Axaxaxas Documentation

Release 1.0

Adam Newgas

September 13, 2015

Contents

Axaxaxas - *making sense of nonsense*.

Axaxaxas is a Python 3.3 implementation of an Earley parser. Earley parsers are a robust parser that can recognize any context-free grammar, with good support for amiguous grammars. They have linear performance for a wide class of grammars, and worst case O(n^3).

CHAPTER 1

Usage

This section assumes you are familiar with the basic terminology involved in parsing Context Free grammars.

1.1 Defining a Grammar

A grammar is stored as a collection of *ParseRule* objects inside a *ParseRuleSet*. Each *ParseRule* is a single production from a "head" (symbol named by string), to a list of *Symbol* objects. Multiple *ParseRule* objects with the same head define alternative productions.

For example, the following Backus-Naur notated grammar:

```
<sentence> ::= <noun> <verb> <noun>
<noun> ::= "man" | "dog"
<verb> ::= "bites"
```

Would be expressed:

```
from axaxaxas import ParseRule, ParseRuleSet, Terminal as T, NonTerminal as NT

grammar = ParseRuleSet()
grammar.add(ParseRule("sentence", [NT("noun"), NT("verb"), NT("noun")]))
grammar.add(ParseRule("noun", [T("man")]))
grammar.add(ParseRule("noun", [T("dog")]))
grammar.add(ParseRule("verb", [T("bites")]))
```

1.2 Invoking the parser

Having defined our grammar, we can attempt to parse it. Parsing operates on an iterator of token objects, There is no lexer included. In the example above we have assumed that the tokens are Python strings, but they can be anything. Often a formal lexer is not needed - we can use `string.split` to produce lists of strings.

The parser is invoked with the *parse* function:

```
from axaxaxas import parse
parse_forest = parse(grammar, "sentence", "man bites dog".split())

print(parse_forest.single())
# (sentence: (noun: 'man') (verb: 'bites') (noun: 'dog'))
```

1.3 Parse results

parse returns a *ParseForest* that contains a collection of *ParseTree* objects. By calling *single()* we check that there is exactly one possible parse tree, and return it. See *Handling Ambiguity* for more details.

ParseTree objects themselves are a fairly straightforward representation. `ParseTree.rule` contains the ParseRule used to match the tokens, and `ParseTree.children` contains a value for each symbol of the rule, where the value is the token matched for terminals, or another *ParseTree* for nonterminals.

1.4 Optional, Star and Plus

Any *Symbol* can be declared with `optional=True` to make it nullable - it must occur zero or one times. Similarly, `star=True` allows any number of occurences, min zero, and `plus=True` allows any number of occurences, min one. In the resulting parse tree, skipped optional symbols are represented with `None`. star and plus elements become tuples:

```
grammar.add(ParseRule("relative", [T("step", optional=True), T("sister")]))
grammar.add(ParseRule("relative", [T("great", star=True), T("grandfather")]))

print(parse(grammar, "relative", "sister".split()).single())
# (relative: None 'sister')

print(parse(grammar, "relative", "step sister".split()).single())
# (relative: 'step' 'sister')

print(parse(grammar, "relative", "grandfather".split()).single())
# (relative: () 'grandfather')

print(parse(grammar, "relative", "great great grandfather".split()).single())
# (relative: ('great', 'great') 'grandfather')
```

1.5 Greedy Symbols

Like in a regular expression, you can mark parts of the grammar as `greedy` or `lazy`. In case of ambiguity the parser will preferentially prefer the parse tree with the more (or fewer) number of occurrences. lazy and greedy can only be used in combination with optional, plus or star:

```
grammar.add(ParseRule("described relative", [NT("adjective", star=True), NT("relative")]))
grammar.add(ParseRule("adjective", [T("awesome")]))
grammar.add(ParseRule("adjective", [T("great")]))

print(parse(grammar, "described relative", "great grandfather".split()).single())
# -- raises AmbiguousParseError

grammar.add(ParseRule("described relative 2", [NT("adjective", star=True, greedy=True), NT("relative"

print(parse(grammar, "described relative 2", "great grandfather".split()).single())
# (described relative 2: ((adjective: 'great'),) (relative: () 'grandfather'))
```

Greediness only trims ambiguous possibilities, so will never cause a sentence fail to parse.

It only affects choices the parser makes when reading from left to right, which means you will still get ambiguity if the leftmost symbol isn't marked.

For non-terminals, settings `prefer_early` and `prefer_late` work analogously. They instruct the parser that if there are several possible productions that could be used for a given symbol, to prefer the first (or last) one in order of definition in the grammar:

```
grammar.add(ParseRule("dinner order", [T("I"), T("want"), NT("item", prefer_early=True)]))
grammar.add(ParseRule("item", [T("ham")]))
grammar.add(ParseRule("item", [T("eggs")]))
grammar.add(ParseRule("item", [T("ham"), T("and"), T("eggs")]))
grammar.add(ParseRule("item", [NT("item", prefer_early=True), T("and"), NT("item", prefer_early=True)

print(parse(grammar, "dinner order", "I want eggs and ham".split()).single())
# (dinner order: 'I' 'want' (item: (item: 'eggs') 'and' (item: 'ham')))

print(parse(grammar, "dinner order", "I want ham and eggs".split()).single())
# (dinner order: 'I' 'want' (item: 'ham' 'and' 'eggs'))
```

1.6 Penalty

As mentioned above, greedy and related settings only trim ambiguity when the two options have so far parsed identically.

In some circumstances, you wish to avoid a particular rule, no matter how different the alternatives are. You can associate a penalty with each rule. The parser sums up all the penalties associated with a given parse, and choose only possibly parses with the lowest sum. This can have wide ranging effects on eliminating ambiguity. Penalties can be viewed as very lightweight support for probabilistic parsing:

```
grammar.add(ParseRule("sentence", [NT("noun"), T("like"), T("a"), NT("noun")]))
grammar.add(ParseRule("sentence", [NT("noun"), T("flies"), T("like"), T("a"), NT("noun")]))
grammar.add(ParseRule("noun", [T("fruit"), T("flies")], penalty=1))
grammar.add(ParseRule("noun", [T("fruit")]))
grammar.add(ParseRule("noun", [T("banana")]))

print(parse(grammar, "sentence", "fruit flies like a banana".split()).single())
# (sentence: (noun: 'fruit') 'flies' 'like' 'a' (noun: 'banana'))
```

In the example above the parser chose to avoid the other possible parse `(sentence: (noun: 'fruit' 'flies') 'like' 'a' (noun: 'banana'))` because it contains a rule with a penalty.

Penalties can be considered an experimental feature. Most of the time, you can just add more greedy settings to get the desired effect.

CHAPTER 2

Customization

2.1 Tokens

The parser works on a stream of tokens. Tokens can be any python object, they are not expected to have any particular behaviour. You may want to provide useful `__repr__` and `__str__` methods to give better error messages.

Thus, the *`parse`* function can work just as effectively on a stream of `str`, `bytes` or a stream of single characters (a scannerless parser). A common technique is for the lexer to produce some sort of Token object that includes a text string and additional annotations. For example the Natural Language Toolkit can mark each token with the relevant part of speech.

2.2 Symbols

Symbols are objects used to define the right hand side of a *`ParseRule`* production. Two Symbols, *`NonTerminal`* and *`Terminal`* are provided in the `symbols` module. Anything that duck-types the same as these can be used however.

This is mostly useful for re-defining *`Terminal.match`*, which is the method responsible for determining if a given token matches the terminal. The default *`Terminal`* class matches by equality, but, for example, you may have terminals that match entire classes of tokens.

2.3 Customizing ParseTrees

There is no way to customize the *`ParseTree`* class. But you can avoid using it entirely by writing your own *`Builder`*. Builders specify a semantic action to take at each step of the parse, allowing you to build your own parse trees or abstract syntax trees directly from a *`ParseForest`*. See ***Builders*** for more details.

2.4 Customizing Grammars

You can override *`ParseRuleSet.get`* with anything that returns a list of *`ParseRule`* objects. As there is no preprocessing done on the rules, you can generate a grammar on the fly. You can use this feature to parse context sensitive grammars, by passing any relevant context as part of the head, and adjusting the non-terminals of the returned rules to forward on relevant context. This will probably lead to very long parse times unless care is applied.

CHAPTER 3

Handling Ambiguity

The `parse` function returns a `ParseForest`. A `ParseForest` is an efficient shared representation of multiple possible `ParseTree` objects. For some grammars, therefore, you must be careful dealing with `ParseForest` objects as they may contains exponentially many possible parses.

3.1 single, all, count, and __iter__

These are the basic methods for extracting parse trees from the forest.

`ParseForest.single()` returns the unique tree in the forest, if there is one, or throws `AmbiguousParseError` (see *Errors and Edge Cases*).

`ParseForest.count()` returns a count of all the trees.

`ParseForest.all()` returns a list of all the trees in the forest. It can be quite large.

`ParseForest.__iter__()` iterates over all the trees in the forest. It is quite a bit slower than `all`, but it doesn't load all the trees into memory at once.

3.2 Greedy Rules

Using the `greedy`, `lazy`, `prefer_early`, `prefer_late` and `penalty` settings described in *Greedy Symbols* allows you to eliminate alternative parses. In the extreme case of marking every nonterminal with `prefer_early` and every `optional`, `star` and `plus` symbol with `greedy`, then you will never have an ambiguous parse.

3.3 Builders

Builders are an advanced API that give you fine control over interpreting the parse. You can explicitly control behaviour in ambiguity by handling `Builder.merge()`.

CHAPTER 4

Builders

Builders are an advanced way to process a *ParseForest*. Builders can take advantage of the shared representation the parse trees inside a forest, and choose how to handle ambiguity.

To use builders, you must define your own builder class inheriting from *Builder*:

```
class MyBuilder(Builder):
    def start_rule(self, context):
        ...

    def end_rule(self, context, prev_value):
        ...

    def terminal(self, context, token):
        ...

    def skip_optional(self, context, prev_value):
        ...

    def begin_multiple(self, context, prev_value):
        ...

    def end_multiple(self, context, prev_value):
        ...

    def extend(self, context, prev_value, extension_value):
        ...
```

Then you can run your builder against a parse forest using *ParseForest.apply*. The parse forest will then invoke methods on your builder as it walks over the possible parse tree. Each method is given some context, and the value currently being built, and returns a new value updated for what occured in the parse. In this way, you can build a complete picture of the parse tree, one step at a time.

The passed `context` is a *BuilderContext* with fields *rule*, *symbol_index*, *start_index* and *end_index* that give details about where in the rule and where in the token stream this invocation is occuring.

First *apply* will call *start_rule* for the matched rule. The result from that is passed to the other methods.

- *start_rule* is called at the start of each parsed rule.
- *end_rule* is called at the start of each parsed rule.
- *terminal* is called when a terminal is parsed.
- *extend* is called when a given symbol in a rule has been parsed. It is passed both the previous value for that rule and the extension_value describing what was parsed for that symbol.

- *skip_optional* is called in place of *extend* when an `optional` symbol is skipped over.
- *begin_multiple* and *end_multiple* are caused when a `star` or `plus` symbol is first encountered and left. Between them, any number of *extend* calls may be made, all corresponding to the same symbol.

You may find it easier to study the definitions of `CountingBuilder` and `SingleParseTree` builder, which are internal classes used for implementing *ParseForest.count()* and *ParseForest.single()*, as they are both fairly straightforward. `SingleParseTree` can be easily adapted to building arbitrary abstract syntax trees, or performing other semantic actions according to the parse.

4.1 Example

For example, suppose that the parse forest only contains a single parse tree, which looks like this:

```
(rule 1: "a" (rule 2: "b") "c")
```

In other words, we've parsed the token stream `["a", "b", "c"]` with the following grammar:

```
rule1 = ParseRule("rule 1", [T("a"), NT("rule 2"), T("c")])
rule2 = ParseRule("rule 2", [T("b")])
```

Then the following methods would get invoked during *apply* (though not necessary in this order):

```
v1 = builder.start_rule({rule2, 0})
v2 = builder.terminal({rule2, 0}, 'b')
v3 = builder.extend({rule2, 0}, v1, v2)
v4 = builder.end_rule({rule2, 1}, v3)
v5 = builder.start_rule({rule1, 0})
v6 = builder.terminal({rule1, 0}, 'a')
v7 = builder.extend({rule1, 0}, v5, v6)
v8 = builder.extend({rule1, 1}, v7, v4)
v9 = builder.terminal({rule1, 1}, 'c')
v10 = builder.extend({rule1, 2}, v8, v9)
v11 = builder.end_rule({rule1, 3}, v10)
```

4.2 Ambiguity

When there are multiple possible parse trees, sequences of builder results that are shared between parse trees will only get invoked once, then stored and re-used. This is why some context is omitted from builder methods, as the call may be used in several contexts. This is also why it is important not to mutate the passed in prev_value, as it may be used in other contexts. You should always return a fresh value that represents whatever change you need to make to prev value.

The easiest way to handle amiguity is to use utility methods *make_list_builder* and *make_iter_builder*. These methods accept a builder with no ambiguity handling, and returns a new builder that simply treats every possible parse tree independently, and return a list or iterable respectively. They directly correspond to the *ParseForest.all* and *ParseForest.__iter__* methods, which include some additional details.

If you do wish to directly handle ambiguity. You must override either the *merge* method, or both the *merge_horizontal* and *merge_vertical* methods. All these methods work the same way: you are passed a list of values that each represent an alternative parse of the same sequence of tokens for the same parse rule or symbol, and you must return a single value aggregating them.

A merge is "vertical", and calls *merge_vertical* when there are multiple possible *ParseRule* objects with the same head that match the same sequence of tokens. The *BuilderContext* indicates the non terminal symbol

they both match. Conversely, *merge_horizontal* is called when there are multiple possible parses for a single *ParseRule*. In most use cases, these methods will share the same implementation, so you are free to override *merge* instead.

Here is an example of the call sequence for an ambiguous parse of `["hello"]` by grammar:

```
rule1 = ParseRule("sentence", [T("hello")])
rule2 = ParseRule("sentence", [T("hello")])

v1 = builder.start_rule({rule1, 0})
v2 = builder.terminal({rule1, 0}, 'hello')
v3 = builder.extend({rule1, 0}, v1, v2)
v4 = builder.end_rule({rule1, 1}, v3)
v5 = builder.start_rule({rule2, 0})
v6 = builder.terminal({rule2, 0}, 'hello')
v7 = builder.extend({rule2, 0}, v5, v6)
v8 = builder.end_rule({rule2, 1}, v7)
v9 = builder.merge_vertical({None, 0}, [v8, v4])
```

(Note that in this special case where the top level symbol itself is ambiguous, then `None` is passed in as the rule being merged).

Here's another example, ambiguously parsing `["a"]`:

```
sentence = ParseRule("sentence", [NT("X"),NT("Y")])
X        = ParseRule("X", [T("a", optional=True)])
Y        = ParseRule("Y", [T("a", optional=True)])

v1 = builder.start_rule({Y, 0})              # After token 0
v2 = builder.skip_optional({Y, 0}, v1)
v3 = builder.end_rule({Y, 1}, v2)
v4 = builder.start_rule({X, 0})
v5 = builder.terminal({X, 0}, 'a')
v6 = builder.extend({X, 0}, v4, v5)
v7 = builder.end_rule({X, 1}, v6)
v8 = builder.start_rule({sentence, 0})
v9 = builder.extend({sentence, 0}, v8, v7)
v10 = builder.start_rule({Y, 0})              # Before token 0
v11 = builder.terminal({Y, 0}, 'a')
v12 = builder.extend({Y, 0}, v10, v11)
v13 = builder.end_rule({Y, 1}, v12)
v14 = builder.skip_optional({X, 0}, v4)
v15 = builder.end_rule({X, 1}, v14)
v16 = builder.extend({sentence, 0}, v8, v15)
v17 = builder.extend({sentence, 1}, v9, v3)
v18 = builder.extend({sentence, 1}, v16, v13)
v19 = builder.merge_horizontal({sentence, 2}, [v17, v18])
v20 = builder.end_rule({sentence, 2}, v19)
```

The two above examples give a visual indication of the terminology "vertical" and "horizontal". In the first, `rule1` and `rule2` are ambiguous and in vertically column in the grammar definition. In the second, `X` and `Y` are ambiguous, and are horizontally next to each other in a single grammar rule.

CHAPTER 5

Errors and Edge Cases

There are 3 possible ways that parsing can fail. All of them raise subclasses of *ParseError*. All instances of *ParseError* contain a `message`, and fields `start_index`, `end_index` indicating where in the token stream the error is occurring.

5.1 No parse

When the parser can find no possible parse for a token stream, it raises *NoParseError*. The location will be the furthest the parse got before there were no possible parses. Additional fields are included:

- `encountered` - The token we failed at, or `None` if the end of the stream was reached.
- `expected_terminals` - All the terminals that were evaluated against encountered (and failed)
- `expected` - Similar to `expected_terminals`, except the parser includes any non-terminals that could have started at the encountered token, and hides any terminals or non-terminals that are implicitly covered another one. This is usually a higher level summary of what was expected at any given point.

Note, you can override method *ParseRuleSet.is_anonymous()* to return true for some heads. Any anonymous rule will never be eligible to appear in `expected`. This is useful if you are transforming or generating the grammar, and some rules don't make sense to report.

5.2 Ambiguous Parse

Ambiguous parses are not an immediate error. The *parse* function simply returns a forest which contains all possible parse trees. However, if you call *ParseForest.single* and there is ambiguity, then *AmbiguousParseError* will be thrown. It will indicate the earliest possible ambiguity, but there may be others. The error will contain a field called `values` containing the possible alternatives parses. However, it only contains a subtree of the full parse tree, and additionally, it may be only halfway through building a rule, so the subtree may be missing elements. These limitations ensure that `values` is a short list. It is recommended you do not use *ParseForest.single* if you need more detail on ambiguity.

5.3 Infinite Parse

In some obscure grammars it is possible to define rules that have an infinite number of possible parses. Here is a simple example:

```
ParseRule("s", [NT("s")]
ParseRule("s", [T("word")]
```

When parsing `["word"]`, all the following are valid parse trees:

```
(s: word)
(s: (s: word))
(s: (s: (s: word)))
...
```

In this circumstance, *parse* throws *InfiniteParseError*. You can avoid this error with the right use of greedy and penalty settings as they are evaluated before checking for infinite parses.

It's possible to improve support for infinite parses if there is demand. Let me know.

5.4 Other notes

The order of evaluation for trimming ambiguity is:

- `penalty`
- `greedy` and `lazy`
- `prefer_early` and `prefer_late`

The *unparse* method can be used to convert *ParseTree* objects back into lists of tokens.

CHAPTER 6

Reference

6.1 Parsing

class axaxaxas.**ParseRule**(*head*, *symbols*, *, *penalty=0*)
Represents a single production in a context free grammar.

class axaxaxas.**ParseTree**(*rule*, *children=None*)
Tree structure representing a sucessfully parsed rule

class axaxaxas.**ParseForest**(*top_partial_rule*)
Represents a collection of related *ParseTree* objects.

__iter__()
Iterators over the list of contained *ParseTree* objects. Calling *all* is somewhat faster

all()
Returns a list of the contained *ParseTree* objects

apply(*builder*)
Constructs a result step at a time using the given *Builder*.

count()
Returns a count of the contained *ParseTree* objects

single()
Returns the only *ParseTree* in the collection, or throws if there are multiple.

class axaxaxas.**ParseRuleSet**
Stores a set of *ParseRule*, with fast retrieval by rule head

add(*rule*)
Adds a new *ParseRule* to the set

get(*head*, *lookahead_token=None*)
Returns a list of *ParseRule* objects with matching head

is_anonymous(*head*)
Returns true if a given head symbol should be omitted from error reporting

axaxaxas.**parse**(*rule_set*, *head*, *tokens*, *, *fail_if_empty=True*)
Parses a stream of tokens according to the grammer in rule_set by attempting to match the non-terminal specified by head.

axaxaxas.**unparse**(*parse_tree*)
Converts a *ParseTree* back to a list of tokens

6.2 Errors

class axaxaxas.**ParseError**(*message*, *start_index*, *end_index*)
Base parse error

class axaxaxas.**AmbiguousParseError**(*message*, *start_index*, *end_index*, *values*)
Indicates that there were multiple possible parses in a context that requires only one

class axaxaxas.**NoParseError**(*message*, *start_index*, *end_index*, *encountered*, *expected_terminals*, *expected*)
Indicates there were no possible parses

class axaxaxas.**InfiniteParseError**(*message*, *start_index*, *end_index*)
Indicates there were infinite possible parses

6.3 Building

class axaxaxas.**BuilderContext**
Contains information about the location and rule currently being parsed. The exact meaning is specific to each method of *Builder*.

rule
The relevant *ParseRule*. Can be None for *merge_vertical* calls at the top level.

symbol_index
context.rule.symbols[context.symbol_index] indicates the relevant symbol of the rule. Note *symbol_index* may be len(context.rule.symbols) in some circumstances.

start_index
The first token in the relevant range of tokens.

end_index
After the last token in the relevant range of tokens.

class axaxaxas.**Builder**
Abstract base class for constructing parse trees and other objects from a *ParseForest*. See *Builders* for more details.

begin_multiple(*context*, *prev_value*)
Called when a plus or star symbols is enountered

end_multiple(*context*, *prev_value*)
Called when there are no more matches for a plus or star symbol

end_rule(*context*, *prev_value*)
Called when a new rule is completed

extend(*context*, *prev_value*, *extension_value*)
Called when a symbol is matched. Potentially multiple times for a star or plus symbol

merge(*context*, *values*)
Called when there are multiple possible parses, unless *merge_vertical* and *merge_horizontal* is overriden.

merge_horizontal(*context*, *values*)
Called when there are multiple possible parses of a *ParseRule*.

merge_vertical(*context*, *values*)
Called when multiple possible *ParseRule* objects could match a non terminal

skip_optional(*context*, *prev_value*)
Called when an `optional` symbol is skipped over

start_rule(*context*)
Called when a new rule is started

terminal(*context*, *token*)
Called when a terminal is matched

axaxaxas.**make_list_builder**(*builder*)
Takes a *Builder* which lacks an implementation for *merge_horizontal* and *merge_vertical*, and returns a new *Builder* that will accumulate all possible built parse trees into a list

axaxaxas.**make_iter_builder**(*builder*)
Takes a *Builder* which lacks an implemenation for *merge_horizontal* and *merge_vertical*, and returns a new *Builder* that will accumulate all possible built parse trees into an iterator.

6.4 Symbols

class axaxaxas.**Symbol**(*, *star=False*, *optional=False*, *plus=False*, *name=None*, *greedy=False*, *lazy=False*)
Base class for non-terminals and terminals, this is used when defining ParseRule objects

class axaxaxas.**NonTerminal**(*head*, *prefer_early=False*, *prefer_late=False*, ***kwargs*)
Represents a non-terminal symbol in the grammar, matching tokens according to any ParseRules with the specified `head`

class axaxaxas.**Terminal**(*token*, ***kwargs*)
Represents a terminal symbol in the grammar, matching a single token of the input

match(*token*)
Returns true if token is matched by this Terminal

a

Symbols

A

B

C

E

G

I

M

N

P

R

S

T

U