Building modular parsers. In Deep into Pharo, page 36. Square Bracket Associates, Sept. 2013.

5. Kursa Jan, Vrany Jan, Ghafari Mohammad, Lungu Mircea, Nierstrasz Oscar. Science of Computer Programming. Volume 161, 1 September 2018, Pages 57-88, DOI:10.1016/j.scico.2017.12.001

6. Fabio Mascarenhas R. I., Medeiros Sergio. On the relation between context free grammars and parsing expression grammars. CoRR, abs/1304.3177, pp. 235-250, 2013.

7. Kurs J., Lungu M., and Nierstrasz O. Top-down parsing with parsing contexts. In Proceedings of International Workshop on Smalltalk Technologies (IWST 2014), pp.127-133, 2014.

8. Moonen L. Generating robust parsers using island grammars. In E. Burd, 1090 P. Aiken, and R. Koschke, editors, Proceedings Eighth Working Conference on Reverse Engineering (WCRE 2001), pages 13-22. IEEE Computer Society, Oct. 2001.

9. Kurs J., Vrany J., Ghafari M., Lungu M., and Nierstrasz O. Optimizing parser combinators. In Proceedings of International Workshop on Smalltalk Technologies (IWST 2016), pages 1:1-1:13, 2016.

10. Black A., Ducasse S., Nierstrasz O., Pollet D., Cassou D., and Denker M. Pharo by Example. Square Bracket Associates, 2009, 333p.

11. Frost R. A. and Szydlowski B. Science of Computer Programming, 27(3), pp. 263-288, Nov. 1996.