Санкт-Петербургский государственный университет

Кафедра системного программирования

Мелентьев Кирилл Игоревич

Реализация библиотеки парсер-комбинаторов на основе алгоритма для платформы .NET

Курсовая работа

Научный руководитель:
ст. преп., магистр информационных технологий Григорьев С.В.

Санкт-Петербург
2016

Оглавление

Введение

Традиционно, для создания синтаксических анализаторов используются генераторы синтаксических анализаторов, при этом используется особый язык спецификации грамматики, а семантические действия, как правило, описываются на другом языке программирования. Известным примером является Yacc [2] — генератор синтаксических анализаторов на языке C (существует множество клонов уасс, генерирующих синтакисческие анализаторы на других языках). Другой подход — парсер-комбинаторы — подход, при котором, парсер строится динамически, из простейших парсеров с использованием функций комбинирующих их. Парсер-комбинаторы позволяют реализовать синтаксический анализатор и семантические действия на одном языке, на котором пишется дальнейшая обработка данных. Более того, этот подход обладает другими преимуществами: естественность и понятность описания анализаторов, а также простота отладки, обеспечиваемая свойством модульности.

Однако многие библиотеки парсер-комбинаторов, (например классическая библиотека парсер-комбинаторов Parsec [3] для Haskell, или библиотека FParsec [7] для F#), наследуя свойство нисходящих синтаксических анализаторов, не поддерживают леворекурсивные правила грамматик, или испытывают проблемы с производительностью, при использовании неоднозначных грамматик. Многие стандартные грамматики содержат леворекурсивные правила, например грамматика арифметических выражений, выражений с постфиксными операциями – вызовами функций, обращениями к элементам массива по индексу, обращения к полям структуры/объекта. В общем случае, если парсер-комбинаторы не поддерживают леворекурсивные правила, то придется модифицировать грамматику, удалив леворекурсивные правила, а так же изменить семантические действия. Необходимость этих действий может усложнить разработку, увеличить время разработки. В данной работе мы рассмотрим реализацию одного из решений этой проблемы под платформу .NET.

1. Обзор существующих решений

В данной курсовой работе были рассмотрены возможности и свойства, следующих популярных библиотек парсер-комбинаторов: FParsec (для F#), Attoparsec [4] (для Haskell), gll-combinators [6] (для Scala). Каждая из библиотек обладает различными особенностями, среди которых были выделены следующие характерные возможности: построение синтаксического анализаторы на основе грамматики, содержащей леворекурсивные правила; поддержка работы с абстрактным типом входных данных (многие библиотеки парсер комбинаторов позволяют работать только с потоком символов); Возможность создания инкрементальных синтаксических анализаторов.

1.1. Леворекурсивные правила

Различные библиотеки парсер-комбинаторов предоставляют свои решения проблемы с леворекурсивными правилами, например FParsec имеет специальный интерфейс OperatorPrecedenceParser для создания парсеров арифметических выражений с инфиксными, префиксными и постфиксными операторами. Но непосредственно работать с леворекурсивными правилами (среди рассмотренных решений) умеет только библиотека gll-combinators (Scala), которая и была взята за основу нашего решения.

1.2. Абстрактный тип входа

Возможность работы с абстрактным типом входных данных расширяет набор возможных способов использования библиотеки, например символы входного потока могут быть дополнены какой-либо важной для анализатора мета-информацией. Также, в каких-то случаях, может быть удобно работать с потоком лексем поступающих от лексического анализатора — во-первых синтаксический анализатор упрощается благодаря тому, что лексический анализатор сам обрабатывает и фильтрует такие конструкции, как пробелы и комментарии в исходном коде;

во-вторых, возможно повышение производительности, так как лексический анализатор обычно работает быстро благодаря более простой конструкции, и длина входного потока, поступающего синтаксическому анализаторы в разы меньше, чем длина текста в символах.

1.3. Инкрементальный анализ

Библиотека Attoparsec позволяет создавать инкрементальные лексические анализаторы. Работают анализаторы Attoparsec так: когда анализатор встречает конец входного потока (при этом ожидая какого-либо символа) он остановливается и возвращает новый парсер, который можно применить к новому входному потоку так, будто бы это продолжение предыдущего. Например, пусть мы сконктруировали парсер X для арифметических выражений:

```
E ::= E ('+' | '-') T | T
T ::= T ('*' | '/') F | F
F ::= num | '(' E ')'
```

Применив X к строке "2+(", мы получим парсер Y. Применив парсер Y к строке "3-4)", мы получим итоговый результат для строки "2+(3-4)".

Такой подход может быть удобен в случаях, когда входной поток не доступен весь в один момент, а поступает частями, например при получении данных по сети — пакеты могут быть обработны последовательно, нет необходимости собирать весь входной поток в один буффер. Другим примером использования может быть анализ исходного кода в IDE (для подсветки синтаксических конструкций): например можно сохранять промежуточные результаты парсинга для каждой n-ой позиции, и когда пользователь редактирует код в одном месте, не нужно заново анализировать весь файл – можно продолжить парсинг от ближайшего сохраненного парсера (придется откинуть следующие за ним).

1.4. gll-combinators

Рассмотрим, каким образом библиотека gll-combinators справляется с произвольными КС-грамматиками. В основе реализации этой библиотеки лежит алгоритм GLL, разработанный в 2009 учеными Elizabeth Scott и Andrian Johnstone [5]. GLL – это разновидность нисходящего рекурсивного спуска, благодаря чему он может быть адаптирован для использования комбинаторами парсеров. Алгоритм имеет асимптотику $O(n^3)$ в худшем случае, в отличие от обыкновенного рекурсивного спуска с возвратом, который в худшем случае имеет экспоненциальную сложность.

Daniel Spiewak, автор библиотеки gll-combinators, адаптировал GLL под подход парсер-комбинаторов и разработал реализацию на Scala, на которой и основывается наша работа.

2. Постановка задачи

Целью данной курсовой работы работы является разработка библиотеки парсер-комбинаторов, поддерживающих произвольные контекстно свободные грамматики, для платформы .NET. Для достижения данной цели поставлены следующие задачи.

- Разработать библиотеку парсер-комбинаторов со следующими свойствами и возможностями:
  - платформа реализации .NET;
  - возможность создания синатксических анализаторов на основе произвольных контекстно-свободных грамматик;
  - поддержка работы с абстрактным типом входных данных;
  - возможность создания инкрементального синтаксических анализаторов.
- Провести сравнение производительности с существующими решениями.

3. Реализация

В ходе курсовой работы была данная библиотека была реализована (далее FsGll). Приводятся некоторые детали ее реализации и использования.

3.1. Детали реализации

Реализация базовой части библиотеки повторяет реализацию описанную Daniel Spiewak, за исключением некоторых технических моментов, связанных с использованием языка F#, а также с использованием произвольного типа входа.

Реализация инкрементальной версии потребовала небольших модификаций алгоритма. Во-первых, были заменены структуры данных, основанные на хэш-таблицах, на структуры данных, основанные на деревьях поиска (множества и ассоциативные массивы). Так же все функции, модифицировавшие разделяемое состояние, стали возвращать новый объект состояния. Суть модификации непосредственно алгоритма в следующем: когда анализатор возвращает ошибку конца входного потока, то вместо того что бы вызвать для этой ошибки функцию обрабатывающую результат, мы добавляем этот анализатор и функцию обработки в список (хранящийся в объекте, инкапсулирующем состояние процесса синтаксического анализа) для «перезапуска» при запуске на следующей части входного потока. В конце процесса (когда все пути анализа пройдены) проверяется, не пуст ли данный список «перезапуска», и если он не пуст, то в список результатов добавляется PartialParser, сохранивший этот список.

3.2. Интерфейс библиотеки

Интерфейс библиотеки FsGll предоставляет типы и функции в нескольких пространствах имен:

- `FsGll.Parsers` — базовая реализация на основе изменяемых структур данных

- `FsGll.Parsers.Incremental` — реализация парсер-комбинаторов для инкрементального анализа на основе изменяемых структур
- `FsGll.Parsers.Incremental.Pure` — реализация парсер-комбинаторов для инкрементального анализа на основе неизменяемых структур

Все эти пространства имен предоставляют тип `Parser<'a,'r>`, экземплярами которого являются парсеры.

- `'a` — тип возвращаемого парсером значения
- `'r` — тип элементов входного потока

FsGll.Parsers предоставляет тип `Result<'a,'r>` для описания результата парсинга.

```
type Result<'a,'r> = Success of 'a
                   | Failure of string * Stream<'r>
```

Комбинаторы и функции:

- `runParser<'a, 'r>: Stream<'r> -> [Result<'a,'r>]` — Функция предназначена для запуска анализатора на входном потоке. Она возвращает список всех успешных результатов анализа (поскольку на неоднозначной грамматике их может быть много), либо список всех ошибок.
- `satisfy` — принимает или отвергает элемент входного потока
- `<|>` — альтернатива
- `>>=` — связывание
- `createParserForwardedToRef` — аналог соответсвующей функции из пакета FParsec [8]

(приведены только функции используемые в примере, полный список на странице документации [1])

Пространства имен `FsGll.Parsers.Incremental` и `FsGll.Parsers.Incremental.Pure` расширяют тип `Result`, добавляя конструктор `Partial` (соответственно функции `runParser` в этих пространствах имен также могут в списке результатов вернуть `Partial`):

```
type Result<'a,'r> = Success of 'a
                   | Failure of string * Stream<'r>
                   | Partial of Parser<'a,'r>
```

Далее приведен пример использования комбинаторов из `FsGll.Parsers.Incremental.Pure` (интерфейсы парсер-комбинаторов из других пространст имен аналогичны за исключением отсутсвия `Partial` результата в `FsGll.Parsers`):

```
// utilities
let isPartial = function Partial(_) -> true
                       | _          -> false
let getPartial = List.tryFind isPartial
let parse p s = runParser<_,_> p (stringStream s)

// construct parser
let e, er = createParserForwardedToRef<_>()
let t, tr = createParserForwardedToRef<_>()
let chr c = satisfy ((=)c)
let op = chr '+' <|> chr '-'
     <|> chr '*' <|> chr '/'
let num = satisfy (Char.IsDigit) |>> Int32.Parse
let f = num <|> (chr '(' >>. e .>> chr')')

let sem l x r =
    match x with
    | '+' -> l + r
    | '-' -> l - r
    | '*' -> l + r
```

```
    | '/' -> l - r

er := pipe3 e op t sem <|> t
tr := pipe3 t op f sem <|> f

// run
let f = parse e ''1+2*'' |> getPartial
let g = parse f ''(3'' |> getPartial
let h1 = parse f ''3+4''
let h2 = parse g ''+4)''
```

В данном примере конструируется синтаксический анализатор для грамматики арифметических выражений с односимвольными операнадми и операциями (+, -, *, /):

E ::= E ('+' | '-') T | T
T ::= T ('*' | '/') F | F
F ::= num | '(' E ')'

Анализатор применяется к строке ''1+2*'', которая не является принадлежащей языку, (но может быть дополнена до корректной), после чего из списка результатов извлекается новый анализатор f, который применяется к продолжению строки (''(3'') для получения следующего "промежуточного" анализатора g. Анализатор f также применяется к ''3+4'', после чего в списке результатов h1 содержится корректный результат 11. Анализатор g применяется к ''+4)'' , и возвращает корректный результат 15.

4. Замеры производительности

Тестирование проводилось на компьютере со следующими характеристиками:

- CPU – Intel Core i5-5300U
- RAM – 12 Гб

Каждый тест запускался отдельным процессом, каждый раз предварительно «разогревая» runtime систему коротким тестом.

Первое сравнение проводилось между FsGll и gll-combinators на следующей сильно неоднозначной грамматике, содержащей леворекурсивные правила:

```
S ::= S S S
    | S S
    | '0'
```

График производительности представлен на изображении (рис. 4). Видно, что производительность обоих решений на данной грамматике сопоставима.

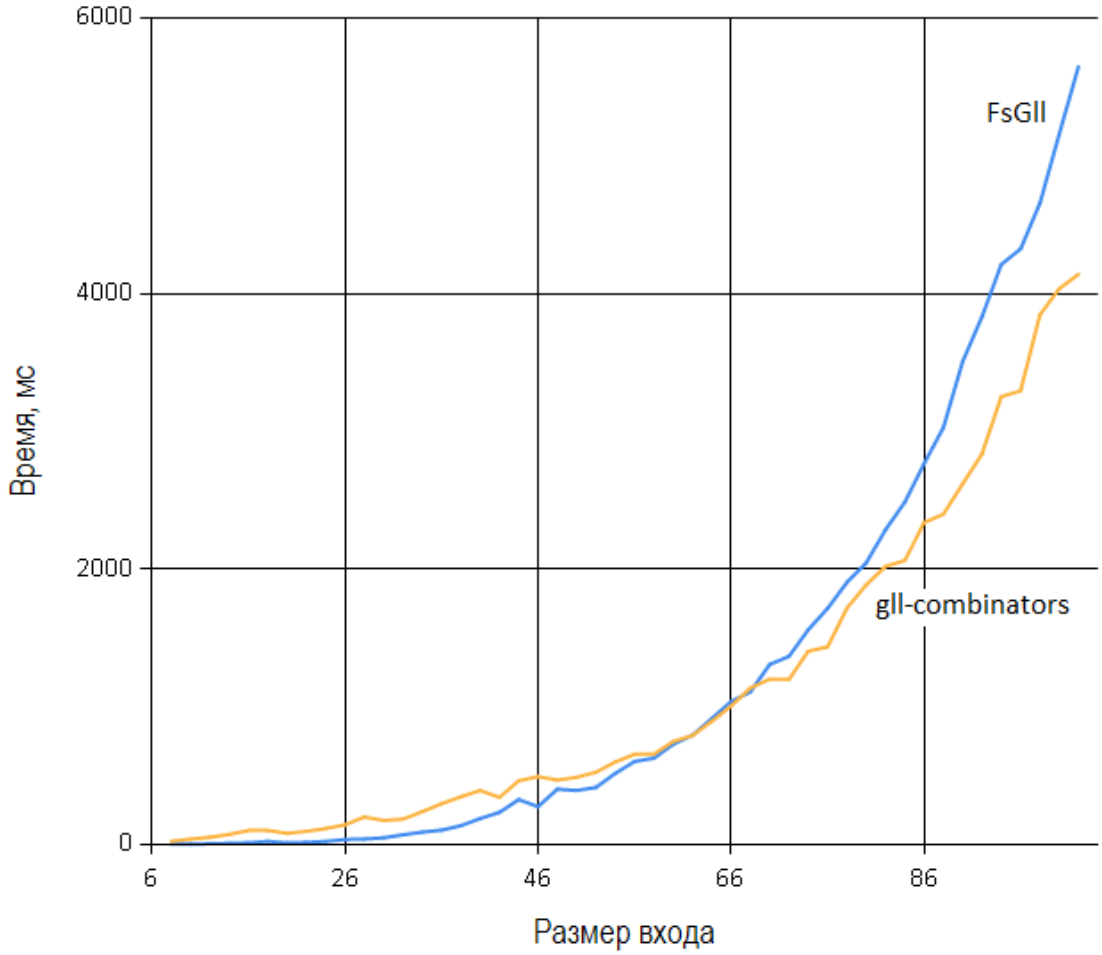

Рис. 1: Сравнение производительности FsGll и gll-combinators на сильно неоднозначной грамматике

Следующий тест сравнивает производительность FsGll и FParsec на грамматике арифметических выражений:

```
E ::= E ‘’(+ | ’‘−) T | T
T ::= T ‘’(* | ‘’/) F | F
F ::= ident | value | ’(’ E ’)’
Stmt ::= ident ’=’ Expr ’;’
P ::= Stmt* E
```

График производительности представлен на изображении (рис. 4). Здесь ”Размер входа” — это количество операндов в выражении.

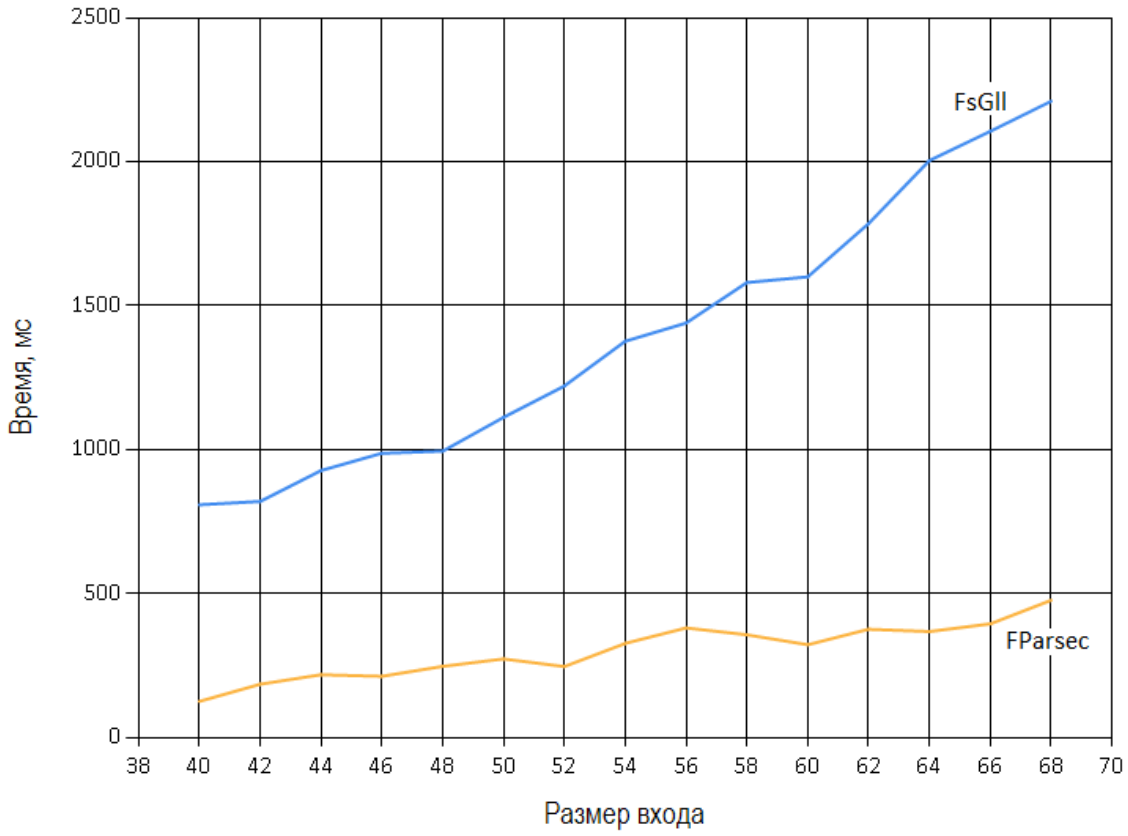

Рис. 2: Сравнение производительности FParsec и FsGll на грамматике арифметических выражений

По графику видно, что по производительности на данном примере FParsec значительно превосходит FsGll. FParsec — промышленная библиотека, довольно хорошо работает на стандартных грамматиках.
Последний тест сравнивает производительность реализаций, предназначенных для создания инкрементальных парсеров. Грамматика вновь:

S ::= S S S
| S S
| '0'

Символы подаются на вход по одному. График производительности представлен на изображении (рис. 4).

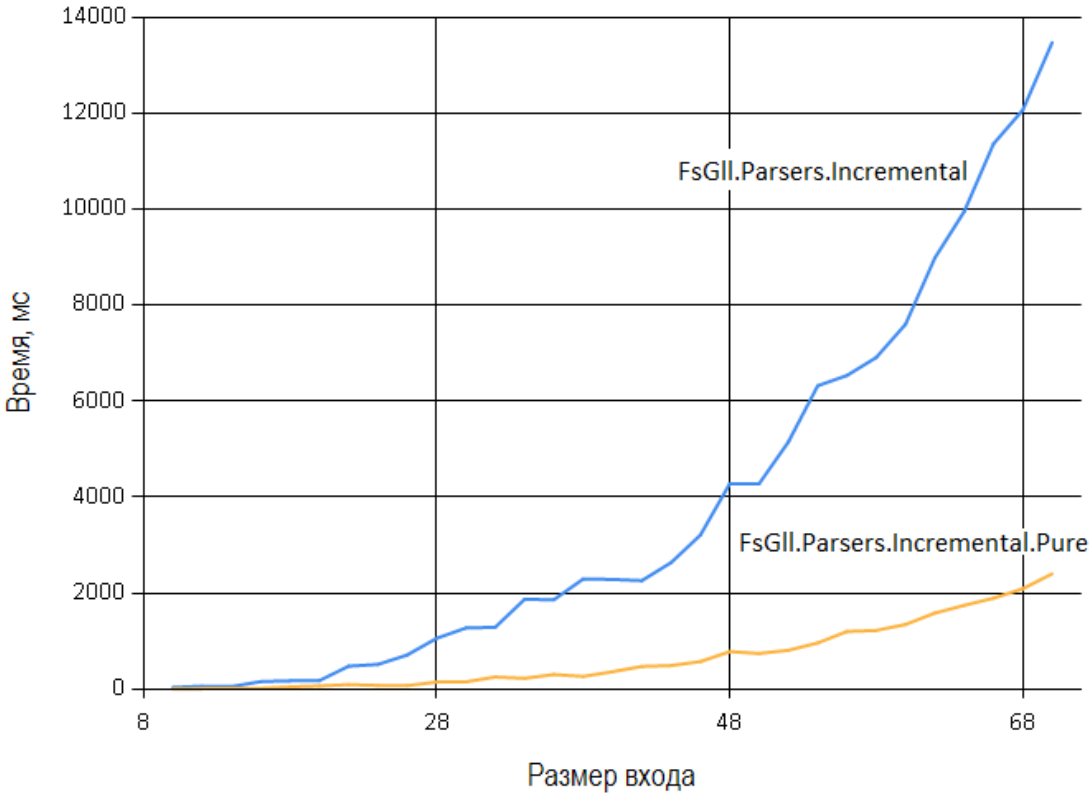

Рис. 3: Сравнение производительности FsGll.Parsers.Incremental и FsGll.Parsers.Incremental.Pure на сильно неоднозначной

Производительность решения реализованного на основе неизменяемых структур данных ожидаемо ниже, в среднем в 6 раз.

Заключение

В рамках данной курсовой работы была реализована библиотека парсер-комбинаторов для платформы .NET, FsGll, обладающая следующими возможностями:

- возможность создания синатксических анализаторов на основе произвольных КС грамматик
- поддержка работы с абстрактным типом входных данных
- возможность создания инкрементального синтаксических анализаторов.

Библиотека сочетает в себе возможности различных существующих библиотек, благодаря чему является довольно универсальным решением.

Также было проведено сравнение производительности с другими известными библиотеками парсер-комбинаторов. Производительность данного решения сравнима со своим прообразом (gll-combinators), но проигрывает другим библиотекам, при использовании сходных грамматик.

Доклад по данной реализации был успешно представлен на конференции «Современные технологии в теории и практике программирования» в СПбПУ.

Исходный код библиотеки доступен по ссылке: https://github.com/YaccConstructor/FsGll

Документация: http://yaccconstructor.github.io/FsGll/

NuGet пакет: https://www.nuget.org/packages/FsGll/

Список литературы

[1] FsGll - documentation. — URL: http://yaccconstructor.github.io/FsGll/.

[2] Johnson Stephen C. Yacc. — URL: http://dinosaur.compilertools.net/yacc/.

[3] Parsec: Direct Style Monadic Parser Combinators for the Real World : Rep. : UU-CS-2001-27 / Department of Computer Science, Universiteit Utrecht ; Executor: Daan Leijen, Erik Meijer : 2001.

[4] O'Sullivan Bryan. Attoparsec. — URL: https://hackage.haskell.org/package/attoparsec.

[5] Scott Elizabeth, Johnstone Adrian. GLL Parsing // Preliminary Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications. — 2009. — P. 113–126.

[6] Spiewak Daniel. Generalized Parser Combinators. — 2010. — URL: https://github.com/djspiewak/gll-combinators.

[7] Tolksdorf Stephan. FParsec. — URL: http://www.quanttec.com/fparsec/.

[8] Tolksdorf Stephan. FParsec:createParserForwardedToRef. — URL: http://www.quanttec.com/fparsec/reference/primitives.html#createParserForwardedToRef.