TatSu Documentation

Release 5.10.0

Juancarlo Añez

Nov 09, 2023

CONTENTS

1 Introduction **3**

2 Rationale **5**

3 Installation **7**

4 Using the Tool **9**

4.1 As a Library 9
4.2 Compiling grammars to Python 11
4.3 The Generated Parsers 12
4.4 Using the Generated Parser 13

5 Grammar Syntax **15**

5.1 Rules 15
5.2 Expressions 15
5.3 Rules with Arguments 23
5.4 Based Rules 23
5.5 Memoization 24
5.6 Rule Overrides 24
5.7 Grammar Name 24
5.8 Whitespace 25
5.9 Case Sensitivity 25
5.10 Comments 25
5.11 Reserved Words and Keywords 26
5.12 Include Directive 26
5.13 Left Recursion 27

6 Grammar Directives **29**

6.1 `@@grammar :: <word>` 29
6.2 `@@comments :: <regexp>` 29
6.3 `@@eol_comments :: <regexp>` 29
6.4 `@@ignorecase :: <bool>` 30
6.5 `@@keyword :: {<word>|<string>}+` 30
6.6 `@@left_recursion :: <bool>` 30
6.7 `@@namechars :: <string>` 30
6.8 `@@nameguard :: <bool>` 30
6.9 `@@parseinfo :: <bool>` 30
6.10 `@@whitespace :: <regexp>` 31

7 Abstract Syntax Trees (ASTs) **33**

8 Semantic Actions **35**

9 Building Models **37**
9.1 Walking Models 37
9.2 Model Class Hierarchies 38

10 Viewing Models as JSON **41**

11 Print Translation **43**

12 Declarative Translation **45**

13 Left Recursion **47**

14 *Calc* Mini Tutorial **49**
14.1 The initial grammar 49
14.2 The Tatsu grammar 49
14.3 Add *cut* expressions 50
14.4 Annotating the grammar 52
14.5 Semantics 53
14.6 One rule per expression type 55
14.7 Object models 57
14.8 Code Generation 60

15 Traces **63**

16 Grako Compatibility **69**

17 Using ANTLR Grammars **71**

18 Examples **73**
18.1 Tatsu 73
18.2 Calc 73
18.3 g2e 73

19 Support **75**

20 Credits **77**

21 Contributors **79**

22 Contributing **81**
22.1 Donations 81

23 License **83**

> *At least for the people who send me mail about a new language that they're designing, the general advice is: do it to learn about how to write a compiler. Don't have any expectations that anyone will use it, unless you hook up with some sort of organization in a position to push it hard. It's a lottery, and some can buy a lot of the tickets. There are plenty of beautiful languages (more beautiful than C) that didn't catch on. But someone does win the lottery, and doing a language at least teaches you something.*
>
> Dennis Ritchie (1941-2011) Creator of the C programming language and of Unix

TatSu is a tool that takes grammars in a variation of EBNF as input, and outputs memoizing (Packrat) PEG parsers in Python.

Why use a PEG parser? Because regular languages (those parsable with Python's `re` package) *"cannot count"*. Any language with nested structures or with balancing of demarcatiors requires more than regular expressions to be parsed.

TatSu can compile a grammar stored in a string into a `tatsu.grammars.Grammar` object that can be used to parse any given input, much like the re module does with regular expressions, or it can generate a Python module that implements the parser.

TatSu supports left-recursive rules in PEG grammars, and it honors *left-associativity* in the resulting parse trees.

CHAPTER ONE

INTRODUCTION

TatSu is *different* from other PEG parser generators:

- Generated parsers use Python's very efficient exception-handling system to backtrack. **TatSu** generated parsers simply assert what must be parsed. There are no complicated *if-then-else* sequences for decision making or backtracking. Memoization allows going over the same input sequence several times in linear time.
- *Positive and negative lookaheads*, and the *cut* element (with its cleaning of the memoization cache) allow for additional, hand-crafted optimizations at the grammar level.
- Delegation to Python's re module for *lexemes* allows for (Perl-like) powerful and efficient lexical analysis.
- The use of Python's context managers considerably reduces the size of the generated parsers for code clarity, and enhanced CPU-cache hits.
- Include files, rule inheritance, and rule inclusion give **TatSu** grammars considerable expressive power.
- Automatic generation of Abstract Syntax Trees_ and Object Models, along with *Model Walkers* and *Code Generators* make analysis and translation approachable

The parser generator, the run-time support, and the generated parsers have measurably low Cyclomatic complexity. At around 5 KLOC of Python, it is possible to study all its source code in a single session.

The only dependencies are on the Python standard library, yet the regex library will be used if installed, and colorama will be used on trace output if available. pygraphviz is required for generating diagrams.

TatSu is feature-complete and currently being used with complex grammars to parse, analyze, and translate hundreds of thousands of lines of input text, including source code in several programming languages.

CHAPTER TWO

RATIONALE

TatSu was created to address some recurring problems encountered over decades of working with parser generation tools:

- Some programming languages allow the use of *keywords* as identifiers, or have different meanings for symbols depending on context (Ruby). A parser needs control of lexical analysis to be able to handle those languages.
- LL and LR grammars become contaminated with myriads of lookahead statements to deal with ambiguous constructs in the source language. PEG parsers address ambiguity from the onset.
- Separating the grammar from the code that implements the semantics, and using a variation of a well-known grammar syntax (EBNF) allows for full declarative power in language descriptions. General-purpose programming languages are not up to the task.
- Semantic actions *do not* belong in a grammar. They create yet another programming language to deal with when doing parsing and translation: the source language, the grammar language, the semantics language, the generated parser's language, and the translation's target language. Most grammar parsers do not check the syntax of embedded semantic actions, so errors get reported at awkward moments, and against the generated code, not against the grammar.
- Preprocessing (like dealing with includes, fixed column formats, or structure-through-indentation) belongs in well-designed program code; not in the grammar.
- It is easy to recruit help with knowledge about a mainstream programming language like Python, but help is hard to find for working with complex grammar-description languages. **TatSu** grammars are in the spirit of a *Translators and Interpreters 101* course (if something is hard to explain to a college student, it's probably too complicated, or not well understood).
- Generated parsers should be easy to read and debug by humans. Looking at the generated source code is sometimes the only way to find problems in a grammar, the semantic actions, or in the parser generator itself. It's inconvenient to trust generated code that one cannot understand.
- Python is a great language for working with language parsing and translation.

CHAPTER

THREE

INSTALLATION

```
$ pip install tatsu
```

Warning: Versions of **TatSu** since 5.0.0 may require Python>=3.8. Python 2.7 is no longer supported

CHAPTER FOUR

USING THE TOOL

4.1 As a Library

TatSu can be used as a library, much like Python's `re`, by embedding grammars as strings and generating grammar models instead of generating Python code.

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

  Compiles the grammar and generates a *model* that can subsequently be used for parsing input with.

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

  Compiles the grammar and parses the given input producing an AST as result. The result is equivalent to calling:

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

  Compiled grammars are cached for efficiency.

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

  Compiles the grammar to the Python sourcecode that implements the parser.

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

  Compiles the grammar and generates the Python sourcecode that implements the object model defined by rule annotations.

This is an example of how to use **Tatsu** as a library:

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

    start = expression $ ;

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

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

(continues on next page)

(continued from previous page)

```
        | 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()
```

And this is the output:

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

JSON
[
  "3",
  "+",
  [
    "5",
    "*",
    [
```

(continues on next page)

(continued from previous page)

```
      "10",
      "-",
      "20"
    ]
  ]
]
```

4.2 Compiling grammars to Python

Tatsu can be run from the command line:

```
$ python -m tatsu
```

Or:

```
$ scripts/tatsu
```

Or just:

```
$ tatsu
```

if **Tatsu** was installed using *easy_install* or *pip*.

The *-h* and *–help* parameters provide full usage information:

```
$ 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
```

(continues on next page)

(continued from previous page)

```
                        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
$
```

4.3 The Generated Parsers

A **Tatsu** generated parser consists of the following classes:

- A `MyLanguageBuffer` class derived from `tatsu.buffering.Buffer` that handles the grammar definitions for *whitespace*, *comments*, and *case significance*.
- A `MyLanguageParser` class derived from `tatsu.parsing.Parser` which uses a `MyLanguageBuffer` for traversing input text, and implements the parser using one method for each grammar rule:

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

- A `MyLanguageSemantics` class with one semantic method per grammar rule. Each method receives as its single parameter the Abstract Syntax Tree (AST) built from the rule invocation:

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

- A `if __name__ == '__main__':` definition, so the generated parser can be executed as a Python script.

The methods in the delegate class return the same AST received as parameter, but custom semantic classes can override the methods to have them return anything (for example, a Semantic Graph). The semantics class can be used as a template for the final semantics implementation, which can omit methods for the rules that do not need semantic treatment.

If present, a `_default()` method will be called in the semantics class when no method matched the rule name:

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

If present, a `_postproc()` method will be called in the semantics class after each rule (including the semantics) is processed. This method will receive the current parsing context as parameter:

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

4.4 Using the Generated Parser

To use the generated parser, just subclass the base or the abstract parser, create an instance of it, and invoke its `parse()` method passing the grammar to parse and the starting rule's name as parameter:

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

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

The generated parsers' constructors accept named arguments to specify whitespace characters, the regular expression for comments, case sensitivity, verbosity, and more (see below).

To add semantic actions, just pass a semantic delegate to the parse method:

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

If special lexical treatment is required (as in *80 column* languages), then a descendant of `tatsu.tokenizing.Tokenizer` can be passed instead of the text:

```
class MySpecialTokenizer(Tokenizer):
    ...

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

The generated parser's module can also be invoked as a script:

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

As a script, the generated parser's module accepts several options:

```
$ 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
```

CHAPTER

FIVE

GRAMMAR SYNTAX

TatSu uses a variant of the standard EBNF syntax. Syntax definitions for VIM and for Sublime Text can be found under the `etc/vim` and `etc/sublime` directories in the source code distribution.

5.1 Rules

A grammar consists of a sequence of one or more rules of the form:

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

If a *name* collides with a Python keyword, an underscore (_) will be appended to it on the generated parser.

Rule names that start with an uppercase character:

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

do not advance over whitespace before beginning to parse. This feature becomes handy when defining complex lexical elements, as it allows breaking them into several rules.

The parser returns an AST value for each rule depending on what was parsed:

- A single value
- A list of AST
- A dict-like object for rules with named elements
- An object, when ModelBuilderSemantics is used
- None

See the Abstract Syntax Trees and Building Models sections for more details.

5.2 Expressions

The expressions, in reverse order of operator precedence, can be:

5.2.1 `# comment`

Python-style comments are allowed.

5.2.2 `e1 | e2`

Choice. Match either `e1` or `e2`.

A | may be used before the first option if desired:

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

5.2.3 `e1 e2`

Sequence. Match `e1` and then match `e2`.

5.2.4 `( e )`

Grouping. Match `e`. For example: `('a' | 'b')`.

5.2.5 `[ e ]`

Optionally match `e`.

5.2.6 `{ e }` or `{ e }*`

Closure. Match `e` zero or more times. The AST returned for a closure is always a `list`.

5.2.7 `{ e }+`

Positive closure. Match `e` one or more times. The AST is always a `list`.

5.2.8 `{}`

Empty closure. Match nothing and produce an empty `list` as AST.

5.2.9 `~`

The *cut* expression. Commit to the current active option and prevent other options from being considered even if what follows fails to parse.

In this example, other options won't be considered if a parenthesis is parsed:

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

There are also options in optional expressions, because `[foo]` is equivalent to `(foo|())`.

There are options also in closures, because of a similar equivalency, so the following rule will fail if `expression` is not parsed after an = is parsed, while the version without the ~ would succeed over a partial parse of the `name '=' expression` ahead in the input:

```
parameters
    =
    ','.{name '=' ~ expression}
    ;
```

5.2.10 `s%{ e }+`

Positive join. Inspired by Python's `str.join()`, it parses the same as this expression:

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

yet the result is a single list of the form:

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

Use grouping if *s* is more complex than a *token* or a *pattern*:

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

5.2.11 `s%{ e }` or `s%{ e }*`

Join. Parses the list of `s`-separated expressions, or the empty closure. It is equivalent to:

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

5.2.12 `op<{ e }+`

Left join. Like the *join expression*, but the result is a left-associative tree built with `tuple()`, in wich the first element is the separator (`op`), and the other two elements are the operands.

The expression:

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

Will parse this input:

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

To this tree:

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

5.2.13 `op>{ e }+`

Right join. Like the *join expression*, but the result is a right-associative tree built with `tuple()`, in wich the first element is the separator (`op`), and the other two elements are the operands.

The expression:

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

Will parse this input:

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

To this tree:

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

(continues on next page)

(continued from previous page)

```
        )
    )
)
```

5.2.14 `s.{ e }+`

Positive *gather*. Like *positive join*, but the separator is not included in the resulting AST.

5.2.15 `s.{ e }` or `s.{ e }*`

Gather. Like the *join*, but the separator is not included in the resulting AST. It is equivalent to:

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

5.2.16 `&e`

Positive lookahead. Succeed if `e` can be parsed, but do not consume any input.

5.2.17 `!e`

Negative lookahead. Fail if `e` can be parsed, and do not consume any input.

5.2.18 `'text'` or `"text"`

Match the token *text* within the quotation marks.

Note that if *text* is alphanumeric, then **TatSu** will check that the character following the token is not alphanumeric. This is done to prevent tokens like *IN* matching when the text ahead is *INITIALIZE*. This feature can be turned off by passing `nameguard=False` to the `Parser` or the `Buffer`, or by using a pattern expression (see below) instead of a token expression. Alternatively, the `@@nameguard` or `@@namechars` directives may be specified in the grammar:

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

or to specify additional characters that should also be considered part of names:

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

5.2.19 `r'text'` or `r"text"`

Match the token *text* within the quotation marks, interpreting *text* like Python's raw string literals.

5.2.20 `?"regexp"` or `?'regexp'` or `/regexp/`

The *pattern* expression. Match the Python regular expression `regexp` at the current text position. Unlike other expressions, this one does not advance over whitespace or comments. For that, place the `regexp` as the only term in its own rule.

The *regex* is interpreted as a Python raw string literal and passed the Python re module using `match()` at the current position in the text. The returned AST has the semantics of `re.findall(pattern, text)[0]` (a *tuple* if there is more than one group), so use `(?:)` for groups that should not be in the resulting AST.

Consecutive *patterns* are concatenated to form a single one.

5.2.21 `/./`

The *any* expression, matches the next position in the input. It works exactly like the `?'.'` pattern, but is implemented at the lexical level, without regular expressions.

5.2.22 `->e`

The "*skip to*" expression; useful for writing *recovery* rules.

The parser will advance over input, one character at time, until `e` matches. Whitespace and comments will be skipped at each step. Advancing over input is done efficiently, with no regular expressions are involved.

The expression is equivalent to:

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

A common form of the expression is `->&e`, which is equivalent to:

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

This is an example of the use of the "*skip to*" expression for recovery:

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

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

statement_recovery = ->&statement ;
```

5.2.23 `` `constant` ``

Match nothing, but behave as if `constant` had been parsed.

Constants can be used to inject elements into the concrete and abstract syntax trees, perhaps avoiding having to write a semantic action. For example:

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

If the text evaluates to a Python literal (with `ast.literal_eval()`), that will be the returned value. Otherwise, string interpolation in the style of `str.format()` over the names in the current AST is applied for *constant* elements. Occurrences of the { character must be scaped to \{ if they are not intended for interpolation. A *constant* expression that hast type `str` is evaluated using:

```
eval(f'{"f" + repr(text)}', {}, ast)
```

5.2.24 ```` ```constant``` ````

A multiline version of `` `constant` ``.

5.2.25 ^ `` `constant` `` and ^ ```` ```constant``` ````

An alert. There will be no token returned by the parser, but an alert will be registed in the parse context and added to the current node's `parseinfo`.

The ^ character may appear more than once to indicate the alert level:

```
assignment = identifier '=' (
    | value
    | ->'&; ^^^`could not parse value in assignment to {identifier}`
```

5.2.26 `rulename`

Invoke the rule named `rulename`. To help with lexical aspects of grammars, rules with names that begin with an uppercase letter will not advance the input over whitespace or comments.

5.2.27 `>rulename`

The include operator. Include the *right hand side* of rule `rulename` at this point.

The following set of declarations:

```
includable = exp1 ;

expanded = exp0 >includable exp2 ;
```

Has the same effect as defining *expanded* as:

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

Note that the included rule must be defined before the rule that includes it.

5.2.28 `()`

The empty expression. Succeed without advancing over input. Its value is `None`.

5.2.29 `!()`

The *fail* expression. This is actually `!` applied to `()`, which always fails.

5.2.30 `name:e`

Add the result of `e` to the AST using `name` as key. If `name` collides with any attribute or method of `dict`, or is a Python keyword, an underscore (`_`) will be appended to the name.

When there are no named items in a rule, the AST consists of the elements parsed by the rule, either a single item or a `list`. This default behavior makes it easier to write simple rules:

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

Without having to write:

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

When a rule has named elements, the unnamed ones are excluded from the AST (they are ignored).

5.2.31 `name+:e`

Add the result of `e` to the AST using `name` as key. Force the entry to be a `list` even if only one element is added. Collisions with `dict` attributes or Python keywords are resolved by appending an underscore to `name`.

5.2.32 `@:e`

The override operator. Make the AST for the complete rule be the AST for `e`.

The override operator is useful to recover only part of the right hand side of a rule without the need to name it, or add a semantic action. This is a typical use of the override operator:

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

The AST returned for the `subexp` rule will be the AST recovered from invoking `expre`.

5.2.33 `@+:e`

Like `@:e`, but make the AST always be a `list`.

This operator is convenient in cases such as:

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

In which the delimiting tokens are of no interest.

5.2.34 `$`

The *end of text* symbol. Verify that the end of the input text has been reached.

5.3 Rules with Arguments

TatSu allows rules to specify Python-style arguments:

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

The arguments values are fixed at grammar-compilation time. An alternative syntax is available if no *keyword parameters* are required:

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

Semantic methods must be ready to receive any arguments declared in the corresponding rule:

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

When working with rule arguments, it is good to define a `_default()` method that is ready to take any combination of standard and keyword arguments:

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

5.4 Based Rules

Rules may extend previously defined rules using the < operator. The *base rule* must be defined previously in the grammar.

The following set of declarations:

```
base::Param = exp1 ;

extended < base = exp2 ;
```

Has the same effect as defining *extended* as:

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

Parameters from the *base rule* are copied to the new rule if the new rule doesn't define its own. Repeated inheritance should be possible, but it *hasn't been tested*.

5.5 Memoization

TatSu is a packrat parser. The result of parsing a rule at a given position in the input is cached, so the next time the parser visits the same input position with the same rule the same result is returned and the input advanced, without repeating the parsing. Memoization allows for grammars that are clearer and easier to write because there's no fear that repeating subexpressions will impact performance.

There are rules that should not be memoized. For example, rules that may succeed or not depending on the associated semantic action should not be memoized if sucess depends on more than just the input.

The @nomemo decorator turns off memoization for a particular rule:

```
@nomemo
INDENT = () ;

@nomemo
DEDENT = () ;
```

5.6 Rule Overrides

A grammar rule may be redefined by using the @override decorator:

```
start = ab $;

ab = 'xyz' ;

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

When combined with the #include directive, rule overrides can be used to create a modified grammar without altering the original.

5.7 Grammar Name

The prefix to be used in classes generated by **TatSu** can be passed to the command-line tool using the -m option:

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

will generate:

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

The name can also be specified within the grammar using the @@grammar directive:

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

5.8 Whitespace

By default, **TatSu** generated parsers skip the usual whitespace characters with the regular expression `r'\s+'` using the `re.UNICODE` flag (or with the `Pattern_White_Space` property if the regex module is available), but you can change that behavior by passing a `whitespace` parameter to your parser.

For example, the following will skip over *tab* (`\t`) and *space* characters, but not so with other typical whitespace characters such as *newline* (`\n`):

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

The character string is converted into a regular expression character set before starting to parse.

You can also provide a regular expression directly instead of a string. The following is equivalent to the above example:

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

Note that the regular expression must be pre-compiled to let **TatSu** distinguish it from plain string.

If you do not define any whitespace characters, then you will have to handle whitespace in your grammar rules (as it's often done in PEG parsers):

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

Whitespace may also be specified within the grammar using the `@@whitespace` directive, although any of the above methods will overwrite the setting in the grammar:

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

5.9 Case Sensitivity

If the source language is case insensitive, it can be specified in the parser by using the `ignorecase` parameter:

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

You may also specify case insensitivity within the grammar using the `@@ignorecase` directive:

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

The change will affect token matching, but not pattern matching. Use *(?i)* in patterns that should ignore case.

5.10 Comments

Parsers will skip over comments specified as a regular expression using the `comments_re` parameter:

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

For more complex comment handling, you can override the `Buffer.eat_comments()` method.

For flexibility, it is possible to specify a pattern for end-of-line comments separately:

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

Both patterns may also be specified within a grammar using the `@@comments` and `@@eol_comments` directives:

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

5.11 Reserved Words and Keywords

Some languages must reserve the use of certain tokens as valid identifiers because the tokens are used to mark particular constructs in the language. Those reserved tokens are known as Reserved Words or Keywords

TatSu provides support for preventing the use of keywords as identifiers though the `@@ keyword` directive,and the `@ name` decorator.

A grammar may specify reserved tokens providing a list of them in one or more `@@ keyword` directives:

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

The `@ name` decorator checks that the result of a grammar rule does not match a token defined as a keyword:

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

There are situations in which a token is reserved only in a very specific context. In those cases, a negative lookahead will prevent the use of the token:

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

5.12 Include Directive

TatSu grammars support file inclusion through the include directive:

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

The resolution of the *filename* is relative to the directory/folder of the source. Absolute paths and `../` navigations are honored.

The functionality required for implementing includes is available to all **TatSu**-generated parsers through the `Buffer` class; see the `EBNFBuffer` class in the `tatsu.parser` module for an example.

5.13 Left Recursion

TatSu supports left recursion in PEG grammars. The algorithm used is Warth et al's.

Sometimes, while debugging a grammar, it is useful to turn left-recursion support on or off:

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

Left recursion can also be turned off from within the grammar using the `@@left_recursion` directive:

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

CHAPTER

SIX

GRAMMAR DIRECTIVES

TatSu allows *directives* in the grammar that control the behavior of the generated parsers. All directives are of the form `@@name :: <value>`. For example:

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

The *directives* supported by **TatSu** are described below.

6.1 `@@grammar :: <word>`

Specifies the name of the grammar, and provides the base name for the classes in parser source-code generation.

6.2 `@@comments :: <regexp>`

Specifies a regular expression to identify and exclude inline (bracketed) comments before the text is scanned by the parser. For `(* ... *)` comments:

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

6.3 `@@eol_comments :: <regexp>`

Specifies a regular expression to identify and exclude end-of-line comments before the text is scanned by the parser. For `# ...` comments:

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

6.4 @@ignorecase :: <bool>

If set to True makes **TatSu** not consider case when parsing tokens. Defaults to False:

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

6.5 @@keyword :: {<word>|<string>}+

Specifies the list of strings or words that the grammar should consider as *"keywords"*. May appear more than once. See the Reserved Words and Keywords section for an explanation.

6.6 @@left_recursion :: <bool>

Enables left-recursive rules in the grammar. See the Left Recursion sections for an explanation.

6.7 @@namechars :: <string>

A list of (non-alfanumeric) characters that should be considered part of names when using the *@@nameguard* feature:

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

6.8 @@nameguard :: <bool>

When set to True, avoids matching tokens when the next character in the input sequence is alfarnumeric or a @@namechar. Defaults to True. See the 'text' expression for an explanation.

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

6.9 @@parseinfo :: <bool>

When True, the parser will add parse information to every AST and Node generated by the parse under a parseinfo field. The information will include:

- rule the rule name that parsed the node
- pos the initial position for the node in the input
- endpos the final position for the node in the input
- line the initial input line number for the element
- endline the final line number for the element

Enabling @@parseinfo will allow precise reporting over the input source-code while performing semantic actions.

6.10 `@@whitespace :: <regexp>`

Provides a regular expression for the whitespace to be ignored by the parser. It defaults to `/(?s)\s+/`:

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

CHAPTER SEVEN

ABSTRACT SYNTAX TREES (ASTS)

By default, an AST is either:

- a *value*, for simple elements such as *token*, *pattern*, or *constant*
- a `tuple`, for *closures*, *gatherings*, and the right-hand-side of rules with more than one element but without named elements
- a `dict`-derived object (`AST`) that contains one item for every named element in the grammar rule, with items can be accessed through the standard `dict` syntax (`ast['key']`), or as attributes (`ast.key`).

AST entries are single values if only one item was associated with a name, or `tuple` if more than one item was matched. There's a provision in the grammar syntax (the `+:` operator) to force an AST entry to be a `tuple` even if only one element was matched. The value for named elements that were not found during the parse (perhaps because they are optional) is `None`.

When the `parseinfo=True` keyword argument has been passed to the `Parser` constructor or enabled with the `@@parseinfo` directive, a `parseinfo` item is added to AST nodes that are *dict*-like. The item contains a `collections.namedtuple` with the parse information for the node:

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

With the help of the `Tokenizer.line_info()` method, it is possible to recover the line, column, and original text parsed for the node. Note that when `ParseInfo` is generated, the `Tokenizer` used during parsing is kept in memory for the lifetime of the AST.

Generation of `parseinfo` can also be controlled using the `@@parseinfo :: True` grammar directive.

CHAPTER EIGHT

SEMANTIC ACTIONS

There are no constructs for semantic actions in **TatSu** grammars. This is on purpose, because semantic actions obscure the declarative nature of grammars and provide for poor modularization from the parser-execution perspective.

Semantic actions are defined in a class, and applied by passing an object of the class to the `parse()` method of the parser as the `semantics=` parameter. **TatSu** will invoke the method that matches the name of the grammar rule every time the rule parses. The argument to the method will be the AST constructed from the right-hand-side of the rule:

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

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

If there's no method matching the rule's name, **TatSu** will try to invoke a `_default()` method if it's defined:

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

Nothing will happen if neither the per-rule method nor `_default()` are defined.

The per-rule methods in classes implementing the semantics provide enough opportunity to do rule post-processing operations, like verifications (for inadequate use of keywords as identifiers), or AST transformation:

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

For finer-grained control it is enough to declare more rules, as the impact on the parsing times will be minimal.

If preprocessing is required at some point, it is enough to place invocations of empty rules where appropriate:

```
myrule = first_part preproc {second_part} ;

preproc = () ;
```

The abstract parser will honor as a semantic action a method declared as:

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

CHAPTER

NINE

BUILDING MODELS

Naming elements in grammar rules makes the parser discard uninteresting parts of the input, like punctuation, to produce an *Abstract Syntax Tree* (AST) that reflects the semantic structure of what was parsed. But an AST doesn't carry information about the rule that generated it, so navigating the trees may be difficult.

TatSu defines the `tatsu.model.ModelBuilderSemantics` semantics class which helps construct object models from abtract syntax trees:

```
from tatsu.model import ModelBuilderSemantics

parser = MyParser(semantics=ModelBuilderSemantics())
```

Then you add the desired node type as first parameter to each grammar rule:

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

`ModelBuilderSemantics` will synthesize a `class AddOperator(Node):` class and use it to construct the node. The synthesized class will have one attribute with the same name as the named elements in the rule.

You can also use Python's built-in types as node types, and `ModelBuilderSemantics` will do the right thing:

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

`ModelBuilderSemantics` acts as any other semantics class, so its default behavior can be overidden by defining a method to handle the result of any particular grammar rule.

9.1 Walking Models

The class `tatsu.model.NodeWalker` allows for the easy traversal (*walk*) a model constructed with a `ModelBuilderSemantics` instance:

```
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)

```

(continues on next page)

(continued from previous page)

```
model = MyParser(semantics=ModelBuilderSemantics()).parse(input)

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

When a method with a name like `walk_AddOperator()` is defined, it will be called when a node of that type is *walked*. The *pythonic* version of the class name may also be used for the *walk* method: `walk__add_operator()` (note the double underscore).

If a *walk* method for a node class is not found, then a method for the class's bases is searched, so it is possible to write *catch-all* methods such as:

```
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__)
```

Predeclared classes can be passed to `ModelBuilderSemantics` instances through the `types=` parameter:

```
from mymodel import AddOperator, MulOperator

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

`ModelBuilderSemantics` assumes nothing about `types=`, so any constructor (a function, or a partial function) can be used.

9.2 Model Class Hierarchies

It is possible to specify a a base class for generated model nodes:

```
additive
    =
    | addition
    | substraction
    ;

addition::AddOperator::Operator
    =
    left:mulexpre op:'+' right:additive
    ;

substraction::SubstractOperator::Operator
    =
    left:mulexpre op:'-' right:additive
    ;
```

TatSu will generate the base class if it's not already known.

Base classes can be used as the target class in *walkers*, and in *code generators*:

```
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}'
```

CHAPTER

TEN

VIEWING MODELS AS JSON

Models generated by **TatSu** can be viewed by converting them to a JSON-compatible structure with the help of `tatsu.util.asjson()`. The protocol tries to provide the best representation for common types, and can handle any type using `repr()`. There are provisions for structures with back-references, so there's no infinite recursion.

```
import json

print(json.dumps(asjson(model), indent=2))
```

The `model`, with richer semantics, remains unaltered.

Conversion to a JSON-compatible structure relies on the protocol defined by `tatsu.utils.AsJSONMixin`. The mixin defines a `__json__(seen=None)` method that allows classes to define their best translation. You can use `AsJSONMixin` as a base class in your own models to take advantage of `asjson()`, and you can specialize the conversion by overriding `AsJSONMixin.__json__()`.

You can also write your own version of `asjson()` to handle special cases that are recurrent in your context.

CHAPTER

ELEVEN

PRINT TRANSLATION

TatSu doesn't impose a way to create translators, but it exposes the facilities it uses to generate the Python source code for parsers.

Translation in **TatSu** is based on subclasses of `Walker` and on classes that inherit from `IndentPrintMixin`, a strategy copied from the new PEG parser in Python (see PEP 617).

`IndentPrintMixin` provides an `indent()` method, which is a context manager, and should be used thus:

```
class MyTranslationWalker(NodeWalker, IndentPrintMixin):

    def walk_SomeNode(self, node):
        with self.indent():
            # ccontinue walking the tree
```

The `self.print()` method takes note of the current level of indentation, so output will be indented by the `indent` passed to the `IndentPrintConstructor`:

```
def walk_SomeNode(self, node):
    with self.indent():
        self.print(walk_expression(node.exp))
```

The printed code can be retrieved using the `printed_text()` method. Other posibilities are available by assigning a text-like object to `self.output_stream` in the `__init__()` method.

CHAPTER

TWELVE

DECLARATIVE TRANSLATION

Translation is one of the most common tasks in language processing. Analysis often sumarizes the parsed input, and *walkers* are good for that. In translation, the output can often be as verbose as the input, so a systematic approach that avoids bookkeeping as much as possible is convenient.

TatSu provides support for template-based code generation ("translation", see below) in the `tatsu.codegen` module. Code generation works by defining a translation class for each class in the model specified by the grammar.

Nowadays the preferred code generation strategy is to walk down the AST and *print()* the desired output, with the help of the `NodWalker` class, and the `IndentPrintMixin` mixin. That's the strategy used by pegen, the precursor to the new PEG parser in Python. Please take a lookt at the mini-tutorial for an example.

Basically, the code generation strategy changed from declarative with library support, to procedural, breadth or depth first, using only standard Python. The procedural code must know the AST structure to navigate it, although other strategies are available with `PreOrderWalker`, `DepthFirstWalker`, and `ContextWalker`.

deprecated

TatSu doesn't impose a way to create translators with it, but it exposes the facilities it uses to generate the Python source code for parsers.

Translation in **TatSu** was *template-based*, but instead of defining or using a complex templating engine (yet another language), it relies on the simple but powerful `string.Formatter` of the Python standard library. The templates are simple strings that, in **TatSu**'s style, are inlined with the code.

To generate a parser, **TatSu** constructs an object model of the parsed grammar. A `tatsu.codegen.CodeGenerator` instance matches model objects to classes that descend from `tatsu.codegen.ModelRenderer` and implement the translation and rendering using string templates. Templates are left-trimmed on whitespace, like Python *doc-comments* are. This is an example taken from **TatSu**'s source code:

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

Every *attribute* of the object that doesn't start with an underscore (`_`) may be used as a template field, and fields can be added or modified by overriding the `render_fields(fields)` method. Fields themselves are *lazily rendered* before being expanded by the template, so a field may be an instance of a `ModelRenderer` descendant.

The `rendering` module defines a `Formatter` enhanced to support the rendering of items in an *iterable* one by one. The syntax to achieve that is:

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

All of `ind`, `sep`, and `fmt` are optional, but the three *colons* are not. A field specified that way will be rendered using:

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

The extended format can also be used with non-iterables, in which case the rendering will be:

```
indent(fmt % render(value), ind)
```

The default multiplier for `ind` is `4`, but that can be overridden using `n*m` (for example `3*1`) in the format.

note

Using a newline character (`\n`) as separator will interfere with left trimming and indentation of templates. To use a newline as separator, specify it as `\\n`, and the renderer will understand the intention.

CHAPTER

THIRTEEN

LEFT RECURSION

TatSu supports direct and indirect left recursion in grammar rules using the the algorithm described by *Nicolas Laurent* and *Kim Mens* in their 2015 paper *Parsing Expression Grammars Made Practical*.

The design and implementation of left recursion was done by Vic Nightfall with research and help by Nicolas Laurent on Autumn, and research by Philippe Sigaud on PEGGED.

Left recursive rules produce left-associative parse trees (AST), as most users would expect, *except if some of the rules involved recurse on the right (a pending topic)*.

Left recursion support is enabled by default in **TatSu**. To disable it for a particular grammar, use the `@@left_recursion` directive:

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

Warning: Not all left-recursive grammars that use the **TatSu** syntax are PEG (the same happens with right-recursive grammars). **The order of rules matters in PEG**.

For right-recursive grammars the choices that parse the most input must come first. The same is true for left-recursive grammars.

CHAPTER

FOURTEEN

CALC MINI TUTORIAL

TatSu users have suggested that a simple calculator, like the one in the documentation for PLY would be useful.

Here it is.

14.1 The initial grammar

This is the original PLY grammar for arithmetic expressions:

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

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )
```

And this is the input expression for testing:

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

14.2 The Tatsu grammar

The first step is to convert the grammar to **TatSu** syntax and style, add rules for lexical elements (`number` in this case), add a `start` rule that checks for end of input, and a directive to name the generated classes:

```
@@grammar::CALC


start
    =
    expression $
    ;


```

(continues on next page)

(continued from previous page)

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


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


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


number
    =
    /\d+/
    ;
```

14.3 Add *cut* expressions

Cut expressions make a parser commit to a particular option after certain tokens have been seen. They make parsing more efficient, because other options are not tried. They also make error messages more precise, because errors will be reported closest to the point of failure in the input.

```
@@grammar::CALC


start
    =
    expression $
    ;


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

```

(continues on next page)

(continued from previous page)

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


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


number
    =
    /\d+/
    ;
```

Let's save the above grammar in a file called `calc_cut.ebnf`. We can now compile the grammar, and test the parser:

```
import json
from pprint import pprint

import tatsu


def simple_parse():
    with open('calc_cut.ebnf') as f:
        grammar = f.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))


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

Save the above in `calc.py`. This is the output:

```
$ python calc.py
```

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

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

14.4 Annotating the grammar

Dealing with ASTs that are lists of lists leads to code that is difficult to read, and error-prone. **TatSu** allows naming the elements in a rule to produce more humanly-readable ASTs and to allow for clearer semantics code. This is an annotated version of the grammar:

```
@@grammar::CALC


start
    =
    expression $
    ;


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

(continues on next page)

(continued from previous page)

```
    ;


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


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


number
    =
    /\d+/
    ;
```

Save the annotated grammar in `calc_annotated.ebnf`, change the grammar filename in `calc.py` and re-execute it to get the resulting AST:

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

14.5 Semantics

Semantic actions for **TatSu** parsers are not specified in the grammar, but in a separate *semantics* class.

```
from pprint import pprint

import tatsu
from tatsu.ast import AST


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

    def term(self, ast):
```

(continues on next page)

(continued from previous page)

```
        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():
    with open('calc_annotated.ebnf') as f:
        grammar = f.read()

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

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


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

Save the above in `calc_semantics.py` and execute it with `python calc_semantics.py`. The result is:

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

14.6 One rule per expression type

Having semantic actions determine what was parsed with `isinstance()` or querying the AST for operators is not very pythonic, nor object oriented, and it leads to code that's more difficult to maintain. It's preferable to have one rule per *expression kind*, something that will be necessary if we want to build object models to use *walkers* and *code generation*.

```
@@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
    ;
```

(continues on next page)

(continued from previous page)

```


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


number
    =
    /\d+/
    ;
```

Save the above in `calc_refactored.ebnf`.

```
from pprint import pprint

import tatsu


class CalcSemantics:
    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


def parse_refactored():
    with open('calc_refactored.ebnf') as f:
        grammar = f.read()

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

    print('# REFACTORED SEMANTICS RESULT')
    pprint(ast, width=20, indent=4)
    print()

```

(continues on next page)

(continued from previous page)

```

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

The semantics implementation is simpler, and the results are the same:

```
# REFACTORED SEMANTICS RESULT
-47
```

14.7 Object models

Binding semantics to grammar rules is powerful and versatile, but this approach risks tying the semantics to the *parsing process*, rather than to the parsed *objects*.

That is not a problem for simple languages, like the arithmetic expression language in this tutorial. But as the complexity of the parsed language increases, the number of grammar rules quickly becomes larger than the types of objects parsed.

TatSu can create typed object models directly from the parsing process which can be navigated (*walked*) and transformed (with *code generation*) in later passes.

The first step to create an object model is to annotate the rule names with the desired class names:

```
@@grammar::Calc


start
    =
    expression $
    ;


expression
    =
    | addition
    | subtraction
    | term
    ;


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


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


term
```

(continues on next page)

(continued from previous page)

```
    =
    | multiplication
    | division
    | factor
    ;


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


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


factor
    =
    | subexpression
    | number
    ;


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


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

Save the grammar in a file name `calc_model.ebnf`.

The `tatsu.objectmodel.Node` descendants are synthetized at runtime using `tatsu.semantics.ModelBuilderSemantics`.

This is how the model looks like when generated with the `tatsu.to_python_model()` function or from the command line with `tatsu --object-model calc_model.ebnf -G calc_semantics_model.py`:

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


class ModelBase(Node):
    pass


```

(continues on next page)

(continued from previous page)

```
class CalcModelBuilderSemantics(ModelBuilderSemantics):
    def __init__(self, context=None, types=None):
        types = [
            t for t in globals().values()
            if type(t) is type and issubclass(t, ModelBase)
        ] + (types or [])
        super(CalcModelBuilderSemantics, self).__init__(context=context, types=types)


class Add(ModelBase):
    left = None
    op = None
    right = None


class Subtract(ModelBase):
    left = None
    op = None
    right = None


class Multiply(ModelBase):
    left = None
    op = None
    right = None


class Divide(ModelBase):
    left = None
    right = None
```

The model that results from a parse can be printed, and walked:

```
import tatsu
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)
```

(continues on next page)

(continued from previous page)

```
def parse_and_walk_model():
    with open('calc_model.ebnf') as f:
        grammar = f.read()

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

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


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

Save the above program in `calc_model.py` and execute it to get this result:

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

14.8 Code Generation

Translation is one of the most common tasks in language processing. Analysis often sumarizes the parsed input, and *walkers* are good for that. In translation, the output can often be as verbose as the input, so a systematic approach that avoids bookkeeping as much as possible is convenient.

TatSu provides support for template-based code generation (translation) in the `tatsu.codegen` module. Code generation works by defining a translation class for each class in the model specified by the grammar.

Nowadays the preferred code generation strategy is to walk down the AST and *print()* the desired output, with the help of the `NodWalker` class, and the `IndentPrintMixin` mixin. That's the strategy used by pegen, the precursor to the new PEG parser in Python.

The following code generator translates input expressions to the postfix instructions of a stack-based processor:

```
import sys

from tatsu.model import Node
from tatsu.walkers import NodeWalker
from tatsu.mixins.indent import IndentPrintMixin
from tatsu.codegen import ModelRenderer

THIS_MODULE = sys.modules[__name__]


class PostfixCodeGenerator(NodeWalker, IndentPrintMixin):

    def walk_Add(self, node: Node, *args, **kwargs):
        with self.indent():
            self.walk(node.left)  # type: ignore
```

(continues on next page)

(continued from previous page)

```
            self.walk(node.right)  # type: ignore
            self.print('ADD')

    def walk_Subtract(self, node: Node, *args, **kwargs):
        with self.indent():
            self.walk(node.left)  # type: ignore
            self.walk(node.right)  # type: ignore
            self.print('SUB')

    def walk_Multiply(self, node: Node, *args, **kwargs):
        with self.indent():
            self.walk(node.left)  # type: ignore
            self.walk(node.right)  # type: ignore
            self.print('MUL')

    def walk_Divide(self, node: Node, *args, **kwargs):
        with self.indent():
            self.walk(node.left)  # type: ignore
            self.walk(node.right)  # type: ignore
            self.print('DIV')

    def walk_int(self, node: Node, *args, **kwargs):
        self.print('PUSH', node)
```

Save the above program in `calc_translate.py` and execute it to get this result:

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

CHAPTER FIFTEEN

TRACES

TatSu compiling and parsing actions have a `trace=` argument (`--trace` on the command line). When used with the `colorize=` option (`--color` on the command line), it produces trace like the following, in which colors mean try, suceed, and fail.

```
start ~1:1
3 + 5 * ( 10 - 20 )
expressionstart ~1:1
3 + 5 * ( 10 - 20 )
expressionexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
expressionexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
expressionexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
expressionexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
termexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
termtermexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
termtermexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
termtermexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
termtermexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
factortermexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
’(’ ~1:1
3 + 5 * ( 10 - 20 )
numberfactortermexpressionstart ~1:1
3 + 5 * ( 10 - 20 )
’3’ /d+/ ~1:2
+ 5 * ( 10 - 20 )
numberfactortermexpressionstart ~1:2
```

```
+ 5 * ( 10 - 20 )
factortermexpressionstart ~1:2
+ 5 * ( 10 - 20 )
termtermexpressionstart ~1:3
+ 5 * ( 10 - 20 )
termtermexpressionstart ~1:3
+ 5 * ( 10 - 20 )
’*’ ~1:3
+ 5 * ( 10 - 20 )
termtermexpressionstart ~1:3
+ 5 * ( 10 - 20 )
termtermexpressionstart ~1:3
+ 5 * ( 10 - 20 )
’/’ ~1:3
+ 5 * ( 10 - 20 )
factortermexpressionstart ~1:3
+ 5 * ( 10 - 20 )
’(’ ~1:3
+ 5 * ( 10 - 20 )
numberfactortermexpressionstart ~1:3
+ 5 * ( 10 - 20 )
” /d+/ ~1:3
+ 5 * ( 10 - 20 )
factortermexpressionstart ~1:3
+ 5 * ( 10 - 20 )
termexpressionstart ~1:2
+ 5 * ( 10 - 20 )
expressionexpressionstart ~1:3
+ 5 * ( 10 - 20 )
expressionexpressionstart ~1:3
+ 5 * ( 10 - 20 )
’+’ ~1:4
5 * ( 10 - 20 )
termexpressionstart ~1:4
5 * ( 10 - 20 )
termtermexpressionstart ~1:5
5 * ( 10 - 20 )
termtermexpressionstart ~1:5
5 * ( 10 - 20 )
termtermexpressionstart ~1:5
5 * ( 10 - 20 )
termtermexpressionstart ~1:5
5 * ( 10 - 20 )
factortermexpressionstart ~1:5
5 * ( 10 - 20 )
’(’ ~1:5
5 * ( 10 - 20 )
numberfactortermexpressionstart ~1:5
5 * ( 10 - 20 )
’5’ /d+/ ~1:6
* ( 10 - 20 )
numberfactortermexpressionstart ~1:6
* ( 10 - 20 )
factortermexpressionstart ~1:6
* ( 10 - 20 )
termtermexpressionstart ~1:7
* ( 10 - 20 )
termtermexpressionstart ~1:7
* ( 10 - 20 )
’*’ ~1:8
( 10 - 20 )
factortermexpressionstart ~1:8
( 10 - 20 )
’(’ ~1:10
10 - 20 )
expressionfactortermexpressionstart ~1:10
10 - 20 )
expressionexpressionfactortermexpressionstart ~1:11
10 - 20 )
expressionexpressionfactortermexpressionstart ~1:11
10 - 20 )
expressionexpressionfactortermexpressionstart ~1:11
10 - 20 )
expressionexpressionfactortermexpressionstart ~1:11
10 - 20 )
termexpressionfactortermexpressionstart ~1:11
10 - 20 )
termtermexpressionfactortermexpressionstart ~1:11
10 - 20 )
termtermexpressionfactortermexpressionstart ~1:11
10 - 20 )
termtermexpressionfactortermexpressionstart ~1:11
10 - 20 )
termtermexpressionfactortermexpressionstart ~1:11
10 - 20 )
factortermexpressionfactortermexpressionstart ~1:11
10 - 20 )
’(’ ~1:11
10 - 20 )
numberfactortermexpressionfactortermexpressionstart ~1:11
10 - 20 )
’10’ /d+/ ~1:13
- 20 )
numberfactortermexpressionfactortermexpressionstart ~1:13
```

```
- 20 )
factortermexpressionfactortermexpressionstart ~1:13
- 20 )
termtermexpressionfactortermexpressionstart ~1:14
- 20 )
termtermexpressionfactortermexpressionstart ~1:14
- 20 )
’*’ ~1:14
- 20 )
termtermexpressionfactortermexpressionstart ~1:14
- 20 )
termtermexpressionfactortermexpressionstart ~1:14
- 20 )
’/’ ~1:14
- 20 )
factortermexpressionfactortermexpressionstart ~1:14
- 20 )
’(’ ~1:14
- 20 )
numberfactortermexpressionfactortermexpressionstart ~1:14
- 20 )
” /d+/ ~1:14
- 20 )
factortermexpressionfactortermexpressionstart ~1:14
- 20 )
termexpressionfactortermexpressionstart ~1:13
- 20 )
expressionexpressionfactortermexpressionstart ~1:14
- 20 )
expressionexpressionfactortermexpressionstart ~1:14
- 20 )
’+’ ~1:14
- 20 )
expressionexpressionfactortermexpressionstart ~1:14
- 20 )
expressionexpressionfactortermexpressionstart ~1:14
- 20 )
’-’ ~1:15
20 )
termexpressionfactortermexpressionstart ~1:15
20 )
termtermexpressionfactortermexpressionstart ~1:16
20 )
termtermexpressionfactortermexpressionstart ~1:16
20 )
termtermexpressionfactortermexpressionstart ~1:16
20 )
```

```
termtermexpressionfactortermexpressionstart ~1:16
20 )
factortermexpressionfactortermexpressionstart ~1:16
20 )
’(’ ~1:16
20 )
numberfactortermexpressionfactortermexpressionstart ~1:16
20 )
’20’ /d+/ ~1:18
)
numberfactortermexpressionfactortermexpressionstart ~1:18
)
factortermexpressionfactortermexpressionstart ~1:18
)
termtermexpressionfactortermexpressionstart ~1:19
)
termtermexpressionfactortermexpressionstart ~1:19
)
’*’ ~1:19
)
termtermexpressionfactortermexpressionstart ~1:19
)
termtermexpressionfactortermexpressionstart ~1:19
)
’/’ ~1:19
)
factortermexpressionfactortermexpressionstart ~1:19
)
’(’ ~1:19
)
numberfactortermexpressionfactortermexpressionstart ~1:19
)
” /d+/ ~1:19
)
factortermexpressionfactortermexpressionstart ~1:19
)
termexpressionfactortermexpressionstart ~1:18
)
expressionexpressionfactortermexpressionstart ~1:19
)
expressionexpressionfactortermexpressionstart ~1:19
)
’+’ ~1:19
)
expressionexpressionfactortermexpressionstart ~1:19
)
expressionexpressionfactortermexpressionstart ~1:19
```

```
)
’-’ ~1:19
)
termexpressionfactortermexpressionstart ~1:19
)
termexpressionfactortermexpressionstart ~1:19
)
expressionfactortermexpressionstart ~1:18
)
’)’
factortermexpressionstart
termtermexpressionstart
termtermexpressionstart
’*’
termtermexpressionstart
termtermexpressionstart
’/’
factortermexpressionstart
’(’
numberfactortermexpressionstart
” /d+/
factortermexpressionstart
termexpressionstart
expressionexpressionstart
expressionexpressionstart
’+’
expressionexpressionstart
expressionexpressionstart
’-’
termexpressionstart
termexpressionstart
expressionstart
start
```

CHAPTER

SIXTEEN

GRAKO COMPATIBILITY

TatSu is routinely tested over major projects developed with Grako. The backwards-compatibility suite includes (at least) translators for COBOL, Java, and (Oracle) SQL.

Grako grammars and projects can be used with **TatSu**, with these caveats:

- The AST type retuned when a sequence of elements is matched is now `tuple` (instead of a descendant of `list`). This change improves efficiency and avoids unwanted manipulations of a value that should be inmutable.
- The Python module name changed to `tatsu`.
- `ignorecase` no longer applies to regular expressions in grammars. Use `(?i)` in the pattern to enable `re.IGNORECASE`
- Left recursion is enabled by default because it works and has zero impact on non-recursive grammars.
- Deprecated grammar syntax is no longer documented. It's best not to use it, as it will be removed in a future version of **TatSu**.

CHAPTER SEVENTEEN

USING ANTLR GRAMMARS

ANTLR is one of the best known parser genrators, and it has an important collection of grammars. The `tatsu.g2e` module can translate an ANTLR grammar to the syntax used by **TatSu**.

The resulting grammar won't be immediately usable. It will have to be edited to make it abide to PEG semantics, and in general be adapted to the way things are done with **TatSu**.

To use `g2e` as a module, invoke one of its translation functions.

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

For example:

```
from tatsu import g2e

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

`g2e` can also be used from the command line:

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

CHAPTER

EIGHTEEN

EXAMPLES

18.1 Tatsu

The file `grammar/tatsu.ebnf` contains a grammar for the **TatSu** grammar language written in its own grammar language. It is used in the *bootstrap* test suite to prove that **TatSu** can generate a parser to parse its own language, and the resulting parser is made the bootstrap parser every time **TatSu** is stable (see `tatsu/bootstrap.py` for the generated parser).

TatSu uses **TatSu** to translate grammars into parsers, so it is a good example of end-to-end translation.

18.2 Calc

The project `examples/calc` implements a calculator for simple expressions, and is written as a tutorial over most of the features provided by **TatSu**.

18.3 g2e

The project `examples/g2e` contains an example ANTLR to **TatSu** grammar translation. The project is a good example of the use `g2e`. It generates the **TatSu** grammar on standard output, but because the model used is **TatSu**'s own, the same code can be used to directly generate a parser from any ANTLR grammar. Please take a look at the examples *README* to know about limitations.

CHAPTER

NINETEEN

SUPPORT

For general Q&A, please use the `[tatsu]` tag on StackOverflow.

CHAPTER TWENTY

CREDITS

- **TatSu** is the successor of Grako, which was built by **Juancarlo Añez** and funded by **Thomas Bragg** to do analysis and translation of programs written in legacy programming languages.
- **Niklaus Wirth** was the chief designer of the programming languages Euler, Algol W, Pascal, Modula, Modula-2, Oberon, and Oberon-2. In the last chapter of his 1976 book Algorithms + Data Structures = Programs, Wirth creates a top-down, descent parser with recovery for the Pascal-like, LL(1) programming language PL/0. The structure of the program is that of a PEG parser, though the concept of PEG wasn't formalized until 2004.
- **Bryan Ford** introduced PEG (parsing expression grammars) in 2004.
- Other parser generators like PEG.js by **David Majda** inspired the work in **TatSu**.
- **William Thompson** inspired the use of context managers with his blog post that I knew about through the invaluable Python Weekly newsletter, curated by **Rahul Chaudhary**
- **Jeff Knupp** explains why **TatSu**'s use of exceptions is sound, so I don't have to.
- **Terence Parr** created ANTLR, probably the most solid and professional parser generator out there. *Ter*, *ANTLR*, and the folks on the *ANLTR* forums helped me shape my ideas about **TatSu**.
- **JavaCC** (originally Jack) looks like an abandoned project. It was the first parser generator I used while teaching.
- **TatSu** is very fast. But dealing with millions of lines of legacy source code in a matter of minutes would be impossible without PyPy, the work of **Armin Rigo** and the PyPy team.
- **Guido van Rossum** created and has lead the development of the Python programming environment for over a decade. A tool like **TatSu**, at under 10K lines of code, would not have been possible without Python.
- **Kota Mizushima** welcomed me to the CSAIL at MIT PEG and Packrat parsing mailing list, and immediately offered ideas and pointed me to documentation about the implementation of *cut* in modern parsers. The optimization of memoization information in **TatSu** is thanks to one of his papers.
- **My students** at UCAB inspired me to think about how grammar-based parser generation could be made more approachable.
- **Gustavo Lau** was my professor of *Language Theory* at USB, and he was kind enough to be my tutor in a thesis project on programming languages that was more than I could chew. My peers, and then teaching advisers **Alberto Torres**, and **Enzo Chiariotti** formed a team with **Gustavo** to challenge us with programming languages like *LATORTA* and term exams that went well into the eight hours. And, of course, there was also the *pirate patch* that should be worn on the left or right eye depending on the *LL* or *LR* challenge.
- **Manuel Rey** led me through another, unfinished, thesis project that taught me about what languages (spoken languages in general, and programming languages in particular) are about. I learned why languages use declensions, and why, although the underlying words are in English, the structure of the programs we write is more like Japanese.
- Marcus Brinkmann has kindly submitted patches that have resolved obscure bugs in **TatSu**'s implementation, and that have made the tool more user-friendly, specially for newcomers to parsing and translation.

- Robert Speer cleaned up the nonsense in trying to have Unicode handling be compatible with 2.7.x and 3.x, and figured out the canonical way of honoring escape sequences in grammar tokens without throwing off the encoding.
- Basel Shishani has been an incredibly throrough peer-reviewer of **TatSu**.
- Paul Sargent implemented Warth et al's algorithm for supporting direct and indirect left recursion in PEG parsers.
- Kathryn Long proposed better support for UNICODE in the treatment of whitespace and regular expressions (patterns) in general. Her other contributions have made **TatSu** more congruent, and more user-friendly.
- David Röthlisberger provided the definitive patch that allows the use of Python keywords as rule names.
- Nicolas Laurent researched, designed, implemented, and published the left recursion algorithm used in **TatSu**.
- Vic Nightfall designed and coded an implementation of left recursion that handles all the use cases of interest (see the Left Recursion topic for details). He was gentle enough to kindly take over management of the **TatSu** project since 2019.

CHAPTER

TWENTYONE

CONTRIBUTORS

The following, among others, have contributed to **TatSu** with features, bug reports, bug fixes, or suggestions:

> Alberto Berti, Andy Wright, Basel Shishani, Daniel Martin, Daniele Nicolodi, 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, Oleg Komarov, 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, by-Exist

CHAPTER

TWENTYTWO

CONTRIBUTING

TatSu development is done on Github. Bug reports, patches, suggestions, and improvements are welcome.

22.1 Donations

If you'd like to contribute to the future development of **TatSu**, please make a donation to the project.

Some of the planned new features are: grammar expressions for left and right associativity, new algorithms for left-recursion, a unified intermediate model for parsing and translating programming languages, and more…

CHAPTER

TWENTYTHREE

LICENSE

TATSU - A PEG/Packrat parser generator for Python