aperiot user-guide

Ernesto Posse

August 29, 2006

Contents

---

1 aperiot at glance

aperiot is both a grammar description language and a parser generator for Python. Its purpose is to provide the means to describe a language's grammar and automatically generate a parser to recognize and process text written in that language.[1] It is intended to be used mainly for programming and modelling languages.

The basic idea is this:

1. Write the grammar of a language you want to parse using the aperiot meta-language described in section 3. Save this in a plain text file with a `.apr` extension.

[1] "aperio" is a Latin word meaning "to uncover," "to unearth." A parser is, after all, a tool that uncovers the structure of text.

2. Use the aperiot grammar compiler script to produce one of two possible grammar representations.

3. In your application, load one of the generated representations using a simple API provided in aperiot, which results in a Python object, the parser, that can parse strings or files given as input.

To illustrate this process, we'll consider a simple language for arithmetic expressions. The application is a simple calculator.

1. Write the grammar below in a plain text file named `aexpr.apr`.

```
# This is a simple language for arithmetic expressions

numbers
    number

operators
    plus   "+"
    times  "*"
    minus  "-"
    div    "/"

brackets
    lpar  "("
    rpar  ")"

start
    EXPR

rules
EXPR -> TERM               : "$1"
      | TERM plus EXPR     : "$1 + $3"
      | TERM minus EXPR    : "$1 - $3"

TERM -> FACTOR               : "$1"
      | FACTOR times TERM    : "$1 * $3"
      | FACTOR div TERM      : "$1 / $3"

FACTOR -> number           : "float($1)"
        | minus FACTOR     : "-$2"
        | lpar EXPR rpar   : "$2"
```

In this file, the sections titled "numbers," "operators," and "brackets" define symbolic names for the input tokens. The last section provides the actual rules. Each rule is annotated with a "Python expression template." This is,

a Python expression that uses *placeholders* (numbers preceded by '$'.) The placeholders refer to the corresponding symbol in the symbol sequence. For example, in `FACTOR times TERM : "$1 * $3"`, `$1` refers to `FACTOR`, and `$3` refers to `TERM`. When parsing, if this rule is applied, the result of applying the actions that yield a `FACTOR` will replace the `$1` entry and the result of applying the actions that yield `TERM` will replace the entry `$3`, and the result of evaluating the full Python expression will be the result of applying this rule.

2. Use the aperiot grammar compiler script to produce one of two possible grammar representations.

   In the comand-line prompt, execute the grammar compiler by typing:

   ```
   apr aexpr.apr
   ```

   This will generate a Python package called `aexpr_cfg` in the same directory where `aexpr.apr` is located. This package contains a module called `aexpr.py`.

3. In your application, load one of the generated representations using a simple API provided in aperiot, which results in a Python object, the parser, that can parse strings or files given as input.

   Assuming that the aperiot package and the directory where you generated `aexpr_cfg` are in the Python path, in your application you can write something like this:

   ```
   from aperiot.parsergen import build_parser
   myparser = build_parser('aexpr')
   text_to_parse = "56 +43* -21/(12-7)"
   outcome = myparser.parse(text_to_parse)
   print outcome
   ```

   Alternatively, you can split the parsing process in two steps: 1) obtaining the parse tree, and 2) applying the rule actions on the parse tree:

   ```
   from aperiot.parsergen import build_parser
   myparser = build_parser('aexpr')
   text_to_parse = "56 +43* -21/(12-7)"
   tree = myparser.parse(text_to_parse, apply_actions=False)
   outcome = myparser.apply_actions(tree)
   print outcome
   ```

   Furthermore, the input provided to the parser could be a file:

   ```
   from aperiot.parsergen import build_parser
   myparser = build_parser('aexpr')
   text_to_parse = file("myfile.txt", 'r')
   ```

```
outcome = myparser.parse(text_to_parse)
text_to_parse.close()
print outcome
```

The scheme described above generates a minimal Python representation of the grammar in the `aexpr.py` module within the `aexpr_cfg` package, and the parser object is built at run-time in the client application by the `build_parser` function. This approach, however, may be time-consuming if the language's grammar is large. `aperiot` provides alternative approach, in which the parser object is built during the grammar compilation and saved into a special file (with a .pkl extension,) which then can be quickly loaded by the application. To do this, use the `-f` command-line option of the `apr` script:

```
apr -f aexpr.apr
```

This will generate other files in the `aexpr_cfg` package, in particular a file called `aexpr.pkl`, containing the parser object itself.

Then, in the client Python application, use the `load_parser` function instead of the `build_parser` function:

```
from aperiot.parsergen import load_parser
myparser = load_parser('aexpr')
text_to_parse = file("myfile.txt", 'r')
outcome = myparser.parse(text_to_parse)
text_to_parse.close()
print outcome
```

Usually you want to report parsing errors in a user-friendly way. To do that, wrap around the parse method invocation with an exception handler as follows:

```
from aperiot.parsergen import load_parser
from aperiot.llparser import ParsingException
myparser = load_parser('aexpr')
text_to_parse = file("myfile.txt", 'r')
try:
    outcome = myparser.parse(text_to_parse)
    print outcome
except ParsingException, e:
    print e
text_to_parse.close()
```

The printout of the parsing error can be made nicer by keeping a separate copy of the source file:

```
from aperiot.parsergen import load_parser
from aperiot.llparser import ParsingException
myparser = load_parser('aexpr')
```

```
text = file(“myfile.txt”, ‘r’)
lines = text.readlines()
text.close()
text_to_parse = file(“myfile.txt”, ‘r’)
try:
    outcome = myparser.parse(text_to_parse)
    print outcome
except ParsingException, e:
    e.pprint(lines[e.linenum-1])
text_to_parse.close()
```

2 Lexing and parsing overview

The process of parsing text is usually performed in two stages:

1. Lexical analysis, or lexing, and
2. Syntactic analysis or parsing

Lexical analysis is performed by a *lexer* which takes as input a stream of characters (the source text,) and produces a string of *tokens* or *lexemes*, this is, “words” or sequences of characters that are to be treated as units, such as numbers, identifiers, keywords, etc.

Syntactic analysis is performed by a *parser*, which takes as input the sequence of tokens produced by the lexer and produces a *concrete syntax tree*, also known as *parse tree*, representing the syntactic structure of the text according to some given *grammar*.

Usually the parse tree itself is processed further by applying actions to it in order to produce some desired outcome. Usually the desired outcome is an *abstract syntax tree*, which contains the structure of the text abstracting away specific details about the text which are irrelevant for any further processing.

A grammar consists of a set of rules which describe the structure or composition of the text in terms of the text’s components. Rules are annotated with *actions* which are applied to the corresponding nodes in the parse tree in order to obtain some desired outcome.

In aperiot, a parser object encapsulates all these operations: the parser object performs lexical analysis, parse tree generation and actions application.

2.1 Context Free Grammars

A *grammar* consists of a set of rules describing the structure of text in a language. The most common type of grammar is known as “context free grammar,” or CFG for short.

A simple rule in a CFG has the form:

$$L \rightarrow w$$

where L is called a *non-terminal symbol* or simply a *non-terminal,* and w is a sequence of symbols or the special symbol ϵ which represents the empty string. The symbols in the sequence w may be non-terminals and other symbols called *terminals.*

Terminal symbols are those symbols or tokens which can appear literally in the text. Non-terminal symbols represent syntactic categories and a rule $L \rightarrow w$ states that the non-terminal L can be replaced by the sequence w. This is, if there is a sequence

$$uLv$$

applying the rule $L \rightarrow w$ yields the sequence

$$uwv$$

In the context of parsing one may interpret a rule $L \rightarrow w$ as saying that if the sequence of symbols w ocurrs in the input then it can be seen as a single ocurrence of the symbol L.

A grammar may also have "composite" rules of the form:

$$\begin{array}{lcl} L & \rightarrow & w_1 \\ & | & w_2 \\ & \vdots & \\ & | & w_n \end{array}$$

In such a rule, each w_i represents an alternative. In other words the rule states that L can be substituted by either w_1, w_2, ..., or w_n. Such rule is simply a short hand for the following set of rules:

$$\begin{array}{lcl} L & \rightarrow & w_1 \\ L & \rightarrow & w_2 \\ & \vdots & \\ L & \rightarrow & w_n \end{array}$$

A grammar has a distinguished non-terminal symbol called the *start* symbol. This symbol represents the topmost syntactic category.

A given text is succesfully parsed if it can be reproduced by the following procedure:

1. begin with the start symbol S
2. find a rule $S \rightarrow w$, and replace S by w
3. choose a non-terminal symbol N in w
4. find a rule $N \rightarrow w'$ and replace the ocurrence of N in w by w'
5. repeat from step 3 until there are no non-terminals left.

2.2 Types of parsers

The procedure specified above provides a possible way to determine whether some text conforms to a grammar, but it is by no means the only way to do so. There are different types of parsers, which generally are classified as either "top-down" or "bottom-up" parsers.

Top-down parsers follow a mechanism similar to the one described above, beginning with the start symbol and attempting to match the text by applying the rules. Bottom-up parsers on the other hand, scan the input stream and try to apply the rules "backwards" so that if the start symbol is reached, the text is successfully parsed.

The most common type of top-down parsers are called LL parsers and the most common type of bottom-up parsers are called LR parsers. The main difference between the two is that LL parsers yield the left-most derivation of the text if one exists while LR parsers yield the right-most derivation. The main consequence for the user is that for a given grammar, the generated parse trees may be different.

LL parsers and LR parsers use a parsing table to direct their behaviour and help them decide which rule to apply in any possible situation. This table is generated from the grammar provided.

A grammar is ambiguous if there are rules such that when applied there might be more than one possible alternative because the first symbol is the same for several alternatives. For example, the following rule is ambiguous:

$$\begin{array}{rcl} A & \rightarrow & bm \\ & | & bn \end{array}$$

This is an example of *direct ambiguity*, but rules may also be *indirectly ambiguous*, as is the case in the following set of rules:

$$\begin{array}{rcl} A & \rightarrow & Bm \\ & | & Cn \\ B & \rightarrow & x \\ C & \rightarrow & x \end{array}$$

There are several ways of dealing with ambiguous grammars.

A common approach is to use backtracking: whenever the parser finds more than one rule that could be applied, it remembers the current state and chooses one of the rules. If at some point parsing fails, it returns or "backtracks" to the more recently stored "choice point," and attempts another rule.

A second approach is to "look ahead" in the input stream and use more than one input token to decide which rule to apply. An LL parser (resp. LR parser) that requires looking ahead k input tokens is called an LL(k) parser (resp. LR(k) parser.) k is called the *lookahead* of the parser.

A third approach is to transform the grammar from an ambiguous grammar to a non-ambiguous grammar. One possibility is to apply the following

transformation to the given grammar. Suppose there is a rule of the form:

$$\begin{array}{rcl} A & \rightarrow & Bm \\ & | & Bn \\ & | & p \end{array}$$

Then we can rewrite the rule as

$$\begin{array}{rcl} A & \rightarrow & BB' \\ & | & p \end{array}$$

where B' is a new non-terminal symbol, and we add a rule:

$$\begin{array}{rcl} B' & \rightarrow & m \\ & | & n \end{array}$$

Now, this new rule may be itself ambiguous, so we must apply the transformation repeatedly until there are no more rules to transform.

aperiot's parsing algorithm is an LL(1) parser but it deals with ambiguous grammars by transformation as explained above (as well as additional transformations to optimize the grammar.) Unfortunately aperiot does not handle indirect ambiguity in grammars.

3 Describing grammars: the aperiot (meta-)language

aperiot is the name of the language used to describe context-free grammars. An aperiot file usually has a `.apr` extension, and is divided in sections which specifiy the set of terminal symbols, as well as the set of rules together with some associated actions.

The aperiot language is sensitive to indentation and newlines. It is also case sensitive.

Notation: In the following description we use `typewriter font` to write the actual tokens of the aperiot language, and *italic roman* font enclosed in < and > as meta-symbols to refer to syntactic categories of the aperiot language. A <*symbolic_name*> must be any valid identifier, i.e. an alphanumeric string (possibly with underscore characters) which must begin with a letter.

3.1 Sections of an aperiot file

The general form of an aperiot file is as follows:

<*import section*>
<*terminals section*>
<*rules section*>

3.1.1 Comments

Comments can be added anywhere in the file. A comment begins with the # symbol and ends with a newline.

3.1.2 The import section

The import section lists Python packages that will be used by the actions associated to the rules, if any are needed. It is of the form:

```
import
    <python_package_name_1>
    <python_package_name_2>
    ...
    <python_package_name_n>
```

3.1.3 The terminals section

The terminals section provides symbolic names for specific tokens and classes of tokens to be used in the rules. There are two types of declaration in this section: symbolic names alone, and symbolic names with an associated string literal.

A symbolic name with an associated literal is simply a name for the literal, so if the symbolic name occurs in some rule, the corresponding literal must be in the appropriate position in the input stream for the rule to be applicable. This kind of token declaration is used for keywords, operators, separators and brackets.

A symbolic name without an associated string literal is a name that represent *a class* of tokens rather than a specific literal. This includes identifiers, numbers and strings. If the symbolic name occurs in some rule, then a token of the corresponding class must occur in the input stream for the rule to be applicable.

The terminal section consists of the following parts, all of which are optional:

```
<indentation_and_newlines_part>
<keywords_part>
<operators_part>
<separators_part>
<brackets_part>
<identifiers_part>
<numbers_part>
<strings_part>
```

Indentation and newlines The indentation and newlines section specifies whether the grammar will be sensitive to indentation and newline characters.

It consists of two keywords (both optional) each of which must be in its own line, and unindented:

To state that the grammar is sensitive to indentation, use the following directive:

```
useindents
```

By specifying this directive, the lexer will generate "indent" and "dedent" tokens whenever change of indentation occurs. The corresponding symbolic names to be used in the rules section are:

```
indent
```

and

```
dedent
```

To state that the grammar is sensitive to newline characters, use the following directive:

```
usenewlines
```

By specifying this directive, the lexer will generate "newline" tokens at the end of each line[2], except for consecutive blank lines which are ignored. This means that consecutive blank lines are treated as a single newline. The corresponding symbolic name to be used in the rules section are:

```
newline
```

The useindents directive must appear before the usenewlines directive.

Keywords The keywords section describes alphanumeric (usually only alphabetic) strings used as keywords or reserved words in the target language. It has the form

```
keywords
    <keyword_declaration_1>
    <keyword_declaration_2>
    ...
    <keyword_declaration_n>
```

Each keyword declaration has the form

<symbolic_name> "*<string>*"

or

"*<string>*"

The string must start with a letter and contain only letters, digits or the underscore character.

If the first form is used, the symbolic name is to be used in the rules section to refer to that keyword. If the second form is used, the string itself (without quotes) may be used as its own symbolic name.

Giving a symbolic name is the preferred form, and it is mandatory if the keyword is the same as a Python keyword, in order to avoid conflict.

[2] On Windows the combination of CR (carriage return) and LF (line feed) is treated as a single newline token.

Operators The operators section describes symbolic strings used as operators in the target language. It has the form

```
operators
    <operator_declaration_1>
    <operator_declaration_2>
    ...
    <operator_declaration_n>
```

Each operator declaration has the form

<symbolic_name> "*<string>*"

The string can contain any sequence of symbols but must not start with a letter or digit. It cannot contain any space characters.

The symbolic name is to be used in the rules section to refer to that operator.

Separators The separators section describes symbolic strings used as delimiters or symbols to separate significant parts of the text in the target language. It is actually the same as the operators section but one may want to describe certain symbols as separators rather than operators for the sake of clarity, documentation or to differentiate them conceptually. It has the form

```
separators
    <separator_declaration_1>
    <separator_declaration_2>
    ...
    <separator_declaration_n>
```

Each separator declaration has the form

<symbolic_name> "*<string>*"

The string can contain any sequence of symbols but must not start with a letter or digit. It cannot contain any space characters.

The symbolic name is to be used in the rules section to refer to that separator.

Brackets The brackets section describes symbolic strings used as delimiters or symbols to contain significant parts of the text or describe nesting in the target language. It is actually the same as the separators section but one may want to describe certain symbols as brackets rather than separators for the sake of clarity, documentation or to differentiate them conceptually. It has the form

```
brackets
    <bracket_declaration_1>
    <bracket_declaration_2>
    ...
    <bracket_declaration_n>
```

Each bracket declaration has the form

<symbolic_name> "*<string>*"

The string can contain any sequence of symbols but must not start with a letter or digit. It cannot contain any space characters.

The symbolic name is to be used in the rules section to refer to that bracket.

Identifiers The identifiers section lists symbolic names used for identifiers, i.e. alphanumeric strings which are not keywords. Since an identifier is any such string, an identifier declaration only specifies the symbolic name to be used in the rules section. It has the form

```
identifiers
    <identifier_declaration_1>
    <identifier_declaration_2>
    ...
    <identifier_declaration_n>
```

Each identifier declaration has the form

<symbolic_name>

The symbolic name is to be used in the rules section to refer to an identifier.

Numbers The numbers section lists symbolic names used for number literals, i.e. numeric strings[3]. Since a number is any such string, a number declaration only specifies the symbolic name to be used in the rules section. It has the form

```
numbers
    <number_declaration_1>
    <number_declaration_2>
    ...
    <number_declaration_n>
```

Each number declaration has the form

<symbolic_name>

The symbolic name is to be used in the rules section to refer to a number literal.

Strings The strings section lists symbolic names used for string literals, i.e. arbitrary character strings enclosed in double-quotes. Since a string literal can be any sequence, a string declaration only specifies the symbolic name to be used in the rules section. It has the form

```
strings
    <string_declaration_1>
    <string_declaration_2>
    ...
    <string_declaration_n>
```

[3]Currently the lexer recognizes only positive integers and floating point numbers in decimal format but not in exponential format. Only base 10 numbers are recognized.

Each string declaration has the form

<symbolic_name>

The symbolic name is to be used in the rules section to refer to a string literal.

3.1.4 The rules section

The rules section consists of two parts: the start symbol section and the rules section proper.

The start symbol section specifies the start symbol of the grammar and must be a non-terminal which appears in the left-hand side of some rule in the rules section. It has the form:

```
start
    <symbolic_name>
```

The rules section contains the set of rules and their associated actions. It has the form:

```
rules
<composite_rule_1>
<composite_rule_2>
...
<composite_rule_n>
```

Note that each composite rule is *not* indented.

Each composite rule is of the form:

```
<non-terminal> ->  <word_1>    : "<action template 1>"
               |   <word_2>    : "<action template 2>"
               ...
               |   <word_n>    : "<action template n>"
```

where <non-terminal> is a <symbolic-name>, <word> is either the keyword

```
empty
```

or a space-separated sequence of symbols: either symbolic names or string literals:

<symbol_1> <symbol_2> ... <symbol_m>

If the symbol is a string literal (enclosed in double quotes) it will be interpreted as a terminal symbol, i.e. as a token which must be matched exactly by the input.

If the symbol is a *<symbolic name>*, then it can be either a symbolic name declared in a terminal section or a non-terminal.

If the symbolic name is a terminal, it must be the symbolic name declared for one of the following: a keyword, an operator, a separator, a bracket, an identifier, a number, a string, the `newline` symbolic name if the `usenewlines` directive was

declared, or the `indent` or `dedent` symbolic name if the `useindents` directive was declared.

If the symbolic name is a non-terminal, it must appear as the left-hand side of some rule.

Rule actions Each action template must be a valid Python expression that may use numbered *placeholders* `$1`, `$2`, ..., `$n`, where there must be at most n symbols in the corresponding rule[4]. Placeholders refer to the corresponding symbol in the symbol sequence. When the parser generates the concrete syntax tree, each node in the tree is annotated with the rule that was used to produce the node. Hence, if the right hand side of the rule has n symbols, any tree node that was created with this rule will have n children. Once the parser has generated the full parse tree, the actions are applied by traversing the parse tree in a depth-first fashion. For each node, all its children are visited recursively and the corresponding actions applied. After visiting all the children and collecting the respective results, the action corresponding to the rule of the node is applied with the placeholders replaced by the corresponding values obtained by visiting the children. Hence, a placeholder `$k` will be replaced by the result of applying the actions that yield the k-th symbol of the sequence. Note that even if the word is the empty string keyword `empty`, there must be an action. This would usually be some constant.

4 The grammar compiler command-line options

The grammar compiler is called `apr`. The basic usage is as follows:

`apr` [*options*] <*file*>

where <*file*> is an aperiot file (with `.apr` extension,) and [*options*] is a combination of the options described in table 1.

The aperiot compiler will parse the given aperiot file and depending on the options, it will generate one of two possible representations: a minimalistic representation and an optimized representation.

The minimialistic representation is to be used with the build_parser function, as explained in section 5. The optimized representation is to be used with the load_parser function as explained in section 5.

Whatever the representation, the generated parser for a file named <*filename_base*>`.apr` is saved in a Python package with same name as the base name of the given file with '`_cfg`' appended, i.e. a directory called <*filename_base*>`_cfg`. This directory will be saved by default in the same directory where the source file <*filename_base*>`.apr` is located. This can be modified by using the `-d` option.

The minimalistic representation for a file named <*some_name*>`.apr` will create

[4] Hence if the word is the empty string keyword `empty`, there must not be any placeholders.

`-a`	[Default] Use ASCII format for the generated `.pkl` file. This option is only relevant in conjunction with the `-f` option. The generated file is larger, slower to load, but it is guarranteed to be portable across platforms.
`-b`	Use binary format for the generated .pkl file. This option is only relevant in conjunction with the `-f` option. The generated file is smaller file, faster to load, but may not be portable.
`-m`	[Default] Minimal save: saves only the basic Python representation file. It is enough when using `build_parser`.
`-f`	Full save: saves the generated parser and actions. Necessary when using `load_parser` instead of `build_parser`.
`-d` <*dir*>	Save the generated package in the given target directory <*dir*>.
`-t`	Use the hard coded meta-grammar instead of the bootstrapped meta-grammar. This is used only for debugging and in certain circumstances for setting up the package.
`-v`	Verbose; shows various internal messages.

Table 1: aperiot compiler options.

- *<some_name>*`_cfg`: A directory containing a Python package with that name and which contains the following files:

`__init__.py`	A file to identify the *<some_name>*`_cfg` directory as a Python package.
<some_name>`.py`	The minimalistic Python representation for the grammar.

The optimized representation for a file named *<some_name>*`.apr` will create:

- *<some_name>*`_cfg`: A directory containing a Python package with that name and which contains the following files:

`__init__.py`	A file to identify the *<some_name>*`_cfg` directory as a Python package.
<some_name>`.py`	The minimalistic Python representation for the grammar.
<some_name>`.pkl`	A Python pickled file containing the parser object itself (including the lexer and the CFG object.)
<some_name>`_cfg_ll1_cfg_wan.py`	A Python module containing the normalized form of the Python representation of the grammar.
<some_name>`_cfg_ll1_actions.py`	A Python module containing the source code for the actions associated with grammar rules.

5 The Python API for parsers

Python programs can use a parser generated by `aperiot` by importing the `aperiot.parsergen` module. This module provides the functions shown in table 2.

The `ASCII` and `BINARY` formats are defined also in the same module (`aperiot.parsergen`.)

The `Parser` object returned by `build_parser` and `load_parser` is an instance of the `Parser` class from the `aperiot.llparser` module, and it has the methods shown in table 3.

The `ParseTreeNode` has a method `pprint` for pretty-printing.

Function	Parameters	Returns	Description
`build_parser`	`module_name, verbose=False`	`Parser`	Builds the parser object of the given module, where the module is the minimialistic Python representation produced by `aprcompiler.py`.
`load_parser`	`module_name, verbose=False`	`Parser`	Loads the parser object of the given module, where the module is the full Python representation produced by `aprcompiler.py`.
`save_parser`	`parser, dir, module_name, pickle_format=ASCII, verbose=False`	-	Saves a Parser object in a package named *<module_name>*`_cfg` under the directory *<dir>* using the given pickle format for serialization. The format is either `ASCII` or `BINARY`.

Table 2: `aperiot.parsergen` functions.

Method	Parameters	Returns	Description
`parse`	`input_stream, apply_actions=True`	Any or `ParseTreeNode`	Given an input stream, which may be a `string` or a `file` object, returns the result of parsing and applying the grammar's actions, if `apply_actions` is `True`, or returns the parse tree if `apply_actions` is `False`.
`apply_actions`	`node`	Any	Applies the grammar's actions to a given a `ParseTreeNode`.

Table 3: `Parser` class methods.