Parsing revisited: a transformation-based approach to parser generation.

Ernesto Posse

Modelling, Simulation and Design Lab
School of Computer Science
McGill University
Montreal, Quebec, Canada

Abstract

We present a new parser generator called aperiot based on grammar transformation. This approach retains the complexity and performance advantages of LL(1) parsers while allowing to parse a large set of non-LL(1) grammars and without imposing a lookahead limit. We demonstrate the practicality of this approach with a case-study of a realistic language.

1 Introduction

Parsing is generally considered a "closed" field of research. It appears as if there was a general agreement that all questions have been answered. But are the existing tools the only possible way of doing things? All standard parser generators suffer from different limitations. The big question from a pragmatic point of view is how to negotiate the available trade-offs. Sometimes it pays to revisit an old concept from a fresh angle.

The most common types of parser generators produce either top-down parsers (LL(k)) or bottom-up parsers (LR(k)). Usually the resulting parsers depend on the number (k) of *lookahead* tokens required to decide which rule to apply in case of ambiguities. This results in limitations on the languages that can be recognized as well as incrementing the space complexity of the resulting parser: the number of recognizable grammars grows with larger lookahead, but the space required to store the parsing table also grows, which in turn affects negatively the parser's performance. When k is 1 we have very efficient parsers, but the set of recognizable grammars is seriously restricted, resulting in less expressiveness and more work for grammar designers.

Our approach is simple: transform a non-LL(1) grammar into an LL(1) grammar which can then be parsed efficiently. Although there are grammars which we cannot transform with our current approach, the set of non-LL(1) grammars is large enough to allow us to handle many realistic grammars, and in particular we do not impose a lookahead limit.

It has been long known that any context-free grammar can be translated into an equivalent grammar in Chomsky Normal Form (see for instance [4],) but the drawback is the large number of new rules in the resulting grammar. This appears at first glance to be an impractical approach. Furthermore, most parser generators must deal with actions associated with the rules, and it is not clear what to do with these actions when the grammar is transformed. Here we show that by using a transformation similar to Chomsky normalization, coupled with some simplifying transformations we reduce the number of generated rules enough to make it practical. Furthermore, we show how to deal with rule actions.

We have implemented this approach in a parser generator called aperiot, written entirely in Python, which generates parsers to be used by Python programs.

The rest of the paper is organized as follows: section 2 reviews the basic concepts of context-free grammars and parsing. Section 3 introduces the aperiot grammar description language and the Python API for client applications by means of a simple example. Section 4 discusses ambiguous grammars and the process of translating them into an LL(1) form. Section 5 describes how rule actions are handled by the transformation. Section 6 discusses this approach with respect to traditional approaches to parsing and parser generation. Section 7 discusses a more complex case study of a realistic language. Finally, section 8 provides some final remarks.

2 Background

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. Normally 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 application of actions.

2.1 Context Free Grammars

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:

$$L \to w$$

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 $L \to w$ states that the non-terminal L can be replaced by the sequence w. This is, if there is a sequence

$$uLv$$

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

$$uwv$$

In the context of parsing one may interpret a rule $L \to w$ as stating that if the sequence of symbols w occurs in the input then it can be seen as a single occurrence of the symbol L.

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

$$\begin{array}{rcl} L & \to & w_1 \\ & | & w_2 \\ & \vdots & \\ & | & w_n \end{array}$$

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 a rule is simply a short hand notation for the following set of rules:

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

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

A given text is successfully 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 occurrence 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".

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.

3 A simple example

Using aperiot to generate parsers is straightforward. The basic idea is this: 1) describe the target grammar using aperiot's meta-language, a language similar to standard BNF notations. 2) Use the aperiot grammar compiler to produce one of two possible grammar representations. 3) In the client application, load one of the generated representations using a simple API provided by aperiot, which results in a Python object, the parser, that can parse strings or files given as input.

To illustrate this process as well as aperiot's meta-language, we'll consider a simple language for arithmetic expressions. The application is a simple calculator.

1. The following is a typical grammar for arithmetic expressions written in aperiot's meta-language (saved in a 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 -> FACT             : “$1”
      | FACT times TERM  : “$1 * $3”
      | FACT div TERM    : “$1 / $3”

FACT -> number           : “int($1)”
      | minus FACT       : “-$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 `FACT times TERM : “$1 * $3”`, the placeholder `$1` refers to `FACT`, and `$3` refers to `TERM`. When parsing, if this rule is applied, the result of applying the actions that yield a `FACT` 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. From a grammar file we generate a suitable Python representation of the grammar using aperiot’s grammar compiler.

   On the command-line, we execute the grammar compiler by:

   ```
   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 the client application, we load the generated grammar representation using a simple API provided in aperiot. Loading this representation results in the parser itself, a Python object which can process strings or files given as input.

Assuming that the aperiot package and the directory where we generated `aexpr_cfg` are in the Python path, in the client application we can write, for example:

```
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 outcome is, by default, the result of applying the rule actions. The process of generating the parse tree and applying actions can be explicitly divided by passing an extra argument to the parse method as follows:

```
tree = myparser.parse(text_to_parse,
                      apply_actions=False)
outcome = myparser.apply_actions(tree)
```

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 grammar compilation and saved into a pickle file (with a `.pkl` extension,) which then can be quickly loaded by the application. To do this, we 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 compiled parser object itself. Then, in the client Python application, we use the `load_parser` function instead of the `build_parser` function.

4 From ambiguous grammars to LL(1) grammars

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:

$$\begin{array}{lcl} A & \rightarrow & bm \\ & \mid & bn \end{array}$$

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

$$\begin{array}{lcl} A & \rightarrow & Bm \\ & \mid & Cn \\ B & \rightarrow & x \\ C & \rightarrow & x \end{array}$$

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. An LL(1) parser cannot recognize any ambiguous grammar.

A third approach, and the one used by aperiot, is to transform the grammar from an ambiguous grammar to a non-ambiguous grammar.

The core transformation, known as *left-factoring* (see [1]), is as follows. Suppose there is a rule of the form:

$$\begin{array}{lcl} A & \rightarrow & Bm \\ & \mid & Bn \\ & \mid & p \end{array}$$

Then we can rewrite the rule as

$$\begin{array}{lcl} A & \rightarrow & BA' \\ & \mid & p \end{array}$$

where A' is a new non-terminal symbol, and we introduce a rule:

$$\begin{array}{lcl} A' & \rightarrow & m \\ & \mid & n \end{array}$$

Now, this new rule may itself be ambiguous, so we must apply the transformation repeatedly until there are no more rules to transform, i.e. until we reach a fixed-point.

This basic transformation is generalized to any ambiguous rule where there might be more than two alternatives starting with the same symbol.

This approach is the core of Chomsky normalization[1]. The resulting grammar will recognize the same language but it will also have too many rules, many of which are superfluous. To avoid having too many rules we apply a second simplifying transformation, which gets rid of all *simple* or *unit* rules, i.e. rules that have only one alternative: For any simple or unit rule $A \rightarrow w$, where A is not the start symbol, replace all occurrences of A by w in all other rules, and eliminate the rule $A \rightarrow w$. We repeat this until there are no changes in the grammar. We call this transformation *inlining.* Since we only "inline" non-composite rules, the transformation is guaranteed to terminate.

aperiot's parser generator applies the inlining transformation first, then left-factoring and finally inlining again. In this way we can deal with a large class of ambiguous grammars, even many indirectly ambiguous grammars which would otherwise be rejected by similar parser generators. For example,

$$\begin{array}{lcl} S & \rightarrow & pAq \\ A & \rightarrow & Bm \\ & \mid & Cn \\ B & \rightarrow & x \\ C & \rightarrow & x \end{array}$$

[1]Technically, in Chomsky Normal Form, all rules have the form $A \rightarrow BB'$ or $A \rightarrow x$ where B and B' are non-terminals and x is terminal. In our approach, we do not require B to be non-terminal, and this results in generating less rules.

is an indirectly ambiguous grammar which cannot be handled by a standard LL(1) parser, but aperiot can recognize it by first applying the inlining transformation which results in

$$\begin{array}{lcl} S & \rightarrow & pAq \\ A & \rightarrow & xm \\ & | & xn \end{array}$$

which is then left-factored into

$$\begin{array}{lcl} S & \rightarrow & pAq \\ A & \rightarrow & xA' \\ A' & \rightarrow & m \\ & | & n \end{array}$$

which finally is transformed by inlining again into

$$\begin{array}{lcl} S & \rightarrow & pxA'q \\ A' & \rightarrow & m \\ & | & n \end{array}$$

5 Dealing with rule actions

Grammar rules are annotated with actions to perform computation on the parse tree. Usually these actions involve constructing an abstract syntax tree, but in general they may be any operation on the nodes of the parse tree. As shown in section 3, in aperiot, actions are "Python template expressions," i.e. expressions which use *placeholders* (numbers preceded by '$',) to refer to symbols in the rule.

These actions are transformed by aperiot as Python functions. For example consider the rule

```
TERM -> FACT times TERM : "$1 * $3"
```

from section 3. The generated action looks like this:

```
def term_action_1(x1, x2, x3):
    return x1 * x3
```

where the parameters x1, x2 and x3 represent the outcomes of the rules for the corresponding non-terminal symbols or the tokens for terminal symbols. Note that this function is *not* recursive as might be expected. This is because in aperiot these actions are actually *post-actions* of a depth-first traversal of the parse tree. The parse tree is first built and then visited. When visiting a node the sub-trees are visited first recursively each yielding an outcome from applying actions. These outcomes are collected as an ordered tuple and then the node's action is applied passing as arguments the elements of this tuple.

This basic approach works for LL(1) grammars, but what if the grammar needs to be transformed? We introduce new functions for each transformation.

In the case of the inlining transformation, for each unit rule

$$B \rightarrow \beta \ : g(b_1, ..., b_m)$$

where $g(b_1, ..., b_m)$ is the associated action with m being the length of β, and any rule

$$A \rightarrow \alpha B \gamma \ : f(a_1, ..., a_n, b, c_1, ..., c_k)$$

where $f(a_1, ..., a_n, b, c_1, ..., c_k)$ is the associated action with n and k being the length of α and γ respectively, we eliminate both rules and add a new rule

$$A \rightarrow \alpha\beta\gamma \ : f'(a_1, ..., a_n, b_1, ..., b_m, c_1, ..., c_k)$$

where the new function f' is defined as

$$f'(a_1, ..., a_n, b_1, ..., b_m, c_1, ..., c_k) \overset{def}{=} f(a_1, ..., a_n, g(b_1, ..., b_m), c_1, ..., c_k)$$

Now we describe how to deal with left-factoring: for each composite rule

$$\begin{array}{lcll} A & \rightarrow & B\alpha_1 & : f_1(b, a_1, ..., a_n) \\ & | & B\alpha_2 & : f_2(b, a_1, ..., a_m) \\ & | & \beta & : g(b_1, ..., b_k) \end{array}$$

we replace it by

$$\begin{array}{lcll} A & \rightarrow & BA' & : f'(b, c) \\ & | & \beta & : g(b_1, ..., b_k) \\ A' & \rightarrow & \alpha_1 & : f'_1(a_1, ..., a_n) \\ & | & \alpha_2 & : f'_2(a_1, ..., a_m) \end{array}$$

where f', f'_1 and f'_2 are new functions defined as follows:

$$\begin{array}{rcl} f'(b, c) & \overset{def}{=} & c(b) \\ f'_1(a_1, ..., a_n) & \overset{def}{=} & \lambda x.f_1(x, a_1, ..., a_n) \\ f'_2(a_1, ..., a_m) & \overset{def}{=} & \lambda x.f_2(x, a_1, ..., a_m) \end{array}$$

These functions guarantee the same outcome as the original. To see this, suppose that the rule $A \to B\alpha_1$ is applied in the original grammar, yielding $f_1(b, a_1, ..., a_n)$ as a result, assuming that the outcome of B was b and for the sequence α the outcomes are $a_1, ..., a_n$. We show that the new grammar will produce the same outcome. In the new grammar the rule actions of $A' \to \alpha_1$ and $A \to BA'$ are applied[2]. The first rule yields $f'_1(a_1, ..., a_n)$ and therefore the second rule yields $f'(b, f'_1(a_1, ..., a_n))$ which evaluates to

$$(f'_1(a_1, ..., a_n))(b)$$

this is,

$$(\lambda x. f_1(x, a_1, ..., a_n))(b)$$

which reduces to[3]

$$f_1(b, a_1, ..., a_n)$$

as required.

While this approach works correctly, one problem arises: we introduce many new functions, which take up a lot of space and time to call. Nevertheless we can deal with this problem effectively by doing λ-term reduction when we introduce a new rule and then eliminating all new functions which are not referred to by another function. This strategy eliminates most newly added functions dramatically as demonstrated in section 7.

In order to perform this λ-term reduction, aperiot embeds a simple λ-calculus interpreter which performs these operations. Using Python's own lambda construct does not work because Python only performs execution and does not provide a means to do term simplification.

6 Analysis and comparison with other approaches

Our approach produces LL(1) grammars and therefore the complexity analysis is the same as for LL(1) parsers: parsing time complexity is linear in the length of the input stream, as processing each input token is a constant-time operation consisting mainly of a table lookup. The space used is $O(nm)$ where n is the number of non-terminals and m is the number of terminal symbols. This compares favourably against LL(k) parsers which require in the worst case $n\frac{m!}{(m-k)!}$ entries in the parsing table and parsing time is $O(Nk)$ in the worst case, where N is the length of the input stream. This is due to the fact that each access to the table requires comparing k tokens instead of 1.

There are two fundamental limitations to our approach from the point of view of expressiveness: no handling of left-recursion in rules and limited handling of indirectly ambiguous rules. The first limitation is inherent to LL parsing and can be dealt with using a simple transformation (see [1] for example.) Therefore it is not a good criterion for comparison with other LL parsers. The second limitation is more serious. Section 4 described how by using the inlining transformation before left-factoring we can handle certain grammars with indirect ambiguity, namely grammars such as:

$$\begin{array}{lcl} S & \to & pAq \\ A & \to & Bm \\ & | & Cn \\ B & \to & x \\ C & \to & x \end{array} \tag{1}$$

Here, the indirect ambiguity is caused by the non-terminals B and C, but by the use of inlining we eliminate the problem. aperiot's limitation becomes then obvious: if the rules for B or C were composite rather than simple, for example as in

$$\begin{array}{lcl} S & \to & pAq \\ A & \to & Bm \\ & | & Cn \\ B & \to & x \\ & | & y \\ C & \to & x \\ & | & z \end{array} \tag{2}$$

then the inlining transformation doesn't apply and therefore the ambiguity remains.

[2]They are applied in this order since actions are applied in a post-order traversal of the parse tree, as explained above.

[3]According to the standard semantics of the λ-calculus, see [2].

Inlining composite rules is not a feasible alternative because it amounts to explicitly generating the whole language and is non-terminating.

Nevertheless, by transforming grammars into LL(1) form, aperiot does not require lookahead to handle grammars such as (1) whereas an LL(k) parser will not be able to handle ambiguities beyond k tokens. The set of languages recognized by aperiot is not a subset of LL(k) for any $k \in \mathbb{N}$: for example, the following is not recognized by an LL(2) parser but is recognized by aperiot:

$$\begin{array}{rcl} S & \rightarrow & A \\ A & \rightarrow & abcd \\ & \mid & abce \end{array}$$

The trade-off is then apparent: with aperiot we give up some expressiveness with respect to LL(∞) grammars but we gain in parsing time, space used, parsing generation time and expressiveness with respect to LL(k) grammars.

7 Case study: kiltera

We have used aperiot to generate the parser for a realistic language called "kiltera" (see [3].) Syntactically, kiltera borrows from several languages including Python, ML, Erlang and OCCAM, and therefore has many familiar constructs. Its syntax uses indentation-based nesting.

From the syntax analysis point of view there are a couple of constructs which are particularly interesting, and which are problematic for LL(k) grammars. There are two closely related constructs called "parallel composition" and "indexed parallel composition" which have a similar form but must nevertheless be distinguished. The main syntactic category is called a "process."

A normal parallel composition has the form:

```
par
    process1
    process2
    ...
    processn
```

An indexed parallel composition has the form

```
par
    process1
    process2
    ...
    processn
for pattern in sequence_expression
```

A standard LL(k) parser would not be able to discern between the two and pick up the required rule if the number of tokens in the list of processes exceeds k. Yet, in aperiot this case is easily handled by the following rules:

```
PROCESS -> ...
  |  par SUITE                    : "..."
  |  par SUITE for PATT in EXPR : "..."
SUITE -> newline indent PLIST dedent : "..."
PLIST -> PROCESS        : "..."
       | PROCESS PLIST  : "..."
```

From the point of view of performance it is interesting to see the effect of introducing new rule actions and doing action simplification as described in section 5. We carried out the tests on a 3 GHz Pentium IV on both Linux and Windows XP.

The first version of aperiot did not perform action simplification. It just generated new actions as Python functions. This resulted in massive files which took too long to load. Generating the Python representation for kiltera's grammar took on all runs about 1 minute, and produced a 13MB file which on average took between 9 and 10 seconds to load in client applications. Parsing time of a 3K file took on average about 2 seconds.

The last version of aperiot does perform action simplification at compile time, and the results show a dramatic improvement. Generating the Python representation for kiltera's grammar takes between 3 and 4 seconds, and produces a 300KB file which on average takes less than 1 second to load in client applications. Parsing time of a 3K file is imperceptible.

We obtained similar results doing bootstrapping of aperiot's own meta-language.

8 Final remarks

We have introduced a new parser generator based on the principle of grammar transformation. The main contributions are: 1) showing how to deal with a large class of indirectly ambiguous grammars by means of combining well known transformations, and 2) showing how to deal with rule actions in these transformations. We believe that this approach breathes new air into LL parsing, and in particular allows us to take advantage of the performance benefits of LL(1) parsing while preserving a great deal of expressiveness in the recognizable grammars. Considering this, as well as the light-weight nature of aperiot, we believe it provides a useful alternative to parsing in Python applications.

There are several issues to be addressed in future versions of aperiot. In particular we are interested in adding transformations to eliminate left-recursion and ϵ-rules, as well as adding some syntactic sugar to the meta-language to make it closer to standard BNF notations. Finally, a more flexible lexer will be added as well.

aperiot can be obtained at
`http://moncs.cs.mcgill.ca/projects/aperiot`.

kiltera can be obtained at
`http://moncs.cs.mcgill.ca/projects/kiltera`.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] H. P. Barendregt. *The Lambda Calculus*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, revised edition, 1991.

[3] Ernesto Posse. kiltera: a language for concurrent, interacting, timed mobile systems. Technical Report SOCS-TR-2006.4, School of Computer Science, McGill University, 2006.

[4] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1997.