竜 TatSu

> По крайней мере для людей, которые отправляют мне письма о новом языке, который они проектируют, главный совет: делайте это, чтобы узнать как написать компилятор. Не ожидайте, что кто-то будет использовать его, только если Вы не работаете [?!?] с организацией, которая могла бы его развить. Это лотерея, и некоторые могут купить много билетов. Есть множество красивых языков (красивее, чем C), которые не получили популярность. Но кто-то выигрывает в лотерею, и создание языка по крайней мере чему-то Вас научит.
>
> Деннис Ритчи (1941-2011), Создатель языка программирования C и Unix.

竜 TatSu (компилятор грамматики [?!?]) — инструмент, принимающий грамматики в форме EBNF в качестве входных данных, и выдающий мемоизирующие (Packart) PEG-парсеры в Python.

竜 TatSu может скомпилировать грамматику, хранящуюся в строке, в объект `tatsu.grammars.Grammar`, который можно использовать для анализа любого заданного ввода так же, как модуль `re` делает это через регулярные выражения, или сгенерировать модуль Python, реализующий синтаксический парсер.

竜 TatSu поддерживает леворекурсивные правила в PEG-грамматиках и *учитывает левую ассоциативность* в результирующих деревьях разбора.

Введение

竜 TatSu отличается от других генераторов PEG-парсеров:

- Сгенерированные парсеры используют эффективную обработку исключений Python для возврата [?!?]. Сгенерированные 竜 TatSu парсеры просто декларируют что должно быть проанализировано [?!?]. Отсутствуют сложные последовательности *if-then-else* для принятия решений и вывода. Мемоизация позволяет обрабатывать одну и ту же последовательность за линейное время.
- Прямые и обратные проходы вместе с *вырезкой* элемента (с его удалением из кэша мемоизации) дают дополнительные возможности для ручной оптимизации на уровне грамматики.
- Делегирование лексем модулю Python `re` (Perl-стиль) для мощного и эффективного лексического анализа.
- Использование контекстного менеджера Python значительно уменьшает размер сгенерированных парсеров для чистоты кода и уменьшает кэш-промахи CPU.
- Включение файлов [?!?], наследование правил и включение правил [?!?] дают грамматике 竜 TatSu значительную выразительность.
- Автоматическая генерация Абстрактных Синтаксических Деревьев (AST) и Объектных Моделей вместе с Model Walkers и Генераторами Кода [?!?] даёт возможность для анализа и трансляции [?!?].

Генератор парсеров, поддержка run-time и сгенерированные парсеры имеют относительно низкую цикломатическую сложность. Приблизительно за 5 KLOC [?!?] Python можно изучить весь исходный код за один раз/сессию [?!?].

Зависит лишь от стандартной библиотеки Python, однако будет использоваться `regex`, если установлен, в этом же случае `colorama` для вывода трассировки и `pygraphviz` для генерации диаграмм.

竜 TatSu полнофункционален и в настоящее время используется для анализа сложных грамматик, анализа и перевода сотен тысяч строк входного текста, включая исходный код нескольких языков программирования.

Обоснование

竜 TatSu был создан для решения некоторых повторяющихся проблем, возникавших в течение десятилетий работы с инструментами генерации парсеров:

- Некоторые языки программирования используют *ключевые слова* как идентификаторы или имеют несколько значений для символов в зависимости от контекста (Ruby). Парсеру нужно контролировать лексический анализ, чтобы работать с такими языками.
- LL- и RL-грамматики загрязняются мириадами lookahead выражений [?!?], чтобы управляться с неоднозначными конструкциями в исходном коде. PEG-парсеры устраняют неоднозначность изначально.
- Отделение грамматики от кода, который реализует семантику, использование варианта хорошо известного грамматического синтаксиса (EBNF) даёт полную декларативную силу в описании языка. Языки общего назначения не относятся к задаче.
- Семантические действия [?!?] не относятся к грамматике. Она создают ещё один язык программирования для парсинга и трансляции: исходный язык, язык грамматики, язык семантики, сгенерированный парсером язык, целевой язык. Большинство грамматических парсеров не проверяют синтаксис встроенных семантических действий [?!?], из-за чего выводят ошибки в неудобное время, и по сгенерированному коду, а не по грамматике.
- Предварительная обработка (например, работа с включениями, фиксированными форматами столбцов [?!?] и сквозным отступом) относится к хорошо спроектированному языку, не к грамматике.
- Можно легко найти информацию по такому распространённому языку программирования как Python, но трудно для сложных grammar-description [?!?] языков. Грамматики 竜 TatSu не в духе *Translators and Interpreters 101 course* (Если трудно объяснить что-то студенту, то это либо слишком сложно, либо недостаточно изучено).
- Генерируемые парсеры должны быть просты в чтении и отладке для людей. Анализ исходного сгенерированного кода иногда является единственным способом найти ошибки в грамматике, семантических действиях [?!?] или самом генераторе. Опасно доверять сгенерированному коду, который невозможно понять.
- Python — отличный язык для работы с синтаксическим анализом и переводом [?!?].

Установка

```
$ pip install tatsu
```

Предупреждение:

Версии 竜 TatSu старше 5.0.0 требуют Python>=3.8

Python 2.7 более не поддерживается

Применение

В качестве библиотеки

竜 TatSu может использоваться как библиотека, как `re` в Python, встраиванием грамматик в виде строк и генерации грамматических моделей вместо генерации кода Python.

```
tatsu.compile(grammar, name=None, **kwargs)
```

Компилирует грамматику и генерирует *модель*, которая в последствии может быть использована для парсинга ввода.

```
tatsu.parse(grammar, input, start=None, **kwargs)
```

Компилирует грамматику и парсит входные данные, возвращая AST в качестве результата. Результат эквивалентен вызову:

```
model = compile(grammar)
ast = model.parse(input)
```

Скомпилированные грамматики кэшируются для эффективности.

```
tatsu.to_python_sourcecode(grammar, name=None, filename=None, **kwargs)
```

Компилирует грамматику в исходный код Python, реализующий парсер.

```
to_python_model(grammar, name=None, filename=None, **kwargs)
```

Компилирует грамматику и генерирует исходный код Python, реализующий объектную модель, определяемую аннотациями правил [?!?].

Пример использования Tatsu в качестве библиотеки:

```
GRAMMAR = '''
    @@grammar::Calc

    start = expression $ ;

    expression
        =
        | term '+' ~ expression
        | term '-' ~ expression
        | term
        ;

    term
        =
        | factor '*' ~ term
```

```
        | factor '/' ~ term
        | factor
        ;

    factor
        =
        | '(' ~ @:expression ')'
        | number
        ;

    number = /\d+/ ;
'''


def main():
    import pprint
    import json
    from tatsu import parse
    from tatsu.util import asjson

    ast = parse(GRAMMAR, '3 + 5 * ( 10 - 20 )')
    print('PPRINT')
    pprint.pprint(ast, indent=2, width=20)
    print()

    print('JSON')
    print(json.dumps(asjson(ast), indent=2))
    print()


if __name__ == '__main__':
    main()
```

Вывод:

```
PPRINT
[ '3',
  '+',
  [ '5',
    '*',
    [ '10',
      '-',
      '20']]]
JSON
[
  "3",
  "+",
  [
    "5",
    "*",
    [
      "10",
      "-",
      "20"
    ]
  ]
]
```

Компилирование грамматик в Python

Tatsu может быть запущен из командной строки:

```
$ python -m tatsu
```

Или:

```
$ scripts/tatsu
```

Или просто:

```
$ tatsu
```

Если Tatsu был установлен с помощью *easy_install* или *pip*.

Параметры *-h* и *-help* предоставляют полную информацию об использовании:

```
$ python -m tatsu -h
usage: tatsu [--generate-parser | --draw | --object-model | --pretty]
             [--color] [--trace] [--no-left-recursion] [--name NAME]
             [--no-nameguard] [--outfile FILE] [--object-model-outfile
FILE]
             [--whitespace CHARACTERS] [--help] [--version]
             GRAMMAR

TatSu takes a grammar in a variation of EBNF as input, and outputs a
memoizing
PEG/Packrat parser in Python.

positional arguments:
GRAMMAR               the filename of the Tatsu grammar to parse

optional arguments:
--generate-parser     generate parser code from the grammar (default)
--draw, -d            generate a diagram of the grammar (requires --
outfile)
--object-model, -g    generate object model from the class names given as
                        rule arguments
--pretty, -p          generate a prettified version of the input grammar

parse-time options:
--color, -c           use color in traces (requires the colorama library)
--trace, -t           produce verbose parsing output

generation options:
--no-left-recursion, -l
                        turns left-recursion support off
--name NAME, -m NAME  Name for the grammar (defaults to GRAMMAR base name)
--no-nameguard, -n    allow tokens that are prefixes of others
--outfile FILE, --output FILE, -o FILE
                        output file (default is stdout)
--object-model-outfile FILE, -G FILE
                        generate object model and save to FILE
```

```
--whitespace CHARACTERS, -w CHARACTERS
                        characters to skip during parsing (use "" to
disable)

common options:
--help, -h              show this help message and exit
--version, -v           provide version information and exit
$
```

Сгенерированные парсеры

Сгенерированный Tatsu парсер состоит из следующих классов:

- Класс `MyLanguageBuffer`, производный от `tatsu.buffering.Buffer`, обрабатывающий грамматические определения пробелов, комментариев и значения регистра.
- Класс `MyLanguageParser`, производный от `tatsu.parsing.Parser`, который использует `MyLanguageBuffer` для обхода входного текста и реализует парсер, используя один метод для каждого правила грамматики:

  ```
  def _somerulename_(self):
      ...
  ```

- Класс `MyLanguageSemantics`, одним семантическим методом для одного правила грамматики. Каждый метод получает в качестве единственного параметра Абстрактное Синтаксическое Дерево (AST), построенное из вызова правила [?!?]:

  ```
  def somerulename(self, ast):
      return ast
  ```

- Определение `if __name__ == "__main__":`, благодаря которому сгенерированный парсер может быть выполнен как скрипт Python.

  Методы в delegate class [?!?] возвращают то же AST, полученное в качестве параметра, но пользовательские семантические классы могут переписывать методы так, чтобы они возвращали что-либо (например, Семантический Граф). Семантический класс может быть использован как шаблон для конечной реализации семантики, который может опускать методы для правил, которые не нуждаются в семантической обработке.

  Метод `_default()`, если реализован, будет вызываться в случае, когда ни один метод не подходит для правила:

  ```
  def _default(self, ast):
      ...
      return ast
  ```

  Метод `_postproc()`, если реализован, будет вызван в семантическом классе после обработки каждого правила (включая семантики). Метод принимает текущий контекст парсинга в качестве параметра:

```
def _postproc(self, context, ast):
    ...
```

Использование сгенерированного парсера

Для использования сгенерированного парсера, просто создайте объект базового или абстрактного парсера и вызовите метод `parse()`, передав грамматику и начальное имя правила как параметры:

```
from tatsu.util import asjson
from myparser import MyParser

parser = MyParser()
ast = parser.parse('text to parse', rule_name='start')
print(ast)
print(json.dumps(asjson(ast), indent=2))
```

Основные конструкторы парсеров принимают именованные аргументы для определения пробельных символов, регулярные выражения для комментариев, чувствительность к регистру, verbosity [?!?] и не только (см. ниже).

Для добавления семантических действий [?!?], передайте semantic delegate to parse method [?!?]:

```
model = parser.parse(text, rule_name='start', semantics=MySemantics())
```

Если требуется специальное лексическое обработка (как в 80-столбцовых языка), тогда представитель [?!?] `tatsu.buffering.Buffer` может быть передан вместо текста:

```
class MySpecialBuffer(MyLanguageBuffer):
    ...

buf = MySpecialBuffer(text)
model = parser.parse(buf, rule_name='start', semantics=MySemantics())
```

Парсер, сгенерированный в модуль, также может быть вызван как скрипт:

```
$ python myparser.py inputfile startrule
```

В качестве скрипта, модуль сгенерированного парсера имеет несколько опций:

```
$ python myparser.py -h
usage: myparser.py [-h] [-c] [-l] [-n] [-t] [-w WHITESPACE] FILE
[STARTRULE]

Simple parser for DBD.

positional arguments:
    FILE                  the input file to parse
    STARTRULE             the start rule for parsing

optional arguments:
```

```
    -h, --help            show this help message and exit
    -c, --color           use color in traces (requires the colorama
library)
    -l, --list            list all rules and exit
    -n, --no-nameguard    disable the 'nameguard' feature
    -t, --trace           output trace information
    -w WHITESPACE, --whitespace WHITESPACE
                          whitespace specification
```

Синтаксис грамматики

竜 TatSu использует вариант синтаксиса EBNF. Определения для синтаксиса для VIM и Sublime Text могут быть найдены в директориях **etc/vim** и **etc/sublime** в исходном коде пакета.

Правила

Грамматика состоит из последовательности одного или нескольких правил вида:

```
name = <expre> ;
```

Если *name* совпадает с ключевым словом Python, подчёркивание [?!?] (_) будет добавлено перед ним в сгенерированном парсере.

Имя правила начинается с заглавного [?!?] символа:

```
FRAGMENT = /[a-z]+/ ;
```

do not advance over whitespace [?!?] перед началом парсинга. Эта особенность становится полезной при определении сложных лексических элементов, так как позволяет разбить их на несколько правил.

Парсер возвращает значение AST для каждого правила в зависимости от того, что была распарсено:

- Единственное значение
- Список AST
- dict-like [?!?] объект для правил с именованным элементами
- Объект, когда используется `ModelBuilderSemantics`
- None

За подробностями обращайтесь к *Абстрактные Синтаксические Деревья* и *Модели Сборки* [?!?].

Выражения

Выражения, в обратном порядке приоритета операторов, могут быть:

comment

Допускаются комментарии в стиле Python.

e1 | e2

Выбор. Сопоставление `e1` или `e2`.

`/` может быть первой опцией, если потребуется.

```
choices
    =
    | e1
    | e2
    | e3
    ;
```

e1 e2

Последовательность. Сопоставляется `e1`, затем `e2`.

(e)

Группировка. Сопоставляется `e`. Например: `('a' | 'b')`.

[e]

Опциональное сопоставление `e`.

{ e } или { e }*

Замыкание. Сопоставляет `e` ноль или более раз. AST, возвращённое замыканием, всегда список.

{ e }+

Положительное замыкание. Сопоставляет `e` один или более раз. AST всегда список.

{}

Пустое замыкание. Сопоставляет ничего и возвращает пустой список в качестве AST.

~

Выражение вырезки [?!?]. Выбрать текущую опцию и предотвратить выбор других опций даже если это ведёт к ошибке парсинга.

В этом примере, другие опции не будут выбраны, если скобки распарсены [?!?]:

```
atom
    =
    '(' ~ @:expre ')'
    | int
    | bool
    ;
```

s%{ e }+

Положительное объединение. Вдохновлено `str.join()` из Python. Парсится так же, как следующее выражение:

```
e {s ~ e}
```

пока результат единственный список вида [?!?]:

```
[e, s, e, s, e....]
```

Используйте *группировку*, если `s` сложнее, чем *токен* или *паттерн* [?!?]:

```
(s t)%{ e }+
```

s%{ e } или s%{ e }*

Объединение. Парсит список выражений, разделённых `s`, или пустое замыкание.

Эквивалентно:

```
s%{e}+|{}
```

op<{ e }+

Левое объединение. Схоже с `join`, но возвращает левоассоциативное дерево в виде кортежа (`tuple`), в котором первый элемент есть разделитель (`op`), а другие два элемента — операнды.

Выражение:

```
'+'<{\/d+/}+
```

Обработает следующий вход:

```
1 + 2 + 3 + 4
```

В такое дерево:

```
(
    '+',
    (
        '+',
        (
            '+',
            '1',
            '2'
        ),
        '3'
    ),
    '4'
)
```

op>{ e }+

Правое объединение. Схоже с `join`, но возвращает правоассоциативное дерево в виде кортежа (`tuple`), в котором первый элемент есть разделитель (`op`), а другие два — операнды.

Выражение:

```
'+'>{/\d+/}+
```

Обработает следующий вход:

```
1 + 2 + 3 + 4
```

В такое дерево:

```
(
    '+',
    '1',
    (
        '+',
        '2',
        (
            '+',
            '3',
            '4'
        )
    )
)
```

s.{ e }+

Положительная *группировка* gather [?!?]. Схоже с положительным `join`, но не добавляет разделитель в результирующее AST.

s.{ e } или s.{ e }*

Группировка gather [?!?]. Схоже с `join`, но не добавляет разделитель в результирующее AST.

Эквивалентно:

```
s.{e}+|{}
```

&e

Положительный lookahead [?!?]. Успешно [?!?], если `e` может быть разобрано, но не требует входа [?!?].

!e

Отрицательный lookahead [?!?]. Неверно [?!?], если `e` может быт разобрано, и не получило ввода [?!?].

’text’ или "text"

Match [?!?] токен `text` в кавычках.

Обратите внимание, что если `text` алфавитно-цифровой [?!?], то 竜 TatSu выполнит проверку алфавитно-цифровой [?!?] ли следующий символ. Это сделано для предотвращения токенизации вроде *IN*, когда текст в начале *INITIALIZE* [?!?]. Эта особенность может быть отключена передачей `nameguard=False` в `Parser` или `Buffer`, или применения паттерна вместо токена [?!?] (см. ниже). В качестве альтернативы, директивы `@@nameguard` или `@@namechars` могут быть объявлены в грамматике:

```
@@nameguard :: False
```

или для указания дополнительных символов, которые также должны рассматриваться как часть имён:

```
@@namechars :: '$-.'
```

r’text’ или r"text"

Разбирает токен `text` в кавычках, интерпретируя `text` как raw string literal из Python.

?"regexp" или ?’regexp’ или /regexp/

Pettern выражение [?!?]. Сопоставляет [?!?] регулярное выражение Python в нынешней [?!?] позиции текста. В отличие от других выражений, это не advance over whitespace или комментариев [?!?]. Для этого используйте `regexp` в качестве единственного элемента [?!?] правила.

`regex` интерпретируется как raw string literal в Python и передаётся с параметрами `regexp.MULTILINE | regexp.UNICODE` в модуль Python `re` (или `regex`, если доступен), используя `match()` в нынешней позиции текста [?!?]. Сопоставленный текст является AST для выражения.

Последовательные шаблоны объединяются в один.

/./

Любое выражение, соответствующее следующей позиции в тексте. Работает в точности как `?’.’`, но реализовано на лексическом уровне, без регулярных выражений.

->e

Выражение *«пропуска»*. Полезно для написания *recovery* [?!?] правил.

Парсер будет пропускать ввод по одному символу, пока не встретится `e`. Пробелы и комментарии будут пропускаться на каждом шаге. Проход по входу сделан эффективно, без регулярных выражений.

Выражение эквивалентно:

```
{ !e /./ } e
```

Стандартная форма выражения `->&e`, эквивалентная:

```
{ !e /./ } &e
```

Пример использования *«перейти к»* [?!?] для recovery [?!?]:

```
statement =
    | if_statement
    # ...
    ;

if_statement
    =
    | 'if' condition 'then' statement ['else' statement]
    | 'if' statement_recovery
    ;

statement_recovery = ->&statement ;
```

`constant` [?!?]

Ничего не сопоставляет [?!?], не реагирует, если `constant` был встречен.

Константы могут быть использованы для встраивания элементов в явные [?!?] или абстрактные синтаксические деревья, возможно для обхода потребности написания семантических действий. Например:

```
boolean_option = name ['=' (boolean|`true`) ] ;
```

rulename

Создаёт правило с именем `rulename`. Для упрощения лексических аспектов грамматик, правила с именами, начинающимися с символа в верхнем регистре [?!?], не будут advance over whitespace and comments [?!?].

>rulename

Оператор включения/встраивания [?!?]. Включает/встраивает *правостороннее* правило [?!?] с именем `rulename` в данной позиции.

Следующий набор объявлений:

```
includable = exp1 ;
expanded = exp0 >includable exp2 ;
```

Даёт такой же эффект, как и объявленное следующим образом `expanded`:

```
expanded = exp0 exp1 exp2 ;
```

Обратите внимание, что встраиваемое правило должно быть объявлено перед правилом, встраивающим его.

()

Пустое выражение. Успешно [?!?] без обработки входа. Его значение `None`.

!()

Неверное выражение. Простое применение `!` к `()`, что всегда неверно [?!?].

name:e

Добавляет результат `e` к AST, используя `name` в качестве ключа. Если `name` совпадает с каким-либо атрибутом или методом `dict`, или это ключевое слово Python, подчёркивание (_) будет добавлено к имени.

name+:e

Добавляет результат `e` к AST, используя `name` в качестве ключа. Принудительно приводит вход к `list`, даже если единственный элемент добавлен. Совпадения с атрибутами `dict` и ключевыми словами Python разрешаются добавлением нижнего подчёркивания к `name`.

@:e

Оператор перезаписи. Make the AST for the complete rule be the AST for `e` [?!?].

Оператор перезаписи полезен для сохранения только части правой части правила [?!?] без необходимости именовать его или добавлять семантическое действие.

Типичный пример использования оператора перезаписи:

```
subexp = '(' @:expre ')' ;
```

Возвращённое AST для правила `subexp` будет AST, полученное [?!?] от вызванного `expre`.

@+:e

Такое же, как `@:e`, но AST всегда будет `list`.

Этот оператор применим в подобных случаях:

```
arglist = '(' @+arg {',' @+:arg}* ')' ;
```

Когда разделяющие токены не интересны/не учитываются [?!?].

$

Символ *конца текста*. Проверяет, достигнут ли конец входного текста.

Если нет именованных элементов в правиле, AST состоит из элементов, разобранных правилом, либо единственного элемента или `list`. Это стандартное поведение упрощает написание несложных правил:

```
number = /[0-9+/] ;
```

Без необходимости писать:

```
number = number:/[0-9+/] ;
```

Если правило содержит именованные элементы, не именованные исключаются из AST (игнорируются).

Правила с аргументами

竜 TatSu позволяет указывать аргументы правил в Python-стиле:

```
addition(Add, op='+')
    =
    addend '+' addend
    ;
```

Значения аргументов фиксируются во время компиляции грамматики.

Альтернативный синтаксис в случае, если не требуются keyword аргументы [?!?]:

```
addition::Add, '+'
    =
    addend '+' addend
    ;
```

Семантические методы должны быть готовы принимать [?!?] любые аргументы, объявленные в соответствующем правиле:

```
def addition(self, ast, name, opt=None):
    ...
```

При работе с аргументами правил лучше определить метод `_default()`, принимающий любую комбинацию стандартных и keyword [?!?] аргументов:

```
def _default(self, ast, *args, **kwargs):
    ...
```

Расширение правил / Расширенные правила [?!?]

Правила могут расширять раннее созданные правила с помощью оператора `<`. *Базовое правило* должно быть раннее определено в грамматике.

Следующая последовательность объявлений:

```
base::Param = exp1 ;
extended < base = exp2 ;
```

Даст такой же результат, как объявление `extended` в таком виде:

```
extended::Param = exp1 exp2 ;
```

Параметры из **базового правила** копируются в новое правило, если новое правило не define it`s own [?!?]. Повторное наследование возможно, но *не было протестировано*.

Мемоизация

竜 TatSu это packrat parser [?!?]. Результат парсинга правила в данной позиции ввода [?!?] кэшируется, поэтому при следующей обработке парсером того же ввода в той же позиции этим же правилом, одинаковый результат будет получен и обработка продолжится без повторения парсинга. Мемоизация позволяет писать более чистые и понятные грамматики из-за отсутствия опасений повторяющихся подвыражений, снижающих производительность.

Некоторые правила не нужно мемоизировать. Например, правила, которые may succeed [?!?] или не зависят от ассоциированных семантических действий, не должны быть мемоизированы, если успешность [?!?] зависит не только от ввода.

Декоратор `@nomemo` отключает мемоизацию для конкретного правила:

```
@nomemo
INDENT = () ;

@nomemo
DEDENT = () ;
```

Перезапись правил

Правило грамматики может быть переопределено с помощью декоратора `@override`:

```
start = ab $;

ab = 'xyz' ;

@override
ab = @:'a' {@:'b'} ;
```

В комбинации с директивой `#include` [?!?] перезапись правил может быть использована для создания модифицированной грамматики без переписывания оригинала.

Имя грамматики

Префикс, используемый в классах, сгенерированных 竜 TatSu, могут быть переданы в утилиту командной строки с помощью опции **-m**:

```
$ tatsu -m MyLanguage mygrammar.ebnf
```

Сгенерирует:

```
class MyLanguageParser(Parser):
    ...
```

Имя также может быть объявлено в грамматике с помощью директивы `@@grammar`:

```
@@grammar :: MyLanguage
```

Пробелы

По умолчанию, сгенерированные 竜 TatSu парсеры пропускают обычные пробельные символы регулярным выражением `r'\s+'` с флагом `re.UNICODE` (или со свойством `Pattern_White_Space`, если модуль `regex` доступен), но Вы можете изменить поведение, передав параметр `whitespace` в парсер.

Например, следующий код пропускает **табы** (**\t**) и **пробелы**, но не такие типичные разделители как **переход на новую строку** (**\n**):

```
parser = MyParser(text, whitespace='\t ')
```

Строка символов конвертируется во множество символов [?!?] регулярного выражения перед началом парсинга.

Вы также можете передать регулярное выражение напрямую вместо строки. Следующий код эквивалентен примеры выше:

```
parser = MyParser(text, whitespace=re.compile(r'[\t ]+'))
```

Обратите внимание, что регулярное выражение должно быть предкомпилировано, чтобы 竜 TatSu отличил его от обычной строки.

Если Вы не определяете никакие разделители, тогда нужно обрабатывать их в Вашей грамматике (как зачастую и делается в PEG-парсерах):

```
parser = MyParser(text, whitespace='')
```

Разделители также могут быть указаны внутри грамматики директивой `@@whitespace`, несмотря на то, что [?!?] любой из методов выше перепишет настройку в грамматике:

```
@@whitespace :: /[\t ]+/
```

Чувствительность к регистру

Если исходный язык чувствителен к регистру, можно указать об этом парсеру с помощью параметра `ignorecase`:

```
parser = MyParser(text, ignorecase=True)
```

Вы также можете указать нечувствительность к регистру в грамматике с помощью директивы `@@ignorecase`:

```
@@ignorecase :: True
```

Изменение повлияет на сопоставление [?!?] токенов, но не сопоставление [?!?] паттернов. Используйте `(?i)` в паттернах, игнорирующих регистр.

Комментарии

Парсеры будут пропускать токены, определённые как регулярное выражение, с помощью параметра `comments_re`:

```
parser = MyParser(text, comments_re="\(\*.*?\*\)")
```

Для более сложной обработки комментариев, можно переписать метод `Buffer.eat_comments()`.

Для гибкости можно определить паттерн для комментариев в конце строки отдельно:

```
parser = MyParser(
    text,
    comments_re="\(\*.*?\*\)",
    eol_comments_re="#.*?$"
)
```

Оба паттерна могут быть определены внутри грамматики с помощью директив `@@comments` и `@@eol_comments`:

```
@@comments :: /\(\*.*?\*\)/
@@eol_comments :: /#.*?$/
```

Зарезервированные и Ключевые слова

Некоторые языки резервируют определённые токены в качестве валидных идентификаторов, так как токены используются для обозначения конкретных конструкций языка. Такие зарезервированные токены известны как Зарезервированные слова или Ключевые слова.

竜 TatSu помогает предотвращать использование ключевых слов в качестве идентификаторов с помощью директивы `@@keyword` и декоратора `@name`.

Грамматика может указывать зарезервированные токены, предоставляя их список в одной или более директивах `@@keyword`:

```
@@keyword :: if endif
@@keyword :: else elseif
```

Декоратор `@name` проверяет, не совпадает ли результат грамматического правила с токеном, определённым как ключевое слово:

```
@name
identifier = /(?!\d)\w+/ ;
```

Есть ситуации, в которых токен резервируется только для очень специфичного контекста. В таких случаях, обратный обход предотвратит использование токена:

```
statements = {!'END' statement}+ ;
```

Директивы включения (include)

Грамматики 竜 TatSu поддерживают включение файлов с помощью директивы включения:

```
#include :: "filename"
```

Расположение **filename** относительно *directory/folder* исходника [?!?]. Абсолютные пути и **../** приемлемы [?!?].

Функциональность, необходимая для реализации включений, доступна для всех парсеров, сгенерированных 竜 TatSu, через класс `Buffer`. За примерами обращайтесь к классу `EBNFBuffer` модуля `tatsu.parser`.

Левая рекурсия

竜 TatSu поддерживает левую рекурсию в PEG-грамматиках. Используется алгоритм Warth et al.

Иногда, при отладке грамматики, полезно отключать или включать поддержку левой рекурсии:

```
parser = MyParser(
    text,
    left_recursion=True,
)
```

Левая рекурсия также может быть выключена внутри грамматики с помощью директивы `@@left_recursion`:

```
@@left_recursion :: False
```

Директивы грамматики

竜 TatSu предоставляет *директивы* грамматики для управления поведением сгенерированных парсеров. Все директивы имеют вид `@@name :: <value>`. Например:

```
@@ignorecase :: True
```

Директивы, поддерживаемые 竜 TatSu, описаны ниже.

@@grammar ::

Определяет имя грамматики и предоставляет базовое имя для классов парсера исходного кода генерации [?!?].

@@comments ::

Определяет регулярное выражение для определения и исключения строчных [?!?] (в скобках) комментариев перед сканированием текста парсером. Для **(* ... *)** комментариев:

```
@@comments :: /\(\*((?:.|\n)*?)\*\)/
```

@@eol_comments ::

Определяет регулярное выражение для определения и исключения комментариев в конце строки перед сканированием текста парсером. Для **# ...** комментариев:

```
@@eol_comments :: /#([^\n]*?)$/
```

@@ignorecase ::

Если установлено значение `True`, 竜 TatSu не обращает внимания на регистр символов во время парсинга токенов. По умолчанию `False`:

```
@@ignorecase :: True
```

@@keyword :: {|}+

Определяет список строк или слов, которые грамматика определяет как «ключевые слова». Может встречаться более одного раза. Обращайтесь к Зарезервированные и Ключевые слова для дополнительной информации.

@@left_recursion ::

Включает леворекурсивные правила в грамматике. Обращайтесь к Левая Рекурсия за дополнительной информацией.

@@namechars ::

Список (не alphanumeric [?!?]) символов, определяемых как часть имён, при использовании `@@nameguard`:

```
@@namechars :: '-_$'
```

@@nameguard ::

Если установлено `True`, пропускает [?!?] соответствующие токены, если следующий символ ввода alphanumeric [?!?] или `@@nemachar`. Обращайтесь к 'text' за дополнительной информацией.

```
@@nameguard :: False
```

@@parseinfo ::

Если установлено `True`, парсер будет добавлять информацию о парсинге к каждому сгенерированному AST и узлу [?!?] в поле `parseinfo`.

Информация включает:

- **rule** — имя правила, обработавшего узел
- **pos** — начальная позиция узла во вводе
- **endpos** — конечная позиция узла во вводе
- **line** — начальный номер строки для элемента во вводе
- **endline** — конечный номер строки для элемента во вводе

Включение `@@parseinfo` предоставляет точный отчёт [?!?] по входному исходному коду во время семантических действий.

@@whitespace ::

Определяет регулярное выражение для разделителей, игнорируемых парсером. По умолчанию `/(?s)\s+/`:

```
@@whitespace :: /[\t ]+/
```

Абстрактные Синтаксические Деревья (AST)

По умолчанию, AST либо *список* (для *замыканий* и правил без именованных элементов), либо *dict*-deriver [?!?] объект, содержащий один элемент для каждого именованного элемента в грамматическом правиле. Доступ к элементам может быть осуществлён через стандартный синтаксис *словарей* (`ast['key']`) или атрибуты (`ast.key`).

Точки входа [?!?] AST — единственное значение, если только один элемент был ассоциирован с именем [?!?], или списки, если было ассоциировано больше одного элемента. Есть возможность в синтаксисе грамматики (оператор **+:**) для принудительного приведения входной точки AST к списку, если только один элемент был ассоциирован. Значения для именованных элементов, не найденных при парсинге (возможно, потому они опциональные) `None`.

Когда именованный аргумент `parseinfo=True` был передан в конструктор `Parser`, к узлам AST будет добавлен *dict*-like [?!?] элемент `parseinfo`. Элемент содержит `collections.namedtuple` с информацией парсинга для узла:

```
ParseInfo = namedtuple(
    'ParseInfo,
    [
        'tokenizer',
        'rule',
        'pos',
        'enpos',
        'line',
        'endline',
    ]')
```

С помощью метода `Buffer.line_info()` возможно восстановить строку, столбец, и исходный разобранный текст для узла. Обратите внимание, что когда `ParseInfo` сгенерирован, `Buffer`, использованный для парсинга, остаётся в памяти в течение всего времени жизни AST.

Генерация `parseinfo` также может управляться директивой грамматики `@parseinfo :: True`.

Семантические действия

В грамматиках 竜 TatSu нет конструкций для семантических действий. Это намеренное решение, так как семантические действия усложняют [?!?] декларативную природу грамматик и обеспечивают плохую модульность с точки зрения выполнения синтаксического анализа.

Семантические действия определены в классе и применяются путём передачи объекта класса в качестве параметра `semantics=` в метод парсера `parse()`.

竜 TatSu вызывает [?!?] метод, соответствующий имени грамматики, каждый раз, когда работает правило. Аргументом AST, построенное right-hand-side [?!?] правила:

```
class MySemantics(object):
    def some_rule_name(self, ast):
        return ''.join(ast)

    def _default(self, ast):
        pass
```

Если соответствующий имени правила метод отсутствует, 竜 TatSu вызовет метод `_default()`, если он определён:

```
def _default(self, ast):
    ...
```

Ничего не произойдёт, если ни метод для правила, ни `_default()` не были определены.

Методы для правил в классах, реализующих семантики, предоставляют достаточно возможностей для пост-обработки правил, таких как проверки (для неадекватного использования ключевых слов в качестве идентификаторов) или трансформаций AST:

```
class MyLanguageSemantics(object):
    def identifier(self, ast):
        if my_lange_module.is_keyword(ast):
            raise FailedSemantics('"%s" is a keyword' % str(ast))
        return ast
```

[?!? перевод на f-строки?]

Для более детального контроля достаточно объявить больше правил, так влияние на время парсинга будет минимальным.

Если предварительная обработка требуется в какой-то позиции, достаточно вставить пустые правила где необходимо:

```
myrule = first_part preproc {second_part} ;

preproc = () ;
```

Абстрактный парсер будет воспринимать метод как семантическое действие метод, объявленный следующим образом:

```
def preproc(self, ast):
    ...
```

Построение моделей [?!?]

Именование элементов в правилах грамматики позволяет парсеру пропустить/отбросить [?!?] не интересующие части ввода, такие как пунктуация, чтобы создать AST, отражающее семантическую структуру того, что было обработано/распарсено [?!?]. Но AST не несёт информацию о правиле, которое его сгенерировало, из-за навигация в деревьях может быть затруднительной.

竜 TatSu определяет класс семантик `tatsu.model.ModelBuilderSemantics`, который помогает строить объектные модели из абстрактных синтаксических деревьев:

```
from tatsu.model import ModelBuilderSemantics
parser = MyParser(semantics=ModelBuilderSemantics())
```

Когда Вы добавите желаемый тип узла в качестве первого параметра к каждому грамматическому правилу:

```
addition::AddOperator = left:mulexpre '+' right:addition ;
```

`ModelBuilderSemantics` синтезирует класс `class AddOperator(Node)` и использует его для создания узла. Синтезированный класс будет иметь один атрибут с именем, соответствующим имени элемента в правиле.

Вы также можете использовать встроенные типы Python в качестве типов узлов, и `ModelBuilderSemantics` сделает right thing [?!?]:

```
integer::int = /[0-9]+/ ;
```

`ModelBuilderSemantics` ведёт себя как любой другой класс семантик, поэтому его стандартное поведение может быть переопределено с помощью определения метода для обработки результата любого конкретного правила грамматики.

Модели обхода

Класс `tatsu.model.NodeWalker` позволяет создать trversal [?!?] модель с помощью экземпляра `ModelBuilderSemantics`:

```
from tatsu.model import NodeWalker

class MyNodeWalker(NodeWalker):

    def walk_AddOperator(self, node):
        left = self.walk(node.left)
        right = self.walk(node.right)

        print("ADDED", left, right)

model = MyParser(semantics=ModuleBuilderSemantics()).parse(input)

walker = MyNodeWalker()
walker.walk(model)
```

Когда определён метод на подобии `walk.AddOperator()`, он будет вызван во время *обхода* узла данного типа. Python-вариант имени класса также может быть использован для метода `walk` [обхода ?!?]: `walk__add_operator()` (обратите внимание на двойное подчёркивание).

Если *walk* [?!?] метод для класса узла не был найден, то начнётся поиск метода для class`s bases [?!?], что позволяет писать *catch-call* методы, такие как:

```
def walk_Node(self, Node):
    print("Reached Node", node)

def walk_str(self, s):
    return s

def walk_object(self, o):
    raise Exception("Unexpected tyle %s walked", type(o).__name__)
```

[Опечатка в коде? Перевод на f-строки ?!?].

Предварительно объявленные классы могут быть переданы в экземпляр `ModelBuilderSemantics` через параметр `types=`:

```
from mymodel import AddOperator, MulOperator

semantics = ModelBuilderSemantics(types=[AddOperator, MulOperator])
```

[Форматирование? ?!?]

`ModelBuilderSemantics` ничего не требует от `types=` [?!?], поэтому любой конструктор (функция или частичная функция [?!?]) может быть использован.

Иерархия моделей классов

[Опечатка в исходнике ?!?].

Возможно определить базовый класс для сгенерированных узлов модели:

```
additive
    =
    | addition
    | substraction
    ;

addition::AddOperator::Operator
    =
    left:multexpre op:"+" right:additive
    ;

substraction::SubstractOperator::Operator
    =
    left:multexpre op:"-" right:additive
    ;
```

竜 TatSu сгенерирует базовый класс, если он уже не создан.

Базовые классы могут быть использованы в качестве целевых классов в *обходчиках* [?!?] и **генераторов кода**:

```
class MyNodeWalker(NodeWalker):
    def walk_Operator(self, node):
        left = self.walk(node.left)
        right = self.walk(node.right)
        op = self.walk(node.op)

        print(type(node).__name__, op, left, right)

class Operator(ModelRenderer):
    template = '{left} {op} {right}'
```

Шаблоны и трансляция

Заметка

> Начиная с 竜 TatSu 3.2.0, генерация кода отделена от моделей грамматики через `tatsu.codegen.CodeGenerator`, что позволяет использовать в качестве целевых языки, отличные от Python. Вместе с тем, использование inline [?!?] шаблонов и `rendering.Renderer` не изменилось. Обращайтесь к примеру *regex* за примерами совместного моделирования и генерации кода [?!?].

竜 TatSu не навязывает способ создания трансляторов, но предоставляет возможности [?!?], которые использует для генерации исходного кода Python для парсеров.

Трансляция в 竜 TatSu основана на шаблонах, но вместо создания или использования сложных шаблонизаторов (ещё одного языка), полагается на простой, но мощный `string.Formatter` стандартной библиотеки Python. Шаблоны представляют собой простые строки, встроенные в код в стиле 竜 TatSu.

Чтобы сгенерировать парсер, 竜 TatSu создаёт объектную модель разобранной грамматики. Экземпляр `tatsu.codegen.CodeGenerator` сопоставляет объекты модели классам, наследуемым от `tatsu.codegen.ModelRenderer`, и реализуют трансляцию и rendering [?!?] с помощью строковых шаблонов. Шаблоны left-trimmed [?!?] по пробелам, как *doc-comments* в Python. Пример из исходного кода 竜 TatSu:

```
class LookAhead(ModelRenderer):
    template = '''\
                with self._if():
                {exp:1::}\
                '''
```

Каждый *атрибут* объекта, не начинающийся с нижнего подчёркивания (_), может быть использован в качестве поля шаблона, и поля могут быть добавлены или изменены с помощью переопределения метода `render_fileds(fields)`. Сами поля *лениво rendered* [?!?] перед раскрытием в шаблон [?!?], поэтому поле может быть экземпляром наследника [?!?] `ModelRenderer`.

Модуль `rendering` определяет [?!?] `Formatter` с улучшенный поддержкой рендеринга [?!?] элементов *iterable* [?!?] по одному. Пример синтаксиса:

```
"""
{fieldname:ind:sep:fmt}
"""
```

`ind`, `sep` и `fmt` опциональны, но не три двоеточия. Поле, заданное таким образом, будет отображено [?!?] с помощью [?!?]:

```
indent(sep.join(fmt % render(v) for v in value), ind)
```

Стандартный множитель для `ind` 4, но он может быть переопределён в виде `n*m` (например `3*1`).

Заметка

> Использование символа новой строки [?!?] (**\n**) в качестве разделителя будет мешать обрезке слева и расстановке отступов в шаблонах. Для использования новой строки в качестве разделителя, определите его как `\n` и рендерер Вас поймёт [?!?].

Левая рекурсия

竜 TatSu поддерживает прямую и непрямую левые рекурсии в правилах грамматики с помощью алгоритма, описанного *Nicolas Laurent* и *Kim Mens* в их статье 2015 года *Parsing Expression Grammars Made Practical*.

Дизайн и реализация левой рекурсии сделана Vic Nightfall с research [?!?] и помощью Nicolas Laurent на Autumn, и исследованием [?!?] Philippe Sigaud на PEGGED.

Левые рекурсивные правила создают [?!?] левоассоциативные деревья парсинга (AST), как большинство пользователей ожидает.

Поддержка левой рекурсии в 竜 TatSu включена по умолчанию. Чтобы отключить её для конкретной грамматики, используйте директиву `@@left_recursion`:

```
@@left_recurtion :: False
```

Предупреждение

Не все леворекурсивные грамматики, используемые 竜 TatSu syntax [?!?] PEG. Та же ситуация с праворекурсивными грамматиками. **Порядок правил in matters [?!?] in PEG**.

Для праворекурсивных грамматик вариант [?!?], который анализирует наибольшее число входных данных, должен стоять первым. То же самое и для леворекурсивных грамматик.

Также, для грамматик с **непрямой левой рекурсией правила, содержащие варианты, должны активироваться первыми во время парсинга**. Следующая грамматика верна, но не будет работать, если начальное правило изменить на `start = mul ;`:

```
start = expr ;

expr
    =
    mul | identifier
    ;

mul
    =
    expr "*" identifier
    ;

identifier
    =
    /\w+/
    ;
```

Calc Мини Туториал [?!?]

Пользователи 竜 TatSu полагают, что простой калькулятор, как в документации PLY, может быть полезным.

Вот и он.

Начальная грамматика [?!?]

Исходная грамматика PLY для арифметических выражений:

```
expression : expression + term
           | expression - term
           | term

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )
```

Входное выражение для текста:

```
3 + 5 * ( 10 - 20 )
```

Грамматика Tatsu

Первый шаг — конвертация грамматики в синтаксис и стиль 竜 TatSu, добавление правил для лексических элементов (`number` в этом случае), добавление правила `start`, проверяющее конец ввода [?!?], и директива для имени сгенерированного класса:

```
@@grammar::CALC

start
    =
    expression $
    ;

expression
    =
    | expression "+" term
    | expression "-" term
    | term
    ;

term
    =
    | term "*" factor
    | term "/" factor
    ;

factor
    =
    | "(" expression ")"
    | number
    ;

number
    =
    /\d+/
    ;
```

Добавление *вырезок* [?!?]

Вырезки [?!?] заставляют парсер [?!?] фиксировать конкретный вариант после обнаружения определённого токена. Они делают парсинг более эффективным благодаря тому, что остальные варианты не рассматриваются [?!?]. Они также делают сообщения об ошибке более точными, так как сообщения будут ближе к точке ошибки во вводе. [?!?]

```
@@gramar::CALC

start
    =
    expression $
    ;

expression
    =
    | expression '+' ~ term
    | expression '-' ~ term
    | term
    ;

term
    =
    | term '*' ~ factor
    | term '/' ~ factor
    | factor
    ;

factor
    =
    | '(' ~ expression ')'
    | number
    ;

number
    =
    /\d+/
    ;
```

Теперь можно скомпилировать грамматику и протестировать парсер:

```
import json
from codecs import open
from pprint import pprint

import tatsu

def simple_parse():
    grammar = open("grammars/calc_cut.ebnf").read()

    parser = tatsu.compile(grammar)
    ast = parser.parse("3 + 5 * ( 10 - 20 )")

    print("# SIMPLE PARSE")
    print("# AST")
    pprint(ast, width=20, indent=4)
```

```
    print()

    print("# JSON")
    print(json.dumps(ast, indent=4))


def main():
    simple_parse()


if __name__ == "__main__":
    main()
```

Вывод:

```
$ PYTHONPATH=../.. python calc.py
```

```
# SIMPLE PARSE
# AST
[   '3',
    '+',
    [   '5',
        '*',
        [   '(',
            [   '10',
                '-',
                '20'],
            ')']]]

# JSON
[
    "3",
    "+",
    [
        "5",
        "*",
        [
            "(",
            [
                "10",
                "-",
                "20"
            ],
            ")"
        ]
    ]
]
```

Аннотирование грамматики [?!?]

Работа с AST, являющимися списком списков, ведёт к плохо читаемому, подверженному ошибкам коду. 竜 TatSu позволяет именовать элементы в правиле для создания более читабельных AST и создавать более чистый код семантик. Аннотированная [?!?] версия грамматики:

```
@@grammar::CALC

start
    =
    expression $
    ;

expression
    =
    | left:expression op:'+' ~ right:term
    | left:expression op:'-' ~ right:term
    | term
    ;

term
    =
    | left:term op:'*' ~ right:factor
    | left:term '/' ~ right:factor
    | factor
    ;

factor
    =
    | '(' ~ @:expression ')'
    | number
    ;

number
    =
    /\d+/
    ;
```

Выходное AST:

```
# ANNOTATED AST
{   'left': '3',
    'op': '+',
    'right': {   'left': '5',
                 'op': '*',
                 'right': {   'left': '10',
                              'op': '-',
                              'right': '20'}}}
```

Семантики [Опечатка в оригинале ?!?]

Семантики для парсеров 竜 TatSu определяются не в грамматике, а в отдельном *классе семантик*.

```
from tatsu.ast import AST

class CalcBasicSemantics(object):
    def number(self, ast):
        return int(ast)

```

```
    def term(self, ast):
        if not isinstance(ast, AST):
            return ast
        elif ast.op == "*":
            return ast.left * ast.right
        elif ast.op == "/":
            return ast.left / ast.right
        else:
            raise Exception("Unknown operator", ast.op)
     def expression(self, ast):
        if not isinstance(ast, AST):
            return ast
        elif ast.op == "+":
            return ast.left + ast.right
        elif ast.op == "-":
            return ast.left - ast.right
        else:
            raise Exception("Unknown operator", ast.op)

def parse_with_basic_semantics():
    grammar = open("grammars/calc_annotated.ebnf").read()

    parser = tatsu.compile(grammar)
    ast = parser.parse(
        "3 + 5 * ( 10 - 20 )",
        semantics=CalcBasicSemantics()
    )

    print("# BASIC SEMANTICS RESULT")
    pprint(ast, width=20, indent=4)
```

Результат:

```
# BASIC SEMANTICS RESULT
-47
```

Одно правило на выражение [?!?]

Семантические действия, определяющие что было разобрано через `isinstance()` или обращение к AST за операторами не самый «питоничный» вариант, не объектно-ориентированный, и ведёт к более сложному в управлении коду. Предпочтительнее иметь одно правило на *один вид выражения*, что будет необходимо, если мы хотим создавать объектные модели [?!?] для использования *walkers* [?!?] и генерацию кода.

```
@@grammar::CALC


start
    =
    expression $
    ;

expression
    =
    | addition
```

```
    | subtraction
    | term
    ;

addition
    =
    left:expression op:'+' ~ right:term
    ;

subtraction
    =
    left:expression op:'-' ~ right:term
    ;

term
    =
    | multiplication
    | division
    | factor
    ;

multiplication
    =
    left:term op:'*' ~ right:factor
    ;

division
    =
    left:term '/' ~ right:factor
    ;

factor
    =
    | '(' ~ @:expression ')'
    | number
    ;

number
    =
    /\d+/
    ;
```

```
class CalcSemantics(object):
    def number(self, ast):
        return int(ast)

    def addition(self, ast):
        return ast.left + ast.right

    def subtraction(self, ast):
        return ast.left - ast.right

    def multiplication(self, ast):
        return ast.left * ast.right

    def division(self, ast):
        return ast.left / ast.right
```

```python


def parse_factored():
    grammar = open("grammars/calc_factored.ebnf").read()

    parser = tatsu.compile(grammar)
    ast = parser.parse(
        "3 + 5 * ( 10 - 20 )",
        semantics=CalcSemantics()
    )

    print("# FACTORED SEMANTICS RESULT")
    pprint(ast, width=20, indent=4)
    print()
```

Реализация семантики проще и результат тот же:

```
# FACTORED SEMANTICS RESULT
-47
```

Объектные модели [?!?]

Привязывание семантики к грамматике мощный и гибкий метод, но есть риск связать семантику с *процессом анализа*, а не с *объектами*, которые анализируются.

Это не проблема для простых языков, как язык арифметических выражений в этом туториале [?!?]. Но с ростом сложности анализируемого число правил грамматики растёт быстрее, чем число типов объектов.

竜 TatSu обеспечивает создание типизированных объектных моделей непосредственно во время парсинга [?!?] и навигации (*обхода* [?!?]) и преобразования (кодогенерация) моделей в последующих проходах.

Первый шаг создания объектной модели — аннотировать [?!?] имена грамматических правил с желаемыми именами классов для объектов:

```
@@grammar::Calc

start
    =
    expression $
    ;

expression
    =
    | addition
    | subtraction
    | term
    ;

addition::Add
    =
    left:term op:'+' ~ right:expression
    ;

```

```
subtraction::Subtract
    =
    left:term op:'-' ~ right:expression
    ;

term
    =
    | multiplication
    | division
    | factor
    ;

multiplication::Multiply
    =
    left:factor op:'*' ~ right:term
    ;

division::Divide
    =
    left:factor '/' ~ right:term
    ;

factor
    =
    | subexpression
    | number
    ;

subexpression
    =
    '(' ~ @:expression ')'
    ;

number::int
    =
    /\d+/
    ;
```

Представители [?!?] `tatsu.objectmodel.Node` синтезируются в рантайме [?!?] с помощью `tatsu.semantics.ModelBuilderSemantics`.

Так выглядит модель, сгенерированная функцией `tatsu.to_python_model()`:

```
from tatsu.objectmodel import Node
from tatsu.semantics import ModelBuilderSemantics


class CalcModelBuilderSemantics(ModelBuilderSemantics):
    def __init__(self):
        types = [
            t for t in globals().values()
            if type(t) is type and issubclass(t, ModelBase)
        ]
        super().__init__(types=types)


class ModelBase(Node):
```

```
    pass


class Add(ModelBase):
    def __init__(self,
                 left=None,
                 op=None,
                 right=None,
                 **_kwargs_):
        super().__init__(
            left=left,
            op=op,
            right=right,
            **_kwargs_
        )


class Subtract(ModelBase):
    def __init__(self,
                 left=None,
                 op=None,
                 right=None,
                 **_kwargs_):
        super().__init__(
            left=left,
            op=op,
            right=right,
            **_kwargs_
        )


class Multiply(ModelBase):
    def __init__(self,
                 left=None,
                 op=None,
                 right=None,
                 **_kwargs_):
        super().__init__(
            left=left,
            op=op,
            right=right,
            **_kwargs_
        )


class Divide(ModelBase):
    def __init__(self,
                 left=None,
                 right=None,
                 **_kwargs_):
        super().__init__(
            left=left,
            right=right,
            **_kwargs_
        )
```

Модель, полученная в результате парсинга, может быть выведена [?!?] и walked [?!?]:

```
from tatsu.walkers import NodeWalker


class CalcWalker(NodeWalker):
    def walk_object(self, node):
        return node

    def walk__add(self, node):
        return self.walk(node.left) + self.walk(node.right)

    def walk__subtract(self, node):
        return self.walk(node.left) - self.walk(node.right)

    def walk__multiply(self, node):
        return self.walk(node.left) * self.walk(node.right)

    def walk__divide(self, node):
        return self.walk(node.left) / self.walk(node.right)


def parse_and_walk_model():
    grammar = open("grammars/calc_model.ebnf").read()

    parser = tatsu.compile(grammar, asmodel=True)
    model = parser.parse("3 + 5 * ( 10 - 20 )")

    print("# WALKER RESULT IS:")
    print(CalcWalker().walk(model))
```

Результат:

```
# WALKER RESULT IS:
-47
```

Генерация Кода

Трансляция — одна из наиболее часто встречающихся задач в обработке языков. Анализ часто sumarize [?!?] обработанный ввод, и walkers [?!?] хороши в этом. В трансляции выход часто может быть таким же многословным [?!?] как и вход, поэтому системный подход, максимально избегающий бухгалтерии [?!?], удобен.

竜 TatSu предоставляет поддержку генерации кода (трансляции) на шаблонах в модуле `tatsu.codegen`. Генерация кода работает с помощью определения класса трансляции [?!?] для каждого класса в модели, определённой грамматикой.

Следующий генератор кода переводит входные выражения в постфиксные инструкции стэкового [?!?] процессора:

```
from tatsu.codegen import ModelRenderer
from tatsu.codegen import CodeGenerator

THIS_MODULE =  sys.modules[__name__]


```

```
class PostfixCodeGenerator(CodeGenerator):
    def __init__(self):
        super().__init__(modules=[THIS_MODULE])


class Number(ModelRenderer):
    template = '''\
    PUSH {value}'''


class Add(ModelRenderer):
    template = '''\
    {left}
    {right}
    ADD'''


class Subtract(ModelRenderer):
    template = '''\
    {left}
    {right}
    SUB'''


class Multiply(ModelRenderer):
    template = '''\
    {left}
    {right}
    MUL'''


class Divide(ModelRenderer):
    template = '''\
    {left}
    {right}
    DIV'''
```

Кодогенератор может быть использован следующим образом:

```
from codegen import PostfixCodeGenerator


def parse_and_translate():
    grammar = open("grammars/calc_model.ebnf").read()

    parser = tatsu.compile(grammar, asmodel=True)
    model = parser.parse("3 + 5 * ( 10 - 20 )")

    postfix = PostfixCodeGenerator().render(model)

    print("# TRANSLATED TO POSTFIX")
    print(postfix)
```

Результат:

```
# TRANSLATED TO POSTFIX
PUSH 3
PUSH 5
PUSH 10
PUSH 20
SUB
MUL
ADD
```

Трассировка [?!?]

Парсинг и компиляция в 竜 TatSu имеют аргумент `trace=` (`-trace` в командной строке). При использовании опции `colorize=` (`-color` в командной строке), трассировка будет выглядеть как показано ниже, где цвета означают try, succeed и fail [?!?]. ↙start ~1:1 3 + 5 * (10 - 20) ↙expression↙start ~1:1 3 + 5 * (10 - 20) ↙expression↙expression↙start ~1:1 3 + 5 * (10 - 20) ⟲ expression↙expression↙start ~1:1 3 + 5 * (10 - 20) ↙expression↙expression↙start ~1:1 3 + 5 * (10 - 20) ⟲ expression↙expression↙start ~1:1 3 + 5 * (10 - 20) ↙term↙expression↙start ~1:1 3 + 5 * (10 - 20) ↙term↙term↙expression↙start ~1:1 3 + 5 * (10 - 20) ⟲ term↙term↙expression↙start ~1:1 3 + 5 * (10 - 20) ↙term↙term↙expression↙start ~1:1 3 + 5 * (10 - 20) ⟲ term↙term↙expression↙start ~1:1 3 + 5 * (10 - 20) ↙factor↙term↙expression↙start ~1:1 3 + 5 * (10 - 20) ≠'(' ~1:1 3 + 5 * (10 - 20) ↙number↙factor↙term↙expression↙start ~1:1 3 + 5 * (10 - 20) ≡'3' /d+/ ~1:2 + 5 * (10 - 20) ≡number↙factor↙term↙expression↙start ~1:2 + 5 * (10 - 20) ≡factor↙term↙expression↙start ~1:2 + 5 * (10 - 20) ↙term↙term↙expression↙start ~1:3 + 5 * (10 - 20) ≡term↙term↙expression↙start ~1:3 + 5 * (10 - 20) ≠'*' ~1:3 + 5 * (10 - 20) ↙term↙term↙expression↙start ~1:3 + 5 * (10 - 20) ≡term↙term↙expression↙start ~1:3 + 5 * (10 - 20) ≠'/' ~1:3 + 5 * (10 - 20) ↙factor↙term↙expression↙start ~1:3 + 5 * (10 - 20) ≠'(' ~1:3 + 5 * (10 - 20) ↙number↙factor↙term↙expression↙start ~1:3 + 5 * (10 - 20) ≠'' /d+/ ~1:3 + 5 * (10 - 20) ≠factor↙term↙expression↙start ~1:3 + 5 * (10 - 20) ≡term↙expression↙start ~1:2 + 5 * (10 - 20) ↙expression↙expression↙start ~1:3 + 5 * (10 - 20) ≡expression↙expression↙start ~1:3 + 5 * (10 - 20) ≡'+' ~1:4 5 * (10 - 20) ↙term↙expression↙start ~1:4 5 * (10 - 20) ↙term↙term↙expression↙start ~1:5 5 * (10 - 20) ⟲ term↙term↙expression↙start ~1:5 5 * (10 - 20) ↙term↙term↙expression↙start ~1:5 5 * (10 - 20) ⟲ term↙term↙expression↙start ~1:5 5 * (10 - 20) ↙factor↙term↙expression↙start ~1:5 5 * (10 - 20) ≠'(' ~1:5 5 * (10 - 20) ↙number↙factor↙term↙expression↙start ~1:5 5 * (10 - 20) ≡'5' /d+/ ~1:6 * (10 - 20) ≡number↙factor↙term↙expression↙start ~1:6 * (10 - 20) ≡factor↙term↙expression↙start ~1:6 * (10 - 20) ↙term↙term↙expression↙start ~1:7 * (10 - 20) ≡term↙term↙expression↙start ~1:7 * (10 - 20) ≡'*' ~1:8 (10 - 20) ↙factor↙term↙expression↙start ~1:8 (10 - 20) ≡'(' ~1:10 10 - 20) ↙expression↙factor↙term↙expression↙start ~1:10 10 - 20) ↙expression↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ⟲ expression↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ↙expression↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ⟲ expression↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ↙term↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ↙term↙term↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ⟲ term↙term↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ↙term↙term↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ⟲ term↙term↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ↙factor↙term↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ≠'(' ~1:11 10 - 20) ↙number↙factor↙term↙expression↙factor↙term↙expression↙start ~1:11 10 - 20) ≡'10' /d+/

~1:13 - 20) ≡number↙factor↙term↙expression↙factor↙term↙expression↙start ~1:13 - 20)
≡factor↙term↙expression↙factor↙term↙expression↙start ~1:13 - 20)
↙term↙term↙expression↙factor↙term↙expression↙start ~1:14 - 20)
≡term↙term↙expression↙factor↙term↙expression↙start ~1:14 - 20) ≠'*' ~1:14 - 20)
↙term↙term↙expression↙factor↙term↙expression↙start ~1:14 - 20)
≡term↙term↙expression↙factor↙term↙expression↙start ~1:14 - 20) ≠'/' ~1:14 - 20)
↙factor↙term↙expression↙factor↙term↙expression↙start ~1:14 - 20) ≠'(' ~1:14 - 20)
↙number↙factor↙term↙expression↙factor↙term↙expression↙start ~1:14 - 20) ≠" /d+/ ~1:14 -
20) ≠factor↙term↙expression↙factor↙term↙expression↙start ~1:14 - 20)
≡term↙expression↙factor↙term↙expression↙start ~1:13 - 20)
↙expression↙expression↙factor↙term↙expression↙start ~1:14 - 20)
≡expression↙expression↙factor↙term↙expression↙start ~1:14 - 20) ≠'+' ~1:14 - 20)
↙expression↙expression↙factor↙term↙expression↙start ~1:14 - 20)
≡expression↙expression↙factor↙term↙expression↙start ~1:14 - 20) ≡'-' ~1:15 20)
↙term↙expression↙factor↙term↙expression↙start ~1:15 20)
↙term↙term↙expression↙factor↙term↙expression↙start ~1:16 20) ⟲
term↙term↙expression↙factor↙term↙expression↙start ~1:16 20)
↙term↙term↙expression↙factor↙term↙expression↙start ~1:16 20) ⟲
term↙term↙expression↙factor↙term↙expression↙start ~1:16 20)
↙factor↙term↙expression↙factor↙term↙expression↙start ~1:16 20) ≠'(' ~1:16 20)
↙number↙factor↙term↙expression↙factor↙term↙expression↙start ~1:16 20) ≡'20' /d+/ ~1:18
) ≡number↙factor↙term↙expression↙factor↙term↙expression↙start ~1:18)
≡factor↙term↙expression↙factor↙term↙expression↙start ~1:18)
↙term↙term↙expression↙factor↙term↙expression↙start ~1:19)
≡term↙term↙expression↙factor↙term↙expression↙start ~1:19) ≠'*' ~1:19)
↙term↙term↙expression↙factor↙term↙expression↙start ~1:19)
≡term↙term↙expression↙factor↙term↙expression↙start ~1:19) ≠'/' ~1:19)
↙factor↙term↙expression↙factor↙term↙expression↙start ~1:19) ≠'(' ~1:19)
↙number↙factor↙term↙expression↙factor↙term↙expression↙start ~1:19) ≠" /d+/ ~1:19)
≠factor↙term↙expression↙factor↙term↙expression↙start ~1:19)
≡term↙expression↙factor↙term↙expression↙start ~1:18)
↙expression↙expression↙factor↙term↙expression↙start ~1:19)
≡expression↙expression↙factor↙term↙expression↙start ~1:19) ≠'+' ~1:19)
↙expression↙expression↙factor↙term↙expression↙start ~1:19)
≡expression↙expression↙factor↙term↙expression↙start ~1:19) ≠'-' ~1:19)
↙term↙expression↙factor↙term↙expression↙start ~1:19)
≠term↙expression↙factor↙term↙expression↙start ~1:19)
≡expression↙factor↙term↙expression↙start ~1:18) ≡')' ≡factor↙term↙expression↙start
↙term↙term↙expression↙start ≡term↙term↙expression↙start ≠'*'
↙term↙term↙expression↙start ≡term↙term↙expression↙start ≠'/'
↙factor↙term↙expression↙start ≠'(' ↙number↙factor↙term↙expression↙start ≠" /d+/
≠factor↙term↙expression↙start ≡term↙expression↙start ↙expression↙expression↙start
≡expression↙expression↙start ≠'+' ↙expression↙expression↙start
≡expression↙expression↙start ≠'-' ↙term↙expression↙start ≠term↙expression↙start
≡expression↙start ≡start

Совместимость с Grako

竜 TatSu регулярно тестируется на крупных проектах, разработанных на Grako. Пакет обратной совместимости включает (как минимум) трансляторы для COBOL,

, Java и (Oracle) SQL.

Грамматики и проекты Grako могут быть использованы в 竜 TatSu при следующих предостережениях:

- Имя модуля Python изменено на `tatsu`.
- `ignorecase` не применяется к регулярным выражениям в грамматиках. Используйте `(?i)` в паттерне для использования `re.IGNORECASE`.
- Левая рекурсия включена по умолчанию, так как она работает и не влияет на нерекурсивные грамматики.
- Устаревший синтаксис грамматики более не документируется. Лучший вариант — не использовать его, так как он будет удален в будущих версиях 竜 TatSu.

Использование Грамматик ANTLR

ANTLR один из наиболее известных генераторов парсеров, и имеет важную коллекцию грамматик. Модуль `tatsu.g2e` может переводить грамматики ANTLR в синтаксис, используемый 竜 TatSu.

Результирующая грамматика ещё не готова к использованию. Она требует редактирования для соответствия PEG-семантикам и в целом адаптации к работе 竜 TatSu.

Для использования `g2e` в качестве модуля, используйте одну из транслирующих [?!?] функций.

```
def translate(text=None, filename=None, name=None, encoding="utf-8",
trace=False):
```

Например:

```
from tatsu import g2e

tatsu_grammar = translate(filename="mygrammar.g", name="My")
with open("my.ebnf") as f:
    f.write(tatsu_grammar)
```

Также `g2e` можно использовать в командной строке:

```
$ python -m tatsu.g2e mygrammar.g > my.ebnf
```

Примеры

Tatsu

Файл `grammar/tatsu.ebnf` содержит грамматику для языка грамматики 竜 TatSu, написанную на собственном языке программирования. Она используется в наборе тестов *bootstrap* [?!?] для доказательства того, что 竜 TatSu может сгенерировать парсер, анализирующий собственный язык, и результирующий парсер создаёт bootstrap [?!?]

парсер каждый раз, когда 竜 TatSu stable [?!?] (см. `tatsu/bootstrap.py` для сгенерированного парсера).

竜 TatSu использует 竜 TatSu для перевода грамматик в парсеры, что является хорошим примером end-to-end трансляции.

Calc

Проект `examples/calc` реализует калькулятор для простых выражений, и создан в качестве руководства по большинству возможностей, предоставляемых 竜 TatSu.

---

g2e

Проект `examples/g2e` содержит пример перевода грамматики ANTLR в грамматику 竜 TatSu. Это хороший пример применения `g2e`. Он генерирует грамматику 竜 TatSu on standart output [?!?], но поскольку модель используется самим 竜 TatSu, тот же код может быть использован для непосредственного генерирования из любой грамматики ANTLR. Пожалуйста, ознакомьтесь с примерами в *README*, чтобы узнать об ограничениях.

Поддержка

Для общих Q&A [вопросов ?!?], пожалуйста, используйте тэг `tatsu` на StackOverflow.

Благодарности

- 竜 TatSu преемник Grako, созданного **Juancarlo Añez** и финансируемого **Thomas Bragg**, для анализа и трансляции программ, написанных на устаревших языках программирования.
- **Niklaus Wirth** был главным дизайнером языков Euler, Algol W, Pascal, Modula, Modula-2, Oberon и Oberon-2. В последней главе его книги 1976 года Algorithms + Data Structures = Programs, Вирт создал нисходящий парсер с recovery [?!?] для Паскалеподобного, LL(1) языка программирования PL/0. Структура программы была PEG-парсером, в то время как PEG ещё не был формализован до 2004 года.
- **Bryan Ford**, описавший PEG (parsing expression grammars) в 2004.
- Другие генераторы парсеров, как PEG.js **David Majda**, вдохновивший на создание 竜 TatSu.
- **William Thompson**, вдохновивший [?!?] на использование менеджеров контекста своим постом в блоге, о котором я узнал из бесценной новостной рассылки Python Weekly, курируемой **Rahul Chaudhary**.
- **Jeff Knupp** объяснил почему использование исключений 竜 TatSu звучит [круто ?!?] вместо меня.
- **Terence Parr** создал ANTLR, возможно самый цельный и профессиональный парсер. *Ter*, *ANTLR* и парни на форуме ANTLR помогли мне оформить идеи относительно 竜 TatSu.
- **JavaCC** (в оригинале Jack) выглядит как заброшенный проект. Это был первый генератор парсеров, который я использовал для обучения.

- **竜 TatSu** очень быстрый. Но работа с миллионами строк legacy [?!?] кода за минуты было бы невозможным без PyPy, работы **Armin Rigo** и команды PyPy.
- **Guido van Rossum** создал и возглавлял разработку Python больше десятилетия. Такой инструмент как 竜 TatSu, размером менее 10 тысяч строк кода, невозможно было бы создать без Python.
- **Kota Mizushima** пригласил меня в CSAIL at MIT MIT PEG and Packart parsing mailing list, и сразу предложил идеи и посоветовал мне документацию для реализации вырезок [?!?] в современных парсерах. Спасибо за информацию по оптимизации мемоизации в 竜 TatSu в одной из его статей.
- Мои студенты в UCAB вдохновили меня задуматься о том, как grammar-based [?!?] генераторы парсеров могут быть реализованы более доступно.
- **Manuel Rey** провёл меня через ещё один незаконченный дипломный проект, который научил меня тому, что такое языки (разговорные языки в целом и языки программирования в частности). Я узнал, почему языки используют склонения и почему, вопреки тому, что основные слова написаны на английском, структура программ, которые мы пишем, больше похожа на японский.
- Marcus Brinkmann любезно представил патчи, которые исправили неявные ошибки в реализации 竜 TatSu и сделали инструмент более удобным для пользователя, особенно для новичков в парсинге и трансляции.
- Robert Speer убрал бессмыслицу [?!?] в попытках обеспечить совместимость обработки Unicode с 2.7.x и 3.x, и выяснил канонический способ обработки escape-последовательностей [?!?] в грамматических токенах без исключения [?!?] кодировки.
- Besel Shishani был невероятно проницательным рецензентом [?!?] 竜 TatSu.
- Paul Sargent реализовал алгоритм Wart et al для поддержки прямой и непрямой левой рекурсии в PEG-парсерах.
- Kathryn Long предложила [?!?] улучшить поддержку Unicode при обработке пробелов и регулярных выражений (patterns [?!?]) в целом. Другие её вклады сделали 竜 TatSu более конгруэнтным [?!?] и удобным для пользователя.
- David Röthlisberger предоставил окончательный патч, позволяющий использовать ключевые слова Python в качестве имён правил.
- Nicolas Laurent изучил, спроектировал, реализовал и опубликовал алгоритм левой рекурсии, используемый в 竜 TatSu.
- Vic Nightfall спроектировал и написал реализацию левой рекурсии, которая обрабатывает все интересующие случаи использования (см. Левая рекурсия для деталей). Он был достаточно добр, чтобы любезно взять на себя управление проектом 竜 TatSu с 2019 года.

Контрибьюторы [?!?]

Следующие, среди прочих, внесли свой вклад в 竜 TatSu дополнительным функциями, отчётами об ошибках, исправлениями ошибок или предложениями:

Alberto Berti, Andy Wright, Basel Shishani, David Chen, David Delassus, David Röthlisberger,David Sanders, Dmytro Ivanov, Felipe, Franck Pommereau, Franklin Lee, Gabriele Paganelli, Guido van Rossum, Jack Taylor, Kathryn Long, Karthikeyan Singaravelan, Manuel Jacob, Marcus Brinkmann, Mark Jason Dominus, Max Liebkies, Michael Noronha, Nicholas Bishop, Nicolas Laurent, Nils-Hero Lindemann, Paul Houle, Paul Sargent, Robert Speer, Ryan, Ryan Gonzales, Ruth-Polymnia, S Brown, Tonico Strasser, Vic Nightfall, Victor Uriarte, Vinay Sajip, franz_g, gkimbar,nehz , neumond, pdw-mb, pgebhard, siemer.

Содействие [?!?]

Разработка 竜 TatSu осуществляется на Github. Отчёты об ошибках, патчи, предложения и улучшения приветствуются.

Пожертвования

Donate

Если Вы хотите внести свой вклад в будущее развитие 竜 TatSu, пожалуйста, сделайте пожертвование проекту.

Некоторые из запланированных функций: грамматические выражения для левой и правой ассоциативностей, новые алгоритмы для левой рекурсии, унифицированная промежуточная модель для парсинга и трансляции языков программирования и многое другое.

Лицензия
