ETH zürich

Parser for Go Programs and Specification

Bachelor's Thesis

Nico Berling

March 15, 2022

Advisors: Prof. Dr. Peter Müller, Linard Arquint, Felix Wolf

Department of Computer Science, ETH Zürich

Abstract

Gobra is a verifier for the Go programming language. It allows users to annotate Go programs with specifications and to prove their correctness. In order to provide a seamless verification experience, performance is very important. Unfortunately, Gobra's parser used to read annotated Go files exhibits poor performance, and even makes up a significant part of Gobra's execution time. In addition, it does not always parse correct Gobra files sucessfully. Furthermore, the old parser often emitted error messages that were hard to read.

We have replaced Gobra's old parser with a new one that has significantly improved performance and completeness. This thesis presents details of the new parser, including our new architecture, our process of replacing the parser, and how we can use it to return more helpful error messages. Our evaluation shows that the new parser is significantly faster than the old parser.

Contents

Contents **iii**

1 Introduction **1**

2 Changing the Parser **3**
2.1 The old Gobra Parser . . . 3
2.2 Completeness . . . 5
2.2.1 Indexing and selector expressions . . . 5
2.2.2 Struct Literals in Conditional Expressions . . . 5
2.2.3 Syntax for Predicate Constructors . . . 6
2.3 Alternative Parser Frameworks . . . 6
2.3.1 Performance . . . 6
2.3.2 Maintainability . . . 7

3 Architecture **9**
3.1 Changes in Architecture . . . 9

4 The Gobra Grammar **11**
4.1 ANTLR . . . 11
4.2 The Base Grammar . . . 14
4.2.1 Expression Precedence . . . 14
4.2.2 Semicolon Omission . . . 15
4.2.3 Contributions to the ANTLR Repository . . . 16
4.3 From Go to Gobra . . . 17
4.3.1 Adjusting Gobra's Grammar . . . 17
4.4 Limitations . . . 18

5 Maintainability **21**
5.1 ANTLR to Kiama Transformation . . . 21
5.1.1 The Visitor Interface . . . 21
5.2 Design Choices to Improve Maintainability . . . 22

5.3 Extending Gobra with New Features 22

6 Evaluation **25**
6.1 Performance Comparison 25
6.1.1 Parameters 26
6.1.2 Impact of Warm-Up 26
6.1.3 Speedup of the New Parser 27
6.2 Completeness 27
6.2.1 Limitations 28
6.3 Better Error Messages 29
6.3.1 Messages for Unsupported Go Features 29
6.3.2 Slice Expressions 29

7 Conclusion **31**
7.1 Future Work 31

Bibliography **33**

Chapter 1

Introduction

Gobra [11] is a verification language for Go programs. It extends the Go language with annotations, which are used to specify the intended properties of the code and to help the verification backend to prove these properties. The Gobra verifier takes these annotated Go programs as input and outputs whether the program satisfies the specified properties.

In order to provide a seamless verification experience, performance is very important: The user should get the verification results as quickly as possible. While Gobra's backend verifier has been improved toward this goal, Gobra is currently held back by its parser. The parser is the first step in the verification pipeline and completes the task of translating a written program to some internal representation, used in the later parts of verification. As such, we consider it overhead, as it is not part of the actual verification process.

Despite being part of the overhead, Gobra's parser often takes a long time to parse files. In particular, there are several cases where parsing took significantly longer than verification. This thesis addresses this problem by replacing the old parser with a more performant one.

There are many different algorithms and frameworks used for parsing, some being more flexible than others. When dealing with custom languages, like Gobra's annotation language, we look for an algorithm that offers enough flexibility to accommodate language constructs that are challenging to parse, while still maintaining good overall parsing performance.

In this thesis, we first discuss the shortcomings of Gobra's old parser, and compare it to alternatives (chapter 2). Furthermore, we we justify why we have picked the new parser infrastructure. Afterward, we discuss how to integrate a new parser in the Gobra infrastructure (chapter 3), and delve deeper into the characteristics of the new parser (chapter 4). Lastly, we present an evaluation where we compare the new parser against the old

parser, not only in terms of performance, but also user experience and maintainability (chapters 5 and 6).

Chapter 2

Changing the Parser

The old Gobra parser has been a source of frustration: Parsing takes a long time, in some cases longer than verification itself, many Go files cannot be parsed because several Go features are unsupported, and some operations, like indexing an array, are parsed incorrectly, unless surrounded with extra parentheses. For this reason, we developed a new parser for the Gobra language. This chapter describes the problems with the old parser and motivates the choice of our new parser framework.

2.1 The old Gobra Parser

The old Gobra parser is based on the *parser combinator* framework Kiama [9]. Parser combinators are a functional approach to generating parsers. Traditional parser generators work in the following way: One writes a grammar in some dialect of Extended Backus-Naur form (EBNF), the parser generator analyzes this grammar, and then outputs source code of a parser that recognizes the language specified by this grammar. This generated source code can then be integrated into the target application, in our case the Gobra verifier.

Parser combinators, however, are not based on generating a parser from a static grammar specification. Instead, they build a parser for a language by combining multiple simpler parsers that recognize different structures in the language in various ways.

In this context, we think of parsers as functions that take a string as an input and return a *parsing success* or a *parsing failure*. Usually, a success results in a parse tree, while a failure results in a list of error messages.

A parser combinator framework provides the building blocks for this: On one hand, it provides basic parsers that match a specified string or regular expressions. This most primitive parser is used to recognize the basic sym-

```
1  val p3 = seq(alt("x", "y"), option(p1))
2  val p4 = ("x" alt "y") seq option(p1)
3  val p5 = ("x" | "y") ~ ?(p1) // more symbolic function names
```

Listing 2.1: Three equivalent parser definitions.

bols of a language, such as operators and identifiers. On the other hand, it provides *combinators*, i.e., functions that combine parsers in some specific way.

The most basic of these functions is the *concatenation*, or sequential composition, of two parsers. The concatenation of two parsers uses the first parser to match some prefix of the input, and then hands the rest of the input to the second parser. Only if this second parser also succeeds does the concatenation of the two succeed.

While concatenation is implied by writing two rules next to each other in many EBNF dialects, here it is an explicit function, we will call it *seq*. For example, `p1 = seq("a", "b")` recognizes the language $\{ab\}$, and the parser `p2 = seq(p1, "c")` recognizes $\{abc\}$.

Note that the variables `p1`, `p2` do not denote strings: They are parsers that can recognize languages. They are functions that take strings as inputs and return parse trees if they have succeeded, or some error message if the input is not part of the language. `p1("a")` results in a parse failure, while `p1("ab")` is successful.

Combinator functions are also defined for all the other operators in EBNF. Kiama takes advantage of Scala's clean functional syntax and symbolic function names to make parser combinator constructions easier to read (see Listing 2.1).

Because the last parser specification looks almost exactly like EBNF, it is easy to see how parser combinators are a great tool for recognizing languages, especially during prototyping. They are modular and do not require an extra step when recompiling. However, this comes with a cost: The parser does not analyze the grammar or optimize the resulting parser. Instead, it relies on backtracking.

A backtracking parser simply tries each alternative in turn, backtracking through the input when an alternative fails. This can exhibit exponential performance in the worst case. While Kiama uses a more advanced parsing technique [10], the principle is similar and this technique still reparses some parts of the input. This results in a super-linear increase in parse time relative to input size (see Sec. 2.3 and Chapter 6).

```
1  primaryExpression = selection | indexedExpr | ...
2  selection = primaryExpression ~ ("." ~ identifier) // ~ is seq
3  indexedExpr = primaryExpression ~ ("[" ~ expression ~ "]")
```

Listing 2.2: Gobra's definition of selection and indexing

2.2 Completeness

In addition to its bad performance, the old parser also had several issues when it came to correctly parsing some inputs. The following section discusses some of the main issues we encountered.

2.2.1 Indexing and selector expressions

Like most C-like languages, Go allows member selection with a dot, e.g. `myStruct.myValue`, and array indexing with brackets, e.g. `myArray[4]`. Both of these constructs are *primary expressions*, which can be thought of as operands of more complicated expressions. The current definition can be seen in Listing 2.2.

Even though both of these rules seem very similar, the old parser fails when trying to parse the expression `a.b[c]`. Instead, when we want to index a selected member, we have to surround the selection with parentheses: `(a.b)[c]`. Because this is a common operation, adding such parentheses to an entire codebase is tedious.

2.2.2 Struct Literals in Conditional Expressions

Go's syntax contains an ambiguity when it comes to the condition of conditional statements [4]: Both blocks of statements and constructors for structs use curly braces as delimiters. At the same time, conditions do not need to be surrounded by parentheses. Consider the following code snippet:

```
1  if a == b {
2    // Do nothing
3  } // "{ }" here would change the decision.
```

On first glance, this is a simple, if useless, if statement, with condition `a==b`. However, since the parser does not have any type information, `b` might also be a struct type. In this case, we might be comparing `a` to a newly constructed `b`-struct. This ambiguity is only resolved after the second closing brace: Since there is no block following this, we know that the pair of braces we just parsed delimit the block of the if statement, and our initial assessment was correct.

However, Go's parser makes its decisions without looking that far ahead. For this reason, Go will always interpret the condition as `a == b`, and, in

the case that there was another block afterwards (as in the comment), throw an error.

The old parser, however, failed on such inputs. It decides to parse the condition as a comparison between `a` and a new `b`-struct, and then fails because the if statement is missing a block.

2.2.3 Syntax for Predicate Constructors

Gobra supports predicates, which are essentially parametrized assertions. For these predicates, it supports partial application: This means that it is possible to construct a new predicate by fixing some arguments of a base predicate. The intended syntax for constructing a new predicate by applying some of its arguments is as follows: `myPred{_, 10}`. In this example, the resulting predicate would only take one argument instead of two, with the second argument being fixed at 10.

However, this syntax conflicts with Go's own constructors, because they also use curly braces as delimiters. For this reason, Gobra has chosen to avoid this issue temporarily by replacing the curly braces by the alternate delimiters '`!<`' and '`!>`'. This avoids issues with type checking, but is akward to use. Even though this issue lies mostly with the type checker, it is worthwhile to explore solutions in this thesis.

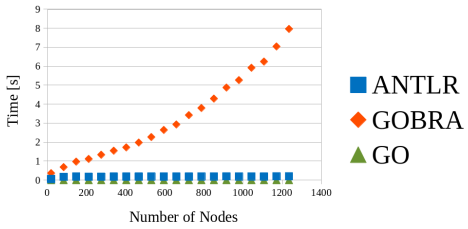
2.3 Alternative Parser Frameworks

In order to alleviate these problems, we have decided to use a different parser framework. We quickly limited our choice to two main alternatives: We wanted to either extend Go's handwritten parser with Gobra's new rules, or we would use *Another Tool for Language Recognition (ANTLR)* [3], a parser framework that had already been used in connection with Gobra previously [6]. Both of these have their own distinct advantages and disadvantages. This section explains the choice we made.

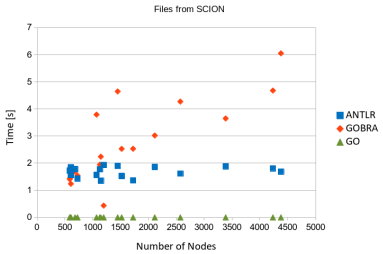
2.3.1 Performance

When it comes to performance, Go's own handwritten parser is a clear winner: It parses most files in just a couple of miliseconds. However, we must keep in mind that this is just the time used to generate a parse tree within Go: In order to import this parse tree into Gobra, we would have to use *JNI*. JNI is a tool that can be used to integrate C code into Java [2]. Since Go can be compiled to a C library, this would allow us to call Go functions from Java or Scala.

This process somewhat diminishes Go's performance lead: Calling C functions and accessing structs defined in C through the JNI layer is quite costly.



(a) Performance comparison on generated files.



(b) Performance comparison on some unannotated SCION files.

Figure 2.1: Preliminary performance evaluation of the three parser frameworks. The number of nodes denotes the number of components in Go's internal representation.

Our other option, ANTLR, is slower than Go's parser, but still outperforms Gobra, while being written entirely in Java. This allows for almost seamless interoperability with Scala.

We also conducted some preliminary tests on small filesets: First, we generated files containing different amounts of code to get an idea of the performance characteristics of the different parsers. Then, we chose some files from Github to further compare our two candidate parsers. The results of these measurements can be see in Fig. 2.1.

While this option sounds promising, we cannot definitively evaluate its viability yet, since we are only considering pure Go code without Gobra annotations. The final analysis in Sec. 6.1, however, confirms these preliminary measurements.

2.3.2 Maintainability

Both of our options require us to translate the parse tree from some intermediate representation to the representation used for Gobra's further verification steps. That means, that additional code is necessary that connects the parse tree to Gobra's existing pipeline. Since using the Go parser essentially requires two translation steps, ANTLR has an advantage here.

We also need to consider the limitations of each specific parser: While ANTLR is designed as a general parser generator, and can thus parse almost any grammar, Go's parser is designed specifically for the Go language.

In particular, it is designed so that any language construct must be identifiable by its first symbol. This can be seen, for example, in the design of top-level declarations. Each different kind starts with its own keyword: `const` for constants, `func` for functions, et cetera. Gobra's old grammar does not conform to this, which means we might have to carefully reformulate parts of it in order to integrate it into Go's handwritten parser.

For these reasons, we went forward with the ANTLR parser generator. It offers both good performance, as well as easy integration with Gobra's current codebase.

Chapter 3

Architecture

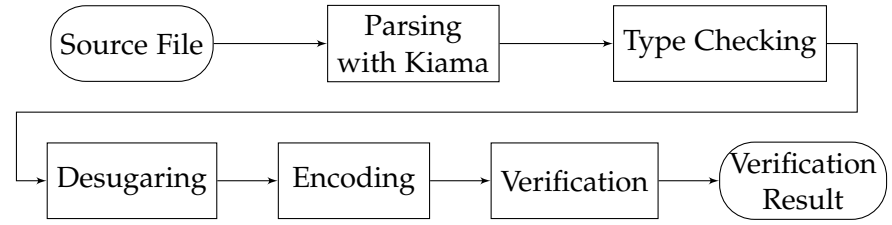
By using a new parser, we introduce an extra step into Gobra's pipeline: Translating the new parser's internal representation of the parse tree to the Gobra abstract syntax tree (AST). This approach was chosen instead of completely rewriting the type checking and encoding steps to ensure interoperability, avoid blocking any other concurrent work on improving Gobra, and avoid unneccessary work. Furthermore, we have altered some preprocessing steps slightly. In the following sections, this will be discussed in more detail.

3.1 Changes in Architecture

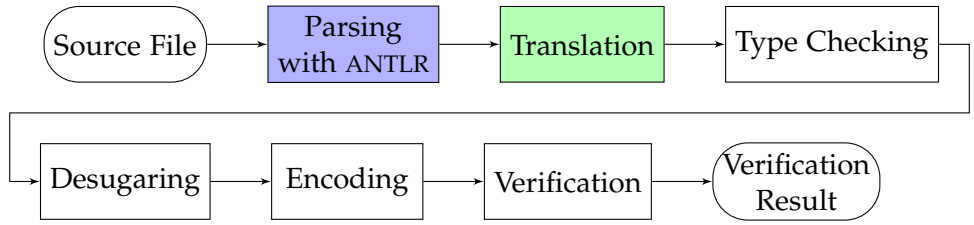
With the old parser, Gobra interprets and verifies files as follows: Gobra first parses the input, resulting in a parse tree. Gobra then type checks this parse tree. Given that type checking has succeeded, Gobra translates the parse tree into an internal AST. This step involves simplifying the language by removing syntactic sugar and augments the AST with information useful for the encoding. Gobra then encodes the internal AST into an equi-satisfiable Viper AST, which is then verified using one of Viper's backend verifiers such as Silicon.

As mentioned, we attempt to reduce the necessary changes to the existing pipeline. However, using a new parser entails receiving a differently structured parse tree than before. Therefore, we add an additional translation step between parsing and type checking. The new step takes the parse tree produced by ANTLR and translates it into Gobra's old parse tree format. The new architecture is displayed in Fig. 3.1.

In this way, all of the other steps can remain unchanged. The new step is described mainly in Sec. 5.1. Here, we just describe some peculiarities that need to be taken into account to ensure a seamless transition.



(a) The old pipeline



(b) The new pipeline

Figure 3.1: Comparison of Gobra's old and new pipeline. The step highlighted in blue was altered significantly, while the step highlighted in green is completely new.

Positioning On verification failure, we want to give the user detailed information, which includes the location of the failed assertion or proof step. Most parsers encode this location information in their parse trees. The old parser, however, used a separate map to associate a given AST node with a position. Previously, this map was computed automatically by the parser generator framework. Because the new parser handles positional information differently, the translation from new to old parse AST has to also compute this positions map.

Errors Parse errors must also be emitted as a specific type of object to be reported by Viper. For this purpose, ANTLR provides an *error listener* interface. This is an object with a specific entry point that gets called whenever an error is encountered. We can then override this entry point to analyze the error and wrap it as a Gobra error.

Chapter 4

The Gobra Grammar

In order to utilize ANTLR to parse Gobra programs, we formulate a grammar for Gobra programs in ANTLR's specification syntax. This process is split in two parts: defining a grammar for the Go language, hereafter referred to as the *base grammar*, and then extending this grammar to include Gobra annotations. ANTLR hosts a repository of contributed grammars, including one for the Go Language. This grammar can be used as a starting point for the first step, however, it contains some errors that we have fixed.

First we discuss relevant aspects of ANTLR's parser algorithm (Sec. 4.1). Afterwards, we present our changes to the original Go grammar hosted by ANTLR, resulting in our base grammar (Sec. 4.2), including limitations of our base grammar. Finally, we present the extensions to the base grammar resulting in the Gobra grammar (Sec. 4.3).

4.1 ANTLR

ANTLRv4, in this paper referred to as just ANTLR, uses the parsing algorithm Adaptive LL* (ALL*) [7]. ALL* is a left-to-right, leftmost derivation (LL) parser with arbitrary lookahead. In this section, we highlight the differences between ALL* and traditional LL parsers. We first look at the structure of a more traditional LL parser.

The first step of most parsers is to split the input into terminal symbols, or *tokens* (for example '`if`' or '`:=`'). These tokens serve as an abstraction so that the parser does not have to deal with a raw character stream. After this step, often called *lexing*, the actual parser will step through these tokens to decide whether the input matches the grammar or not, beginning at a specified *start rule*.

Because a rule specified in EBNF has multiple alternatives, some of which may match the input while others may not, we need a way to decide which

```
1  func assignment() {
2    // inspect k tokens and return the predicted alternative for
3    // the assignment rule
4    switch inspectLookahead(_Assignment) { // ANTLR: adaptivePredict
5      case 1: // rule alternative 1: IDENTIFIER '=' expression
6        consume(IDENTIFIER); consume(EQUALS); expression();
7      case 2: // rule alternative 2: IDENTIFIER ':=' expression
8        consume(IDENTIFIER); consume(COLON_EQUALS); expression();
9    }
```

Listing 4.1: The recursive descent function for assignments.

alternative to go ahead with. A *recursive descent* parser usually defines a separate procedure for each rule. Within these procedures, it looks at the next k tokens, and decides which of the alternatives specified within the rule should be used. When it has decided, it consumes any tokens specifed directly in the rule, and recursively calls the respective procedures for any sub-rules contained in the alternative. See Listing 4.1 for an example based on a simplified rule for variable assignments.

For ANTLR, this basic structure remains, but instead of a static lookahead function that looks at the next k tokens (the *lookahead* tokens) and decides on an alternative, ANTLR employs an alogrithm called *adaptivePredict*. This algorithm analyzes the grammar with ATN simulation and incrementally builds a cache of Deterministic Finite Automata (DFA) that map input sequences to parsing decisions. [7]

The Augmented Recursive Transition Network (ATN) is essentially a syntax diagram of the grammar. To analyze the grammar, ANTLR steps through the input tokens while using the ATN to keep track of all viable alternatives until there is only one such alternative left, or an error is reached. If only one alternative remains, the DFA for this rule is updated with transitions for each token we have traversed, and an accepting state representing the alternative we have decided on.

To illustrate, consider the code snippet in Fig. 4.1, and assume that the cache is initially empty. We will focus on the cache for the *type* rule. ANTLR first checks the cache for this rule, resulting in a miss.

Thus it falls back to ATN simulation. Before reading the first token, it has no information, so the set of all viable alternatives is simply the set of all alternatives. However, once it reads the token '[', only the two rule alternatives for array types remain viable. It will add a new state to the DFA and connect it to the initial state with an arc for '['.

Because there are still two viable alternatives, ANTLR will read another token, in this case a number, leaving only explicitly sized arrays as a viable alternative. It will create a new accepting state, labeled with the corresponding alternative number. This results in DFA 4.1b

```
1    var i [10]int;
2    var j [...]int;
```

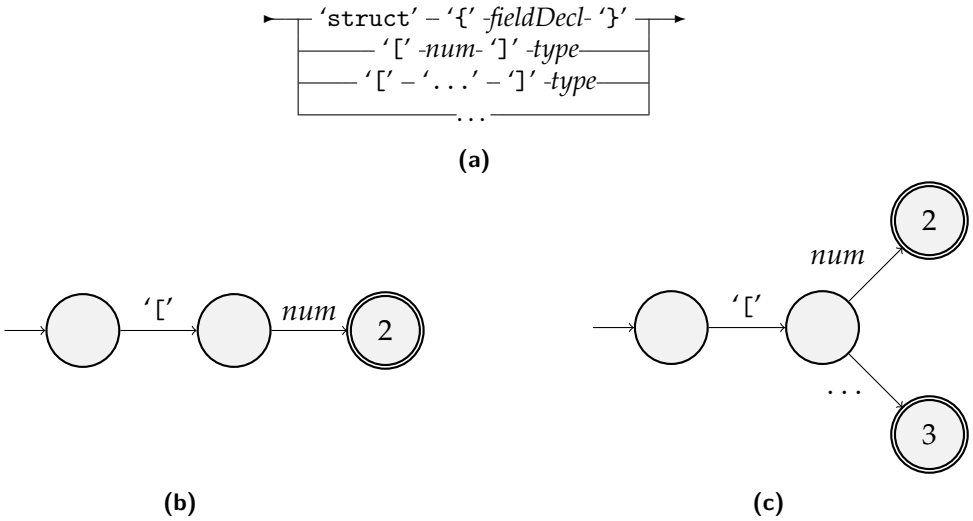


Figure 4.1: The input code, followed by the syntax diagram for the rule *type* (a), the DFA for this rule after reading the first line (b), as well as the DFA after reading both lines (b). Note that this is only one DFA out of many. ANTLR maintains a separate DFA for each rule.

For the type in the second line, ANTLR will follow the first transition in the DFA cache. However, from this state, there is no edge matching the ellipsis symbol. Therefore, it will go back to ATN simulation as above, ending up with a new final state for rule alternative number 3, resulting in DFA 4.1c.

When ANTLR encounters another array type later in the input, the lookahead tokens will match the DFA, and ANTLR knows which rule alternative to choose without doing any analysis of the syntax diagram. We call this a *cache hit*. Whenever the respective DFA does not match the input, and the algorithm for ATN simulation is invoked, this is a cache miss.

Since most programming languages are LL(k) for some fairly small k, this process leads to a big increase in performance: ATN simulation will only be done for a small number of decisions, while the bulk of decisions is made using very small DFAs.

By only maintaining a single DFA per rule, however, the parser does not take into account previously consumed tokens. Usually, this is not an issue, but consider the following grammar from [7]:

$$S \to xB \mid yC \quad B \to Aa \quad C \to Aba \quad A \to b \mid \varepsilon$$

On lookahead *ba* inside the rule A, there is no way to decide whether to choose the first or the second alternative of A. If the surrounding rule is

C, we must choose ε, if it is B, we have to consume b. The grammar is still context free and LL, but it is no longer strong LL (SLL).

In the context of recursive descent parsers, the concept of surrounding rules can be thought of as the call stack. This also makes it clear why the distinction between LL and SLL is important for ANTLR: Creating a DFA for every possible call stack leads to an immense number of DFA, and consequently a very low cache hit rate.

ANTLR has therefore chosen a hybrid approach: It will make decisions without taking the call stack into account, until it reaches an ambiguity or an error. Because it cannot tell whether this is a true conflict or just a dependence on the call stack, it re-examines the input while taking into account the call stack. This gives it the full power of LL(*) parsing, while also being performant for most grammars. This needs to be taken into account when defining the grammar for Go and Gobra's specifications.

Summary From the previous sections, we can gather the following key points: ANTLR is able to handle all LL grammars. However, the performance of ANTLR depends on two main factors: how much lookahead is needed for common[1] rules, and whether there are many rules that depend on the call stack. With this in mind, let us take a look at the ANTLR's Go grammar.

4.2 The Base Grammar

ANTLR's Go grammar is based almost one-to-one on the Go Specifications. It is able to parse almost any correct input, however, it does not always do so correctly: The actual grammar used in Go's internal parser is slightly different from the specification grammar. The Go parser is a handwritten recursive descent parser [1], and as such relies on having an unambiguous SLL(1) grammar, while the grammar presented in the specification is optimized for readabillity, and therefore ambiguous in some places, most notably in expressions.

4.2.1 Expression Precedence

Expressions with operator precedence are challenging to parse [5]: We must choose between a complicated grammar that explicitly encodes operator precedence, or a compact grammar that is aided by some special parsing mechanism to assign precedence to operators. The Go specification is formulated as a simple EBNF grammar, with expressions defined as follows:

[1] Because there is a separate cache for each rule, the performance of one rule does not impact the performance of other rules. We can therefore tolerate it if some rarely used language constructs require large lookahead and have a low cache hit rate.

Expression ::= *UnaryExpr* | *Expression binary_op Expression*

UnaryExpr ::= *PrimaryExpr* | *unary_op UnaryExpr*

binary_op ::= '||' | '&&' | *rel_op* | *add_op* | *mul_op*

EBNF does not specify a way to assign precedence to ambiguous rules: Any order of association is equally valid in this grammar. Since we want a unique parse tree that expresses the order of operations correctly, we need to change the grammar. Often, this is done by creating a new rule for each precedence level, but ANTLR explicitly defines a way it deals with ambiguous operator rules: Precedence will be assigned in the order that the operators are specified in the rule. The maintainers have thus rewritten the expression rule as follows.

Expression ::= *UnaryExpr*
| *Expression mul_op Expression*
...
| *Expression* '||' *Expression*

Now, multiplication is assigned the highest precedence. However, the transformation only applies to operators directly defined within the *Expression* rule. *UnaryExpr* will therefore be assigned low precedence, even though it is the first alternative. Consequently, the parser has no indication as to its precedence. Because ANTLR is designed to output a single parse tree, it will always treat these types of rule alternatives as primary expressions with low precedence, e.g. incorrectly parsing `-a+b` as $-(a + b)$, and report an ambiguity to notify us that our grammar contains problems.

Therefore, we need to include *UnaryExpr* in ANTLR's precedence transformation: This means removing the indirection, giving us a rule where all operators on expressions are defined directly within the *Expression* rule:

Expression = *primaryExpr*
| *unary_op Expression*
| *Expression mul_op Expression*
...

This approach will also be used when integrating Gobra's new operators, such as implications and quantifications into the grammar.

4.2.2 Semicolon Omission

In Go, semicolons are automatically inserted after statements that end in a line break. The Go lexer contains a simple state machine that keeps track of whether a line break can be treated as a semicolon. This mechanism is not trivial to implement when using an existing parser framework instead of a handwritten parser.

Gobra had a preprocessing step adding semicolons. This preprocessing step is implemented using regular expression matches. This old approach has several disadvantages: it is hard to extend, maintain, and not flexible enough for annotations introduced by Gobra.

ANTLR's Go grammar implemented this feature with complicated semantic predicates. Semantic predicates are conditions that are checked during parsing, which, when they evaluate to false, will remove this rule alternative from the current analysis, even if the input tokens may still match it. With this method, ANTLR's Go grammar never induced incorrect semicolons, but also did not induce some correct semicolons. Consider the following list of statements:

```
1  i := a
2  + b
```

This code should be parsed as two separate statements, but since ANTLR's Go grammar only checked to include semicolons in places where they were needed for a successful parse, it did not induce the semicolon after the identifier *a*, parsing both lines as a single assignment. Fixing this with more semantic predicates checking for line terminators inside expressions is tedious and likely detrimental for performance. Instead, our approach is inspired by the state machine of the Go lexer.

The state machine in the Go lexer is represented by a single boolean flag. It is set to true only after the appropriate tokens, and unset as soon as the next character is consumed. ANTLR's lexer supports *lexer modes*. Lexer modes allow the lexer to apply different lexer rules to different parts of the input. With these lexer modes, we can simulate Go's state machine.

Because Go's state machine consists of only two states, we need to add a single new mode, called *NLSEMI*, in addition to the default mode. Whenever the Go lexer toggles the boolean flag, our parser enters this mode, for example after an identifier is read (Line 1 of Listing 4.2).

Within this mode, white space and single-line comments are ignored, while semicolons and line breaks are interpreted as `EOS`-tokens, which are equivalent to semicolons in the parser. The rule '`OTHER`' (Line 5) only matches the empty string. Because lexers always choose the longest match to proceed, this rule is only matched if none of the others are successful.

4.2.3 Contributions to the ANTLR Repository

This issue, as well as some other problems relating to control flow statements and floating point literals, do not only concern Gobra, but any tool using ANTLR's grammar to parse Go programs. As a contribution of this thesis, we have merged our improved Go grammar into ANTLR's repository.

```
1  IDENTIFIER : LETTER (LETTER | UNICODE_DIGIT)* -> mode(NLSEMI);

2  mode NLSEMI;

3  WS: //[omitted, match any white space and comments here]
4  EOS: ; | [\r\n]+ | EOF;
5  OTHER: -> mode(DEFAULT_MODE);
```

Listing 4.2: A subset of Gobra's lexer rules. Rules below the mode-statement are only matched in the `NLSEMI` mode, while rules above belong to the default mode.

```
1  rep1sep(parameterDecl, ",")
```

```
1  parameterDecl (COMMA parameterDecl)* COMMA?
```

Listing 4.3: Equivalent definitions for parameter lists in Kiama and ANTLR.

4.3 From Go to Gobra

Gobra introduces many new syntactical elements to the Go language. To reduce maintenance, we use ANTLR's inheritance mechanism to extend the base grammar with support for Gobra annotations.

The base grammar is imported similarly to package imports in a programming language, enabling us to directly access rules defined in the base grammar, and override the rules that need to be changed. Unlike inheritance in object-oriented programming, there is no way to access the original rule. This leads to code duplication. Since code duplication is unavoidable, we have added comments documenting which parts of a rule are inherited and which parts are new.

4.3.1 Adjusting Gobra's Grammar

The old grammar is written for Kiama. Kiama's syntax is somewhat more complicated than basic EBNF, but it still consists mainly of the same operators. There are some useful operators in Kiama that cannot be replicated in ANTLR. For example, Kiama provides a function for elements that are repeated and separated by some token, like a comma. Rules using this construct have to be expressed in ANTLR using the basic EBNF rules (see Figure 4.3).

These differences are generally easy to handle. Because ANTLR can deal with almost all grammars, the old grammar can simply be transcribed to ANTLR's syntax and integrated into our new grammar. In most cases, this will work, however due to performance considerations and differences in the structure of the base grammar compared to Gobra's implementation of the standard Go grammar, some changes were made.

One simple example are expressions: In the old Gobra grammar, expressions are defined in an ambiguous way. Consider the following version of the grammar.

Expression ::= *PrimaryExpr* | ...

PrimaryExpression ::= ... | *GhostPrimaryExpression* | ...

GhostPrimaryExpression ::= ... | *Forall* | ...

Forall ::= '`forall`' *BoundVariables Triggers Expression*

Precedence is not specified: The rule *Forall* is shaped like a unary expression, with an un-delimited *Expression* at the end, but it should have lower precedence. We have encountered a similiar problem in Section 4.2.1, but here, both ANTLR and Kiama disambiguate this in our favour: Rules of this form are assigned low precedence. However, for ANTLR, ambiguities lead to fewer DFA cache hits: ANTLR will not be able to construct a small lookahead DFA, since we never reach a state with a single alternative.

The solution to this problem is almost identical to section 4.2.1, except that the we will add the new rule as the last alternative specified in the expression rule, expressing its low precedence relative to binary expressions.

We also move toward the Go language in the way we handle type identifiers: Parsing them as normal identifiers, and then looking them up in our symbol table during type checking. This not only works better with the base grammar but also removes some errors of the old parser.

In the old parser, identifiers and predeclared names occupy two distinct namespaces. Therefore, Gobra allows redefining these identifiers without shadowing the original type. This is not consistent with Go: Redefining built-in identifiers is allowed and will shadow the built-in type.

Gobra's symbol table already supports some built-in functions, so we only needed to extend it to be able to handle types as well.

4.4 Limitations

Because ANTLR uses a lexer to split the input into tokens before parsing, operators will have to emit their own tokens. This has the consequence that some words, like `in`, the set membership operator, can no longer be used as identifiers. This is hard to mitigate, so we have decided to implement clear error messages when these tokens are used as identifiers to avoid confusion.

With the exception of these limitations regarding identifiers, the new parser can parse a superset of the old one. Some features that the parser can correctly recognize, but are not part of Gobra's specification yet are Go's control

flow statements, such as `goto` and `fallthrough`, as well as imported embedded types.

Chapter 5

Maintainability

Because Gobra relies on Kiama's tree structure for its internal processes, the parse tree produced by ANTLR needs to be translated. The transformation is straightforward in most places, but to easen maintenance and extension, one must be careful to take a principled approach. The following sections discuss the structure of this transformation and explain how it can be extended by taking the addition of type parameters as an example.

5.1 ANTLR to Kiama Transformation

The parse tree generated by ANTLR consists of a Java object for each rule, which contains references to all of its children in the parse tree. Therefore, we will have to walk the entire parse tree to translate the Java objects, usually called *contexts*, to Kiama nodes. ANTLR provides a visitor interface to simplify this walk.

5.1.1 The Visitor Interface

In a visitor interface, traversal of the parse tree is done by a *visitor* object. This object contains a function for each context type, for example the *ExpressionContext* for our expression rule. The default implementation simply calls a method that visits all direct children of this context, and returns some aggregation, for this example a vector, of all the contexts. This means that, by default, it will traverse the parse tree in a depth-first manner.

For example, if we were to visit a simple expression like `a + b`, we would receive a vector of length three, containing the result of visiting `a`, then the addition operator, and the result of visiting `b`. This is where we override the visitor function. In this case, we use the results of visitng the left and the right side of the expression to create an AST node representing an addition (see Fig. 5.1), and return this as the result of our visitor function.

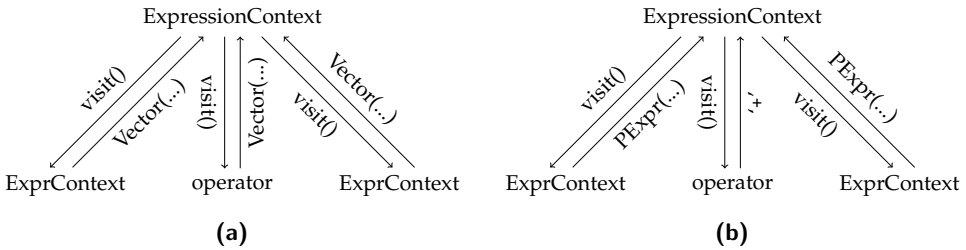


Figure 5.1: The results of the default and overridden implementation are shown to the left and right, respectively. Note that the results on the right hand side correspond to Gobra AST nodes.

Rules that are not represented as a new AST node, for example a parenthesized expression, are not overridden: The default implementation will simply return the result of the inner expression. This is the main advantage compared to manually visiting the tree.

5.2 Design Choices to Improve Maintainability

Whenever a rule can match inputs of multiple shapes, for example a parameter declaration that may or may not contain an identifier, we have to inspect the context to find out which elements are present and which aren't. For more complicated rules, this can easily result in cluttered code.

We have applied the following two principles to address this problem. First, we refactor rules such that alternatives that are not closely related to each other are split into separate rules. For any remaining alternatives, we consistently use pattern matching to distinguish between the alternatives. In some cases, however, the structure of the parse tree does not match the structure of Gobra's AST. In these cases, we carefully annotate any deviations from our standard patterns in documentation comments.

5.3 Extending Gobra with New Features

Changing the input language's grammar potentially affects three locations in the parser: The grammar specification itself, corresponding functions in the translator, and the type checker. Of course, we also need to encode these changes for the verifier, but we limit ourselves to changes relating to the parse tree. For our example, we focus on the changes in the corresponding visitor function.

To illustrate this with an example, we look at the introduction of type parameters to Go. Type parameters will be supported in Go beginning with release 1.18, slated for spring 2022. They allow the definition of generic structures and functions. Gobra does not yet support this feature, and no concrete

```
PTypeDef(r: PType, typeParams: Vector[PTypeElement], l: PIdnDef)
```

```
// ... (visit the other members)
val typeParams = if (ctx.typeParameters() != null) {
  visitNode[Vector[PTypeParameter]](ctx.typeParameters)
} else {
  Vector.empty
}
// ...
PTypeDef(right, typeParams, left)
```

Listing 5.1: Changes to the AST node and visitor function for type definitions.

plans to do so are in place. However, it is useful to sketch the process of adding new features, since it can be applied to other changes as well.

We look more closely at type definitions. For type definitions, type parameters are provided in square brackets after the identifier that is bound to the type. The name of a generic list looks like `list[T any]`, for example. The changes will reflect in the grammar roughly as follows:

TypeDef ::= *Identifier* [*TypeParameters*] *Type*

TypeParameters ::= '[' [*TypeParamDecl* { ',' *TypeParamDecl* } [','] ']'

TypeParameters is a completely new rule, while TypeDef has simply been extended with a new optional component. TypeParameters is structurally similar to function parameters, so it will suffice to copy, and slightly alter the visitor function for regular function parameters. Therefore, it is more interesting to consider how type definitions will be changed.

We will extend Gobra's AST node for type definitions with a new member that represents the type parameters. Like function parameters, this will simply be a vector of type elements, which may be empty if the user does not provide type parameters (see Fig. 5.1).

Because type parameters are completely optional, unlike regular parameters where the parentheses are still required, we have to handle the case where no type parameters are specified. If the context does not contain any type parameters, the function `ctx.typeParameters()` will return `null`.

Due to the visitor's modular nature, this is the only change we have to perform. The introduction of type parameters will also alter some other rules, where changes will happen analogously. Changes to Gobra's specifications will follow nearly the same process, but additionally, we will need to consider how the new rule affects grammatical structure: We should not create rules with large amounts of lookahead.

Chapter 6

Evaluation

In this chapter, we confirm that our changes in the parsing process indeed address Gobra's limitations regarding performance and completeness: First, we measure parsing performance of the new parser in comparison to the old one for multiple test sets. Then, we evaluate the completeness of the new parser by attempting to parse files that were known to fail with the old parser and unit tests for the Go language, which include tests designed specifically to test Go's own parser against edge cases. Finally, we evaluate the parser's error messages on files containing syntax errors.

6.1 Performance Comparison

In order to measure performance, we benchmark the parser alone and in context using three test sets. The first set consists of the files used to unit-test Gobra. These files cover almost all language constructs of Gobra, and offer a good overview over the parser's performance and capability.

In order to take into account real-life workloads, we also use VerifiedSCIONs annotated files as a test set. VerifiedSCION is a project that aims to verify the source code of SCION, an attempt at creating a new, more secure internet architecture [8].

Finally, we use Go's own unit tests, which are designed to test both the Go parser as well as other parts of the Go environment, to evaluate the new parser's performance on pure Go files, as well as its completeness.

We see a speedup between 24x and 60x. Note that this is only the speedup during the parsing stage. Time spent in the other stages of Gobra did not change. Table 6.1 shows the total amount of speedup for each of the test sets.

In the next sections, we describe the evaluation setups for the tests and evaluate our results.

Test Set	Gobra	ANTLR	Speedup
Gobra Unit Tests	4min 46s	11.5s	24x
SCION Files	5min 30s	6.5s	51x
Go Unit Tests	18min 15s	18.3s	60x

Table 6.1: Comparison of mean parsing times summed over each test set.

6.1.1 Parameters

All tests were run on a machine with an i5-4670k processor and 16GB of RAM, with a JVM stack size of 128MB. Each file was parsed 5 times, for ANTLR the DFA cache was reset between each iteration (see the following section for more detail). Unless explicitly specified, a run does not include parsing other files, for example imports, since this is dependent on semantic content of the file, and not the file itself. The best and worst result for each file is discarded. The time associated with each file represents the mean over the remaining three iterations.

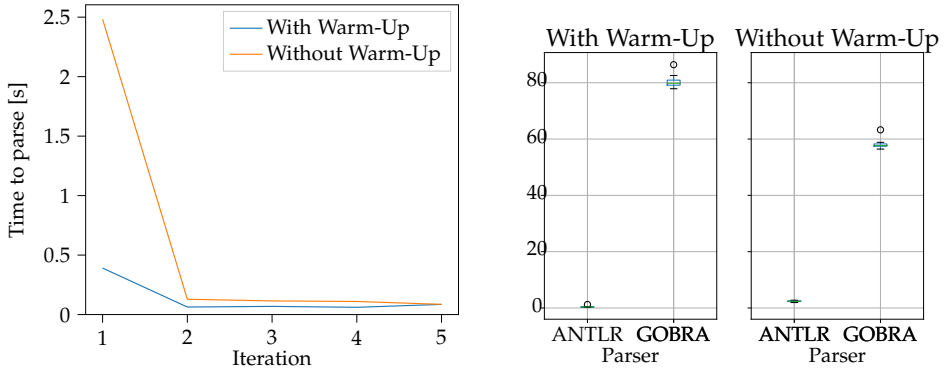
6.1.2 Impact of Warm-Up

For evaluation, it is important to recall that ANTLR builds up DFA caches while running and that these caches are not per-file but per-JVM instance. In practice, this means that ANTLR performs best when already warmed up. However, when measuring performance, there is one important consideration: Running ANTLR repeatedly on the same file and then removing the best and worst run will heavily skew results.

This is because after having parsed a file once, ANTLR has included each decision it encountered in this file in its DFA caches. When parsing the same file again, we observe a 100% cache hit rate. As can be seen in Fig. 6.1a, this effect is considerable: Without warm-up, ANTLR took over two seconds to parse this file for the first time, and just over one tenth of a second in any subsequent iterations. Even with warm-up, the first iteration takes almost five times longer than subsequent runs.

While we avoid this effect in our measurements, which are intended to mimic behaviour when manually verifying a large codebase, this is actually a huge advantage when it comes to IDE integration. In an IDE, we usually have a continually running JVM, and a limited set of files that are parsed each time the user changes them. Because the user only changes small parts at a time, performance will be similar to the later iterations in Fig. 6.1a. In some cases, this can lead to a speedup of over 200x compared to the old parser.

Another interesting observation is the following: While ANTLR clearly bene-



(a) When repeatedly parsing the same file in one instance of the JVM, ANTLR gains immensely in performance. When the DFA cache is already warmed up using different files, this effect is reduced, but still considerable

(b) Parsing a reference file 20 times with each parser. On the left, the JVM is warmed up with a subset of Gobra's unit tests, while on the right, the reference file is the only file parsed in this JVM instance.

Figure 6.1: The effects of warm-up and repetition on the two parsers

fits from a warmed up JVM, Gobra's performance is diminished. This was observed over many runs, and exemplified in Fig. 6.1b. It is not clear what this effect can be attributed to.

6.1.3 Speedup of the New Parser

As mentioned in the introduction, we generally reach a speedup between one and two orders of magnitude. Fig. 6.2 shows the relative speedup that the new parser exhibits when parsing files from VerifiedSCION. As you can see, most files experience between 10x and 100x speedup. However, you can also see some values below one. These are particularly small files, where the new overhead that comes with ANTLR outweighs any of its performance benefits.

6.2 Completeness

In this section, we compare parsing power between the old and the new parser. First, we consider the weaknesses described in Sec. 2.2. Then, we use Go's unit tests to check both parsers for any weaknesses that have remained unknown so far: These tests include many examples of constructs that used to be parsed incorrectly by Go's own handwritten parser.

Indexing expressions now work as intended and conditional expressions followed by empty blocks will not cause an error. With this fix, however, a slight issue was introduced: While unparenthesized struct literals inside conditional expressions are explicitly disallowed in the Go specification [4], be-

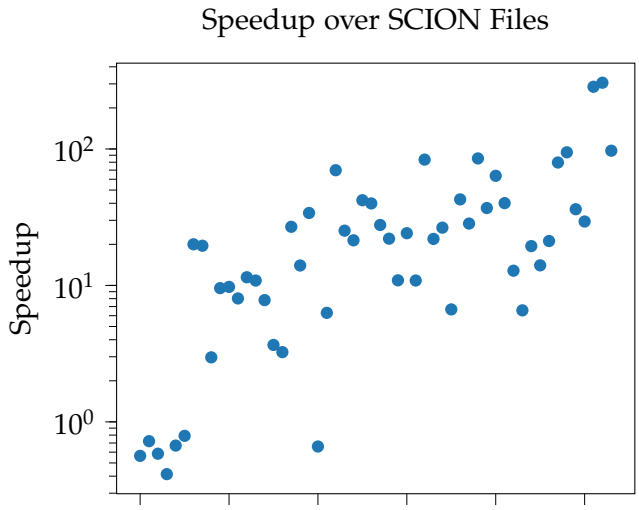


Figure 6.2: Speedup over SCION's annotated Gobra files. The files are ordered on the x-axis by the time required to parse them with the old parser.

cause the handwritten recursive descent parser cannot handle these, ANTLR is able to successfully parse these inputs.

This does not actually affect correctness, as these "illegal" struct literals can simply be surrounded by parentheses without changing the program's meaning. We have also put in place mechanisms that emit a warning when such structures appear, but since the verification framework does not currently support warnings, they are not yet shown to the user.

Another issue mentioned in Chapter 2 was the awkward syntax for predicate constructors. We have made progress toward supporting the intended syntax using curly braces instead of the temporary work-around. Parsing and type checking such operations is now mostly supported, however, some further work is required in the desugaring step.

6.2.1 Limitations

While the new parser is able to parse a much larger set of files, it has introduced some new limitations: Because it is a lexing parser, language keywords can no longer be parsed as identifiers, even when there is no ambiguity between the two interpretations. This means that some words, for example `in`, which are often used as variable names, are now reserved. We have begun work on a solution to this problem.

To avoid having to refactor files with large amounts of these reserved words as identifiers, but few or even no uses of them as Gobra keywords, progress has been made toward introducing a setting that parses the reserved words

as identifiers by default, while requiring a backslash before them to act as keywords. For example, the expression `myElem in mySet` could be rewritten as `myElem \in mySet`. This setting can be set for all files, or overridden on a per-file basis to allow the choice between refactoring reserved keywords or Gobra's reserved words, depending on which alternative entails less work.

Furthermore, features that are not supported by Gobra such as complex numbers, are now partially supported: Complex numbers are parsed by ANTLR, but then cause an error during translation to notify the user that this type of number is not supported by Gobra yet. This is another advantage compared to the old parser, since such constructs would often emit confusing parser errors that do not inform the user that these features of Go are not supported by Gobra yet.

6.3 Better Error Messages

On parse failure, the old parser often emitted confusing error messages that did not help the user in identifying and fixing the issue. In this section, we compare error messages between the parsers, discuss how ANTLR can provide better error messages, and discuss how we can customize ANTLR's error messages to be more specific for some common programming mistakes.

6.3.1 Messages for Unsupported Go Features

Some features of Go are not supported in Gobra. This includes some data types like imaginary numbers, but also more advanced features like embedding imported interfaces. When encountering such constructs, the old parser does not give the user any indication that their input is correct, but contains an unsupported language feature: It just emits a regular syntax error. Since ANTLR supports these features, and returns a valid parse tree, we can detect such usages and emit specific error messages.

6.3.2 Slice Expressions

Go allows taking *slices* of arrays. Slices are variable length sequences containing a single type. Slices can be thought of as sub-arrays, and as such, their basic constructor takes a base array, a low index and a high index. For instance, `myArray[2:10]` constructs a slice containing the elements between index 2 and 10 from the `myArray`. In this form, one or both of these indices can be omitted, with the default values being 0 and the length of the array, respectively.

There is another constructor, called a *full slice expression*. Here, we not only give a low and a high index, but also a *capacity*. In Go, the length of a slice

can be extended until it matches this capacity. In the full slice expression, only the low index can be omitted, while the high index and the capacity are required. Omitting one of these is a common mistake: Go's handwritten parser has implemented specific error messages for this case.

In Gobra, the error message emitted by this mistake exemplifies its issues. The message reads "'(' expected, but ']' found". This does not communicate the issue to the user. While the suggested token, an opening parenthesis, may appear here as the beginning of a parenthesized expression, the user has to discover this "hidden" meaning themselves.

ANTLR's default error message is not much more helpful, indicating only that the syntax inside the slice expression is wrong. However, ANTLR can give us much more control over the error messages. Inside ANTLR's error handler, we have full access to the parser's current state, including which rule is currently being parsed and all tokens that might lead to a viable alternative. With this information, it is easy to detect exactly this case and emit a custom error message, reading "In a 3-index slice, the 2nd and 3rd are required.".

With this same method, we also emit special messages for errors involving mistaken usages of the *declare-assign* operator, ':='. However, there are still some syntax errors where ANTLR is nearly as unhelpful as the old parser. For this reason, we have documented the error listener well, so that any future input about confusing error messages can quickly be addressed with specific error messages.

Chapter 7

Conclusion

In this paper, we have addressed Gobra's bad performance by switching to a different parser framework. By switching to the parser generator framework ANTLR, we have decreased parsing time significantly: Large workloads that used to take several minutes are now completed in a few seconds. This is possible not only due to the new framework's advanced parsing techniques, but also because we took this framework's performance characteristics into account when implementing a grammar for Gobra.

At the same time, we have also improved user experience by emitting more helpful error messages. This was done by taking advantage of ANTLR's error listener interface that allows us to inspect each error's context and detect common errors. With this information, we can emit more detailed error messages, and are not bound by the parser framework's own error messages.

7.1 Future Work

Even though the new parser supports more Go features, we have also introduced some limitations: Gobra adds new keywords to the language, and these may conflict with identifiers in existing Go programs. One approach to address this has been proposed in this thesis, namely, to introduce a parsing mode where all Gobra keywords need to be preceded by a backslash, or some other prefix such as '`gobra.`'.

This method is quite heavy-handed: Even if just one of Gobra's keywords is used as an identifier, switching to this new mode requires us to prefix all Gobra keywords. Instead, the parser could detect these usages individually and require the prefix for just those keywords used as identifiers in a given file.

It is also possible to refactor all usages of identifiers in the ANTLR grammar to not only accept the `IDENTIFIER` token, but also the tokens representing

the reserved keywords. With this approach, care would have to be taken to avoid ambiguities or reductions in performance.

Another interesting area is IDE integration. Previously, the poor performance of the parser would have hindered user experience in this context. ANTLR, on the other hand, is perfectly suited for this task: Repeatedly parsing the same file with slight modification benefits greatly from ANTLR's caching mechanism, potentially leading to exceptional performance in this use case. Work in this direction also includes allowing for partial parsing of files, enabling rapid feedback during development.

Bibliography

[1] parser.go - Go. https://cs.opensource.google/go/go/+/refs/tags/go1.17.1:src/cmd/compile/internal/syntax/parser.go. [Online; accessed 15. Mar. 2022].

[2] Java Native Interface Specification: 1 - Introduction. https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html#java-native-interface-overview, Dec 2021. [Online; accessed 15. Mar. 2022].

[3] ANTLR. https://www.antlr.org, Feb 2022. [Online; accessed 15. Mar. 2022].

[4] The Go Programming Language Specification - The Go Programming Language. https://go.dev/ref/spec, Mar 2022. [Online; accessed 15. Mar. 2022].

[5] Robert Floyd. Syntactic analysis and operator precedence. *Journal of the ACM*, 10:316–333, 1963.

[6] Eva Charlotte Mayer, Peter Müller, and Alexander Pretschner. Assertion-based testing of go programs. Master's thesis, ETH Zürich.

[7] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll(*) parsing: The power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on object oriented programming systems languages & applications*, OOPSLA '14, pages 579–598. ACM, 2014.

[8] Adrian Perrig. *SCION: a secure internet architecture*. Information security and cryptography. Springer, Cham, 2017.

[9] Anthony M Sloane. Lightweight language processing in kiama, 2011.

[10] Tony Sloane. Experiences with domain-specific language embedding in scala. In *Domain-specific program development*, page 7, 2008.

[11] F A Wolf, L Arquint, M Clochard, W Oortwijn, J C Pereira, and P Müller. Gobra: Modular specification and verification of go programs. pages 367–379. Springer International Publishing, 2021.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Parser for Go Programs and Specification

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Berling	Nico

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 15.03.22	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.