PROJECT REPORT

Opeg - A PEG parser generator for OCaml

Nikolaus Huber

Uppsala University, Uppsala, Sweden

Abstract
Parsing Expression Grammars (PEGs) are a way of specifying the structure (i.e. the grammar) of formal languages. They offer certain traits that make them easier to work with than the more widely used formalism of Context Free Grammars (CFGs). In this project we develop a prototype of a PEG based parser generator for the general purpose programming language OCaml and compare its performance against a framework called Menhir, which relies on a more traditional parsing strategy.

Keywords: compiler; parsing; parsing expression grammar; context free grammar; parser generation

1. Introduction

When tasked with the design and implementation of a new programming language, one of the first software artefacts to be implemented is usually a parser. A parser is a program, which can determine if a given input string forms a correct program according to the grammar of the intended language. The rather mechanical nature of the definition and implementation of parsers has led to the development of so called *parser generators*, which consume a formal description of the grammar of a language and turn it into executable code that in turn can recognize programs written in that language.

Many parser generators are built upon the framework of Context Free Grammars (CFGs), which were first formalized by Chomsky [1]. CFGs offer a convenient way of writing down the structure of a language as a set of (mutually recursive) derivation rules. While powerful, parsing algorithms for general CFGs do not offer linear parsing time in the length of the input.

One way to guarantee linear parsing time is to restrict the input grammar to a subgroup of CFGs for which efficient algorithms exist. Many different subgroups of CFGs have been defined together with algorithms that can parse them efficiently. However, many mainstream programming languages in use today do not fall strictly into one of these subgroups, therefore they often employ tricks and workarounds to push them into a shape that is according to the chosen subgroup. As an example, Python's reference implementation CPython employed a parser generator (named *pgen*) that could create code for an LL(1) grammar while Python's grammar itself is not strictly LL(1). Therefore pgen included workarounds that made it possible to still formalize Python's grammar in it.

A contrary idea building upon a different formalism for grammars arose in the 1970s (see for example [2]) through the development of parsing expression grammars (PEGs). While visually quite similar, the semantics of a grammar given as a PEG is quite different when compared to a CFG. PEGs rely on an ordered choice operator for each derivation rule, thereby making the grammar deterministic. While the theory was developed early on, PEGs have not been widely used, mostly due to their higher memory requirements. Momentum surrounding PEGs has started up again since Bryan Ford demonstrated an efficient implementation scheme in his master thesis [3]. What used to be an academic curiosity is slowly seeing adoption in the developer community. For example, from version 3.10 on CPython is using a PEG parser.

This report showcases the development of a PEG parser generator for the programming language OCaml. It is structured in the following way:

Section 2 will give a general introduction into PEG parsing and sets the objectives for this project. Section 3 illustrates the design and highlights some of the implementation details of Opeg. One of the limiting factors in certain subgroups of CFGs is the lack of support for left recursive rules, therefore we will highlight how Opeg supports left recursion in section 4. To get an idea of Opeg's performance section 5 presents a benchmark and compares to the state of the art tool Menhir [4]. Finally, section 6 lists current limitations and possibilities for further work, before we conclude in section 7. Throughout the report, it is assumed that the reader has prioir experience with CFGs.

2. PEG parsing

In this section we will see an example of a PEG parser for simple arithmetic expressions only consisting of addition, multiplication, and parenthesised subexpressions. The grammar for this simple language can be formalized as a PEG in the following way:

```
expr:
    / term "+" expr
    / term
term:
    / atom "*" term
    / atom
atom:
    / NUM
    / "(" expr ")"
```

The semantics of this grammar are as follows: There are three derivation rules **expr**, **term**, and **atom**. An expression is either an additive clause, consisting of a term on the left hand and an expression on the right hand, or it is simply a term. Similarly, a term is either a multiplicative clause or a simple atom, and an atom can either be a numerical constant, or a parenthesised expression.

The above given grammar visually resembles a CFG describing the same language, the only difference being the usage of the choice operator ("/") instead of the usual bar operator ("|") commonly used in CFGs. The choice operator makes PEGs unambiguous by design. When trying to parse an expression, we we always try to parse an additive clause first, and only if that results in a failure to parse we will try the next alternative. In CFGs, the different right hand sides of a derivation rule have no defined order, so the CFG analogon of the above grammar would result in a shift/reduce conflict.

This also means, that we need to pay attention to the order in which we define the choices. If we would have defined the **expr** rule different (i.e. by switching the order of the two alternatives), the additive clause would never get tried, given that there would be a choice of higher priority that parses a prefix of the same input the additive clause parses.

The alert reader may have realized, that both the addition and multiplication actually have the wrong associativity in the given grammar, i.e. mathematically they are left associative, however, due to the right recursive structure of the respective clauses, they are parsed as if they were right associative. We will cover this in section 4 and just note here, that for addition and multiplication the associativity does not matter, since the evaluation will be the same.

The structure of a PEG translates easily into a concept know as *recursive decent parsing*. We can therefore write down the above code in the following way:

```
type 'a parse_result =
    | Parse of 'a
    | No_parse

let rec expr =
    let alt1 () =
        let* t = term in
        let* _ = expect "+" in
        let* e = expr in
        Parse (* Create AST node here *)
    in

    let alt2 () =
        let* t = term in
        Parse (* Create AST node here *)
    in

    match alt1 () with
    | Parse x -> Parse x
    | No_parse -> (* backtrack *);
        match alt2 () with
        | Parse x -> Parse x
        | No_parse -> (* backtrack *); No_parse

and term = (* ... *)
```

The above code is a slightly simplified version of what Opeg would generate for our example grammar. We can see, that the **expr** rule has been turned into a recursive function, which itself defines two inner functions, each corresponding to one choice in the original grammar. All of these functions return a result of type **parse_result**, which either holds the parsed Abstract Syntax Tree (AST) node, or the special value **No_parse**. The **let*** is a custom let expression which allows for easy chaining of parsing expressions. It is similar to a monadic bind operation in that it will forward a parsed result to the next operation, but circumvent additional computation should one of the parse functions fail to parse at the current position.

As mentioned, the above code is simplified in that it leaves out parts that are not relevant for understanding the basic functioning. In a real implementation each parse function would take in additional parameters such as the current position in the input. We also only hinted on the code responsible for *backtracking*, i.e. resetting the parsers current position in the input in case that a choice (or rule) fails to parse and we therefore try the next one.

If we look closely at the code above we realize, that this method of parsing does not scale linearly with the length of the input. For example, if the first alternative for an expression fails to parse at a particular position, the second alternative will try to parse a term at exactly the same position. However, a term was also the first symbol in the first alternative, therefore by that time we will have already tried to parse a term at that position.

This realization lead to the development of the *packrat parsing technique*, first described and implemented in the programming language Haskell by Bryon Ford in his master thesis [3]. The basic idea is, that we remember each parse result at each position that we have tried, so that no redundant computation needs to be performed. This can easily be done by using hash tables, which use a combination of rule name and input position as a key to store parsed results. In functional languages that employ a lazy, non-strict evaluation strategy, one can do without hash tables as well. Since OCaml is an eager and strict language, we need to use memoization instead. For any grammar with k rules, the memoization table for an input of length n can only have $k \cdot n$ different parse results, and since the list of rules for a given grammar is constant, this means that parsing with memoization will be linear in the length of the input (assuming that operations related to memoization happen in constant time).

3. Development of Opeg

Section 2 hinted on how a PEG parser can be implemented by hand. While always possible, the repetitive nature of the code together with the clear translation scheme from PEG rule to the resulting code lends itself well to automation. Therefore, this section presents some of the implementation details of Opeg, the PEG parser generator developed for this project.

Opeg consumes a grammar description given in a custom definition language, which was developed alongside the tool, and produces OCaml code, which implements a parser that recognizes strings of the given grammar.

In principle, PEG parsers can take care of both the lexical and syntax analysis of a given input string. However, to simplify the implementation, and in order for the tool to act as a drop-in replacement for Menhir, we chose to keep the traditional distinction between these two phases. Therefore, we still need to tokenize the string first, which can be done with *ocamllex*, a tool that comes with the standard OCaml distribution.

As already hinted in the previous section, in order to guarantee linear parsing time we need to incorporate memoization. Opeg therefore creates a hash table for each rule that is not the starting rule (we usually don't call the starting rule multiple times on the same input position).

The development of a parser generator is quite similar to that of a simple compiler. Specifically, it has the same standard structure of phases, starting with a front-end which is responsible for reading in a grammar description, a middle-end performing analysis and optimization on the grammar, and a back-end that generates the actual parser code. So the question arises, how does one develop the parser of a parser generator? There are two obvious answers: One could write the parser by hand, or one could rely on a different parser generator. While the first one is more work intensive, the second one adds an additional build dependency to the project. Since the standard package manager of OCaml, opam[1], installs packages by building them from source, this would mean, that anyone installing Opeg would also need to install another parser generator. In order to remove that dependency, the Opeg project actually incorporates two parser generators.

The first, named *Opeg_boot* relies on Menhir in order to create the parser for the custom defined grammar description language. With the help of Opeg_boot we can then generate a parser for the grammar description language by defining the grammar of this grammar language in a meta-grammar. Once OCaml code has been produced for this parser it does not rely on Menhir anymore. This bootstrapped parser can then be installed without any dependencies outside of the project. The meta-grammar for Opeg's grammar description language is shown in appendix A.

4. Left recursion in Opeg

In section 2 we showed a simple grammar for arithmetic expressions including only addition and multiplication. This choice was not only done for simplicity, but also for the fact that for these two operations associativity does not matter, i.e. $(1 + 2) + 3$ evaluates to the same as $1 + (2 + 3)$. It was therefore possible to write down the grammar in a right-recursive way.

[1]https://opam.ocaml.org

Mathematically speaking, addition is left associative though, which is lost in the structure of the grammar. While we could rewrite the grammar in this case, if we would like to incorporate subtraction, this is not as easy, given that $(1-2)-3$ is not not the same as $1-(2-3)$. It is a well known fact, that left recursive CFGs can be rewritten so that they are right recursive while still accepting the same language. However, these transformations alter the structure of the grammar, and obscure the notion of associativity. Therefore, we incorporated support for a specific type of left recursion in Opeg.

There are three different types of left recursion that can occur. The simplest one is *direct left recursion*, where at least one alternative for a rule refers to the rule itself as the first symbol in the derivation:

```
expr: expr "+" term
```

Indirect left recursion occurs when a concatenation of derivations leads to the same rule being applied at the same input position:

```
a: b "A"
b: a "B"
```

Hidden left recursion happens when the current rule is used as a non-terminal symbol after a sequence of other symbols that might accept the empty string:

```
a: "A"? a
b: "B"* b
```

Here **a** is hidden left recursive, since the first symbol is optional, and **b** is hidden left recursive, since the first symbol accepts a list of **"B"**s, which might be empty. In general, hidden left recursion is difficult to find, since the sequence of symbols before the self reference might include other non-terminal (i.e rule) symbols, for which we then need to decide if they can accept the empty string.

Opeg currently only supports direct left recursion, since it seems to be the most common and natural way that left recursion appears in the grammar of modern programming languages.

The way Opeg incorporates support for direct left recursion roughly follows the algorithm presented in [5] with the addition of using the memoization cache as suggested by Guido van Rossum in a series of blog posts[2] about the development of a PEG parser generator for the Python programming language. We will repeat the basic idea here and refer the interested reader to the above mentioned paper for further information and a proof of the correctness of the method.

Assume we want to parse the string "1+2+3" according to this direct left recursive rule, where **NUM** is a terminal symbol that stands for an integer constant:

```
add: add "+" NUM / NUM
```

The idea is to iterate over this rule, always using the last parsed result for the occurrence of the **add** symbol on the right side of the rule, until the parsed string becomes maximal. For the string "1+2+3" this would look as follows:

If we assume that the first time we want to call **add** on the righthand side of the rule it returns no parse result (i.e. add_0 = **No parse**), we will then continue using the second choice, which is just an

[2]https://medium.com/@gvanrossum_83706/peg-parsing-series-de5d41b2ed60

integer constant. This will successfully parse the first number in the expression, i.e. $add_1 = 1$. We try the rule again, which means this time we will get $add_2 = add_1$**"+" NUM** which successfully parses the substring "1+2". Iterating a third time gives us $add_3 = add_2$**"+" NUM** = (1 **"+"** 2) **"+" NUM**, which successfully parses the whole string. If we now try to iterate again, the first rule will fail (there is no more **"+"** or **NUM** token to consume), therefore the second choice will be taken again, this time only matching the first number of the string.

Therefore, since the last iteration gave a result that is shorter than the longest result we had previously parsed, we will return this longest parse. The idea presented by Guido van Rossum cleverly uses the hash table to iterate over a direct left recursive rule, i.e. we can always get the last parsed result from the hash table until the next matched result is shorter, at which point the hash table already has the longest possible parse.

5. Performance analysis

To get an idea of the performance of Opeg we compare it to the state of the art tool for parser generation in OCaml, a tool called Menhir [4]. Menhir is a parser generator that consumes a grammar belonging to the LR(1) subgroup of CFGs. Menhir offers an extensive set of additional features (i.e. a standard library for dealing with separated lists, optional symbols, etc.), and offers methods of dealing with ambiguities in the grammar that are difficult or unnatural to express well in a CFG (e.g. precedence of binary operators). In order to compare Opeg to Menhir, a parser needs to be implemented in both, and sufficient test programs (i.e. input strings) must be available. Due to the author's time constraints, developing two full parsers for a mainstream programming language was infeasible. On the other side, using a toy grammar means that there won't be a big enough corpus of test programs available to thoroughly compare the two generated parsers.

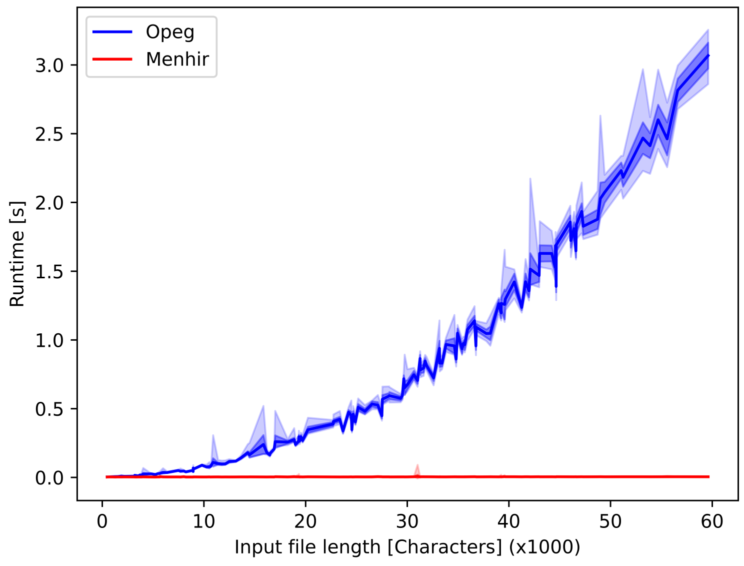

Figure 1. Runtime comparison results for different input lengths.

Therefore, a somewhat niche language has been chosen. Fexl[3] is a simple functional language with a syntax close to lambda calculus. It has a (somewhat formal) description of its grammar available online[4]. The input grammars to both Menhir and Opeg are given in appendix B and C respectively.

The test inputs come from Fexl's standard library. Since we are only interested in parsing syntactically valid files (i.e. they do not have to make sense semantically) we can concatenate files to create longer (syntactically valid) inputs. We can therefore run both parsers on inputs of arbitrary length by randomly combining the files of the standard library.

Figure 1 shows the measured results. Each input file has been run 50 times with both generated parsers, and the mean, standard deviation, minimum and maximum runtime recorded. The solid line represents the measured mean, the dark shaded area around it the (symmetric) standard deviation, and the light shaded area illustrate the whole range of measured values per input file.

The result illustrates, that while Menhir clearly performs better at all input lengths, the parsing time for the parser generated by Opeg grows at most linearly with the input length.

6. Limitations and future work

There are currently multiple limitations in Opeg, some of which we would like to highlight in this section and give ideas about future work that might alleviate them.

At the moment, Opeg only supports direct left recursion. Existing work [5, 6] shows methods of supporting other types of left recursion as well, which would make the grammar description language that Opeg can accept more expressive.

Currently, only minimal analysis is performed on the input grammar to Opeg before code is produced. There are no warnings generated in case that unsupported forms of left recursion are used, or if there is an infinite loop in the code (i.e. a way of deriving the same symbol again without consuming any input). It seems that these types of analysis are similar to those performed by CFG based parser generators, and rely on standard methods from graph theory. Incorporating a more thorough analysis pass in Opeg would make it easier and safer to use. An analysis particular to PEG parsers would be to look for redundant choices, e.g. when a developer defined two clauses where one is a prefix of the other but is itself a choice of higher priority. This would mean that the longer clause will never be used, since the shorter one accepts a prefix of it and comes first. A user should be warned about this type of mistake early on.

Right now, Opeg has very limited error reporting. This makes debugging grammars particularly difficult. A tighter integration with the lexer generator could help give better error messages.

While it was a particular design choice for this project to separate lexing and parsing into two distinct phases, PEG parsers have the potential to unify them. It would be an interesting exercise to incoporate support for regular expressions into the language of Opeg so that one could do away with separate lexical analysis.

7. Conclusion

This report illustrated the development of Opeg, a PEG parser generator for the OCaml programming language. We showcased the general idea behind PEG parsing, the design principles and implementation details of the developed tool, and compared its performance against the state of the art.

[3]http://fexl.com
[4]http://fexl.com/grammar/

While currently clearly offering worse performance compared to the competitor, Opeg offers certain traits that can make it an interesting alternative for particular usage cases:

The code that is produced by Menhir is obscure and difficult to understand. It bears little resemblance to the input grammar given by the developer, therefore one cannot easily alter and play round with the generated code. Especially for educational purposes, it might be interesting to use a tool where there is a clear connection between the input grammar and the generated code.

The ability to combine both lexing and parsing into one framework has interesting applications. It allows to define parsers for only parts of a language, which can then be combined to parse bigger fragments. While this is also possible with CFG based parsers and has in fact been around for quite a while under the name of parser combinators, there are certain problems to that method. The combination of two unambiguous CFGs does not necessarily result in an unambiguous CFG, even less so when restricted to a certain subgroup. PEG parsers offer closure under composition, and are unambiguous by design. This makes them an interesting alternative for projects like Miking [7], where language fragments can be combined to form more expressive languages. One could for example imagine a whole language development framework, where parsers can be interactively combined and the result being illustrated on a given test input. For such a use case, PEG parsers offer many advantages over CFG based ones. A somewhat related project is the experimental programming language Katahdin [8], which allows changing the language's syntax during runtime. It uses PEGs to dynamically describe the grammar of the language.

Currently, to the best of the author's knowledge, there are not other available PEG based parser generators for OCaml. The only related project is a tool called *Aurochs*[5] that turns a PEG based description of a language's grammar into custom byte code. This byte code can then be executed by an accompanying interpreter which is available in different languages including OCaml. However, Aurochs does not offer semantic actions, therefore one cannot easily create an AST from a given input. It also seems like the project is unmaintained, since the last recorded development effort was in 2010.

The source code for Opeg is available online[6].

[5]http://aurochs.fr
[6]https://github.com/nikolaushuber/opeg

References

[1] N. Chomsky. "Three models for the description of language". In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124. DOI: 10.1109/TIT.1956.1056813.

[2] Alexander Birman and Jeffrey D. Ullman. "Parsing algorithms with backtrack". In: *11th Annual Symposium on Switching and Automata Theory (swat 1970)*. 1970, pp. 153–174. DOI: 10.1109/SWAT.1970.18.

[3] Bryan Ford and M. Frans Kaashoek. *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*. 2002.

[4] François Pottier and Yann Régis-Giannis. *The Menhir parser generator*. http://gallium.inria.fr/~fpottier/menhir/. Accessed: 2022-09-01.

[5] Alessandro Warth, James R. Douglass, and Todd Millstein. "Packrat Parsers Can Support Left Recursion". In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 103–110. ISBN: 9781595939777. DOI: 10.1145/1328408.1328424. URL: https://doi.org/10.1145/1328408.1328424.

[6] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. "Left recursion in Parsing Expression Grammars". In: *Science of Computer Programming* 96 (2014). Selected and extended papers of the Brazilian Symposium on Programming Languages 2012 (SBLP 2012), pp. 177–190. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2014.01.013. URL: https://www.sciencedirect.com/science/article/pii/S0167642314000288.

[7] David Broman. "A Vision of Miking: Interactive Programmatic Modeling, Sound Language Composition, and Self-Learning Compilation". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 55–60. ISBN: 9781450369817. DOI: 10.1145/3357766.3359531. URL: https://doi.org/10.1145/3357766.3359531.

[8] Chris Seaton. *Katahdin: Mutating a Programming Language's Syntax and Semantics at Runtime*. Nov. 2015.

A. Parser generator meta-grammar

```
token:
    | TOK_KW_PARSER "parser"
    | TOK_KW_TOKEN "token"
    | TOK_HEADER <string>
    | TOK_SEM_ACTION <string>
    | TOK_NAME <string>
    | TOK_TYPE <string>
    | TOK_DOUBLE_DOTS ":"
    | TOK_CHOICE "/"
    | TOK_BAR "|"
    | TOK_EQUALS "="
    | TOK_OPTION "?"
    | TOK_STAR "*"
    | TOK_PLUS "+"
    | TOK_SEC_DIVIDE "%%"
    | TOK_EOF "eof"

parser "parse" start

%%

start <Parsetree.t>:
    / hd = TOK_HEADER? "token" ":" tl = tok+ "parser" name = TOK_NAME
    start_deriv = TOK_NAME "%%" rl = rule+ "eof"
        {
            {
                header = hd;
                parser_name = name;
                start_deriv = start_deriv;
                tokens = tl;
                rules = rl;
            }
        }

tok <Parsetree.pt_tok>:
    / "|" n = TOK_NAME t = TOK_TYPE? short = TOK_NAME? { (n, t, short) }

rule <Parsetree.pt_rule>:
    / n = TOK_NAME t = TOK_TYPE ":" dl = alt+ { (n, t, dl) }

alt <Parsetree.pt_alt>:
    / "/" sl = symbol* act = TOK_SEM_ACTION { (sl, act) }

symbol <Parsetree.pt_symb>:
    / syn = TOK_NAME "=" n = TOK_NAME suff = suffix? { (n, Some syn, suff) }
    / n = TOK_NAME suff = suffix? { (n, None, suff) }

suffix <Parsetree.symb_suffix>:
    / "?" { Optional }
    / "+" { Plus }
    / "*" { Star }
```

B. Fexl - Menhir grammar

```
%token          SEMI            ";"
%token          BACKSLASH       "\\"
%token          EQUAL           "="
%token          DOUBLE_EQUAL    "=="
%token          LPAREN          "("
%token          RPAREN          ")"
%token          LBRACK          "["
%token          RBRACK          "]"
%token          LCURLY          "{"
%token          RCURLY          "}"
%token <string> NAME
%token <string> STRING
%token          EOF             "eof"

%start <bool> fexl

%%

fexl:
    | expr* "eof" { true }

expr:
    | term { true }
    | ";" { true }
    | "\\" NAME "=" term { true }
    | "\\" NAME "==" term { true }
    | "\\" NAME { true }
    | "\\" ";" { true }
    | "\\" "=" { true }

term:
    | "(" expr* ")" { true }
    | "[" items? "]" { true }
    | "{" items? "}" { true }
    | NAME { true }
    | STRING { true }

items:
    | term items? { true }
    | ";" expr { true }
```

C. Fexl - Opeg grammar

```
token:
    | SEMI ";"
    | BACKSLASH "\\"
    | EQUAL "="
    | DOUBLE_EQUAL "=="
    | LPAREN "("
    | RPAREN ")"
    | LBRACK "["
    | RBRACK "]"
    | LCURLY "{"
    | RCURLY "}"
    | NAME <string>
    | STRING <string>
    | EOF "eof"

parser "parse" fexl

%%

fexl <bool>:
    / expr "eof" { true }

expr <bool>:
    / term expr { true }
    / ";" expr { true }
    / "\\" _ = NAME "=" term expr { true }
    / "\\" _ = NAME "==" term expr { true }
    / "\\" _ = NAME expr { true }
    / "\\" ";" expr { true }
    / "\\" "=" expr { true }
    / { true }

term <bool>:
    / "(" expr ")" { true }
    / "[" items "]" { true }
    / "{" items "}" { true }
    / _ = NAME { true }
    / _ = STRING { true }

items <bool>:
    / term items { true }
    / ";" expr { true }
    / { true }
```