

# Функциональное программирование

КОМБИНАТОРНЫЕ ПАРСЕРЫ ДЛЯ ПРОСТЫХ СМЕРТНЫХ

## Немного о проекте

---

1. Распределенный API: по 22 сервера в 2-х датацентрах (Америка, Европа).
2. Разнообразные клиентские приложения: 40 desktop и web приложений.
3. Датацентр обрабатывает 12 000 запросов в минуту.
4. Ресурсоёмкие запросы и пакетные запросы.

# С чего все началось?

---

Возникла необходимость добавить новый источник данных HBase.

## Что такое HBase?

---

- NoSql - <ключ значение>;
- написана на Java;
- аналог Google Big table;
- не является заменой SQL;
- интерфейсы взаимодействия: REST, Java API, Apache THRIFT.

# AVRO

---

1. Apache Avro – система сериализации данных.
2. Система использует [JSON](#) для определения структуры данных (схемы), которые сериализуются в компактный бинарный формат.

# AVRO – схема

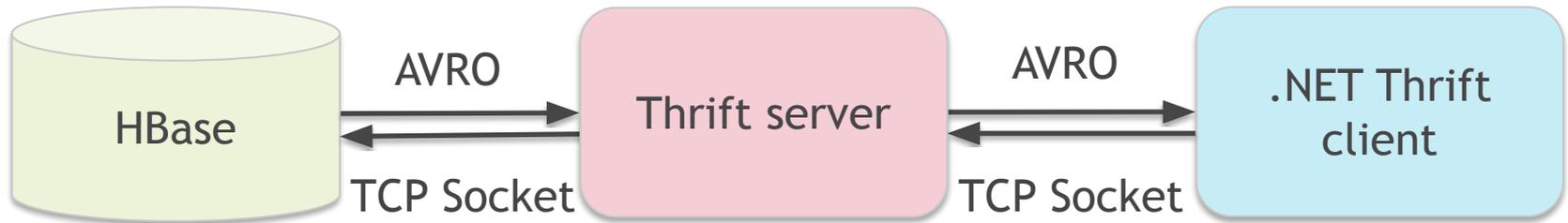
```
{
  "namespace": "com.acme",
  "protocol": "HelloWorld",
  "doc": "Protocol Greetings",

  "types": [
    {"name": "Greeting", "type": "record", "fields": [
      {"name": "message", "type": "string"}]},
    {"name": "Curse", "type": "error", "fields": [
      {"name": "message", "type": "string"}]}
  ],

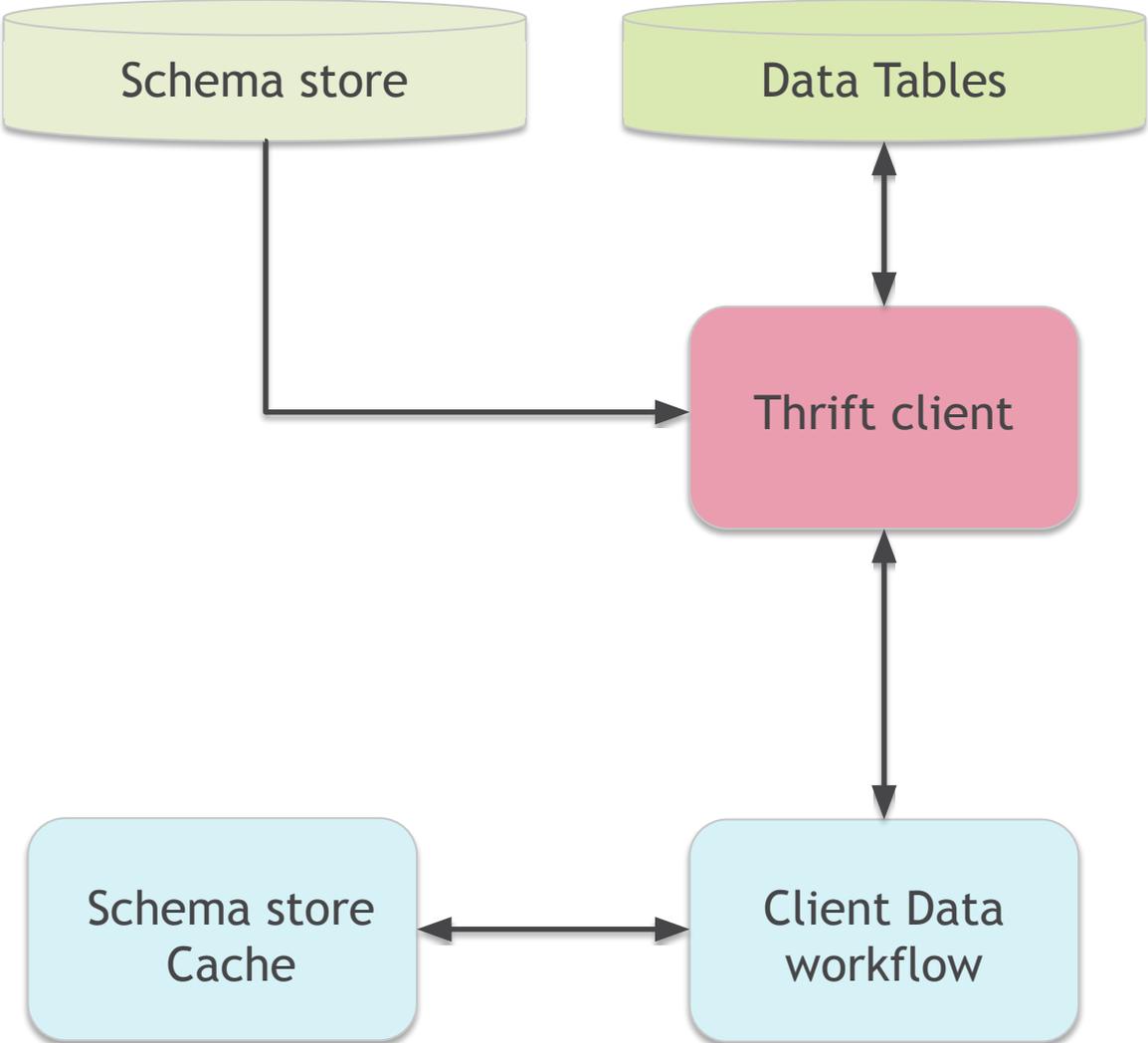
  "messages": {
    "hello": {
      "doc": "say hello.",
      "request": [{"name": "greeting", "type": "Greeting" }],
      "response": "Greeting",
      "errors": ["Curse"]
    }
  }
}
```

# HBase + Thrift-AVRO + .NET = ?

---



# Workflow



# Требования

---

1. МЕТА-DRIVEN;
2. результат - простая таблица;
3. возможность выполнять Map/Reduce;
4. сохранить отношения данных;

# Проблемы

---

- большие вложенные AVRO схемы до 1 000 000 строк;
- AVRO не дружит с .NET;
- после десериализации AVRO Dictionary<string, object>.

Что делать?

---

**А давайте напишем свой DSL!**

# Разработка синтаксиса

```
query(view: "TABLENAME") {  
  
  //simple selector  
  Selector  
  
  //nested-field selector  
  Selector.Next  
  
  Selector.Next.Value.ID  
  Selector.Next.Value.InnerValue  
  
  //nested-object selector  
  Selector.Next.Value {  
    ID  
    InnerValue  
  }  
  
  //nested-array selector  
  Selector.Next.Values((ID >= 1 AND ID <= 100) AND (InnerValue = "Hello" OR InnerValue="World")) [  
    ID  
    InnerValue  
  ]  
}
```

## А давай еще **Join's**

```
query(view: "first_table") {
  //ID Selector
  Selector.ID

  //other selectors
  Value
} with(view: "second_table" fetchkeys: first_table:Selector.ID) {
  //ID Selector
  Target.ID

  //other selectors
  Values.Next
} leftjoin (

  first_table:Selector.ID = second_table:Target.ID

) with(view: "third_table" fetchkeys: first_table:Selector.ID second_table:Target.ID) {
  //ID Selector
  Company.ID
  Apple.Value

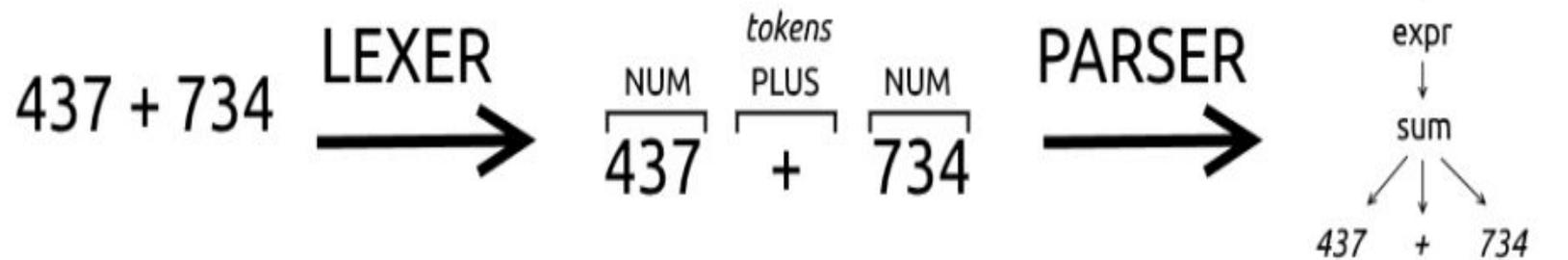
  //other selectors
  Values.Next
} leftjoin (
  first_table:Selector.ID = third_table:Company.ID
  AND
  second_table:Target.ID = third_table:Apple.Value
)
```

А как быстро написать парсер?

---

# Парсинг(синтаксический анализ текста)

## ОБЩИЙ ПОДХОД



# Выбор

---

1. Парсер генератор на основе формальных языков (ANTLR).
2. Подключаемая библиотека комбинаторных парсеров (Sprache, SuperPower).
3. ...

Написать руками



## ANTLR (Another Tool for Language Recognition)

```
grammar Expr;
prog:   (expr NEWLINE)* ;
expr:  expr ('*' | '/' )
      |  expr ('+' | '-' )
      |  expr
      |  INT
      |  '(' expr ')'
      ;
NEWLINE : [\r\n]+ ;
INT      : [0-9]+ ;
```

Расширенная форма Бэкуса — Наура

# Генератор на основе формальных языков

## Плюсы

- декларативный синтаксис;
- строгое соблюдение грамматики;
- не нужно думать о performance;
- не нужно писать документацию.

## Недостатки

- строгие грамматики;
- интеграция с системой сборки;
- тяжело отлаживать;
- кастомные ошибки;
- медленные парсеры;
- проблемы с кодировками;
- проблемы переносимости.

# Комбинаторные парсеры (**Sprache** и т.д.)

## Плюсы

- реализован как библиотека языка;
- модульный и поддерживаемый;
- полуавтоматическая генерация сообщений об ошибках;
- backtracking и look ahead;
- возможности в runtime;
- не требует предварительной токенизации(лексера).

## Недостатки

- компромисс между декларативностью и производительностью;
- проблемы с левой рекурсией;
- придется учить API;
- только для вашего языка.
- может не иметь предварительной токенизации.

# Рукописные парсеры

## Плюсы

- никакого внешнего кода;
- поддается к индивидуальным требованиям;
- потенциально быстр, как только это возможно.

## Недостатки

- все писать с 0;
- создание быстрых парсеров требует опыта;
- выражения с инфиксными операторами(приоритеты).

Причем тут функциональное прог-ие?

---

## Основные концепции:

- неизменяемые структуры данных;
- чистые функции;
- функции высших порядков;
- рекурсия.

# Императивное vs Функциональное

```
List<int> values = new List<int>();  
  
for (int i = 0; i < 42; i++)  
    values.Add(i);  
  
List<int> evenNums = new List<int>();  
foreach (var value in values)  
{  
    if (value % 2 == 0)  
        evenNums.Add(value);  
}  
  
return evenNums;
```

```
var evenNums = Enumerable.Range(0, 42)  
    .Where(num => num % 2 == 0)  
    .ToList();
```

```
let evenNums = [0..42]  
    |> List.where(fun num -> num % 2 = 0)
```

Don't panic it's monadic!

---

1. Написать примитивные парсеры(функции).
2. Написать функции для комбинирования.
3. Скомбинировать простые парсеры в более сложные.
4. PROFIT!

# Сигнатура нашего парсера

---

```
type Parser = String -> Tree
```

```
type Parser = String -> (String, Tree)
```

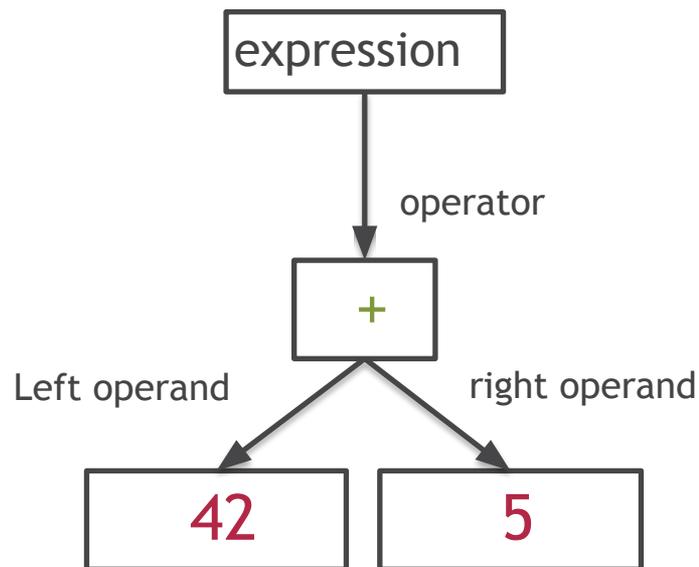
```
type Parser<T> = String -> (String, T)
```

```
type Parser<TInput, T> = TInput -> (TInput, T)
```

# Пишем простой комбинаторный парсер

Выражение : “42 + 5”

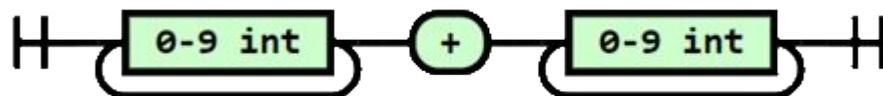
42 + 5



# Парсим простое выражение : “42 + 5”

```
Int ::= [0-9]+  
AddExpr ::= Int '+' Int
```

Грамматика в БНФ



Railroad - диаграмма

# Библиотеки с готовым набором комбинаторных парсеров для C#

## Sprache

<https://github.com/sprache/Sprache>

- небольшая библиотека, совместима .NET Core;
- простой, но богатый API;
- много используется в “боевых проектах”: R#, Octostache, EasyNetQ, Seq.



## Superpower

<https://github.com/datalust/superpower>

- форк Sprache с блэк джеком и ...;
- использует токенизатор;
- парсит потоки токенов, а не поток символов;
- хорошие сообщения об ошибках, легко кастомизируются;
- работает быстрее.



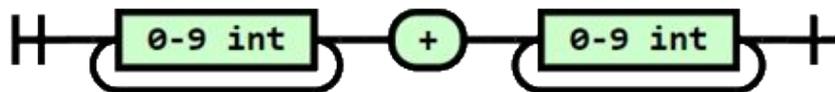
## Готовый код парсера для “42 + 5”

```
public static class CalcParserExample
{
    private static readonly Parser<ConstantExpression> Constant =
        from number in Parse.Number.Token()
        select Expression.Constant(int.Parse(number));

    private static readonly Parser<char> AddOperator =
        from addOperator in Parse.Char('+').Token()
        select addOperator;

    private static readonly Parser<BinaryExpression> SumExpression =
        from lop in Constant
        from addOperator in AddOperator
        from rop in Constant
        select Expression.Add(lop, rop);

    public static Expression<Func<int>> ParseExpression(string input)
    {
        var body = SumExpression.Parse(input);
        return Expression.Lambda<Func<int>>(body);
    }
}
```



## Ну, а как же без TDD?

```
[TestFixture]
internal class ParserTests
{
    [Test]
    public void CalculatorTest()
    {
        string input = "42 + 5";

        Expression<Func<int>> expression = CalcParserExample.ParseExpression(input);
        int result = expression.Compile>();

        Assert.AreEqual(47, result);
    }

    [Test]
    public void CalculatorFailTest()
    {
        Assert.Catch<Exception>(() =>
        {
            string input = "f + 14";

            Expression<Func<int>> expression = CalcParserExample.ParseExpression(input);
            int result = expression.Compile>();
        }, "unexpected f, expected: Numeric character");
    }
}
```

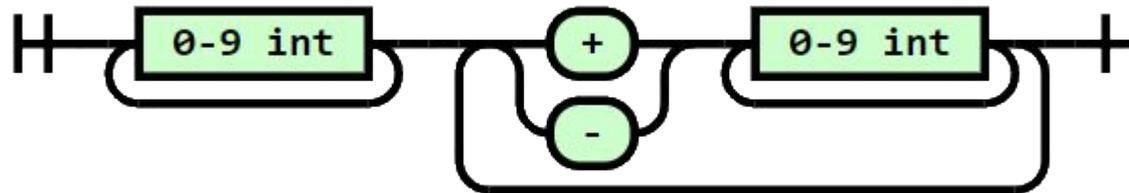
Давайте усложним : “42 + 5 - 7”

Int ::= [0-9]+

Operator ::= '+' | '-'

Expr ::= Expr Operator Expr | Int

Грамматика в БНФ



Railroad - диаграмма

# Парсер на Sprache “42 + 5 - 7”

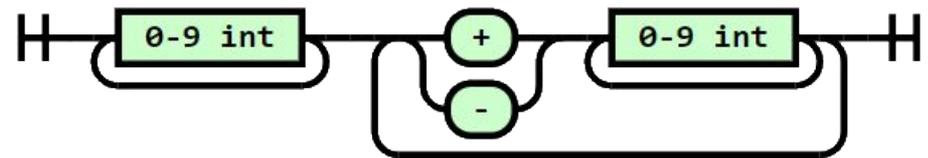
```
public static class SpracheCalcParser
{
    private static Parser<ExpressionType> CreateOperator(string op, ExpressionType type)
    {
        return Parse.String(op).Token().Return(type);
    }

    private static readonly Parser<ExpressionType> Add = CreateOperator("+", ExpressionType.Add);
    private static readonly Parser<ExpressionType> Subtract = CreateOperator("-", ExpressionType.Subtract);

    private static readonly Parser<Expression> Operand =
        from number in Parse.Number.Token()
        select Expression.Constant(int.Parse(number));

    private static readonly Parser<Expression> MainExpression =
        Parse.ChainOperator(
            Add.Or(Subtract),
            Operand,
            Expression.MakeBinary);

    public static Expression<Func<int>> ParseExpression(string input)
    {
        var body = MainExpression.Parse(input);
        return Expression.Lambda<Func<int>>(body);
    }
}
```



# Когда и зачем писать свои парсеры?

---

- DSL (Domain-specific language) Пример: XPath, SQL;
- использование NoSQL;
- не хватает всей “Мощи” XML;
- разработка IDE или плагинов к ним;
- клиентские приложения, например, поиск;
- анализ документов.

# Что внутри?

---

```
public delegate Result<TValue> Parser<TValue>(Input input);
```

```
public static class ParserExtensions
{
    public static T Parse<T>(this Parser<T> parser, string input)
    {
        var result = parser(new Input(input));

        if (result.WasSuccessful)
            return result.Value;

        throw new Exception(result.Message);
    }
}
```

## Тип Result<T>

```
public struct Result<T>
{
    public T Value { get; }
    public bool WasSuccessful { get; }
    public string Message { get; }
    public Input Remainder { get; }

    public Result(string errorMessage)
    {
        Value = default(T);
        Remainder = default(Input);

        Message = errorMessage;
        WasSuccessful = false;
    }

    public Result(T value, Input remainder, bool wasSuccessful, string errorMessage)
    {
        Value = value;
        WasSuccessful = wasSuccessful;
        Message = errorMessage;
        Remainder = remainder;
    }

    public static Result<T> Success(T inputCurrent, Input remainder)
        => new Result<T>(inputCurrent, remainder, true, string.Empty);

    public static Result<T> Failure(string errorMessage) => new Result<T>(errorMessage);
}
```

## Тип Input (обертка над string)

```
public struct Input
{
    public string Source { get; }
    public int Position { get; }

    public char Current => Source[Position];
    public bool AtEnd => Position == Source.Length;

    public Input(string source)
    {
        Source = source;
        Position = 0;
    }

    public Input(string source, int position)
    {
        Source = source;
        Position = position;
    }

    public Input GetRemainder() => new Input(Source, Position + 1);
}
```

# Начнем с простых парсеров

```
public static class Parse
{
    // predicate: (current == "a")
    // input: "abc" -> result (success, value: "a", remainder: "bc")
    public static Parser<char> Char(Func<char, bool> predicate, string description)
    {
        return i =>
        {
            if (!i.AtEnd)
            {
                return predicate(i.Current)
                    ? Result<char>.Success(i.Current, i.GetRemainder())
                    : Result<char>.Failure($"unexpected {i.Current}, expected: {description}");
            }

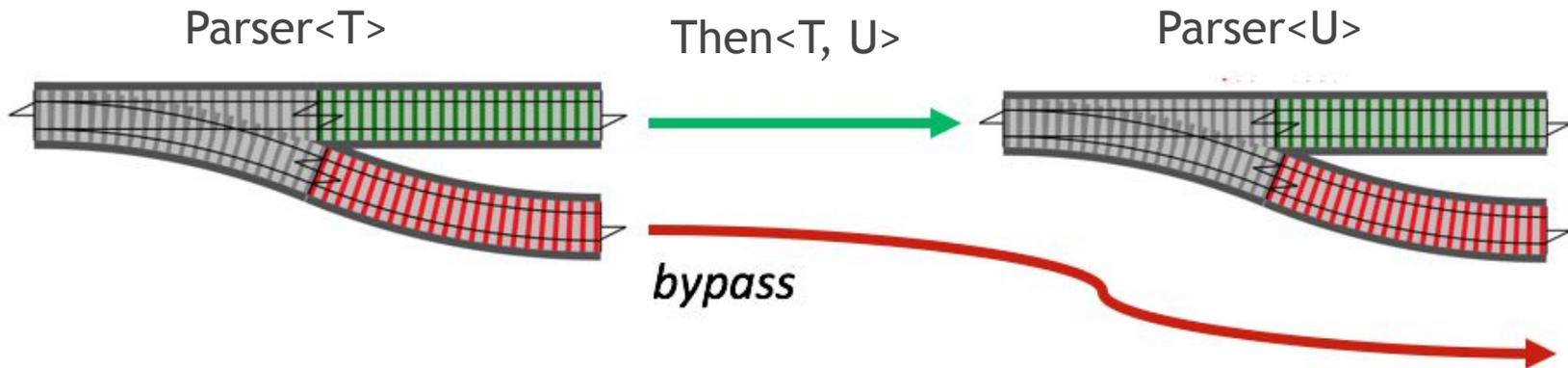
            return Result<char>.Failure("Unexpected end of string");
        };
    }

    // target: 'a'
    // input: "abc" -> result (success, value: "a", remainder: "bc")
    public static Parser<char> Char(char target)
    {
        return Char(current => current == target, target.ToString());
    }

    // input: " abc" -> result (success, value: " ", remainder: "abc")
    public static readonly Parser<char> WhiteSpace = Char(char.IsWhiteSpace, "WhiteSpace");
    // input: "11 + 2" -> result (success, value: "1", remainder: "1 + 2")
    public static readonly Parser<char> Numeric = Char(char.IsNumber, "Numeric character");
}
```

# Пишем первый комбинирующий парсер

```
public static Result<U> IfSuccess<T, U>(this Result<T> result, Func<Result<T>, Result<U>> next)
{
    return result.WasSuccessful
        ? next(result)
        : Result<U>.Failure(result.Message);
}
```



```
// parse('-') then parse('Number')
// input: "-1 + 5" -> result (success, value: "-1", remainder: "+ 12")
public static Parser<U> Then<T, U>(this Parser<T> first, Func<T, Parser<U>> second)
{
    return i => first(i).IfSuccess(s => second(s.Value)(s.Remainder));
}
```

## Строим путь к LINQ's query синтаксису

---

```
//maps T to Parser<T>
public static Parser<T> Return<T>(T value)
{
    return i => Result<T>.Success(value, i);
}

//maps/selects result from T to U.
public static Parser<U> Select<T, U>(this Parser<T> parser, Func<T, U> convert)
{
    return parser.Then(t => Return(convert(t)));
}

// maps Parse<IEnumerable<char>> to Parse<string>
public static Parser<string> Text(this Parser<IEnumerable<char>> characters)
{
    return characters.Select(chs => new string(chs.ToArray()));
}
```

# Комбинируем

```
// like LINQ's SelectMany
public static Parser<V> SelectMany<T, U, V>(
    this Parser<T> parser,
    Func<T, Parser<U>> selector,
    Func<T, U, V> projector)
{
    return parser.Then(t => selector(t).Select(u => projector(t, u)));
}

// input parser: Parser<int>
// input: " 123 + 12" -> result (success, value: "123", remainder: "+ 12")
public static Parser<T> Token<T>(this Parser<T> parser)
{
    return from leading in WhiteSpace.Many()
           from item in parser
           from trailing in WhiteSpace.Many()
           select item;
}
```

Полный код на **GitHub**

---



# HASL – HBase Avro Snapshot Language



Фрагмент из клипа Thrift shop 😊

# Разработка

---

- прототип был написан за 1 вечер на Sprache;
- был переписан на Superpower;
- в дальнейшем все изменения занимали 1-2 часа.

# Что получилось достичь за **1** день?

```
query(view: "TABLENAME") {  
  
  //simple selector  
  Selector  
  
  //nested-field selector  
  Selector.Next  
  
  Selector.Next.Value.ID  
  Selector.Next.Value.InnerValue  
  
  //nested-object selector  
  Selector.Next.Value {  
    ID  
    InnerValue  
  }  
  
  //nested-array selector  
  Selector.Next.Values((ID >= 1 AND ID <= 100) AND (InnerValue = "Hello" OR InnerValue="World")) [  
    ID  
    InnerValue  
  ]  
}
```

1. Транформирует объекты в таблицу.
2. Ориентирован на AVRO-схему.
3. Селекторы для всех типов: `object`, `type[]`, примитивы.
4. Фильтры для `[]`.
5. API функций для сложных вычислений и фильтраций.
6. Поддержка Join'ов.

# Перформанс **HASL** - Основа

---



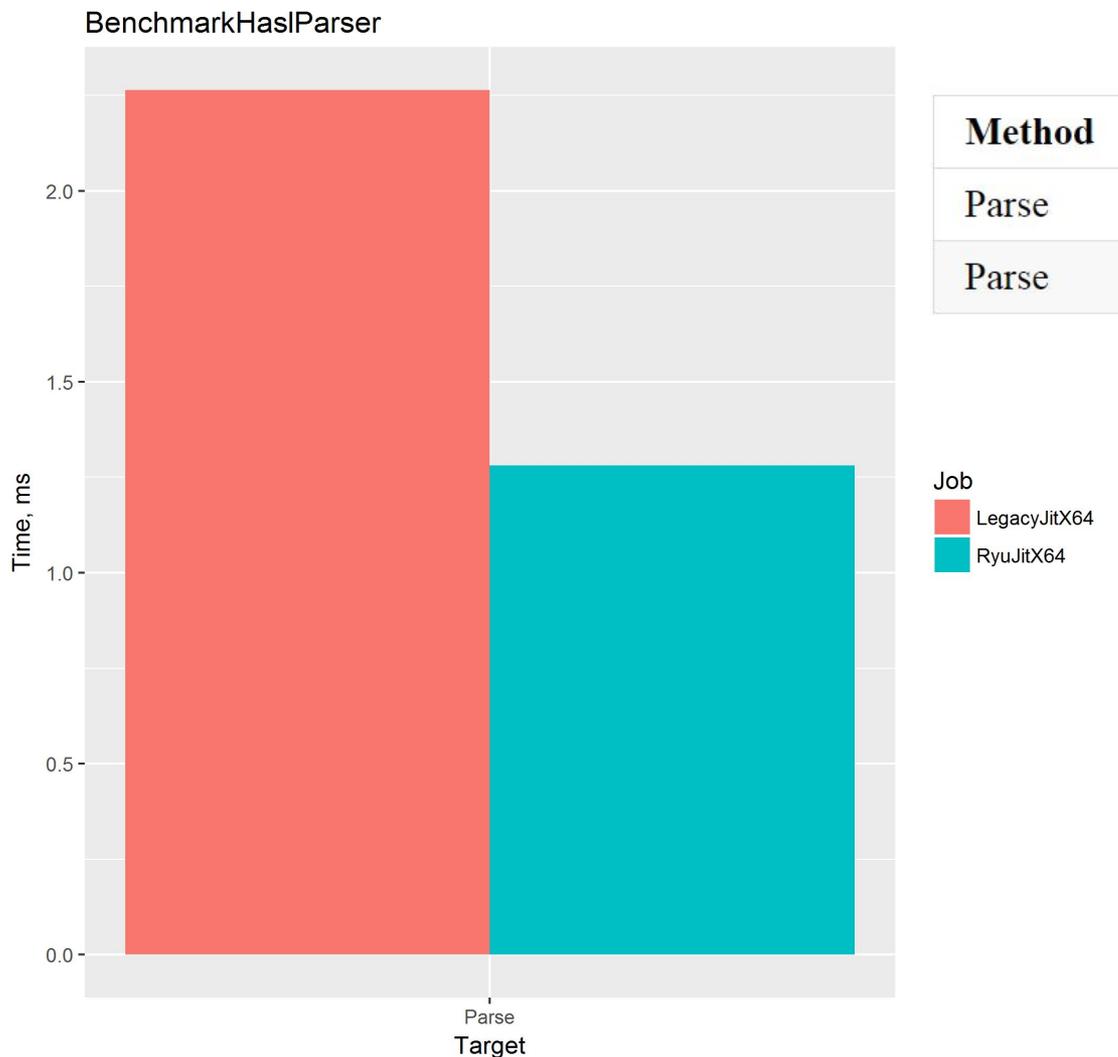
1. Core-I7 6700 3.40 GHz.
2. Windows server 2010.
3. x64.

## Перформанс **HASL** – Тестовые данные

---

1. 3 424 знаков.
2. 120 строк.
3. 3 Join'а.
4. 102 селектора.
5. 25 с фильтрами.
6. Полностью отформатирован.

# Перформанс **HASL** - Результаты



Method	Job	Jit	Mean
Parse	LegacyJitX64	LegacyJit	2.2644 ms
Parse	RyuJitX64	RyuJit	1.2813 ms

# Схема

```
1 {
2   "Apple":{
3     "Id":"long",
4     "Name":"String"
5   },
6   "AppleTraders":{
7     "AppleTrader":[
8       {
9         "Id":"long",
10        "Name":"String",
11        "TraderCompany":{
12          "Id":"long",
13          "Name":"String"
14        }
15      }
16    ]
17  }
18 }
```

## Пример запроса

```
1 - query {
2   //bad part
3   Apple.Id
4   AppleTraders.AppleTrader.Name
5   AppleTraders.AppleTrader.TraderCompany.Id
6   AppleTraders.AppleTrader.TraderCompany.Name
7
8   //cool part
9   Apple.Id
10 - AppleTraders.AppleTrader(Name="Antonovka" AND TraderCompany.Id = 405000) [
11     Name
12 -   TraderCompany {
13     Id
14     Name
15   }
16 ]
17 }
```

# Результат

Apple.Id	AppleTraders.AppleTrader.Name	TraderCompany.Id	TraderCompany.Name
1	Company1	405000	“Antonovka”
1	Company2	405000	“Antonovka”
1	Company3	405000	“Antonovka”
2	Company1	405000	“Antonovka”

Спасибо за внимание!

---

**Sprache**



**Superpower**



# Вопросы?

