Top-Down Parsing with Parsing Contexts

A Simple Approach to Context-Sensitive Parsing

Jan Kurš Mircea Lungu Oscar Nierstrasz

Software Composition Group,
University of Bern, Switzerland
scg.unibe.ch

Abstract

The domain of context-free languages has been extensively explored and there exist numerous techniques for parsing (all or a subset of) context-free languages. Unfortunately, some programming languages are not context-free. Using standard context-free parsing techniques to parse a context-sensitive programming language poses a considerable challenge. Implementors of programming language parsers have adopted various techniques, such as hand-written parsers, special lexers, or post-processing of an ambiguous parser output to deal with that challenge.

In this paper we suggest a simple extension of a top-down parser with contextual information. Contrary to the traditional approach that uses only the input stream as an input to a parsing function, we use a parsing context that provides access to a stream and possibly to other context-sensitive information. At a same time we keep the context-free formalism so a grammar definition stays simple without mind-blowing context-sensitive rules. We show that our approach can be used for various purposes such as indent-sensitive parsing, a high-precision island parsing or XML (with arbitrary element names) parsing. We demonstrate our solution with PetitParser, a parsing-expression grammar based, top-down, parser combinator framework written in Smalltalk.

Keywords Parsing Expression Grammars, Semi-Parsing, Top-Down Parsing, PetitParser, Context-Sensitive Parsing

1. Introduction

Context-free grammars (CFGs) [1], which are used to describe context-free languages, are very popular among parser developers. There are numerous parsers for a subset of CFGs (LALR, LR, LL and others [2, 3]) and there are techniques for a full set of CFGs as well (GLR [4], GLL [5]).

Parsing Expression Grammars (PEGs) are another formalism for describing languages [6]. PEGs are closely related to top-down parsing and they are syntactically similar to context-free grammars. PEGs can handle some context-sensitive grammars (CSG), *e.g.*, $a^n b^n c^n$ [6], but they cannot handle all of them.

Unfortunately, some computer languages cannot be expressed with either CFGs or PEGs. For example, `C` is not context-free because of its `typedef` feature. Python [7] has a context-free grammar definition,[1] but it requires a special lexer "on steroids" to generate indent and dedent tokens. Many languages might have an ambiguous context-free grammar because of the famous *dangling else* problem.[2] Even an XML-like language with arbitrary element names (contrary to a finite set of element names) cannot be expressed in CFG. The common approaches to overcome the limitations of context-free parsers is to write a parser manually (*e.g.*, Ruby), or add pre-processing (*e.g.*, Python) or post-processing phase (*e.g.*, XML, dangling else problem). Such approaches are not automated and can be time-consuming and error-prone.

In this paper we suggest a simple extension to top-down parsers that allows for context-sensitive behaviour. We propose to extend the input parameter of a parsing function from an input string to a *parsing context*. A parsing context contains an input string, but it can also contain other information. Any parsing function can access whole context information and is allowed to change it. Because the result of parsing can depend on something other than an input stream, we can increase the computational power of a parser. For example, if a parsing context contains a stack we can reach the computational power of a Turing machine [8]. [3]

[1] `https://docs.python.org/3/reference/grammar.html`

[2] `http://en.wikipedia.org/wiki/Dangling_else`

[3] If a pushdown automaton (context-free parser) is extended with a second stack it can simulate a Turing machine. The first stack simulates a tape to the left of the current position, the second stack simulates a tape to the right of the current position.

Yet we don't want to give up the simplicity and comprehensibility of context-free grammars. We therefore keep the context-free formalism (*i.e.*, rules of the form `N ← X`, where N is nonterminal and X is a sequence of nonterminals and terminals), and we hide the context-sensitivity behind nonterminals that refer to parsing functions utilizing parsing contexts. The rules are universal and can be used (and re-used) in multiple grammars.

We have implemented our idea in PetitParser [9], a top-down PEG-based parser combinator [10] framework using packrat parsing [11] written in Smalltalk.

The contributions of this paper are (i) a description of a simple extension that enables the implementation of context-sensitive features in top-down parsers; (ii) an implementation in PetitParser; and (iii) a brief description how to use parsing contexts to implement universal nonterminals for indentation sensitive parsing, XML parsing or high-precision island parsing.

The paper is organized as follows: section 2 describes our extension. Section 3 shows how to implement our extension in PetitParser and discusses the backtracking, memoization and modularity issues. Section 4 presents how to use our extension to implement context sensitive features, namely indentation-sensitive parsing and high-precision island parsing. Section 6 presents the related work and the section 7 concludes this paper.

The implementation of our extension is available online.[4] The reader is invited to download and explore the examples.

2. Parsing Contexts in a Nutshell

The basic idea of parsing contexts is very simple: Use a parsing context as an input to a parsing function instead of an input stream. A parsing context encapsulates an input stream as well as possibly other information. Any parsing function can access the context and can modify it.

This greatly increases the computational power of a traditional context-free parser without introducing hard-to-read grammatical rules. We seek to parse programming languages that are a subset of context-sensitive languages while aiming for simplicity and comprehensibility. For this reason the rules adhere to a context-free formalism. The rules are still in a form `N ← X`, where N is nonterminal and X is a sequence (possibly empty) of nonterminals and terminals. The context-sensitive behaviour is hidden behind universal nonterminals that can be used in various use cases.

Our solution is applicable to any top-down parser. Top-down parsers use backtracking [12], which provides unlimited lookahead, while using memoization (*i.e.*, caching) to avoid exponential complexity that arises when the same text is repeatedly parsed in backtracking alternatives [11, 13]. It is therefore essential for parsing contexts to support these techniques.

[4] `http://smalltalkhub.com/#!/~JanKurs/PetitParser/`

To enable backtracking, it must be possible to remember and restore contexts. When a top-down parser approaches a decision point, the parsing context is saved; then an alternative is selected. If the given alternative proves to be a good one, parsing continues as usually. If the given alternative fails, the parser restores the context (which might have changed while parsing the alternative) and tries another alternative.

To support memoization, a context has to provide a key to the lookup table of cached results. A standard memoizing parser stores the results of parsing under a key consisting of an $(\mathit{input}, \mathit{position})$ pair. With parsing contexts, the key becomes $(\mathit{input}, \mathit{position}, \mathit{context})$. This functionality overlaps with the remember and restore functionality used in backtracking.

Parsing contexts do not change the semantics of context-free parsing function (*e.g.*, choice, sequence) and imposes almost no performance overhead.

2.1 Using Parsing Contexts

To access a parsing context we use an anonymous parsing function. Its only parameter is a parsing context. The body of the parsing function can be almost any code in a target language. The contract of the parsing function is to return a consumed input (possibly empty) in the case of a successful parse, or to return failure f in the case of an unsuccessful parse. To define a parsing function p, we use the following syntax `N ←[:context | ... ]`. We extend rules of the form `N ← X` with the variant `N ← p` to define context-sensitive nonterminals.

The idea of parsing contexts emerges in combination with a parsing framework that predefines some important context-sensitive parsing functions. These functions are then referred to by universally applicable non-terminals. Thus, a grammar implementor does not need the specialized form `N ← p` and can stick with the familiar context-free formalism.

Take for example an XML-like language. The rule:

```
R ← '<'ID'>' '</'ID'>'
```

is not context-free if we want arbitrary `ID`s to match (and if there is an unbounded number of possible `ID`s [8]), so it cannot be expressed in a context-free form. A developer has to define a context-free grammar that accepts a superset of XML with any `ID` pairs and implement an extra pass to verify if the `ID` pairs match.

Yet, if a framework predefines context-sensitive nonterminals `OPENTAG` and `CLOSETAG` representing the opening and closing of an XML element, we can simply use these nonterminals. The context-sensitive XML-like grammar looks just like a context-free one:

```
start   ← element
element ← OPENTAG content CLOSETAG
content ← element*
ID      ← letter+
```

```
OPENTAG ← [:context |
            result ← ID parse: context.
            context elemStack push: result.
            ↑ result
          ]

CLOSETAG ← [:context |
            result ← ID parse: context.
            (context popStack == result)
              ifTrue: [
                ↑ result.
              ].
            ↑ Failure
          ]
```

Listing 1. Implementation of `OPENTAG` and `CLOSETAG` using parsing contexts and Smalltalk as an implementation language.

The developer does not need to know that there is a hidden (possibly complex) code using parsing contexts, as Listing 1 shows.

Another example, in the case of C-like languages, is a `TYPEDEF` nonterminal that stores the type name into the *type table* in a parsing context, and a `TYPE` nonterminal that succeeds only if it sees the identifier that is in the *type table*.

In case of Python-like layout sensitive-languages, we can define `INDENT` and `DEDENT` tokens. From the user's point of view, they are just non-terminals and their complexity is hidden.

3. PetitParser Implementation

PetitParser is a popular PEG implementation for Smalltalk (see Appendix A). PetitParser is easy to adapt to parsing contexts. It suffices to change the `parse:` method from:

```
PetitParser>>parse: stream
    ...
```

to the context-aware parsing method:

```
PetitParser>>parse: aContext
    ...

Object subclass: #Context
    instanceVariables: 'stream'

Context>>stream
    ↑ stream
```

For convenience we extend `Context` with the `Stream` protocol as depicted in Listing 2.

Contexts are extensible with the help of a *properties* protocol. Properties are stored in a dictionary instance variable and can be accessed and set via `getProperty:` and `setProperty:to:` methods.

```
Context>>next
    "Mimic stream behaviour"
    ↑ stream next

Context>>peek
    "Mimic stream behaviour"
    ↑ stream peek
```

Listing 2. A `Context` mimicry to provide a `Stream` protocol.

```
Context>>remember
    | memento |
    memento ← ContextMemento new.
    memento stream: stream copy.
    self rememberProperties: memento.

    ↑ memento

Context>>restore: memento
    stream ← memento stream copy.
    self restoreProperties: memento.

Context>>rememberProperties: memento
    properties keysAndValuesDo:
    [:key :value |
        memento setProperty: key
                to: value copy.
    ]

Context>>restoreProperties: memento
    memento propertiesKeysAndValuesDo:
    [:key :value |
        self setProperty: key
             to: value copy.
    ]
```

Listing 3. A memento protocol of `Context`.

To save and restore contexts we apply the memento pattern [14] (Listing 3). To ensure that a memento cannot be accidentally changed, setters and getters are implemented using a `copy`.

We use the memento to support backtracking, and to implement a memoizing parser that adopts the memento as a key to the memoization table (Listing 4).

4. Case Studies

We now present two advanced applications of parsing contexts, one to support indentation-sensitive parsing, and another to support island parsing.

4.1 Indentation Sensitive Parsing

Indentation-sensitivity (used in Python, Haskell, and F#) is an interesting feature that is hard to implement in context-free parser.

```
MemoizingParser>>parseOn: context
    | key result |

    key ← context remember.
    result ← memoTable at: key

    result ifNotNil: [
        ↑ result
    ].

    ...
```

Listing 4. An implementation of `MemoizingParser` compatible with `Context`.

Python uses a special lexer to produce context-sensitive tokens *indent* and *dedent*, that represent an increased or decreased indentation of a first word on a line. Python's lexer maintains a special stack to track indentation levels. As the indents token is recognized a new value is pushed to the stack. As the dedent token is recognized a value is popped from the stack.

PetitParser is designed to build scannerless parsers. As such, it offers no easy way to track indentation levels. By extending PetitParser with parsing contexts, we can easily track this additional information with the help of an *indentation stack*. We define two new parsers, `IndentParser` and `DedentParser` that mimic the Python-like indent and dedent tokens. Indent succeeds if a line starts in a column greater than the current one in the indentation stack. Dedent succeeds if a line starts on the column that matches the one at the top of the stack. See Listing 5 and Listing 6. Both parsers access the `Context` and modify the `#indentation` property containing the indentation stack.

We demonstrate their use in Listing 7 where we define an indentation-sensitive rule `suite`. A `suite` is a block of code whose statements are all at the same indentation level.

4.2 Bounded Seas

Island parsing [15] is a form of semi-parsing used to recognise just certain parts of interest in a source file (*i.e.*, the "islands") and ignore the rest (*i.e.*, the "water"). The traditional approach of island parsing defines water as *"anything if everything else fails"*. Such a water is easy to define but it ignores the structure of a grammar.

To illustrate, consider an XML file with a list of items as in Listing 8. Each item contains a set of values. Suppose the XML file is malformed and contains broken `value` pairs. The island grammar allows the malformed `value` pairs to be ignored, but it cannot say which `item` the `value` belongs to. This problem can be solved by using *bounded seas*.

A *bounded sea* is an expression that searches for an island in a scope limited by a boundaries. Boundaries are ex-

```
PPParser subclass: #IndentParser.

IndentParser>>parse: context
    | column indentation stack |

    " If at the begining of a line"
    " consume leading whitespaces "
    (context isBeginOfLine) ifFalse: [
        ↑ Failure
    ].
    context consumeLeadingWhitespace.

    " Save the current column "
    column ← context stream column.
    stack ← (context propertyAt: #indent).
    indentation ← stack top.

    (column > indentation) ifTrue: [
        stack push: column.
        ↑ #indent
    ].
    ↑ Failure
```

Listing 5. An implementation of `IndentParser` that detects a Pyton-like indent token.

```
PPParser subclass: #DedentParser.

DedentParser>>parse: context
    | column referenceIndentation stack |

    (context isBeginOfLine) ifFalse: [
        ↑ Failure
    ].
    context consumeLeadingWhitespace.
    column ← context stream column.

    " Restore previous column from the "
    " stack and compare with current "
    stack ← (context propertyAt: #indent).
    stack pop.
    referenceIndentation ← stack top.

    (column == referenceIndentation) ifTrue
    : [
        ↑ #dedent
    ].
    ↑ Failure
```

Listing 6. An implementation of `DedentParser` that detect a Python-like dedent token.

```
suite     ← (newline indent
              statement+
             dedent)

indent    ← IndentParser new
dedent    ← DedentParser new
newline   ← #newline asParser
statement ← suite / if / for / ···
```

Listing 7. Grammar for a layout-sensitive `suite` rule.

```
<list>
    <item>
        <value>a</value>
        <value>b <value> <!-- Malformed -->
        <value>c</value>
    </item>

    <item>
        <value>d</value>
        <value>e</value>
    <item>
</list>
```

Listing 8. An example of a XML file to parse.

```
start     ← '<list>'
              item*
            '</list>'
item      ← '<item>' valueSea* '</item'>
valueSea  ← ~value~
value     ← '<value>'content'</value>'
content   ← ...
```

Listing 9. A fault-tolerant XML grammar that uses bounded seas.

pressions that appear before and after the island. We use the syntax `~island~` syntax to create a bounded sea from `island`.

To parse a malformed XML file (*e.g.*, the one as in Listing 8) we define a grammar as in Listing 9. `value` from `valueSea` is always searched between `'<item>'` and `'</item'>`. There could be water (*e.g.*, malformed `value`, comments, *etc.*) both before and after `value`. Because the bounded sea never crosses `'<item>'` or `'</item>'` the parser exactly knows which `item` a `value` belongs to.

Such a sea behaviour is not context-free. Boundaries are by definition context-sensitive, because they are basically the rules used before and after a sea. As a result the `valueSea` being called from the rule `start` from Listing 9 fails on input:

‘*</item><item><value>a</value>*’

But the very same rule `valueSea` succeeds being called from `R`, where `R ←valueSea`.

As it turns out, a bounded sea can be implemented as a context-sensitive non-terminal using parsing contexts. Parsing contexts are used to keep a stack of invoked rules. Subsequently, a bounded sea can access the stack and use it to compute its boundaries.

5. Discussion

Albeit very simple and straightforward, the current implementation of a parsing contexts is guilty of exposing global state. Presently, parsing contexts behave as global environments, that can be accessed and modified from any rule. It is a matter of our further research to implement parsing contexts that supports reduced visibility of data.

Our solution sacrifices the linear complexity of Packrat parsing [11] to unlimited complexity, depending on the implemented extensions. For example, the complexity of the indentation-sensitive extension is quadratic in the worst case (at each position we can detect at most n indentation levels, where n is a size of an input). The average complexity is probably better, but this is a matter of further research.

6. Related Work

Attribute grammars [16] extend the possibilities of context-free grammars by introducing attributes and by evaluating them in the nodes of an abstract-syntax tree. Parsing contexts resemble attribute grammars with some important differences: (i) parsing contexts do not filter ambiguous results and are therefore suitable even for non-ambiguous grammars such as PEGs; (ii) parsing contexts directly use the attributes to determine a parsing result; and (iii) parsing contexts hide the attributes, so that a grammar looks looks like a normal context-free grammar (without attributes). Parsing contexts still allow for attributes and do not limit their use.

Context-sensitive grammars [1] are primarily used in linguistics, because context-free grammars cannot describe the phenomena of natural language. Yet, the complexity (PSPACE [17]), understandability and poor semantic suitability led developers to alternatives. In order to specify formal properties of a spoken language, Joshi introduced a mildly context-sensitive grammars [18, 19], that are by definition parsable in polynomial time. There are mildly context-sensitive grammars such as tree adjoining grammars [20], linear context-free rewriting systems [21], or multiple context-free grammars [22].

In contrast to other indentation-sensitive approaches, such as Erdweg *et al.* or Adams [23, 24], our solution a) does not extend BNF notation and b) does not require generalized parsing [23]. The solution we present is specific to Python; the general indentation-sensitive extension is described in a bachelor's thesis [25].

7. Conclusion

In this paper we present a simple extension of PetitParser that allows us to add support for a context sensitive behaviour of XML-like grammars, indentation-sensitive grammars and for high-precision and composable island parsing. We extended an input to a parsing function with a parsing context that can contain information other than an input stream. Parsing contexts are suitable for any top-down parsing technique, because they support memoization, backtracking. Parsing contexts in PetitParser are also extensible and backward compatible.

7.1 Future work

In our future work we plan to investigate the capabilities of parsing-context to capture the most common context-sensitive features of programming languages. Furthermore, we plan to investigate a parsing contexts that supports reduced visibility of data.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

References

[1] N. Chomsky, Three models for the description of language, IRE Transactions on Information Theory 2 (1956) 113–124, http://www.chomsky.info/articles/195609--.pdf.

[2] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques and Tools, Addison Wesley, Reading, Mass., 1986.

[3] A. V. Aho, J. D. Ullman, The Theory of Parsing, Translation and Compiling Volume I: Parsing, Prentice-Hall, 1972.

[4] M. Tomita, Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems, Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[5] E. Scott, A. Johnstone, Gll parsing, Electron. Notes Theor. Comput. Sci. 253 (7) (2010) 177–189. doi:10.1016/j.entcs.2010.08.041.

[6] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 2004, pp. 111–122. doi:10.1145/964001.964011.

[7] Python, http://www.python.org.

[8] M. Sipser, Introduction to the Theory of Computation, 2nd Edition, Course Technology, 2005.

[9] L. Renggli, S. Ducasse, T. Gîrba, O. Nierstrasz, Practical dynamic grammars for dynamic languages, in: 4th Workshop on Dynamic Languages and Applications (DYLA 2010), Malaga, Spain, 2010.

[10] G. Hutton, E. Meijer, Monadic parser combinators, Tech. Rep. NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996).

[11] B. Ford, Packrat parsing: simple, powerful, lazy, linear time, functional pearl, in: ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, Vol. 37/9, ACM, New York, NY, USA, 2002, pp. 36–47. doi:10.1145/583852.581483.

[12] A. Birman, J. D. Ullman, Parsing algorithms with backtrack, IEEE Conference Record of 11th Annual Symposium on Switching and Automata Theory, 1970 (1970) 153–174doi:10.1109/SWAT.1970.18.

[13] P. Norvig, Techniques for automatic memoization with applications to context-free parsing, Computational Linguistics 17 (1) (1991) 91–98, cited By (since 1996).

[14] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional, Reading, Mass., 1995.

[15] L. Moonen, Generating robust parsers using island grammars, in: E. Burd, P. Aiken, R. Koschke (Eds.), Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001), IEEE Computer Society, 2001, pp. 13–22. doi:10.1109/WCRE.2001.957806.

[16] D. Knuth, Semantics of context-free languages, Mathematical systems theory 2 (2) (1968) 127–145. doi:10.1007/BF01692511.

[17] M. Garey, D. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman, San Francisco, 1979.

[18] A. K. Joshi, Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?, Cambridge University Press, 1985.

[19] K. Laura, Parsing Beyond Context-Free Grammars, Springer-Verlag, 2010.

[20] A. Joshi, Y. Schabes, Tree-adjoining grammars (1997).

[21] K. Vijay-Shanker, D. J. Weir, A. K. Joshi, Characterizing structural descriptions produced by various grammatical formalisms, in: Proceedings of the 25th Annual Meeting on Association for Computational Linguistics, ACL '87, Association for Computational Linguistics, Stroudsburg, PA, USA, 1987, pp. 104–111. doi:10.3115/981175.981190.

[22] H. Seki, T. Matsumura, M. Fujii, T. Kasami, On multiple context-free grammars, Theoretical Computer Science 88 (2) (1991) 191 – 229. doi:10.1016/0304-3975(91)90374-B.

[23] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann, Layout-sensitive generalized parsing, in: SLE, 2012, pp. 244–263. doi:10.1007/978-3-642-36089-3_14.

[24] M. D. Adams, Principled parsing for indentation-sensitive languages: Revisiting Landin's offside rule, in: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '13, ACM, New York, NY, USA, 2013, pp. 511–522. doi:10.1145/2429069.2429129.

[25] A. S. Givi, Layout sensitive parsing in Petit Parser framework, Bachelor's thesis, University of Bern (Oct. 2013).

[26] E. Visser, Scannerless generalized-LR parsing, Tech. Rep. P9707, Programming Research Group, University of Amsterdam (Jul. 1997).

A. PetitParser

PetitParser [9] is a parsing framework using four methodologies: a) Parsing Expression Grammars (PEGs); b) Scannerless Parsers [26] that combine lexical and context-free syntax into one grammar; c) Parser Combinators that are building blocks for parsers modeled as a graph of composable objects (they are modular and maintainable, and can be changed, recomposed, transformed and reflected upon); d) and Packrat Parsers that improve performance of PEGs by using memoization.

A.1 Parsing Expression Grammars

PEGs were first introduced by Ford [6] and the formalism is closely related to top-down parsing. PEGs are syntactically similar to CFGs [1], but they have different semantics. The main semantic difference is that the choice operator in PEG is ordered — it selects the first successful match — while the choice operator in CFG is ambiguous. PEGs are composed using the operators in Table 1.

Operator	Description
''	Literal string
[]	Character class
.	Any character
(e)	Grouping
$e?$	Optional
$e*$	Zero-or-more repetitions of e
$e+$	One-or-more repetitions of e
$\&e$	And-predicate, does not consume input
$!e$	Not-predicate, does not consume input
$e_1\ e_2$	Sequence
e_1 / e_2	Prioritized choice

Table 1. Operators for constructing parsing expressions

A.2 PetitParser in Smalltalk

To create a parsing expression as in Table 1, PetitParser uses internal DSL. In this paper we will use a DSL as in Table 2.

Each of the operators is implemented as a subclass of `PPParser`. `PPParser` contains an abstract method `parse:` that accepts an input as an argument and performs the parsing and returns a result or a failure.

Operator	Description
`'abc' asParser`	Literal string
`#any asParser`	Any character
`#newline asParser`	A new line
`(p)`	Grouping
`p ?`	Optional p
`p *`	Zero-or-more repetitions
`p +`	One-or-more repetitions of p
`p and`	And-predicate
`e not`	Not-predicate
`p1 p2`	Sequence
`p1 / p2`	Prioritized choice

Table 2. Operators for constructing parsing expressions