A Validated Parser for Stan

Brian Ward *Advisors: Joseph Tassarotti, Jean-Baptiste Tristan*

Boston College Computer Science Department

Abstract

Before any code can be interpreted, compiled, or otherwise processed, it must first be parsed. This process of mechanically reading and interpreting code is a well-trodden area of compiler design, and one that has many existing tools and methods. A programmer can succinctly specify the syntax of a language and use a variety of tools, known as parser generators, to create code which implements the translation from that syntax into a tree structure for evaluation or compilation.

Recent developments in the field allow for a way to be certain that this translation is done following the desired specification. Rather than simply trusting a handwritten parser, or trusting those who write parser generators such as `yacc`, it is now possible to use formal methods to *prove* that the parsing code is correct with respect to the specification that was used to create it.

This thesis is an exploration of using said methods to produce a verified parser for the probabilistic programming language Stan, a common statistical modeling and computation language. To this end, an existing tool for parser generation and verification, Menhir, was extended to allow error reporting functionality alongside its validated output.

Contents

1 Introduction

The parsing of a program happens immediately following the reading of a source file, and it is the step that enables everything that follows. As soon as a program has been read and split into tokens (a stage called lexical analysis, or more simply, lexing), it is compared to the formal specification of the language and interpreted into a more organized form than mere text. This can be done through a variety of methods, with one of the most common being the use of tools known as parser generators to automatically create code from simple specifications of the grammar.

Through the use of one such parser generator, Menhir, it is now possible to generate parsers that are also accompanied by automatically-verified proofs of certain desirable qualities. Most notably, one can now be certain that the language the generated code recognizes will precisely match the formal specification given. This thesis will investigate the use of this tool to parse the language Stan and also extend it with some useful functionality for practical implementations seeking to gain the advantages granted by this verified mode.

This section is intended to bring someone generally familiar with the related methods and theory up to speed on the salient details of verification, parsing, and the Stan language. It is not intended as a reference for learning from scratch: the reader is referred to the many authoritative sources on the subjects [3, 8, 14, 16].

1.1 Parsing

Parsers are functions that receive a stream of tokens and verify that they match the required specification of the language being parsed. In this context, the term *language* simply means a specific set of strings. If this verification succeeds, the parser usually produces a parse tree which is then passed on to the remainder of the compiler. This result is commonly referred to as an abstract syntax tree, or simply AST, and it represents the source program in a hierarchical structure. If instead the parser rejects the input, it is typical to give an error message describing the problem with the input string. There are a variety of methods for parsing, including both hand-written approaches and the parser generators discussed in this thesis [4].

1.1.1 Context Free Grammars

Any method of parsing must begin with a formal specification of the syntax of the language. The main method of specification is through the use of *context-free grammars*, a formal structure defining a language. These constructions can describe a broader class of languages than the more commonly seen regular expressions. Context-free grammars (CFGs) are made up of four main components [3, 4]:

1. A set of *non-terminal symbols* that denote sets of strings.
2. A *start symbol*, a distinguished non-terminal which is used to begin all language derivations.
3. A set of *terminal symbols* that are the basic symbols which make up the language. In most cases, this set corresponds directly to the set of tokens.

4. A set of *productions* that provide rules for translating from a non-terminal to terminal and non-terminal symbols. By convention, a set of productions is generally used as the complete specification of the language, with the start symbol's productions listed first. Productions are written as $nt \rightarrow derivation$ or nt `::=` $derivation$, in a format known as Backus-Naur form [2]. If there are multiple productions for a given non-terminal, the various options can be written as $x \mid y$.

This is usually easier to understand by an example. Perhaps the simplest non-trivial CFG is the following single production:

$$expr \;\; \texttt{::=} \;\; \text{‘}\mathbf{x}\text{’} \mid expr \;\text{‘}+\text{’}\; expr$$

This grammar contains one non-terminal (*expr*) and two terminal (x, +) symbols. It defines an infinite language of repeated additions. Some members of this language are x, $x + x$, and $x + x + x + x + x$. Because this production features the same non-terminal on both sides of the translation, it is said to be a recursive rule. It is possible to be more specific in some cases, e.g. a rule a `::=` a **‘.’** | **‘.’** is not just recursive but *left*-recursive, as the only recursive application is to the left of a non-recursive string. Right recursion for grammar rules is similarly defined [4].

Parsing a language specified by a CFG is the process of finding a *derivation* for a given string using the grammar's productions. Beginning with the start symbol, at each step a non-terminal is re-written using one of the options that appear in the body of its corresponding production.[1] Any string for which a derivation exists is said to be *in the language* or *accepted* by the grammar. A derivation step is conventionally denoted by $\Rightarrow$. One derivation of the string $x + x + x$ in the above language is $expr \Rightarrow expr \;+\; expr \Rightarrow x + expr \Rightarrow x + x$. It is immediately obvious that this derivation is not unique — in the penultimate step, one could choose either *expr* to replace first. By preferring the left instance, we have created a *leftmost-derivation*. In more complicated grammars, it may be unclear which order productions are applied in, even if a derivation direction is preferred. A grammar with more than one leftmost- or rightmost-derivation is said to be *ambiguous* [4].

```
   +               *
  / \             / \
 4   *           +   2
    / \         / \
   3   2       4   3
```

Fig. 1: Two possible parse trees for the same expression, $4 + 3 * 2$, which lead to different numerical evaluations.

Ambiguity can lead to several issues, most notably when it comes to associativity and precedence of mathematical expressions in the AST produced by the parser [4]. The specific

[1] This describes the top-down conception of parsing. An alternative form known as bottom-up parsing is actually used in generated parsers, but the distinction is not important for this work, and bottom-up parsing is less intuitive.

derivation the parser enacts is reflected in the structure of the tree created. While the most basic of grammars would yield several ways to parse "1 + 2 * 3 - 1 - 1" and confirm it is a valid form of arithmetic, not all of these would produce a corresponding tree that evaluates the multiplicative portion before the addition in compliance with standard precedence while also enforcing the left-associativity of the minus operator.

There are two common ways of resolving ambiguity in parsing. The first is altering the grammar so that it is unambiguous – this often includes introducing additional non-terminals into the grammar to force the parsing of certain subexpressions before others, and the use of left- or right-recursive productions to ensure left or right associativity respectively [4, 8]. The second method, enabled by some parser generators (see 1.1.2), is the ability to annotate tokens with associativity or precedence information separately from the rules of the grammar [15]. Both of these will methods be discussed further in Section 3.

1.1.2 Parser Generators

Parser generators are programs for automatically producing parsers from specifications. The idealized parser generator takes as input a context-free grammar that has been annotated with semantic actions to perform whenever a production rule is executed and outputs a piece of code that parses the language specified by that grammar. In practice, not all context-free grammars are allowed as the input of a parser generator. The generated parsers are typically LR(k) parsers, This means the parser scans the input from left to right ("L"), produces a rightmost derivation ("R"), and views the next k input symbols for each parsing action (typically $k = 0$ or $k = 1$). Sometimes an even further restriction on the grammar, known as LALR or lookahead-LR, is imposed. In practice, the grammars of most programming languages can be parsed by LR/LALR parsers, but in general there are CFGs which cannot be [4].

The structure of a parser produced by a parser generator differs greatly from many hand-written approaches, but the internal workings are not necessary to understand. To make some of the later discussion clearer, suffice it to know that generated parsers generally model *pushdown automata*, meaning finite automata with the ability to use a stack [8]. This is why the terms "automaton" and "stack" will both figure into discussions of the verified parser.

1.2 Stan

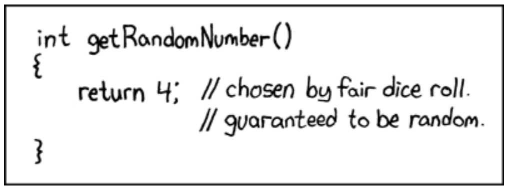
Stan is a compiled probabilistic programming language (PPL) [16]. This means that it allows users to specify probabilistic models in code, and then inference for these models is done (semi-)automatically. This is interesting from the standpoint of a compiler writer because a compiler for Stan does *very* different things than a standard compiler.

A Stan program is an implementation of a probability density function of a random distribution.[2] The compiler generates code that can produce samples from this probability density. In the case of existing Stan compilers, this code is in usually in a high-level language such as C++ [1]. This is in contrast to a 'traditional' compiler that translates from a high-level language to assembly code or other low-level representation with the goal being

[2] For numerical accuracy, it is actually an implementation of the log-density, but the idea is the same.

the preservation of semantic meaning during this translation. Stan programs must change meaning during compilation from a description of a random process into a program that simulates that randomness. It is worth noting that while the output programs may be pseudo-random, the compilation itself is deterministic.

While the nature of Stan as a PPL is not necessary to understand in order to parse it, this does provide one of the philosophical underpinnings for this project. Testing that the compilation translation was done properly can be very difficult because programs that simulate randomness are particularly hard to test by traditional means. Take for example a function that returns a value between 0 and 1 uniformly at random. How does one test that this function is well-behaved? It is hard to say much more about an individual run of this function than "the result is between 0 and 1". Confirming even this is difficult through testing - if it actually returned 1.1 with a probability of 1 in $1,000,000$, it is likely any run of tests would never catch this behavior. However, even if one could test this, a function that returns 0.5 every time would pass this requirement, and still be remarkably incorrect compared to the desired behavior. While this is a contrived example, there are demonstrated instances of bias in the programs existing Stan compilers produced, which were very difficult to spot [7].



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Fig. 2: A very poor randomized program, via xkcd.

One could instead record the results of many runs of said function and then confirm afterward that they are reasonably close to what would be expected from the specified distribution – essentially, an appeal to the Law of Large Numbers. Ignoring the question of what "reasonably close" would mean, this strategy works only if the distribution is well-known and feasible to calculate many times. For programs that are more complicated than our toy example it is easy to imagine how this can quickly become difficult to manage. This is why we instead seek to use formal methods (See 1.3) to *prove* the correctness of the program, rather than *test* it after the fact. The task of formally verifying a compiler for Stan begins with parsing, hence the existence of this work.

Actually parsing Stan is not greatly influenced by its nature as a PPL. The language features a largely C-like syntax for expressions, control flow, and functions, with block structured programs. A simple example is showing in Figure 3. The meaning of these blocks is unique to Stan [16, Ref. Manual §8], but the syntactic structure easily recognizable to users of other languages.

In addition to these unique blocks, there are a few operators (such as ~) that are uncommon, and the ability to give constraints (bounds) to the domain of declared variables. The full syntax is given in the Reference Manual as a Backus-Naur form context free grammar [16, §1.11], though there are notable differences between this grammar and those actually used in practice in the existing Stan compiler. These will be discussed more in Section 3.

```
data {
        int<lower=0> N;
        vector[N] x;
        vector[N] y;
}
parameters {
        real alpha;
        real beta;
        real<lower=0> sigma;
}
model {
        y ~ normal(alpha + beta * x, sigma);
}
```

Fig. 3: An example linear regression Stan model [16, User Guide §1.1].

1.3 Formal Verification

Formal Verification is the practice of proving properties of a program through standard mathematical tools. Such proofs are often performed in a semi-automatic fashion with the aid of a *proof assistant*. These are tools that automate some basic parts of proofs while requiring the programmer to provide the remaining steps. Coq is both a proof assistant and a functional programming language [14]. This means one can both write programs and prove properties about them in the same language. That is to say one writes software that is *correct by construction*.

There are several intellectual hang-ups associated with formal verification of software. If software can be written correctly without testing, why is that not the standard practice? Is it even possible to have a proof written in a way computers can understand, check, and assist with?

The answer to both of these hinges on the work the programmer must do. It is easy to prove many simple things on a computer, and the computer can assist with repetitive, simple tasks — indeed, this is what computers do all the time in other facets of work. Unlike many proofs in mathematics, proofs of software correctness are often intellectually simple, it is just the size of the problem that makes it infeasible to prove 'by hand'. There are few requirements for 'tricks' or the invention of new techniques, just the repeated application of many basic principles such as induction or case analysis. The computer can be quite good at solving incredibly large and repetitive (but ultimately quite 'dumb') proofs.

Writing proofs in such a way to leverage this capability can a difficult and time-consuming task for the programmer, but it is far from impossible. We will discuss in Section 1.3.1 one example of a large fully verified program which shows that it truly can be done in practice. Additionally, as discussed above, some problems lend themselves more naturally to formal methods; for problems that the more traditional means of testing would also require a good deal of work, or for which complete testing is infeasible, there is a prime opportunity to use the tools of formal verification to avoid these pitfalls.

Coq is written in OCaml and has many similar features as a programming language. It features the ability to be "extracted," a process of mechanically translating Coq code to OCaml or other languages. Coq's proof abilities are based on the formal language of the Calculus of Inductive Constructions [9]. It is not essential to understand these details – indeed, one could spend many years trying – they are merely brought up as a starting point for the curious.

1.3.1 The CompCert C Compiler

CompCert is a compiler for a large subset of the language C. It is of particular interest because it is a *formally verified* C compiler written in Coq [12, 13]. It serves as the 'prior art' for this work, and it led to the development of the primary tools used herein, including the verified Menhir mode discussed in Section 2 [11].

CompCert provides both a proof positive of the feasibility of the broader goal of verifying a compiler and a specific model for parsing in a verified environment. Its methods are used as both a basis and a point of comparison for the rest of this work.

The original paper describing CompCert [12] also provides many alternative justifications to the earlier question of why to pursue formal verification above and beyond standard testing. To summarize one of their arguments that also applies to compiling Stan, great effort is often taken to ensure that the source code of a program does what is intended. In C, this may mean static analysis or program proof to ensure the correct thing is done by the code. In Stan, considerable time may be spent ensuring that the program exactly matches the probability distribution desired. However, in both cases, this analysis and preparation are dependent on the compilation being correct – an error in compilation can completely obviate any guarantees made at the source level, where most reasoning about programs is done.

2 Validated Parsing in Menhir

Menhir [15] is a parser generator for the language OCaml. It is a successor to `ocamlyacc`, which as implied by the name is inspired by the original C `yacc`. It features many improvements over `ocamlyacc`, including parameterized non-terminals, the ability to generate parsers for a larger family of grammars (LR(1), compared to `ocamlyacc`'s LALR(1), see 1.1.2), and the ability to name semantic values explicitly as opposed to the traditional `$1, $2,` ... syntax.

Menhir's 'killer feature,' for the purposes of this thesis, is the ability to generate parsers with multiple backends, including a backend for Coq. This produces a parser automaton implemented in Coq that comes with proofs of soundness (the parser only accepts valid inputs), completeness (the parser accepts *all* valid inputs), and safety (the parser will not produce an internal error) in the accompanying library, coq-menhirlib [11]. We will interchangeably refer to this as both a *validated* and a *verified* parser.

2.1 Basics

The existing Stan parser already used Menhir as its parser generator (see 3), but directly using this grammar was not possible due to several restrictions placed on Menhir's Coq mode compared to typical usage. They are enumerated in full in the manual [15], but to summarize the most relevant changes: the grammar must be unambiguous, must not rely on associativity and precedence directives, must not use the Menhir standard library, and must have a valid Coq type for every non-terminal. Additionally, many of the standard ways of handling errors are unavailable (see 2.3). Finally, the semantic actions of the parser must be expressed in Coq code, not OCaml.

Once a specification is created in compliance with these requirements, it can be turned into a parser through passing the `--coq` flag to Menhir. This generates a file with two modules, one containing the grammar and semantic actions, and the other containing the parser automaton. Each entry point[3] of the grammar has a function that parses its portion of the language and accompanying theorems and proofs for said entry point. There are some important details that a user should understand about this parser. One key detail is that termination is not guaranteed directly (and in fact, cannot be in general), but rather the parser uses a "fuel" based approach, where a maximum number of steps is provided as an input parameter, in order to satisfy Coq's requirements for termination [11, 15].

Each parsing function returns a sum type with three branches:

```
Inductive parse_result :=
| Fail_pr: parse_result
| Timeout_pr: parse_result
| Parsed_pr: symbol_semantic_type (NT (start_nt init)) ->
                Stream token ->
                parse_result.
```

There is a failure case for invalid inputs, a timed-out case for when the fuel has been exhausted before the parse completes (or encounters an error), and a success case for when a valid parse is found. The success case carries a semantic value (usually an AST) and the remainder of the tokens, as one familiar with other parsers might expect. In practice, one can assume that the timeout case will not occur for any non-pathological program, and it is often ignored, meaning only the other two return types must be considered.

2.2 Error Messages and Real-World Parsing

The failure outcome of the parse function is, in some ways, just as interesting as the success outcome.

One of the most useful (and certainly most visible) features of most parsers is their ability to report errors back to the programmer. When they encounter a program that is not syntactically sound, they can inform the programmer of the problem and give hints on what should be done to address it. Making these messages useful is imperative, as "Bad

[3] Menhir allows multiple start symbols for a grammar. These are referred to as the entry points, and each one receives its own parsing function.

input program" can take a massive amount of time to resolve, but "Missing) on line 32" is fixed almost instantly.

After successfully transforming the Stan reference parser into one suitable for verification by the Coq mode of Menhir, we considered how best to generate these error messages. Based on the existing CompCert example, and the rest of the Menhir ecosystem, we identified three options for proceeding:

1. Follow the lead of CompCert and use a second – **unverified** – parser that sits in front of the verified parser and uses Menhir's "incremental" (or "table") mode. This is a backend that, like the Coq mode, changes the style of parser produced and is used in Menhir's standard error messaging techniques. This would be a simple solution that clearly has been used before.

2. Modify Menhir's Coq mode significantly to allow it to be run in the same incremental style that enables the standard error messaging (and additional error recovery features) that are available in Menhir's "table" backend.

3. Modify Menhir's Coq mode to return the parser's state and other contextual information when an error state is entered.

We considered each choice in turn.

2.2.1 A Second Parser

The most straightforward choice would be to use a second parser to handle the error messaging. This is done in CompCert's C compiler, which also does lexical feedback through this initial parser [10]. The grammar structure of this additional parser would be the same, and Menhir supports flags for automatically generating grammar specification files with the semantic actions removed. This would allow someone who chooses this path to automatically create the second parser as part of their build system with relatively minimal effort.

The main arguments against this approach are twofold. First, this approach adds nothing to the existing tools and methods. It is entirely trodden terrain. Secondly, parsing twice is both inelegant and (at least for Stan) unnecessary.

2.2.2 Verified Incremental Mode

This second option would recreate the existing behavior of Menhir's incremental API, but in the verified mode. This alternative backend produces parsers that perform one step of parsing at a time and then yield their (partial) results to the calling code. Syntax errors can be handled while they occur, rather than the entire parse failing, and this allows both error-messaging and error-recovery to be done in a natural and elegant way. Two concerns made this option undesirable.

First, this is the approach that would require the most changes to Menhir. Any existing use of the tool would need significant changes. Furthermore, the ultimate benefits that those users received would be relatively limited — with a few notable exceptions, such as the Merlin language server [5], the incremental mode is often used as simply a very large hammer to solve the very small problem of error messaging. The additional features it enables, such as allowing error *resolution*, not just recognition, are unnecessary for our (and many other's) use case.

It is these very features that actually lead to a second reason to not pursue this path. Incremental parsing is ultimately driven by the lexer or an external loop, not the parser itself. This means any true verification of the parser would require proof that the lexer and other code also maintain the invariants required by the parser's correctness and completeness theorems. It would be counter to the entire notion of verifying the parser to allow the lexer to feed back unverified data following each and every computation, and we therefore decided against this method.

2.2.3 Meaningful returns on error

Finally, there was the middle road. The parser generated by Menhir's Coq mode has a sum type for its output, as show in Section 2.1. By modifying the `Fail_pr` branch of this type to include information about the state of the parser during the time the error was detected, we can reconstruct a meaningful error message after failure. In particular, we can return the state of the parser and the last token seen. The state is the piece of information used by Menhir's existing error messaging functionality when in incremental mode, and clever use of the token types (as is employed in both CompCert and our parser) allows the retrieval of crucial context information.

This final option was ultimately selected for use. This required modifications to both Menhir and the coq-menhirlib library.

2.3 Modifications

There are two basic pieces of information that are useful for creating meaningful error messages: the state of the parser (which carries information about what was expected and therefore what went wrong) and the position of the error. The first is relatively simple - it is usually encoded as a number, and the existing Menhir tools rely on these numbers for picking which error message is displayed. The second is not immediately available to the parser, but must be given by the lexer. Luckily, it is quite reasonable to include position information in each token the lexer produces. The token causing the error can thereafter be used as a stand-in for the position of the error in the input.

The modifications to Menhir were motivated by making these two pieces of information available. This primarily required modification to the file `Interpreter.v` in the coq-menhirlib library that is linked to the generated Coq parsers. In particular, the return type of the parsing functions was updated to:

```
Inductive parse_result :=
| Fail_pr_full: state -> token -> parse_result
| Timeout_pr: parse_result
| Parsed_pr: symbol_semantic_type (NT (start_nt init)) ->
        Stream token ->
        parse_result.
```

This change was nearly enough on its own, but the existing proofs and usages of this type all ostensibly needed to be updated to recognize the new `Fail_pr_full`. However, none of these actually require this information (this should be obvious, as they predated its inclusion). Therefore, using the Coq notation functionality, `Fail_pr` was set up as an abbreviation for `Fail_pr_full _ _`. This meant no further modifications to the library were needed to complete this change.

As mentioned above, the state is often encoded as a simple integer, but in this return type it is provided as a sum type, `state`. This type is defined in the generated parser. Im order to support retrieval of the numerical encoding of the state, an additional function, `Aut.N_of_state (s:state) : N`, was added to the parser automaton.[4] This number can be handed off to the code generated by the existing `--compile-errors` feature of Menhir: an OCaml function `message (s:int) : string` that maps from the parser's state one of the user-defined error messages.

These together provide enough for first-class error messages from a single, verified parser. The workflow proceeds something like this:

1. The lexer produces tokens which carry a position payload.
2. The parser is run and returns a `parse_result`.
3. If the result is a failure, one can use the function `Aut.N_of_state` to retrieve the state number from the returned state and extract any position information from the token. They can then call the generated `message` function with the state number, and produce an error message with position and syntax information.

The above functionality was merged into Menhir on April 1st, 2021, and released on April 19th, 2021. An example is provided in Appendix A. For further reference, consult the Menhir manual [15] and the provided demo (`demos/coq-syntax-errors`) written by myself as part of the contribution to Menhir.

3 Parsing Stan

There are two existing sources of information on parsing Stan. The first is the Stan Reference Manual. This document contains both a context-free grammar and a table providing associativity and precedence information that can be used to disambiguate it [16, §11.1 and 6.5]. The second is the stanc3 compiler, which serves as the reference compiler for Stan [1]. The Menhir parser specification from the stanc3 compiler was used as a template and cross-referenced with the written materials.

There are several notable differences between these two sources. These range from the inclusion of an element-wise exponentiation operator, `.^`, which is absent from the documentation but present in the existing compiler to the over-specification of some parts of the grammar compared to the parser, such as the inclusion of `integrate_1d` and other built-in functions as terminal symbols in the provided grammar that are not separately lexed or parsed in the actual implementation. In general, preference was given to the working parser

[4] In extracted code, this function is called `Aut.coq_N_of_state`.

over the documentation in areas where they disagreed, as this is believed to represent Stan 'in the real world'. We mainly differed from stanc3's parser to conform to the requirements of the Coq mode for Menhir, but we also made several changes where differences in the AST required them. One main example of this is stanc3's preservation of additional parentheses in the AST, used mainly for pretty-printing, that was decided to not be emulated in this implementation.

3.1 Grammar Transformations

There were multiple edits needed to modify stanc3's Menhir specification into one compatible with the Coq mode.

The first required changes were a number of edits to the grammar being parsed, primarily as a result of the Coq mode's lack of precedence declarations. The existing grammar used these for all the arithmetic and logical operators and to resolve the 'dangling else' problem [1, 4, 10]. Handling the first of these required rewriting the portion of the grammar handling expressions to explicitly include the associativity and precedence of the different operators. This introduced 11 new non-terminal symbols into the grammar to enforce the precedence hierarchy and an accompanying set of productions which are either left- or right- recursive depending on the associativity given in the reference manual [16]. To resolve the if/else ambiguity the standard method of partitioning the parsing of statements into parsing *open* statements, which may contain further nested open statements, and non-open or *closed* statements, which must be 'complete' and contain only closed statements, was used [4, 10, 13]. Both of these changes were first implemented in the existing OCaml/Menhir parser specification, which was then tested against the reference parser on selected inputs from the stanc3 test suite to ensure the behavior of the new parser remained consistent. The resulting grammar, with additional annotations highlighting these changes, can be viewed in Appendix B.

Once the grammar had been disambiguated without the use of precedence and associativity annotations, there were three remaining steps required to produce a parser which operated with the Coq backend. First, an AST type for Stan was written in Coq following the same basic structure as that used in OCaml. Secondly, the semantic actions of the parser had to be rewritten in Coq. This largely required straightforward translations between the syntax of the two languages, but some additional concern was needed to ensure the functions used would satisfy the Coq compiler's requirements for termination, and some elements of the Menhir standard library, such as the parameterized non-terminal `list`, had to be recreated with Coq semantic actions. Finally, a Coq type signature was written for each non-terminal symbol in the language. This parser was then incorporated into a larger body of code based on CompCert where work by Tristan and Tassarotti continues on the later stages of compilation.

3.2 Error Messages and Other Work

Following the changes to Menhir described in Section 2.3, a few further changes were made to the parser to enable error messaging. The types described in the parser were all updated to carry additional location information, and the lexer was updated to provide this information.

The driver code in the larger compiler was modified to display syntax errors, and the relevant portion of this code is available in Appendix A. Finally, the error messages were written for each parser state that can lead to an error. While the `.messages` file from stanc3 was used as a reference, the differences between the grammars meant that these needed to largely be done by hand. Our final parser has 237 possible error-causing states, which correspond to 164 unique error messages.

4 Results

This thesis successfully created a verified parser for Stan for use in a formally verified compiler. This required rewriting the grammar and semantic actions for use with the verified mode provided by the Menhir parser generator. Furthermore, changes were made to Menhir to allow error messaging capability from the verified parser alone, and these changes were included upstream by the Menhir developers.

This work allows the further development of the Stan verified compiler, and it provides greater functionality to anyone seeking to use Menhir to produce realistic, validated parsers.

Another avenue for useful extension of Menhir's Coq mode which was identified, but not pursued, would be the addition of the associativity and precedence declarations which are available in the more typical usage of Menhir. A large amount of manual work was put in to translating the grammar specification from one that used these annotations to one that did not, and this required additional testing to ensure that the grammars still agreed with one another. This serves as both a source of potential human error and a barrier to the adoption of validated parsing. Implementing these would make transitioning from an existing Menhir specification to one which could be used with the Coq backend considerably simpler.

5 Acknowledgments

I would like to thank Joseph Tassarotti and Jean-Baptiste Tristan for their guidance and mentorship throughout the development and writing of this thesis. Additionally, thanks are owed to Robert Muller, for introducing me to them both, and Howard Straubing, for advising me throughout my undergraduate studies and insisting that I write a thesis. Outside of the Boston College Computer Science department, thank you to both François Pottier and Jacques-Henri Jourdan for their assistance in modifying Menhir and their eagerness to include my contributions into the tool.

The work described in this thesis was supported by a gift from Oracle Labs.

Finally, I would like to thank my family, my roommates, and my girlfriend, Olivia, for all helping me finish my studies in this very odd year and nodding intently while I described details of my work that were decidedly outside of their interests and areas of expertise.

References

[1] A new Stan-to-C++ compiler, stanc3. https://github.com/stan-dev/stanc3, 2019.

[2] ISO/IEC 14977:1996(E). Information technology – Syntactic metalanguage – Extended BNF. Standard, International Organization for Standardization, Geneva, CH, December 1996. https://www.iso.org/standard/26153.html.

[3] Alfred Aho. *The theory of parsing, translation, and compiling*. Prentice-Hall, Englewood Cliffs, N.J, 1972.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[5] Frédéric Bour, Thomas Refis, and Gabriel Scherer. Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–15, July 2018.

[6] Lélio Brun. Obelisk. https://github.com/Lelio-Brun/Obelisk, 2017.

[7] Bob Carpenter. Stan 2.10 through Stan 2.13 produce biased samples. *Statistical Modeling, Causal Inference, and Social Science*. https://statmodeling.stat.columbia.edu/2016/12/20/stan-2-10-stan-2-13-produce-biased-samples/, Dec 2016.

[8] Keith D. Cooper and Linda Torczon. *Engineering a compiler*. Elsevier/Morgan Kaufmann, Amsterdam; Boston, 2nd ed edition, 2012.

[9] Coq 8.9.1 documentation. Calculus of inductive constructions. https://coq.github.io/doc/v8.9/refman/language/cic.html.

[10] Jacques-Henri Jourdan and François Pottier. A simple, possibly correct LR parser for c11. *ACM Transactions on Programming Languages and Systems*, 39(4):1–36, September 2017.

[11] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In *Programming Languages and Systems*, pages 397–416. Springer Berlin Heidelberg, 2012.

[12] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[13] Xavier Leroy. The CompCert C verified compiler. https://github.com/AbsInt/CompCert, 2014.

[14] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, May 2018. Version 5.5. https://softwarefoundations.cis.upenn.edu/lf-current/index.html.

[15] François Pottier and Yann Régis-Gianas. *Menhir Reference Manual*. INRIA, November 2020. https://gallium.inria.fr/~fpottier/menhir/manual.html.

[16] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual*, 2019. https://mc-stan.org.

A Error Messaging Code

This code is part of my contributions to the full parser project, built on CompCert [13].

```
let location t : Lexing.position * Lexing.position =
  match t with
  (* These four tokens have a payload we ignore *)
  | STRINGLITERAL sp | REALNUMERAL sp | INTNUMERAL sp | IDENTIFIER sp ->
    snd sp
  (* All of the following tokens have no payload, just a position *)
  | WHILE p | VOID p | VECTOR p | UPPER p | UNITVECTOR p | TRUNCATE p
  | TRANSPOSE p | TRANSFORMEDPARAMETERSBLOCK p | TRANSFORMEDDATABLOCK p
  | TIMESASSIGN p | TIMES p | TILDE p | TARGET p | SIMPLEX p | SEMICOLON p
  | RPAREN p | ROWVECTOR p | RETURN p | REJECT p | REAL p | RBRACK p
  | RBRACE p | RABRACK p | QMARK p | PRINT p | POSITIVEORDERED p | PLUSASSIGN p
  | PLUS p | PARAMETERSBLOCK p | ORDERED p | OR p | OFFSET p | NEQUALS p
  | MULTIPLIER p | MODULO p | MODELBLOCK p | MINUSASSIGN p | MINUS p | MATRIX p
  | LPAREN p | LOWER p | LEQ p | LDIVIDE p | LBRACK p | LBRACE p | LABRACK p
  | INT p | IN p | IF_ p | IDIVIDE p | HAT p | GEQ p | GENERATEDQUANTITIESBLOCK p
  | FUNCTIONBLOCK p | FOR p | EQUALS p | EOF p | ELTTIMESASSIGN p | ELTTIMES p
  | ELTPOW p | ELTDIVIDEASSIGN p | ELTDIVIDE p | ELSE p | DIVIDEASSIGN p
  | DIVIDE p | DATABLOCK p | COVMATRIX p | CORRMATRIX p | CONTINUE p | COMMA p
  | COLON p | CHOLESKYFACTORCOV p | CHOLESKYFACTORCORR p | BREAK p | BAR p
  | BANG p | ASSIGN p | AND p ->
    p

let state_num s =
  let coq_num = Sparser.Aut.coq_N_of_state s in
  let state = Camlcoq.N.to_int coq_num (* convert to caml int *)
  in
  state

let handle_syntax_error file state token =
  let (pos1, pos2) = location token in
  let line = pos2.pos_lnum in
  let st_num = state_num state in
  let col_start = let {pos_cnum;pos_bol} = pos1 in 1 + pos_cnum - pos_bol in
  let col_end = let {pos_cnum;pos_bol} = pos2 in 1 + pos_cnum - pos_bol in
  let msg = try message st_num with (* generated function *)
    | Not_found -> "Unknown error in parser state " ^ string_of_int st_num
  in
  Printf.eprintf  "Syntax error in '%s', line %d, characters %d-%d:\n%s" file line
  ↪  col_start col_end msg;
    exit 1
```

B Final Grammar

The following grammar was generated from the Menhir description file using Obelisk [6]. It is written in a common extension of Backus-Naur form, which allows several shorthand notations similar to those in regular expressions, such as + and ∗ [2].

There are several sections which are annotated with additional notes on the differences between this grammar and either the stanc3 grammar or the specification in the manual.

⟨*program*⟩	::=	[⟨*function_block*⟩] [⟨*data_block*⟩] [⟨*transformed_data_block*⟩] [⟨*parameters_block*⟩] [⟨*transformed_param_block*⟩] [⟨*model_block*⟩] [⟨*generated_quantities_block*⟩] ‘EOF’
⟨*function_block*⟩	::=	‘FUNCTIONBLOCK’ ‘LBRACE’ ⟨*function_def*⟩* ‘RBRACE’
⟨*data_block*⟩	::=	‘DATABLOCK’ ‘LBRACE’ ⟨*top_var_decl_no_assign*⟩* ‘RBRACE’
⟨*transformed_data_block*⟩	::=	‘TRANSFORMEDDATABLOCK’ ‘LBRACE’ ⟨*top_vardecl_or_statement*⟩* ‘RBRACE’
⟨*parameters_block*⟩	::=	‘PARAMETERSBLOCK’ ‘LBRACE’ ⟨*top_var_decl_no_assign*⟩* ‘RBRACE’
⟨*transformed_param_block*⟩	::=	‘TRANSFORMEDPARAMETERSBLOCK’ ‘LBRACE’ ⟨*top_vardecl_or_statement*⟩* ‘RBRACE’
⟨*model_block*⟩	::=	‘MODELBLOCK’ ‘LBRACE’ ⟨*vardecl_or_statement*⟩* ‘RBRACE’
⟨*generated_quantities_block*⟩	::=	‘GENERATEDQUANTITIESBLOCK’ ‘LBRACE’ ⟨*top_vardecl_or_statement*⟩* ‘RBRACE’
⟨*identifier*⟩	::=	‘IDENTIFIER’
	|	‘TRUNCATE’
⟨*decl_identifier*⟩	::=	⟨*identifier*⟩
⟨*function_def*⟩	::=	⟨*return_type*⟩ ⟨*decl_identifier*⟩ ‘LPAREN’ ⟨*arg_decl*⟩$^*_{\text{‘COMMA’}}$ ‘RPAREN’ ⟨*statement*⟩
⟨*return_type*⟩	::=	‘VOID’
	|	⟨*unsized_type*⟩
⟨*arg_decl*⟩	::=	⟨*unsized_type*⟩ ⟨*decl_identifier*⟩
	|	‘DATABLOCK’ ⟨*unsized_type*⟩ ⟨*decl_identifier*⟩
⟨*unsized_type*⟩	::=	⟨*basic_type*⟩
	|	⟨*basic_type*⟩ ⟨*unsized_dims*⟩

⟨*basic_type*⟩	::=	‘INT’
	|	‘REAL’
	|	‘VECTOR’
	|	‘ROWVECTOR’
	|	‘MATRIX’
⟨*unsized_dims*⟩	::=	‘LBRACK’ ‘COMMA’* ‘RBRACK’
⟨*var_decl*⟩	::=	⟨*sized_basic_type*⟩ ⟨*decl_identifier*⟩ [⟨*dims*⟩] [‘ASSIGN’ ⟨*expression*⟩] ‘SEMICOLON’
⟨*sized_basic_type*⟩	::=	‘INT’
	|	‘REAL’
	|	‘VECTOR’ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘ROWVECTOR’ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘MATRIX’ ‘LBRACK’ ⟨*expression*⟩ ‘COMMA’ ⟨*expression*⟩ ‘RBRACK’
⟨*top_var_decl_no_assign*⟩	::=	⟨*top_var_type*⟩ ⟨*decl_identifier*⟩ [⟨*dims*⟩] ‘SEMICOLON’
⟨*top_var_decl*⟩	::=	⟨*top_var_type*⟩ ⟨*decl_identifier*⟩ [⟨*dims*⟩] [‘ASSIGN’ ⟨*expression*⟩] ‘SEMICOLON’
⟨*top_var_type*⟩	::=	‘INT’ ⟨*range_constraint*⟩
	|	‘REAL’ ⟨*type_constraint*⟩
	|	‘VECTOR’ ⟨*type_constraint*⟩ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘ROWVECTOR’ ⟨*type_constraint*⟩ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘MATRIX’ ⟨*type_constraint*⟩ ‘LBRACK’ ⟨*expression*⟩ ‘COMMA’ ⟨*expression*⟩ ‘RBRACK’
	|	‘ORDERED’ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘POSITIVEORDERED’ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘SIMPLEX’ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘UNITVECTOR’ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘CHOLESKYFACTORCORR’ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘CHOLESKYFACTORCOV’ ‘LBRACK’ ⟨*expression*⟩ [‘COMMA’ ⟨*expression*⟩] ‘RBRACK’
	|	‘CORRMATRIX’ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
	|	‘COVMATRIX’ ‘LBRACK’ ⟨*expression*⟩ ‘RBRACK’
⟨*type_constraint*⟩	::=	⟨*range_constraint*⟩
	|	‘LABRACK’ ⟨*offset_mult*⟩ ‘RABRACK’
⟨*range_constraint*⟩	::=	[‘LABRACK’ ⟨*range*⟩ ‘RABRACK’]
⟨*range*⟩	::=	‘LOWER’ ‘ASSIGN’ ⟨*constr_expression*⟩ ‘COMMA’ ‘UPPER’ ‘ASSIGN’ ⟨*constr_expression*⟩
	|	‘UPPER’ ‘ASSIGN’ ⟨*constr_expression*⟩ ‘COMMA’ ‘LOWER’ ‘ASSIGN’ ⟨*constr_expression*⟩

	|	‘LOWER’ ‘ASSIGN’ ⟨*constr_expression*⟩
	|	‘UPPER’ ‘ASSIGN’ ⟨*constr_expression*⟩
⟨*offset_mult*⟩	::=	‘OFFSET’ ‘ASSIGN’ ⟨*constr_expression*⟩ ‘COMMA’ ‘MULTIPLIER’ ‘ASSIGN’ ⟨*constr_expression*⟩
	|	‘MULTIPLIER’ ‘ASSIGN’ ⟨*constr_expression*⟩ ‘COMMA’ ‘OFFSET’ ‘ASSIGN’ ⟨*constr_expression*⟩
	|	‘OFFSET’ ‘ASSIGN’ ⟨*constr_expression*⟩
	|	‘MULTIPLIER’ ‘ASSIGN’ ⟨*constr_expression*⟩
⟨*dims*⟩	::=	‘LBRACK’ ⟨*expression*⟩$^{+}_{\text{‘COMMA’}}$ ‘RBRACK’

Note: The following rules concerning expressions are the most markedly different from both the specification and the existing stanc3 grammar. They have been re-written to enforce the precedence and grammar rules as specified in the specification document without the use of annotations.

⟨*expression*⟩	::=	⟨*or_expression*⟩ ‘QMARK’ ⟨*expression*⟩ ‘COLON’ ⟨*expression*⟩
	|	⟨*or_expression*⟩
⟨*or_expression*⟩	::=	⟨*and_expression*⟩$^{+}_{\text{‘OR’}}$
⟨*and_expression*⟩	::=	⟨*equal_expression*⟩$^{+}_{\text{‘AND’}}$
⟨*equal_expression*⟩	::=	⟨*equal_expression*⟩ ‘EQUALS’ ⟨*comparison_expression*⟩
	|	⟨*equal_expression*⟩ ‘NEQUALS’ ⟨*comparison_expression*⟩
	|	⟨*comparison_expression*⟩
⟨*comparison_expression*⟩	::=	⟨*comparison_expression*⟩ ‘LABRACK’ ⟨*additive_expression*⟩
	|	⟨*comparison_expression*⟩ ‘LEQ’ ⟨*additive_expression*⟩
	|	⟨*comparison_expression*⟩ ‘RABRACK’ ⟨*additive_expression*⟩
	|	⟨*comparison_expression*⟩ ‘GEQ’ ⟨*additive_expression*⟩
	|	⟨*additive_expression*⟩
⟨*additive_expression*⟩	::=	⟨*additive_expression*⟩ ‘PLUS’ ⟨*multiplicative_expression*⟩
	|	⟨*additive_expression*⟩ ‘MINUS’ ⟨*multiplicative_expression*⟩
	|	⟨*multiplicative_expression*⟩
⟨*multiplicative_expression*⟩	::=	⟨*multiplicative_expression*⟩ ‘TIMES’ ⟨*leftdivide_expression*⟩
	|	⟨*multiplicative_expression*⟩ ‘DIVIDE’ ⟨*leftdivide_expression*⟩
	|	⟨*multiplicative_expression*⟩ ‘IDIVIDE’ ⟨*leftdivide_expression*⟩
	|	⟨*multiplicative_expression*⟩ ‘MODULO’ ⟨*leftdivide_expression*⟩
	|	⟨*multiplicative_expression*⟩ ‘ELTTIMES’ ⟨*leftdivide_expression*⟩
	|	⟨*multiplicative_expression*⟩ ‘ELTDIVIDE’ ⟨*leftdivide_expression*⟩
	|	⟨*leftdivide_expression*⟩

⟨*leftdivide_expression*⟩	::=	⟨*prefix_expression*⟩$^{+}_{\text{‘LDIVIDE’}}$
⟨*prefix_expression*⟩	::=	‘BANG’ ⟨*exponentiation_expression*⟩
	|	‘MINUS’ ⟨*exponentiation_expression*⟩
	|	‘PLUS’ ⟨*exponentiation_expression*⟩
	|	⟨*exponentiation_expression*⟩
⟨*exponentiation_expression*⟩	::=	⟨*postfix_expression*⟩ ‘HAT’ ⟨*exponentiation_expression*⟩
	|	⟨*postfix_expression*⟩ ‘ELTPOW’ ⟨*exponentiation_expression*⟩
	|	⟨*postfix_expression*⟩
⟨*postfix_expression*⟩	::=	⟨*postfix_expression*⟩ ‘TRANSPOSE’
	|	⟨*index_expression*⟩
	|	⟨*common_expression*⟩
⟨*index_expression*⟩	::=	⟨*basic_expression*⟩ ‘LBRACK’ ⟨*indexes*⟩ ‘RBRACK’

Note: In addition to the changes from stanc3 related to the above changes to expressions, it is also worth noting that both stanc3 and this parser differs from spec in not explicitly including some built-in functions in the grammar.

⟨*common_expression*⟩	::=	⟨*basic_expression*⟩
	|	⟨*lhs*⟩
⟨*basic_expression*⟩	::=	‘INTNUMERAL’
	|	‘REALNUMERAL’
	|	‘LBRACE’ ⟨*expression*⟩$^{+}_{\text{‘COMMA’}}$ ‘RBRACE’
	|	‘LBRACK’ ⟨*expression*⟩$^{*}_{\text{‘COMMA’}}$ ‘RBRACK’
	|	⟨*identifier*⟩ ‘LPAREN’ ⟨*expression*⟩ ‘BAR’ ⟨*expression*⟩$^{*}_{\text{‘COMMA’}}$ ‘RPAREN’
	|	⟨*identifier*⟩ ‘LPAREN’ ⟨*expression*⟩$^{*}_{\text{‘COMMA’}}$ ‘RPAREN’
	|	‘TARGET’ ‘LPAREN’ ‘RPAREN’
	|	‘GETLP’ ‘LPAREN’ ‘RPAREN’
	|	‘LPAREN’ ⟨*expression*⟩ ‘RPAREN’
⟨*constr_expression*⟩	::=	⟨*additive_expression*⟩
⟨*indexes*⟩	::=	⟨*index*⟩$^{+}_{\text{‘COMMA’}}$
⟨*index*⟩	::=	ϵ
	|	‘COLON’
	|	⟨*expression*⟩
	|	⟨*expression*⟩ ‘COLON’
	|	‘COLON’ ⟨*expression*⟩
	|	⟨*expression*⟩ ‘COLON’ ⟨*expression*⟩

⟨*printables*⟩	::=	⟨*printable*⟩$^{+}_{\text{‘COMMA’}}$
⟨*printable*⟩	::=	⟨*expression*⟩
	|	⟨*string_literal*⟩
⟨*lhs*⟩	::=	⟨*identifier*⟩
	|	⟨*lhs*⟩ ‘LBRACK’ ⟨*indexes*⟩ ‘RBRACK’
⟨*statement*⟩	::=	⟨*closed_statement*⟩
	|	⟨*open_statement*⟩
⟨*atomic_statement*⟩	::=	⟨*lhs*⟩ ⟨*assignment_op*⟩ ⟨*expression*⟩ ‘SEMICOLON’
	|	⟨*identifier*⟩ ‘LPAREN’ ⟨*expression*⟩$^{*}_{\text{‘COMMA’}}$ ‘RPAREN’ ‘SEMICOLON’
	|	‘INCREMENTLOGPROB’ ‘LPAREN’ ⟨*expression*⟩ ‘RPAREN’ ‘SEMICOLON’
	|	⟨*expression*⟩ ‘TILDE’ ⟨*identifier*⟩ ‘LPAREN’ ⟨*expression*⟩$^{*}_{\text{‘COMMA’}}$ ‘RPAREN’ [⟨*truncation*⟩] ‘SEMICOLON’
	|	‘TARGET’ ‘PLUSASSIGN’ ⟨*expression*⟩ ‘SEMICOLON’
	|	‘BREAK’ ‘SEMICOLON’
	|	‘CONTINUE’ ‘SEMICOLON’
	|	‘PRINT’ ‘LPAREN’ ⟨*printables*⟩ ‘RPAREN’ ‘SEMICOLON’
	|	‘REJECT’ ‘LPAREN’ ⟨*printables*⟩ ‘RPAREN’ ‘SEMICOLON’
	|	‘RETURN’ ⟨*expression*⟩ ‘SEMICOLON’
	|	‘RETURN’ ‘SEMICOLON’
	|	‘SEMICOLON’
⟨*assignment_op*⟩	::=	‘ASSIGN’
	|	‘ARROWASSIGN’
	|	‘PLUSASSIGN’
	|	‘MINUSASSIGN’
	|	‘TIMESASSIGN’
	|	‘DIVIDEASSIGN’
	|	‘ELTTIMESASSIGN’
	|	‘ELTDIVIDEASSIGN’
⟨*string_literal*⟩	::=	‘STRINGLITERAL’
⟨*truncation*⟩	::=	‘TRUNCATE’ ‘LBRACK’ [⟨*expression*⟩] ‘COMMA’ [⟨*expression*⟩] ‘RBRACK’

Note: The next two rules are nearly identical, but exist to disambiguate the dangling-else problem. This is not present in the stanc3 grammar due to the use of a precedence annotation to resolve this.

⟨*open_statement*⟩	::=	'IF' 'LPAREN' ⟨*expression*⟩ 'RPAREN' ⟨*simple_statement*⟩
	|	'IF' 'LPAREN' ⟨*expression*⟩ 'RPAREN' ⟨*open_statement*⟩
	|	'IF' 'LPAREN' ⟨*expression*⟩ 'RPAREN' ⟨*closed_statement*⟩ 'ELSE' ⟨*open_statement*⟩
	|	'WHILE' 'LPAREN' ⟨*expression*⟩ 'RPAREN' ⟨*open_statement*⟩
	|	'FOR' 'LPAREN' ⟨*identifier*⟩ 'IN' ⟨*expression*⟩ 'COLON' ⟨*expression*⟩ 'RPAREN' ⟨*open_statement*⟩
	|	'FOR' 'LPAREN' ⟨*identifier*⟩ 'IN' ⟨*expression*⟩ 'RPAREN' ⟨*open_statement*⟩
⟨*closed_statement*⟩	::=	'IF' 'LPAREN' ⟨*expression*⟩ 'RPAREN' ⟨*closed_statement*⟩ 'ELSE' ⟨*closed_statement*⟩
	|	'WHILE' 'LPAREN' ⟨*expression*⟩ 'RPAREN' ⟨*closed_statement*⟩
	|	⟨*simple_statement*⟩
	|	'FOR' 'LPAREN' ⟨*identifier*⟩ 'IN' ⟨*expression*⟩ 'COLON' ⟨*expression*⟩ 'RPAREN' ⟨*closed_statement*⟩
	|	'FOR' 'LPAREN' ⟨*identifier*⟩ 'IN' ⟨*expression*⟩ 'RPAREN' ⟨*closed_statement*⟩
⟨*simple_statement*⟩	::=	'LBRACE' ⟨*vardecl_or_statement*⟩* 'RBRACE'
	|	⟨*atomic_statement*⟩
⟨*vardecl_or_statement*⟩	::=	⟨*statement*⟩
	|	⟨*var_decl*⟩
⟨*top_vardecl_or_statement*⟩	::=	⟨*statement*⟩
	|	⟨*top_var_decl*⟩