Внешние языки DSL на funcparserlib

Андрей Власовских, СПб Политех
@vlasovskikh
#devconf_ru, 2010-05-17

Domain Specific Languages

- DSL — языки, использующие специальную нотацию для задач в определённой предметной области
- Идея: решать задачу в терминах языка её спецификации
- Внутренние и внешние DSL

Внутренний DSL: Django models

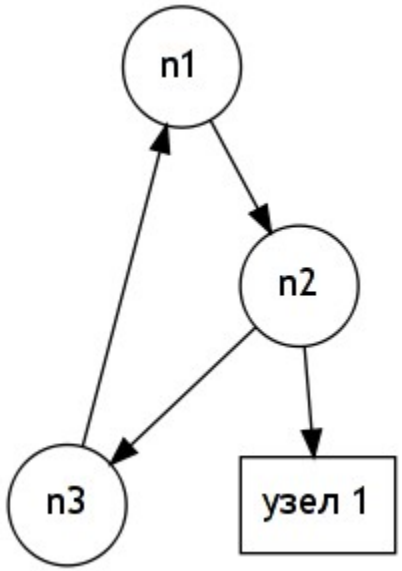
```
from django.db import models

class Permission(models.Model):
  name = models.CharField(_('name'), max_length=50)
  content_type = models.ForeignKey(ContentType)
  codename = models.CharField(_('codename'),
                              max_length=100)
  class Meta:
    unique_together = ['content_type', 'codename']
    ordering = ['content_type__app_label', 'codename']
```

- Другие внутренние DSL: WTForms, lxml E-factory, ...

Внешний DSL: Graphviz DOT

```
digraph {
  nodesep=0.6;
  node [shape=circle];
  n1 -> n2 -> n3 -> n1;
  n2 -> n4;
  n4 [
    label="узел 1",
    shape=box,
  ];
}
```



- Другие внешние DSL: regexp, CSS, JSON, SQL, ..

DSL: внутренние vs внешние

- Синтаксис языка-носителя
- Доступ к средствам языка-носителя
- Ясно как работать: всего лишь библиотека

- Любой синтаксис
- Нет языка-носителя, все средства делаем сами
- Нужно писать парсер языка для превращения во внутреннюю структуру данных

Задача парсера

Строка

```
'''digraph {
  n1 -> n2 -> n3 -> n1;
  n2 -> n4;
  n4 [shape=box];
}'''
```

Дерево*

```
Graph(
  type='digraph',
  stmts=[
    Edge(nodes=['n1', 'n2', 'n3', 'n1'],
         attrs=[]),
    Edge(nodes=['n2', 'n4'], attrs=[]),
    Node(id='n4',
         attrs=[Attr(name='shape',
                     value='box')])])
```

* Abstract Syntax Tree

Считается, что писать парсеры сложно

- Требуется знание теории и средств
  - Терминология, алгоритмы, computer science
  - Средства со множеством деталей и тонкостей
- Проще обойти проблему, чем написать
  - Сделать внутренний DSL
  - Взять готовый язык с парсером

Есть простые подходы к парсингу

- Можно легко создавать парсеры, не углубляясь в теорию
- Теория парсинга занимается в основном оптимизацией, что уже не так актуально
- Простота vs быстродействие

Какое мне дело до парсеров?

- Языки в Веб-разработке
  - Templates, markup, URL maps, complex user input
  - Если не хватает regexp, нужны парсеры
- Столкнулся с языком — знаешь, как с ним быть
  - Разбор готовых или своих собственных DSL
  - Эксперименты с языками
- Интересный подход к парсингу на основе ФП
  - Функциональные комбинаторы парсинга

Функциональные комбинаторы парсинга

- Внутренний DSL для создания парсеров внешних DSL
- На основе известной нотации BNF
- Компактный и ортогональный язык
  - Всего около 10 функций
  - Но можно написать парсер любого* языка
- Простой метод рекурсивного спуска

* С контекстно-свободной грамматикой

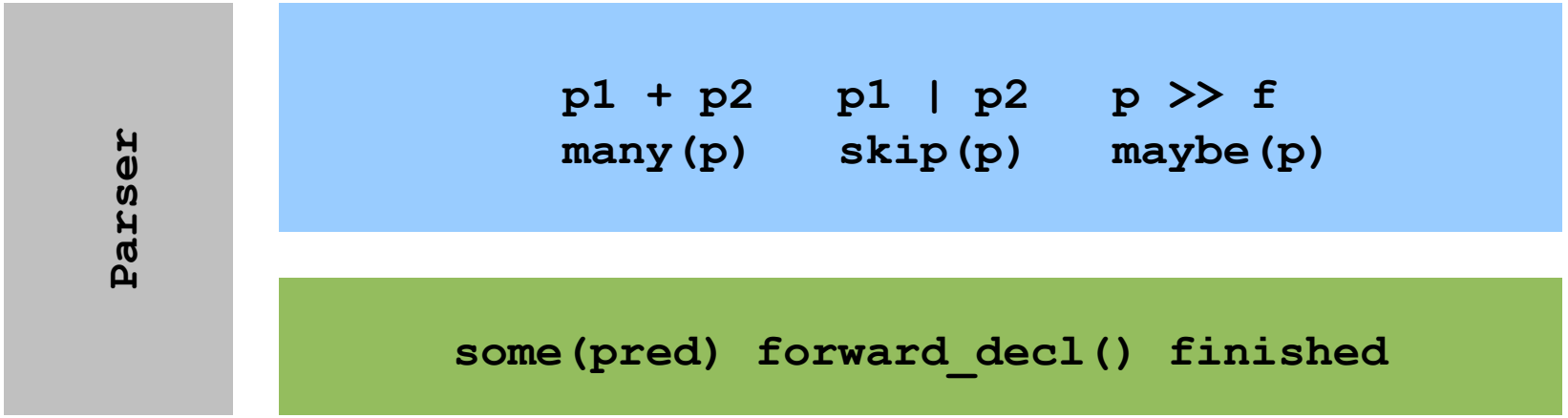
Библиотека funcparserlib

- Pure Python, без зависимостей, 700 SLOC, ок. 10 функций, документация
- Идея заимствована из OCaml и Haskell
- Есть и другие библиотеки, но с некомпактными API
  - Pyparsing, PLY, ANTLR, LEPL, ...

Структура языка комбинаторов

- Проблема парсеров внешних DSL
- Структура языка комбинаторов
- funcparserlib на практике

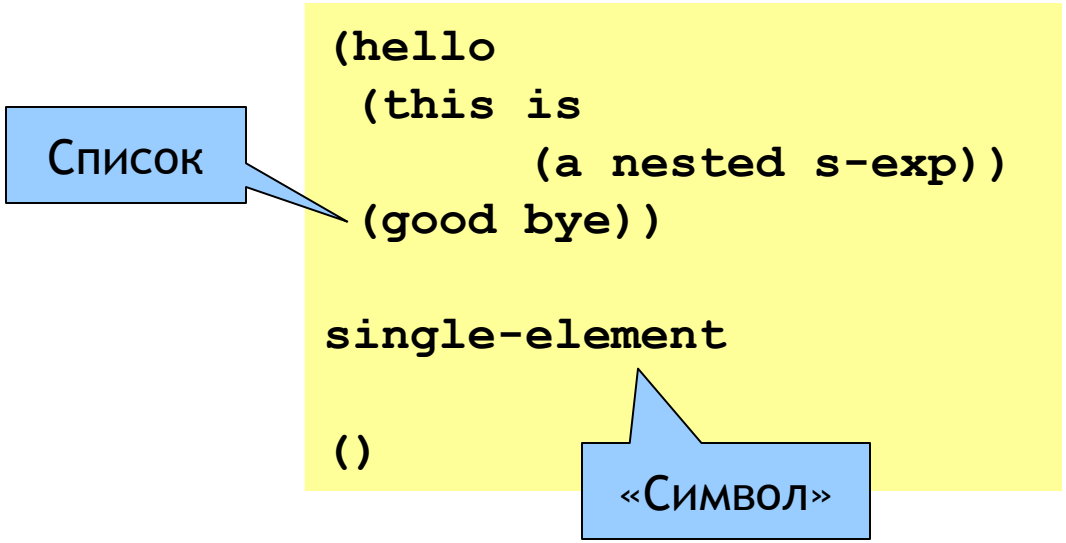
Весь язык комбинаторов



- Это весь язык комбинаторов
  - 1 класс **`Parser`**
  - 3 «конструктора» примитивных парсеров
  - 6 операций композции парсеров

Простой пример: язык S-Exp

- Простой язык для задания вложенных структур данных
- Похож на JSON, XML. Используется в Lisp
- Выражения состоят из символов и списков*



* Подмножество S-Exp

Два языка в одном примере

- Язык S-Exp
  - Как создать парсер внешнего DSL?
- Язык комбинаторов
  - Как устроен внутренний DSL, а не просто API?

Парсер языка S-Exp

Строка

```
'''
(hello
 (this is
       (a nested s-exp))
 (good bye))
'''
```

tokenize(s)

Список токенов

```
[
Token('op',   '('),
Token('sym',  'hello'),
Token('op',   '('),
Token('sym',  'this'),
   ...
Token('op',   '('),
Token('sym',  'good'),
Token('sym',  'bye'),
Token('op',   ')'),
Token('op',   ')'),
]
```

parse(toks)

```
[
  'hello',
  [
    'this',
    'is',
    ['a', 'nested', 's-exp'],
  ],
  ['good', 'bye'],
]
```

Дерево

Готовый токенизатор S-Exp

```
from funcparserlib.lexer import (
  Spec, make_tokenizer, Token)

def tokenize(s):
  'str -> [Token]'
  specs = [
    Spec('space',   r'[ \t\r\n]+'),
    Spec('sym',     r'[A-Za-z_0-9\-]+'),
    Spec('op',      r'[\(\)]'),
  ]
  f = make_tokenizer(specs)
  return [x for x in f(s)
          if x.type != 'space']
```

Шаблоны токенов по regexp

Токенизатор

Объект-токен с полями type, value

Preview: весь парсер S-Exp

```
def mksymbol(tok):
  return tok.value
def op(value):
  return some(lambda t: t.type == 'op' and
                        t.value == value)
def load(s):
  symtok = some(lambda t: t.type == 'sym')
  symbol = symtok >> mksymbol
  list = forward_decl()
  expr = symbol | list
  list.define(
    skip(op('(')) +
    many(expr) +
    skip(op(')'))
  return expr.parse(tokenize(s))
```

Дальше рассмотрим, как он устроен

Парсер одного токена

- Парсер **some(pred)** возвращает токен **t**, если **pred(t)** вернула **True**

```
hello
```

Текущий пример

```
symtok = some(lambda t: t.type == 'sym')
```

Парсер токена-символа

```
def op(value):
  return some(lambda t: t.type == 'op' and
                        t.value == value)
op('(')
op(')')
```

Парсеры токенов-скобок

Использование парсера

```
hello
```

- Успех

```
>>> symtok.parse(tokenize('hello world'))
Token('sym', 'hello')
```

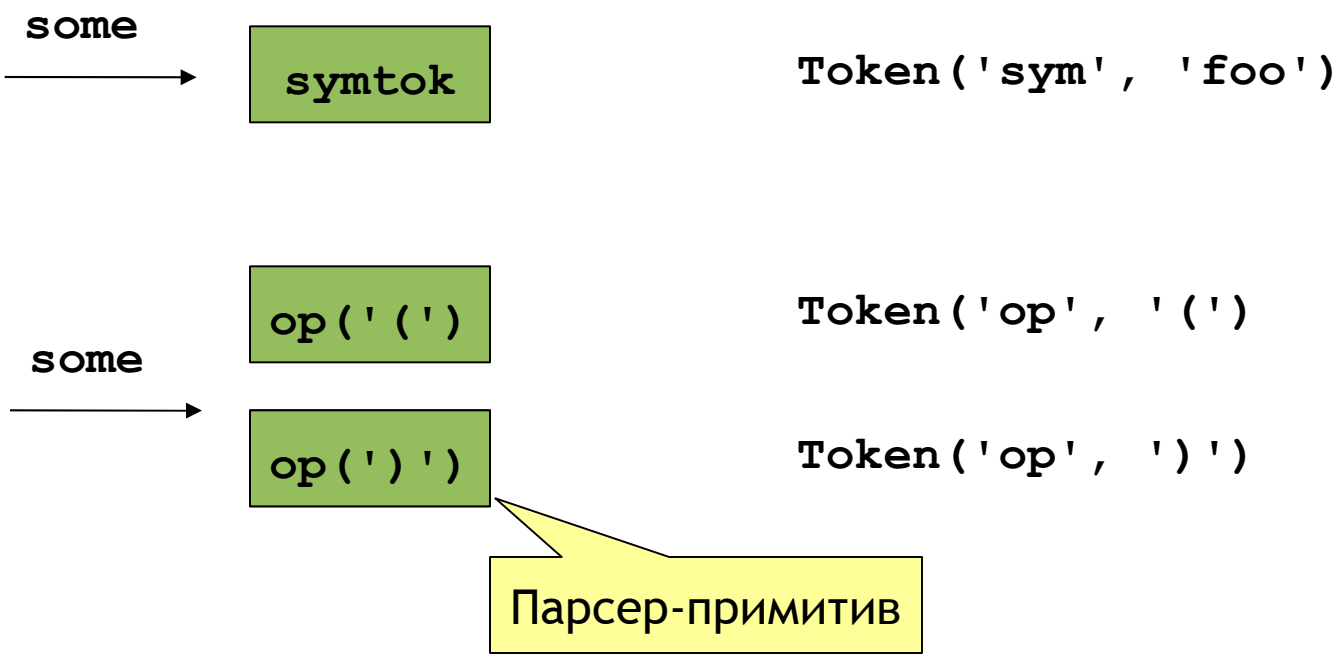
- Неуспех

```
>>> symtok.parse(tokenize('()'))
Traceback (most recent call first):
  ...
SyntaxError: 1,1-1,1: got unexpected token: op '('
```

Примитивные парсеры

```
hello
```

- Парсеры одного токена на основе комбинатора **some**
- Все остальные парсеры — на их основе



Интерпретация результатов

`hello`

- `p >> f` возвращает парсер, применяющий `f` к результату `p`
- Можно преобразовать выходное значение как угодно

```
def mksymbol(tok):
  return tok.value

symbol = symtok >> mksymbol
```

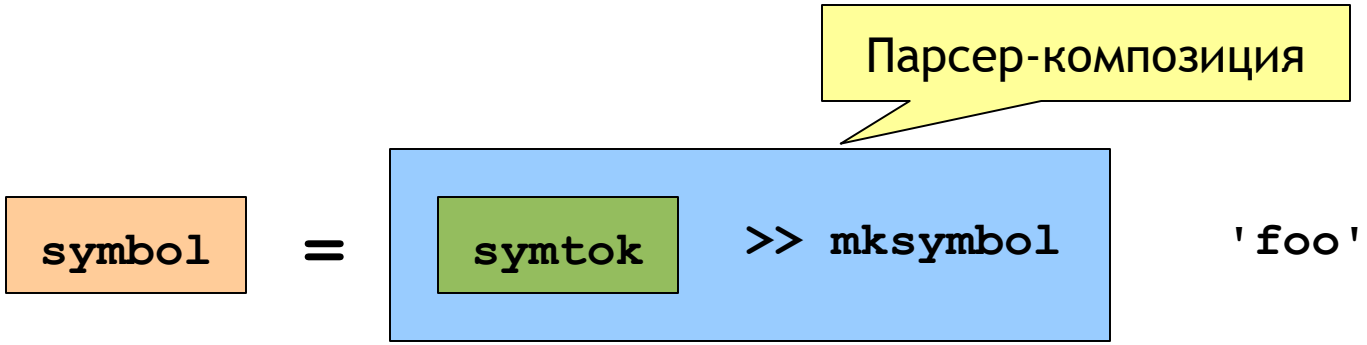
- Использование

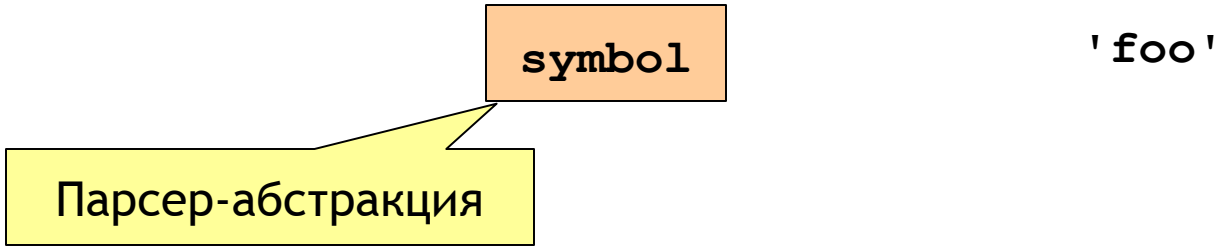
```
>>> symbol.parse(tokenize('hello world'))
'hello'
```

Композиция и абстракция

```
hello
```

- Композиция — составление парсеров из более простых
- Абстракция — составные парсеры выглядят как примитивы





Последовательность

(hello)

- `p1 + p2` запускает один парсер за другим, возвращает кортеж

```
>>> p = op('(') + symbol + op(')')

>>> p.parse(tokenize('(hello)'))
(Token('op',  '('), 'hello', Token('op',  ')'))
```

- `skip(p)` пропускает результат парсера `p`, не включая его в разобранную последовательность

```
>>> p = skip(op('(')) + symbol + skip(op(')'))

>>> p.parse(tokenize('(hello)'))
'hello'
```

Повторение парсера

```
(a b c)
```

- **`many(p)`** применяет парсер **`p`**, пока он успешен, возвращая список результатов

```
list = skip(op('(')) + many(symbol) + skip(op(')'))
```

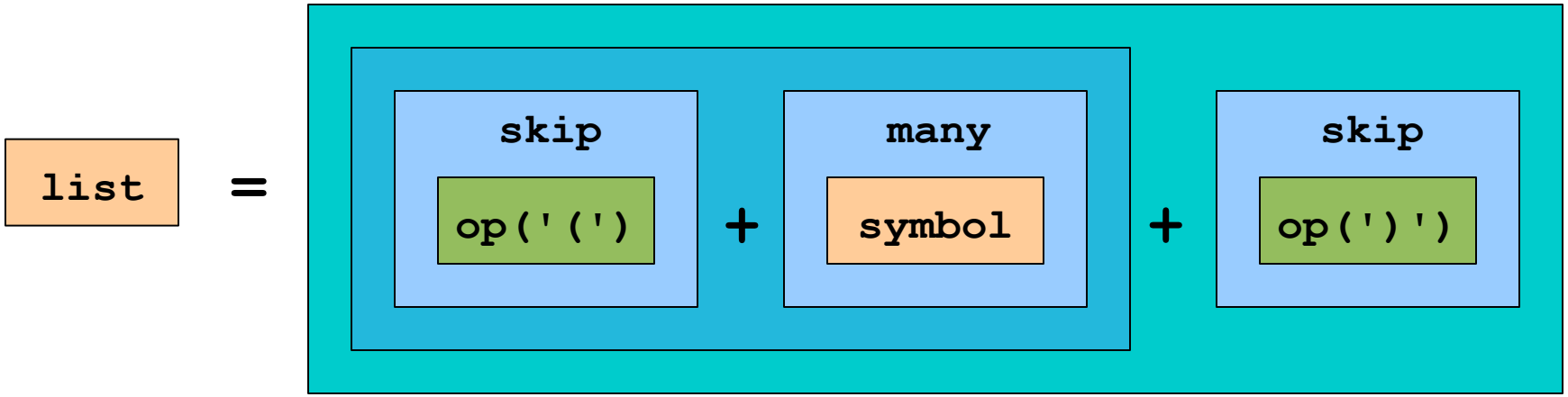
- Использование

```
>>> list.parse(tokenize('(a b c)'))
['a', 'b', 'c']
```

Абстракция в действии

```
(a b c)
```

- Осмысленное сложное выражение становится примитивом




```
list
```

```
['foo','bar']
```

Варианты выбора

```
(a b c)
hello
```

- `p1 | p2` пробует второй парсер, если первый неуспешен

```
expr = symbol | list
```

- Использование

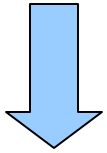
```
>>> expr.parse(tokenize('(a b c)'))
['a', 'b', 'c']

>>> expr.parse(tokenize('hello'))
'hello'
```

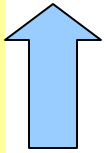
Рекурсивные определения: forward_decl

- Список может включать вложенные списки
- Напрямую нельзя, взаимная рекурсия

```
(a (b (c))
   (d e f))
hello
```



```
expr = symbol | list
list = (
  skip(op('(')) +
  many(expr) +
  skip(op(')'))
```



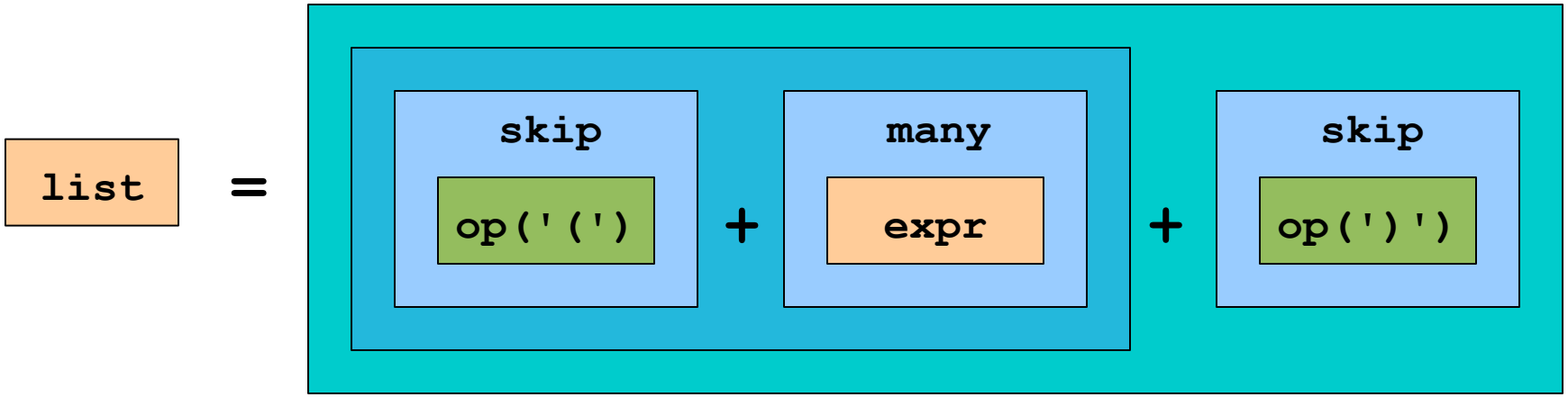
- Опережающее объявление

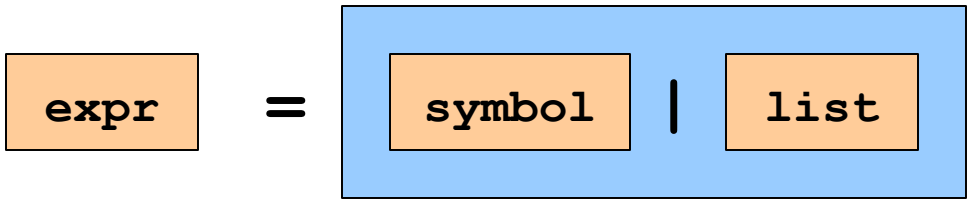
```
list = forward_decl()
expr = symbol | list
list.define(
  skip(op('(')) +
  many(expr) +
  skip(op(')')))
```

Рекурсивно определённые абстракции

- Можно парсить языки с произвольной вложенностью элементов

```
(a (b (c))
   (d e f))
hello
```



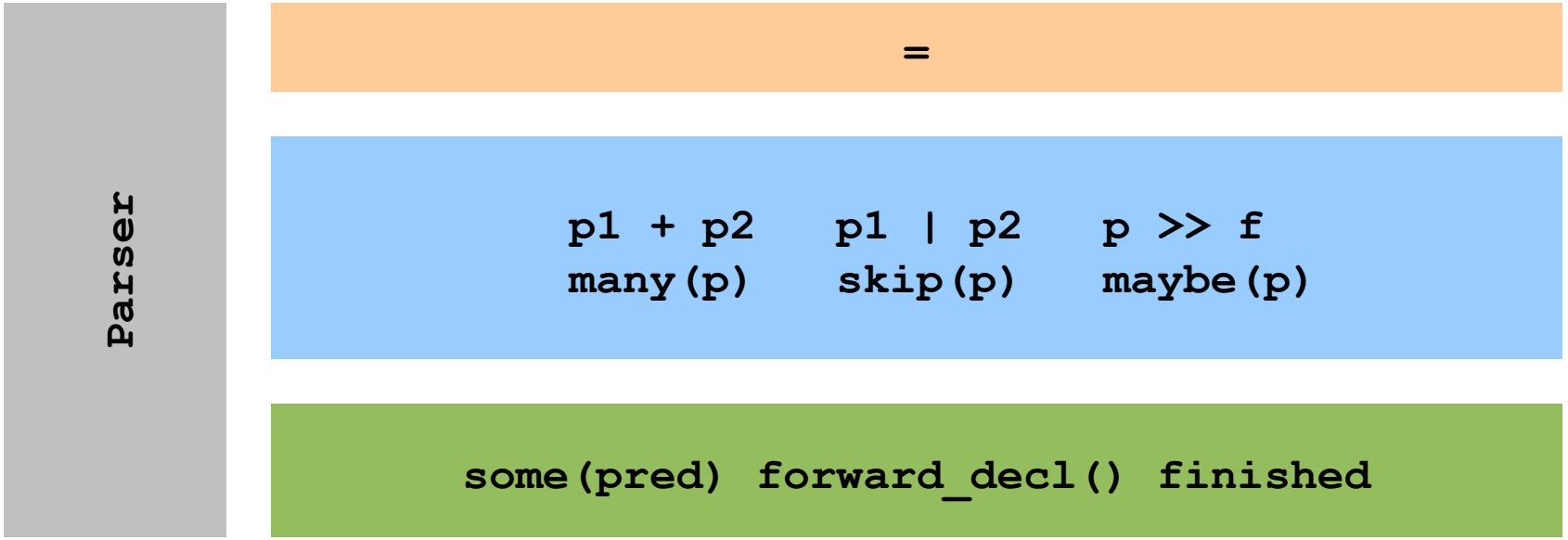



```
['foo',['bar', 'baz']]
'foo'
```

Собранный парсер S-Exp

```
def mksymbol(tok):
  return tok.value
def op(value):
  return some(lambda t: t.type == 'op' and
                        t.value == value)
def load(s):
  symtok = some(lambda t: t.type == 'sym')
  symbol = symtok >> mksymbol
  list = forward_decl()
  expr = symbol | list
  list.define(
    skip(op('(')) +
    many(expr) +
    skip(op(')'))
  return expr.parse(tokenize(s))
```

Примитивы, композиция и абстракция



- Компактность — мало встроенных элементов
- Замыкание — композиция парсеров вновь даёт парсер
- Ортогональность — можно произвольно* сочетать парсеры
- Абстракция — составной парсер можно сделать примитивом

* Почти произвольно

funcparserlib на практике

- Проблема парсеров внешних DSL
- Структура языка комбинаторов
- funcparserlib на практике

Кто использует funcparserlib

- @vlasovskikh
  - Тестовые парсеры JSON и DOT в комплекте funcparserlib
  - Парсер интерфейсов модулей Delphi для отслеживания межмодульных зависимостей
- Другие
  - Парсер алгоритмической музыки Thixotropical
  - Парсер разметки текстов песен на http://musi.cx/
  - Набросок парсера математических выражений WISE

Пример: алгоритмическая музыка Thixotropical

- Внешний DSL для вероятностной алгоритмической музыки

```
channel 0
  inst 0-48
  volume 50

channel 1
  inst 128-48
  volume 60

flute1
  chord 0 for 8 on 2
  chord 80 for 0.125 on 2
```

```
flute1*4
  chord 0 for 4 on 2

slow1*2
  next slow2a slow3a

loop1a
  chord 0 for 1 on 0
  next flute1 chime1 gong
```

Пример: тексты песен на http://musi.cx/

- Внешний DSL для разметки текстов песен

```
No more fear of failure
No more suffering
No more lies, I will arise
From blood-filled rivers of my enemies

>>
Unleash war
Unleash my wrath
Unleash revenge
Unleash my hell
<<

Unleash! (x4)
```

Что умеет funcparserlib

- Комбинаторы парсинга (ок. 400 SLOC с docstrings)
- Токенизатор на regexps (ок. 100 SLOC)
- Сообщения об ошибках по методу длиннейшего разобранного префикса
- Оптимизация подхода для Python
- Логи с трейсом разбора для отладки, читаемые имена парсеров
- Автоопределение незавершающихся парсеров

Вопросы?

http://code.google.com/p/funcparserlib/

@vlasovskikh