Extracting a Rascal Grammar From The Swift Reference Manual

Roman Bell

July 2021

Contents

Chapter 1

Abstract

The purpose of this project is to develop a parsing pipeline which extracts a grammar from the Swift language reference manual. The pipeline will be written entirely in Rascal, and the extracted grammar will be stored as a Rascal grammar, which is designed to contain the production rules and start symbols of a grammar, and therefore is suited for implementing the grammar of Swift.

From there, the grammar is used to parse Swift code, and then undergoes rounds of improvement based on the results of the parsing, to mainly either remove the causes of parse errors in the grammar, or to remove ambiguity.

Chapter 2

Introduction

Grammarware comprises grammars and all grammar-dependent software, i.e., software artefacts that directly involve grammar knowledge[10]. In the case of this project, the software artefact is a grammar for the Swift programming language, extracted from its language reference manual.
A grammar for a programming language can provide a huge amount of insight into the structure of a language, as well as into decisions made in developing the language. Accurate grammars are also difficult to obtain, and are often developed from a variety of sources. One popular source for grammars for a programming language is the manuals and language specifications of that language, although these are known to have issues, and, as is shown later in this paper, do not necessarily represent the language correctly.

This paper contains the methodology of how the grammar is extracted from the Swift language reference manual, and the various tools developed for the extraction. Alongside a description of the tools is the reasoning behind how they have been developed, as well as difficulties that have been encountered during development, and how they have been overcome.

Over the process of extracting the grammar from the manual, and also using the manual to parse Swift code, certain observations and criticisms of the manual became evident. These observations are one of the main deliverables of this overall project, and are backed up with examples of sections from the manual which contain examples of the observations made, and, if relevant, the changes made to the grammar in order to fix them.

Further sections of this paper describe the various changes made to the grammar after extraction, how the changes are made, and the reasoning behind them. Each of these changes represents a necessary improvement made to the grammar, in order to fix some issue in how the language of Swift is represented in the manual.

After these improvements, large amounts of Swift code from various sources are parsed, and the results of these are presented and discussed. This section also contains an overview of the functionality for using the grammar of Swift to parse Swift code, and how it has been developed. These results show that although the grammar has been improved from its initial raw state, it is still imperfect, and doesn't actually consistently parse Swift code. Also, the results vary depending on the sources of Swift code, and the various reasons for this, based on properties on how the code within the sources is structured, are also discussed.

Below is a short overview of this paper, and the contents of each chapter.

In chapter 3 of this paper, the problem is outlined, along with the proposed stages of solving it, and various problems that might arise along the way.

Chapter 4 describes the stages of the parsing pipeline which have been designed, and for each stage the rationale behind how it has been developed, as well as examples of how that particular stage modifies small sections of the Swift manual from one form to another. It also gives information on, once the grammar has fully been extracted from the Swift manual to the Rascal grammar structure, the further changes made to the grammar in order to improve its ability to parse Swift code, and the reasoning behind the changes.

Chapter 5 contains some observations on the manual, which have arisen from the extraction process, and also while using the grammar resulting from the manual to parse Swift code.

Chapter 6 contains a breakdown of the results of parsing various collections of Swift code, as well as observations on the results and possible explanations for the results. It also contains an overview of the structure of the project in terms of the codebase of all tools developed over the course of the project.

Chapter 7 contains observations about the overall project, the process of developing the parsing pipeline, the properties of the resulting grammar after the various rounds of improvements made to it, and the results of parsing.

Chapter 8 contains related work which covers similar topics, and has been useful for reference throughout the whole project, and chapter 9 contains an overview of the outcomes of the overall project.

Chapter 3

Problem statement

The manual of the Swift programming language[3] specifies the lexical composition and syntactic structure of programs written in Swift, alongside other characteristics of the language. The manual is summarised on one page, which contains production rules specifying the correct structure of files containing Swift code.

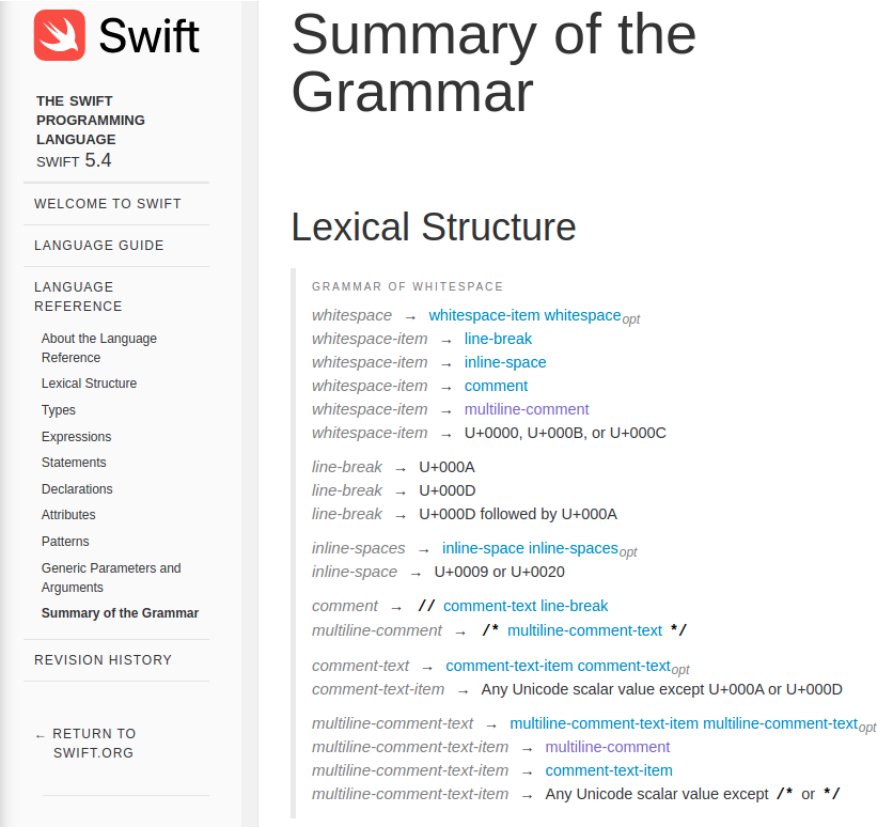

Figure 3.1: A subsection of the manual page

This manual page can be parsed to create a grammar based on the manual. The difficulty of developing the pipeline will largely depend on the structure of the HTML webpage which contains the manual. If there is one clear HTML structure which every rule follows, then a parser can simply iterate over every rule definition. However, if there is a lot of variation in how rules are defined, then a great deal of how rules are traversed will have to be manually implemented. Furthermore, if the HTML structure of a rule doesn't differentiate between subsections of a rule (for example, if literal productions are defined in the same way as syntactic productions), then it would be very difficult to have automated parsing of the grammar, and it would possibly be necessary to have a large part of the parsing be done manually.

This grammar can then be used to parse Swift code, and based on the results of the parsing be further refined to parse Swift code more successfully. This will take place by having a round of parsing, examining the feedback which arises from parsing, fixing one issue which is evident in the feedback, checking

that the fix has been implemented correctly (i.e it fixes the issue properly, and doesn't introduce other issues), and then proceeding with the next round of parsing. This process of changing the grammar from how it is specified in the manual to a grammar that correctly parses Swift code will provide insight into the differences between the language as defined in the manual, and how the language is actually defined and used.

The developed grammar can be tested by using it to parse a large number of Swift files, and then seeing how many files are parsed, parsed with ambiguity, or have parse errors. The main artefacts from this project are the grammar, observations on the manual which arise from the process of developing and refining the grammar, and the tools developed for extracting the Swift manual to a Rascal grammar.

Chapter 4

Approach

The pipeline for parsing the Swift manual page into a Rascal Grammar structure, and modifying it so it can parse Swift code, is as follows:

- The HTML of the manual page[3] is parsed to a data structure developed to hold production rules as defined in the manual, using existing functionality in Rascal.
- All rules which are defined in natural language in the manual are programmatically implemented, and added to the data structure.
- The data structure is converted into a Rascal Grammar.
- Some rules are changed, to implement constraints such as priority, and also some rules are changed significantly from how they are implemented in the manual, in order to allow the finished grammar to parse Swift code more effectively.
- More rules are changed, in order to remove ambiguity from the grammar.

4.1 Parsing the manual page

Initially, the data structure designed to hold rules temporarily while they are being extracted from the manual was as follows:

```
data Rule = rule(str name, list[Alt] alts);

data Alt = alt(list[RuleItem] symbols);

data RuleItem(bool opt = false) =
    ruleRef(str name) |
    literal(str val) |
    plainText(str def);
```

This covered the three cases for items on the right hand side of a production rule in the manual, which are either references to another rule, string literals, or a sentence in natural language defining some literal. However, when implementing the rules in natural language, RuleItem was changed to:

```
data RuleItem(bool opt = false, bool not = false) =
    ruleRef(str name) |
    literal(str val) |
    plainText(str def) |
    rangeItem(int startPoint, int endPoint) |
    keywordItem(str word);
```

The addition of rangeItem enabled long character ranges which are defined in natural language to be implemented far more effectively, as is described in section 4.3.

The HTML of the manual page is parsed using the readHTMLFile[9] function, which results in a large, nested node[18] structure, containing the contents of the page. The node structure in Rascal is designed for representing untyped trees. In general, when representing a section of HTML code, a node will specify which type of HTML code it represents, and also contain a list of the contents of the code, which is either nodes representing further structures contained within the HTML, or text. As an example, the HTML code “ some text ”would be represented by a node as “ “p”([”text”(”some text”)]) ”, while “ some text ” would be represented as “ “div”([”p”(”text”(”some text”))]) ”.

Pattern matching is then used to explore the resulting structure, to extract first the body of the webpage, and then each of the subcategories of the grammar (Lexical Structure, Types, Expressions, Statements, Declarations, Attributes, Patterns and Generic Parameters and Arguments). Each of these subcategories is then further visited, until eventually the nodes containing one individual rule are found.

As an example, below is the rule for numeric literals in the manual:

Figure 4.1: The rule defining numeric-literal in the summary of the Swift grammar

```
▼<p class="syntax-def"> == $0
  ▼<span class="syntax-def-name">
     <a id="grammar_numeric-literal_1278"></a>
     "numeric-literal"
   </span>
   <span class="arrow"> → </span>
   <code>-</code>
   <sub>opt</sub>
  ▼<span class="syntactic-category">
     <a href="../ReferenceManual/LexicalStructure.html#grammar_integer-literal">integer-literal</a>
   </span>
   " | "
   <code>-</code>
   <sub>opt</sub>
  ▼<span class="syntactic-category">
     <a href="../ReferenceManual/LexicalStructure.html#grammar_floating-point-literal">floating-point-literal</a>
   </span>
 </p>
```

Figure 4.2: The HTML code of this rule definition

Below is the HTML of the definition of this rule, annotated with which sections of HTML code define which parts of the rule.

```
// Opening tag of the rule
<p class="syntax-def">

// The span which contains the name of the rule, numeric-literal
<span class="syntax-def-name">
    <a id="grammar_numeric-literal_1276"></a>numeric-literal
</span>

// The span containing the arrow, dividing the name of the rule from its productions
<span class="arrow"> → </span>

// A literal production "-", within a <code> block. It is specified to be optional
// since it is followed by <sub>opt<sub>
<code>-</code><sub>opt</sub>
```

```
// A syntactic production for integer-literal, which contains a reference to another rule,
// as well as a link to where the rule is defined in the manual
<span class="syntactic-category">
    <a href="../ReferenceManual/LexicalStructure.html#grammar_integer-literal">
        integer-literal
    </a>
</span>

// Used to demarcate alternative productions, and isn't contained within a HTML tag
|

// A literal production "-", which is again optional.
<code>-</code><sub>opt</sub>

// Another syntactic production, for floating-point-literal
<span class="syntactic-category">
    <a href="../ReferenceManual/LexicalStructure.html#grammar_floating-point-literal">
        floating-point-literal
    </a>
</span>

//The closing tag of this rule. All rules are contained within their own paragraph.
</p>
```

After this specific fragment of HTML has been parsed with the readHTMLFile function, the resulting node will be:

```
"p"(["span"(["a"([],id="grammar_numeric-literal_1273"),

    "text"("numeric-literal")],class="syntax-def-name"),

    "span"(["text"(" → ")],class="arrow"),

    "code"(["text"("-")]),"sub"(["text"("opt")]),

    "text"(" "),

    "span"(["a"(["text"("integer-literal")],
        href="../ReferenceManual/LexicalStructure.html#grammar_integer-literal")],
    class="syntactic-category"),

    "text"(" | "),

    "code"(["text"("-")]),"sub"(["text"("opt")]),

    "text"(" "),

    "span"(["a"(["text"("floating-point-literal")],
        href="../ReferenceManual/LexicalStructure.html#grammar_floating-point-literal")],
    class="syntactic-category")

],class="syntax-def")
```

As can be seen, although the formatting is different, the same information is present within the node as is within the HTML. Instead of HTML tags, the types of sections of code are now shown through strings. For example, the optional "-" literals, formerly presented as " <code>-</code> _{opt}", are given by " "code"(["text"("-")]), "sub"(["text"("opt")]) ". This also shows that strings within the HTML

Type of change implemented	Number of rules changed
Rules defining character ranges	51
Adding keywords	9
Other	14
Fixing parse errors	10
Reducing ambiguity	11
Total	95

Table 4.1: Types of modifications made to the grammar, and the number of rules which were changed, for each type of change

which formerly had no HTML tags attached to them are now marked as being text. For example, the | for separating alternative productions in the HTML is presented as " "text"(" — ") " in the node structure.

Also, the nested structure of the HTML is preserved within the node. Although there are now no opening and closing tags for sections and subsections, the nesting is now done using brackets.

To convert the node into the Rule data type, the name is extracted by pattern matching, by finding the first element of the node that matches "text". Then a new Rule is declared, with the title of the current rule as its name. For the body of the production, each sub element in the node is processed, and appropriately added to the new Rule. If the element matches the "text"(" | ") tag, then a new Alt is added to the Rule, since a new alternative production will be following. If it otherwise matches "text", it is a definition of a rule in natural language, and is stored in the plainTextDefinitions list (and also added as a reference to a rule). If it otherwise matches "span", the production is a reference to another rule. If it matches "code", it is a literal. Other cases, such as "text"(" "), which is a space, are skipped over. Although the Rule data structure can hold the plain text definitions of rules, these were actually implemented as references to (at this time not existing) rules, so that the rules can be defined in the next step of the pipeline. Whether a production is optional or not is found by whether it is followed by a production that matches "sub", since in the HTML optional productions are followed by $_{opt}$. Then, if the current Rule does not already exist in the rulesArray (of type map[str, Rule]) which is used to keep track of all rules, it is added. If it currently exists, then all productions associated with the current rule are added to the Rule in the rulesArray.

The rule numeric-literal will be implemented as a Rule as follows:

```
rule("numeric-literal", [
    alt(literal("-", opt = true),ruleRef("integer-literal")),
    alt(literal("-", opt = true), ruleRef("floating-point-literal"))
]);
```

In the manual of Swift, there exists multiple definitions per rule, each specifying alternatives for that rule but also | is used in productions to separate different possible productions. Due to how the rules are parsed, in the rulesArray there only exists one instance of each Rule, with its list of Alts containing every possible production specified in the manual.

At this point, every rule specified in the manual now exists in the rules array. For every rule, all its production rules are implemented, with references to other rules and literals being correct, and the natural language definitions implemented as references to (currently) non-existing rules.

4.2 Implementing loosely defined rules

Table 4.1 shows the modifications made to the raw grammar after it has been extracted. Changes above the line were made to the grammar while it was in the Rule data structure. These changes implement specifications stated in other sections of the manual than the summary page, in natural language.

The changes implementing character ranges are necessary since a large number of rules in the manual consist of ranges of characters, defined by their Unicode values. An example of this is "U+00A1–U+00A7"[4]. This is implemented by the following function, which defines the rule under the currently empty definition in the grammar, which is already referenced by other rules:

```
public void U00A1U00A7(RuleItem input){
    list[Alt] newAlts = [alt([rangeItem(161,167)]) ];
    Rule newRule = rule("U00A1U00A7", newAlts);
    newRules += (input.name:newRule);
}
```

All keywords that are reserved in Swift are defined in the manual[4]. One production rule is made for all keywords, and 8 more rules are defined for subsets of keywords that are only used in subsections of the grammar. The Keyword rule is later used in specifications involving keywords. For example, identifiers are not allowed to contain keywords, and this new rule can be used as part of the implementation of that constraint.

Some more constraints are described in natural language, but are more complex than character ranges. An example of this is "Any punctuation except (,), [,], , or ", with punctuation having been defined elsewhere in the manual[4]. This requires more complex implementation than character ranges, but can still be implemented with the functionality available for defining ruleItems:

```
public void anyPunctuationExceptBrackets(){
list[Alt] ruleList = [ alt([literal(".") ]),
        alt([literal(",") ]),
        alt([literal(":") ]),
        alt([literal(";") ]),
        alt([literal("=") ]),
        alt([literal("@") ]),
        alt([literal("#") ]),
        alt([literal("-\>") ]),
        alt([literal("‘") ]),
        alt([literal("?") ]),
        alt([literal("!") ]),
        alt([literal("&") ]) ];
    Rule notBrackets = rule("anyPunctuationExceptBrackets", ruleList);
    newRules += ("anyPunctuationExceptBrackets": notBrackets);
}
```

At this stage, all information defined in the summary of the manual, alongside some information defined in the more specific sections of the manual (such as keywords), is implemented in the Rule data structure. Further specifications will be implemented in the grammar once it has been converted to the Rascal grammar structure, as defined in the next section.

4.3 Converting to Rascal grammar

In this stage of the pipeline, every rule which is currently implemented as a Rule is converted to productions in the Rascal Grammar[17] format. If, in the previous stages, the rule has been found to be one of the subcategories of keyword, then in the grammar the type of the rule is "keywords", and if the rule has been defined under the Lexical Structure section of the manual, then the type of the rule is "lexical", otherwise each rule is defined as being a syntactic production. Each item in the production of each rule is converted into the Grammar format in the generateSymbol function, which takes as its input a RuleItem, and gives a matching Symbol[20] for use in the Grammar:

Rule in RuleItem	Rule as Symbol
literal(“x”)	lit(“x”)
ruleRef(“x”)	sort(“x”)
rangeItem(x,y)	\char-class([range(x, y)])
keywordItem(“x”)	keywords(“x”)

Table 4.2: Rules as stored in the RuleItem data structure, and how they are translated to Rascal

```
public Symbol generateSymbol(RuleItem item){
    Symbol out = lit("empty");
    if(literal(_) := item){
        out = lit(item.val);
    }
    if(ruleRef(_) := item){
        out = sort(fixRulename(item.name));
    }
    if(rangeItem(x,y) := item){
        out = \char-class([range(x, y)]);
    }
    if(keywordItem(_) := item){
        out = lit(item.word);
    }
    if(item.opt){
        out = opt(out);
    }
    if(item.not && literal(_) := item){
        list[int] rangeVals = chars(item.val);
        out = complement(\new-char-class([range(n, n) | n <- rangeVals]) );
    }
    return out;
}
```

This function essentially acts as a mapping function between the two different types of data structures, and since they both are designed to implement a grammar in a similar way, the mapping between the two is quite direct. There are, however, some exceptions to this.
If a RuleItem item is a literal, and item.not is true, then the resulting Symbol is the complement of the characters in the literal. Literals which have previously been designated as keywords are implemented as described previously. Also, in the Rascal grammar format, the names of rules must start with a capital letter, and punctuation is forbidden within the name of a rule. Therefore, all rule names are reformatted to follow this. For example, the rule “numeric-literal” in the manual is defined under “Numericliteral” in the Rascal implementation of the grammar.

The implementation of production rules through \char-class vastly improved the size and efficiency of the resulting grammar. In older versions of the parsing pipeline, as described previously, all productions were either references to other rules or string literals, which meant that ranges of Unicode characters were also implemented as rules with one alternative production per character. Since several of the rules implemented in the previous stage of the parsing pipeline define character ranges over the range of all Unicode scalar values (for example “Any Unicode scalar value except U+000A or U+000D”[4]), the resulting grammar in Rascal had around 2.3 million lines, which was far too many to be useful for parsing files of Swift code. Expanding RuleItem with rangeItems, and implementing them with \char-class has lead to the resulting Swift grammar in Rascal, which has 2484 lines

4.4 Further changes to the grammar

At this stage in the pipeline, there is now the Swift grammar as specified in the manual implemented as a Rascal Grammar structure. In this section, all remaining mutations are applied to the grammar in order that it diverges from what is in the manual, but so that the grammar is more able to parse Swift code. The majority of these changes are based on observed errors when development versions of the grammar were used to parse Swift code, and also some constraints defined in the rest of the Swift reference manual outside

of the summary page. Every change stated below is implemented programmatically to the grammar. For example, below is the fragment of Rascal code for changing the rule Comment, and how the rule is defined in the Rascal grammar.

```
newRulesArray.rules[lex("Comment")].alternatives = {
    pr(lex("Comment"), [lit("//"), sort("Commenttext"), sort("$")]),
    pr(lex("Comment"), [lit("//"),  sort("$")])
};

lexical Comment =
     "//" $
    | "//" Commenttext $
    ;
```

4.5 Fixing parse errors

4.5.1 Layout

Currently, the grammar as implemented in Rascal includes lexical and syntactic rules, but no specifications for layout[14], although a lexical rule Whitespace is derived from the manual[4]. Since the manual specifies that various spaces, tabs, newlines and also the null character are considered to be whitespace, layout is therefore specified, using precedence, as Whitespace* ! >> [\ \t] ; Due to comments also needing to be counted as whitespace, this is expanded to: Whitespace* ! >> [\ \t] ! >> “//” ;

4.5.2 Identifier

Identifier is changed so that keywords are not allowed to be identifiers, unless separated by backticks. All other rules in Identifier that aren't escaped by backticks are given the qualifier that their produced strings are not followed by regular alphanumeric characters, which ensures that strings of characters used in identifiers are fully parsed as one identifier (previously, every substring of an identifier could also be parsed as an identifier, which caused parse errors, ambiguity and also very slow parsing). This matches how identifiers should be processed according to the lexical structure section of the manual:

> In most cases, tokens are generated from the characters of a Swift source file by considering the longest possible substring from the input text, within the constraints of the grammar that are specified below. This behaviour is referred to as longest match or maximal munch. [4]

4.5.3 Comment

Comment is modified in order to allow empty comments (”//” followed immediately by a newline). This is forbidden in the manual, but is very commonly used to break up large comment blocks in actual Swift code.

4.5.4 Multilinecommenttext, Multilinecommenttextitem, CommentChar

Similarly to Comment, multiline comments are also changed to allow empty comments. Additionally, a new rule CommentChar is added to implement the constraint that multiline comments are allowed to contain (possibly multiple) nested multiline comments, but otherwise are not allowed to contain the start and end tokens of a multiline comment (“/*” and “*/””).

4.5.5 EOS, Statement, Endstatement, Prog

Files of Swift code are composed of one or more statements. These statements are either separated by a newline or a semicolon. To implement this, a new rule EOS (end of statement) is added which produces either a literal newline or semicolon, for use as a separator between statements. Since the last statement in a file does not need to be followed by a newline or semicolon, a new rule Endstatement is added, which contains the subtypes of statements similarly to Statement, but not followed by EOS. A rule Prog is also added to act as the starting rule of the grammar, which produces 0 or more instances of Statement, followed

by one Endstatement

4.6 Removing ambiguity

4.6.1 Caseitemlist, Pattern

The manual states that "Expression patterns appear only in switch statement case labels.", however the rule for Caseitemlist contains productions which contain Pattern, and then Pattern can produce an Expressionpattern. To remove ambiguity, Caseitemlist was changed to contain Expressionpattern, and the production to Expressionpattern was removed from Pattern. This also meant that now Expressionpattern can only appear in Caseitemlist, as specified in the manual:

> An expression pattern represents the value of an expression. Expression patterns appear only in switch statement case labels[15].

4.6.2 Identifierhead

An Identifier is composed of an initial head character specified under Identifierhead, followed by an (optional) list of Identifiercharacters. One of the possible head characters is "_", meaning "_" is a valid Identifier. However, this causes ambiguity since two possible productions from Pattern are Wildcardpattern (which is the literal "_"), or an Identifierpattern, which is an Identifier. To remove this ambiguity, Identifierhead was modified so that "_" must be followed by one Identifiercharacter.

4.6.3 Floatingpointliteral

The rule Numericliteral produces either an Integerliteral, or a Floatingpointliteral. Integerliterals are lists of one or more decimal or hexadecimal digits, and Floatingpointliterals are the same, optionally followed by fractions and exponents. Since there are no rules in the grammar which need one type of Numericliteral and not the other, the ambiguity can be removed by making the fractions and exponents in Floatingpointliteral non optional.

4.6.4 Arrayliteralbody, Arrayliteralitems, Arrayliteralitem

Array literals contain a list of array items, separated by commas, with the last item optionally followed by another comma. This is ambiguous since if there is a trailing comma, it can be interpreted as either one of the list separators or the optional comma, and this ambiguity is removed by making the body of an array consist of zero or more items and separating commas, followed by one item, followed by the optional comma.

4.6.5 Classmembers

Classmembers contains one or more Classmember. In the manual this is implemented by having a recursive production rule, however this causes ambiguity, and is changed to a Kleene plus production.

4.6.6 Attributes

Attributes consist of one or more Attribute, so in the same way as Classmembers, the rule is changed from being recursive to having a Kleene plus.

4.6.7 Patterninitializerlist

Patterninitializerlist contains one or more Patterninitializers, and is also changed from being recursive to having a Kleene plus, but with comma separators.

4.6.8 Constantdeclaration

Constantdeclaration has multiple optional productions. To remove ambiguity, Constantdeclaration was given a set of rules which implement every possible option from its productions, which entirely replaces the optional productions.

4.6.9 Literal

Interpolatedstringliteral defines string literals which possibly contain an interpolated value, however the interpolated value is not mandatory, which means a normal Stringliteral can also be parsed as an Interpolatedstringliteral. This causes ambiguity in Literal, since a Literal can be a Stringliteral or an Interpolatedstringliteral. This was removed by removing the production with Stringliteral.

Chapter 5

Observations on the manual

While developing the pipeline for extracting the grammar, and also while using the grammar to parse Swift code, some observations were made about the manual, many of which helped guide the improvements that were made to the grammar.

5.1 Natural language rules

As have been previously discussed, a large number of rules were defined in natural language, and had to be implemented programmatically. In some cases, this does make sense from a user-oriented point of view. For example, the rule "Any Unicode scalar value except /* or */", for text within multiline comments, is very clear in what it defines for a user reading the manual. However, most rules defined in natural language are far less clear. Many of them define rules governing sets or ranges of Unicode characters, but the definitions only reference characters through their Unicode values. An example of this is one of the rules governing what characters can make up the head of an identifier: "U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, or U+00B7–U+00BA". Unlike the previous rule for text within multiline characters, it is unclear which characters are actually specified within this rule, and this would require the user to look up each Unicode definition within the rule to follow it. This added difficulty for the user somewhat defeats the purpose of having rules defined in natural language, which otherwise are easier to understand.

5.2 Incorrect specifications

Some specifications for rules in the manual are incorrect, to the extent that they specify constraints which contradict how Swift code is actually structured, and directly had to be changed in order for Swift code to be parsed. This is obviously a failing in the manuals ability to act as a usable reference for the language of Swift. Examples of these errors are shown in the previous section, alongside the necessary fixes to correct them.

5.3 Lack of clear starting rule

The highest level syntactic structure defined in the language reference manual is for statements. Since a program is composed of one or more statements, the initial implementation of a start rule was simply a list of statements. However, since statements expect to be terminated by a semicolon or a newline, a more complicated implementation had to be developed, such that a file of Swift is composed of one or more statements separated by semicolon or a newline, with the last statement only having optional termination. This is quite complex, and also deals with an important part of the Swift language, and the manual would therefore benefit if it was included.

5.4 Ambiguity

There are some cases where rules are specified in a way which is ambiguous, but not incorrect. An example of this is with numeric literals, as discussed previously. A numeric literal can be an integer literal, or a floating point literal. An integer literal could also be parsed as a floating point literal, since the parts of a floating point literal which specified it as being non integer, such as fractions, were optional. Making these

non optional removed the ambiguity.

However, this change, if present in the manual from the start, would possibly add confusion for someone using it as a reference for the language. This change means that it is impossible for a floating point value to be a whole number, which might cause uncertainty in the user over which numbers to use in which particular situations, and their correct usage. Therefore, although the specification in the manual is ambiguous, this ambiguity leads to clarity for the user.

Chapter 6

Results

For every Swift file which is parsed using the grammar, there are 3 possible outcomes: the file is parsed without ambiguity, parsed with ambiguity, or throws a parse error at some location in the file. In this section, the results are shown for the 3 sources of Swift files that were used throughout the project: the Fixtures subsection of the Swift package manager[5], the overall package manager, and the parsing testbed of Swift, which is used as part of the internal testing of Swift.

6.1 Fixtures

Result	Number of files
Parsed unambiguously	10
Parsed, with ambiguity	144
Parse error	11

Table 6.1: The results of parsing the Fixtures subsection of the Swift package manager

During the development of the grammar, the package manager[5] developed by Apple in Swift was used as a collection of Swift files for parsing. A subset of the files in the package manager is the Fixtures, which are used to initialise a set of objects used as a baseline for testing. As can be seen in table 6.1, the majority of files in the Fixtures are parsed successfully, but with ambiguity, and a small minority are parsed unambiguously or not parsed.

6.2 Package manager

The overall source code of the package manager contains, alongside the Fixtures subsection as described above, Swift code which implements more complex functionality. This is reflected in the comparatively short file lengths within the Fixtures (an average of 10 lines per file), compared to the overall package manager (an average of 184 lines per file). For the package manager, roughly half of all files are parsed (ambiguously), and half cause parse errors when parsed.

Throughout the process of making improvements to the grammar to remove parse errors, short files either from the Fixtures, or manually written in order to contain the specific functionality on Swift which causes the errors, were preferred. This was due to the fact that, although when a parse error is given when parsing a file containing the precise location of the code which has caused the parse error, the error does not specify which rule in the grammar has caused the parse error. For short files, with some familiarity with the

Result	Number of files
Parsed unambiguously	11
Parsed, with ambiguity	219
Parse error	241

Table 6.2: Results from parsing the full Swift package manager

grammar it is often possible to identify which rule has caused the error, based on the code at that point in the file. However, for longer files which also often have a more complex, nested structure, it is very difficult to manually identify which rule has caused a parse error.

6.3 Swift parsing testbed[19]

As part of Apple's internal testing for Swift, it has compiled a large body of Swift code, designed for testing the compiler of Swift. This consists of 186 files containing Swift code. Each of these files roughly covers one small subsection of the functionality of Swift, for example whitespace, import statements or loops.

Each of these files contains multiple statements which are designed to either succeed or throw a specified error when parsed. Lines which are supposed to give errors are directly followed by a comment, starting with "// expected-error", followed by the type of error the compiler expects.

Although each file in the testbed can be simply parsed as if it was a regular Swift file, i.e as a list of Statements, since the testbed files contain code designed to give errors when parsed, parsing reaches the first error in the file, throws a parse error and then stops. To avoid this, each testbed file is broken down into smaller chunks, each of which is then parsed. Initially, subsections of code starting with a "{" and ending with a matched closing "}" are extracted, since in Swift this usually indicates classes, structs, longer functions and other blocks of code which are defined over multiple lines. Then, once these have been extracted and parsed, the rest of the file is split on a line-by-line basis, and then each line is parsed.

If an extracted block contains a comment starting with "expected-error", then the Swift compiler expects an error to come from this block. Quite often, this does not lead to an error during parsing, since the Swift compiler also expects errors due to higher level, non-grammatical problems such as the usage of undeclared variables and functions. For improving the grammar, this has been most interesting in the cases where an error within the block is not expected, but parsing the block of code still gives a parse error.

As an example, below is an example subsection of one of the files in the parsing testbed:

```
let _ = 1 // expected-error{{global variable declaration does not bind any variables}}

func foo() {
  let _ = 1 // OK
}
```

This will be subdivided into two blocks:

```
let _ = 1 // expected-error{{global variable declaration does not bind any variables}}
```

and

```
func foo() {
  let _ = 1 // OK
}
```

Both these blocks will then be parsed as statements. Also, the first block is expected to throw an error, due to the presence of a comment specifying an error, and the second block is not.

Figure 6.1 shows the results from dividing and parsing all files in the Swift parsing testbed. The majority of blocks are either parsed successfully (and without ambiguity), or give parse errors. Many of the parse errors are due to the imperfection of the method of subdividing files, since other structures in Swift (such as conditional compilation clauses) are also spread over multiple lines, but are not demarcated by opening and closing brackets. Since so many of the extracted blocks of code are very short, if they are parsed successfully they are usually parsed without ambiguity.

6.4 Ambiguity

The grammar is ambiguous, and although fixes have been made to remove ambiguity, there are still many rules which are ambiguous. Since, when parsing, files are first attempted to be parsed unambiguously, and then ambiguously, for each file which is parsed ambiguously it is possible to see the rule in the grammar

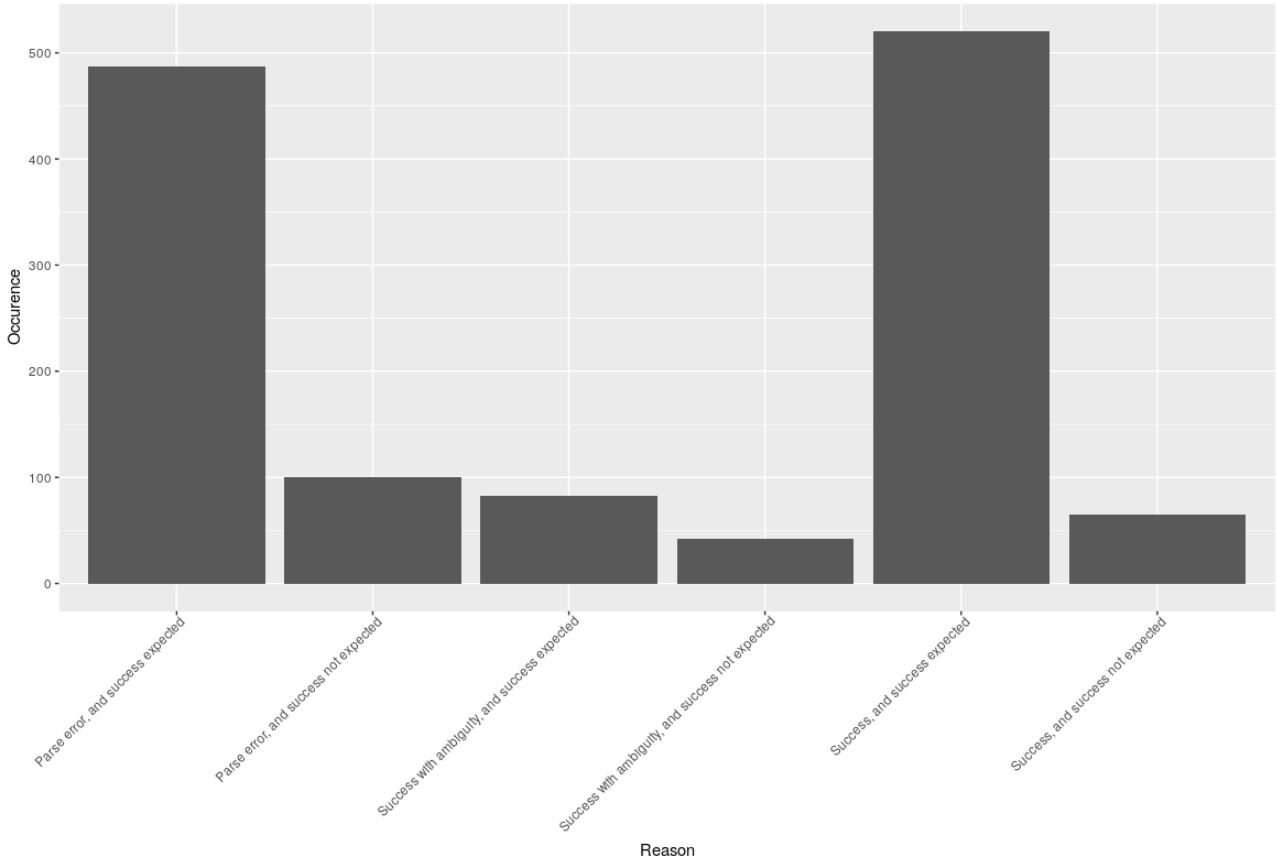

Figure 6.1: Results from parsing the Swift parsing testbed

Result	Number of blocks of code
Parse error, and success expected	487
Success with ambiguity, and success not expected	42
Success, and success expected	520
Parse error, and success not expected	100
Success, and success not expected	65
Success with ambiguity, and success expected	83

Table 6.3: Results from parsing the Swift parsing testbed

which has caused ambiguity. This is useful, since searching for ambiguities in a grammar by hand is a very complex and cumbersome job.[7]

Figure 6.2 shows which rules cause ambiguity when parsing, and how often each rule is a cause of ambiguity. Not every ambiguous rule is present in this chart, since it is possible there are ambiguous rules which are only present in the productions of other ambiguous rules, and therefore aren't reached.

Expression is the most common cause of ambiguity, which is unsurprising since expressions are very commonly used, due to the fact expressions are one of the fundamental structures in a programming language. Expressions, in Swift, are built up from combinations of other expression types. A primary expression is the most basic kind of expression, a postfix expression is a primary expression or a wide range of other, more specific subtypes of expressions, prefix expressions are postfix expressions preceded by specific operators, and a binary expression is a prefix expression preceded by other specific operators. Alongside this general case, each type of expression can also be a wide range of different subtypes of expression, which can also include other kinds of expressions within them. Since there is an overlap between the operators which can precede a postfix and binary expression, then an expression can be parsed as both a binary expression and a prefix expression. Furthermore, both postfix expressions and primary expressions have many production rules to other subtypes of expressions (9 and 12) respectively, which can possibly overlap.

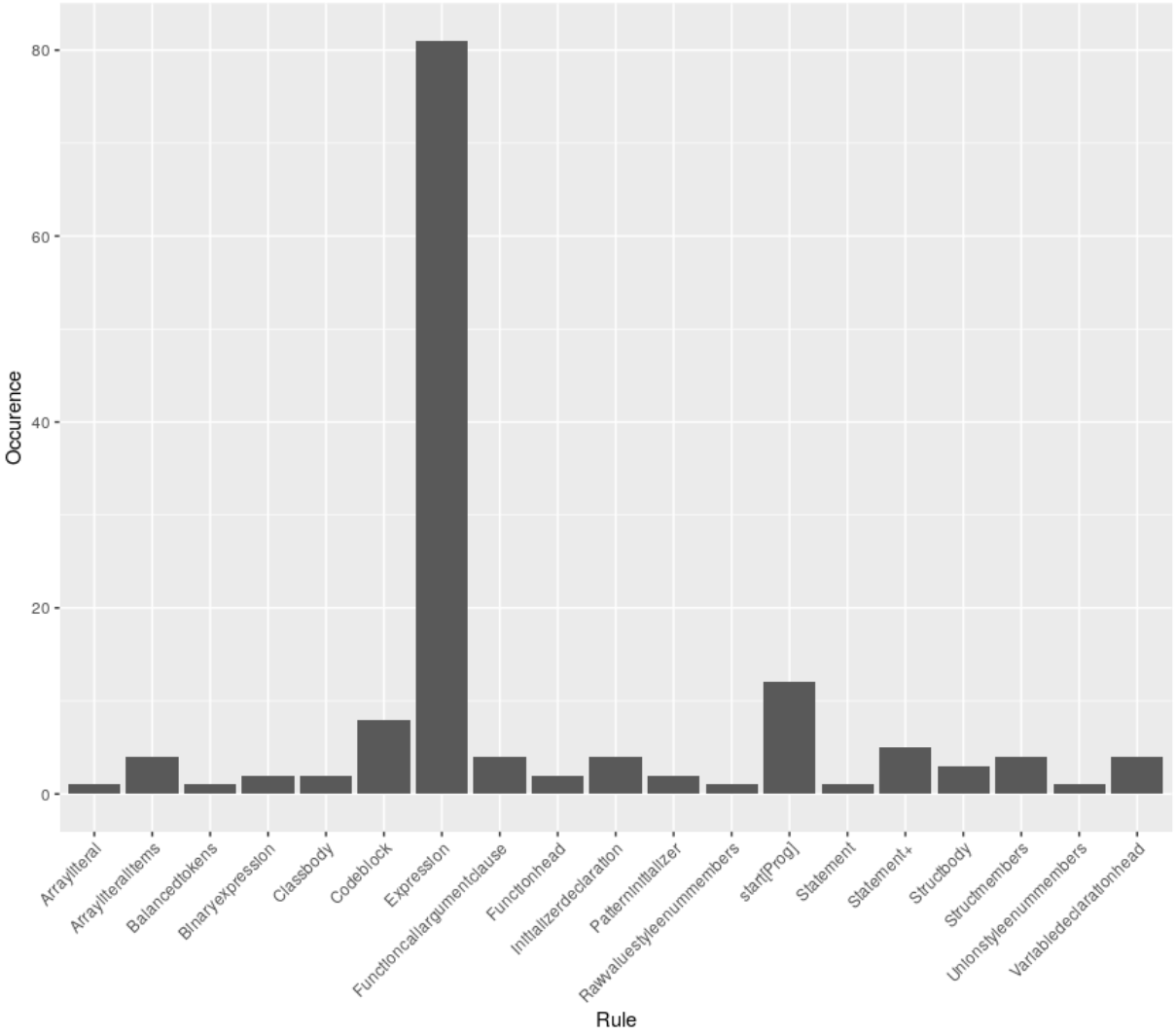

Figure 6.2: Rules causing ambiguity while parsing the fixtures of the package manager, and the number of occurrences

The rules governing primary expressions themselves tend not to cause ambiguity, since primary expressions usually consist of lower level, more fundamental parts of Swift, for example an array of literals (for example [1, 2, 3]), tuples, (for example [a: "1"]), special literals such as #file or even just a wildcard _. Even in combination with some other options for expressions, the expressions remain unambiguous. For example, the expressions:

```
try #file
```

and

```
+ #file
```

are both unambiguous. The first expression is a primary expression, preceded by the optional keyword "try", and the second expression is a prefix expression, consisting of a prefix operator followed by a primary expression. Ambiguity arises for more complex expressions, such as "try + #file", which is interpreted as a prefix expression followed by a binary expression. The ambiguity, in this case, is due to uncertainty over whether the "+" operator belongs to the prefix expression.

It is also worth noting that, although "+ #file" and "try + #file" are both parsed (ambiguously) by the grammar extracted from the manual, they both consist of incorrect Swift code. Both of these expressions are incorrect due to the usage of the "+" operator, which due to operator precedence, is not allowed to be

used as an unary operator.

In Swift, similarly to most other programming languages, there exists precedence for operators. This is acknowledged in the manual, which states:

> At parse time, an expression made up of binary operators is represented as a flat list. This list is transformed into a tree by applying operator precedence. For example, the expression 2 + 3 * 5 is initially understood as a flat list of five items, 2, +, 3, *, and 5. This process transforms it into the tree (2 + (3 * 5)).[3]

However, there is no place within the language reference manual that actually specifies precedence for the existing operators within Swift, so similarly in the grammar in its current form, there exists no precedence for operators. This is possibly also a source for ambiguity within expressions, since, as shown above, expressions often include sub expressions divided by various operators. The precedence for operators is briefly outlined in the language guide section of the manual[21], and is properly defined externally in another source, of Swift documentation[6]. Introducing precedence for operators would also be very difficult without some external source, such as the compiler which specifies precedence for operators, or other sources of Swift documentation.

6.5 Evaluation

Table 6.4 shows all files which are part of the codebase, developed in Rascal, over the course of this project, as well as a brief overview of the purpose of each file.

The file grammar.rsc contains the functionality for the initial extraction of the grammar in the manual. From there, all rules loosely defined in natural language in the grammar are implemented in processLooseRules.rsc, and then the grammar is converted to the Rascal grammar format and corrections are made to it in toRascalGrammar.rsc.

As can be seen, far more code is devoted to making changes to the grammar than extracting it. This is in part due to the fact that, although it took some time to develop the initial functionality for extraction, it is not extremely complex, since the structure of the HTML of the manual page is very regular. As has been discussed in previous sections, every change made to the grammar is made programmatically, so the files which implement these changes must necessarily be large.

analyseSwiftFiles.rsc contains the functionality to visit a folder in the file system, and recursively visit every file and folder within that folder. If the file extension is ".swift", then the file is passed on to parseSwiftCode.rsc. This parses the contents of the file, using the grammar defined in swiftGrammar.rsc, and then passes the results back to analyseSwiftFiles, which keeps track of the results from parsing each file. After the file structure passed to analyseSwiftFiles.rsc is fully visited, the results are exported to JSON. Once a file has been parsed, parseSwiftCode.rsc also can visit the Tree data type in Rascal, which is produced as a result from being able to parse the Swift file. This was extremely useful when making improvements to the grammar, since it meant, that for a Swift file, it was possible to see which rules from the grammar were used to parse that file.

Similar functionality was also developed for processing Swift's own parsing testbed. As described in previous sections, splitParseTestsFiles.rsc subdivides each file in the parsing testbed, then passes these subsections to parseSwiftCode2.rsc. This differs from parseSwiftCode.rsc only in that, instead of being given Swift code as the location of the file containing the code, the Swift code is given as a string. The results from parsing each subsection are passed back to splitParseTestsFiles.rsc, which also, as described previously, keeps track of whether each subsection is expected to be parsed successfully or not.

Other than swiftGrammar.rsc, all files contain functionality which has been manually implemented. swiftGrammar.rsc only contains the grammar for Swift, in the Rascal grammar format, and is completely programmatically generated.

Function	File name	Length of file, in lines
Initial parsing of the HTML of the manual page	grammar.rsc	248
Implementing rules defined in natural language	processLooseRules.rsc	828
Converting to Rascal grammar, and modifying the grammar	toRascalGrammar.rsc	312
The grammar, after all rounds of modifications	swiftGrammar.rsc	2484 lines, 445 rules
Parsing regular Swift files	parseSwiftCode.rsc, analyseFiles.rsc	172, 80
Parsing the Swift parsing testbed	splitParseTestFiles.rsc	218

Table 6.4: All files developed during the course of this project, their purpose and the size of each file

Chapter 7

Discussion

As shown in the results section, the grammar does not perfectly parse Swift code, and indeed for more complex Swift files, the grammar usually cannot finish parsing a file correctly. The grammar is also still highly ambiguous, despite several changes having been made to it in order to remove ambiguity.

The grammar is limited in that it is only recovered from the manual page, meaning the grammar probably bears less relation to the language than a grammar based on the Swift compiler, or even better, a grammar extracted from a range of artefacts and then converged. Grammars purely extracted from manuals are known to have issues:

> a reference manual grammar is normally not suited to be used for parser generation: it is usually ambiguous[11]

and as has been shown in sections regarding changes made to the grammar, some specifications in the manual are wrong in that they contradict the structure of Swift code, and had to be changed. If the grammar was extracted from multiple sources, then even if one of the sources for the grammar is partially incorrect, it is likely that the other sources will be correct in that area. Instead, the process used in refining this grammar was more based on identifying an issue in the grammar, and then trying various modifications to the grammar, until a modification was found which seemed to fix the problem. It is possible that some of the modifications that have been made to the grammar have unknowingly caused further errors with other rules in the grammar, which has not been noticed. There have been cases, when the current changes were being developed, that I have had to revert previous changes, since the fixes to one rule had caused the capacity for unexpected ambiguity or parse errors in other rules.

Although the process of extracting and improving the grammar has raised insights in the structure of Swift and the positives and negatives of how the grammar is presented in the manual, the grammar is only based on Swift, and therefore provides limited insight into the general area of how software languages are specified in manuals.

The complete grammar extraction pipeline is automated, allowing the mutations made to the grammar at every stage to be evident. However, this means that most changes made to the summary page of the manual would cause errors, either by changing the references to rules, or introducing new rules, which will cause the further changes to the grammar made after the initial extraction to be directed towards rules which don't exist. Furthermore, although pattern matching is used in the extraction, changes made to the structure of the file would possibly cause the entire extraction process to stop.

Chapter 8

Related work

Reference manuals and specifications of languages are commonly used artefacts for extracting a grammar from. The reference manuals of C#[22], various Java specifications[13], COBOL[16] and others[1] have had grammars extracted from them. There does currently exist a grammar for Swift implemented in ANTLR[2], however that is based upon the manual, the compiler, and actual behaviour, instead of being purely extracted from the manual.

After all the natural language constraints from the manual have been implemented, the grammar will then exists as a Rascal grammar[17] structure which represents the grammar as defined in the manual, which is structurally equivalent apart from additional rules added to implement rules defined in natural language.

The entire process of extracting the grammar from the manual, before it is refined based on feedback from parsing Swift files, shares the same "core ingredients"[12] as the initial stages of converging the grammars extracted from multiple artefacts:

1. A unified grammar format that effectively supports abstraction from specialities or idiosyncrasies of the grammars as they occur in software artefacts in practice.
2. A grammar extractor for each kind of artefact – e.g., a Java extractor maps Java classes to the unified grammar format[12].

The various stages discussed in this paper over which the grammar is recovered match the levels of grammar recovery described in *Semi-automatic Grammar Recovery*[11] In the case of this project, although the grammar is temporarily stored in an intermediate data structure after it is parsed from the manual, the grammar from the manual is properly at level 1, once it has been converted to the Rascal grammar structure, as defined in section 4.3. From there, in section 4.4 the connectivity of the grammar is modified by adding a start rule for the grammar to increase connectivity. At this point, every rule defined in the lexical structure subsection of the manual will also have been implemented as lexical rules, either programmatically in section 4.3 or manually in 4.2, with a few constraints which can only be implemented in the Rascal grammar structure implemented in 4.4. This means, at this stage, the grammar mostly matches the conditions for being level 3: "no bottom sorts and only sensible top sorts", all lexical productions will have been recovered or implemented, and a starting rule will have been defined. In the rest of section 4.4, the grammar is further refined by parsing Swift code and then modifying the grammar in response to the results from parsing, either to remove the sources of parse errors, or to remove ambiguity in the grammar.

Chapter 9

Conclusion

During this project, the overall process of developing the tools for extracting a grammar for Swift has been relatively straightforward, and the resulting raw grammar that has been extracted clearly represents the language of Swift as represented in the language reference manual. Due to how the parsing pipeline has been developed, it is also clear at each stage how sections of the grammar are extracted, and how they directly map to the same representation of the grammar in the Rascal grammar format.

From this raw form, the grammar is used to parse Swift code, and based on the results from this parsing, changes are made to the grammar to fix issues which the results have made evident. Results which are insightful for this have been the rules from the grammar which are used to parse specific sections of Swift code, whether Swift code is fully parsed or not, and, if the code is not fully parsed, where the parse errors exist. Every change made to the grammar to perform these fixes is done programmatically.

This process of adjusting the grammar has, on the whole, been less clear than the process of developing the parsing pipeline, since there was a greater aspect of testing and adjusting based on personal judgement about which fixes needed to be made to the grammar, instead of having a clear target to aim towards, which was the case with developing the parsing pipeline. However, the changes made do improve the grammar, and for each change the reasoning and outcome of the changes made is described.

These results show that although with these changes the grammar has improved from its raw state in terms of improving how effectively it can parse Swift code, it still cannot consistently parse Swift code. The results of parsing various sections of Swift code are shown in the results section of this paper, along with discussion on the results, and why they vary depending on the structure of the Swift code which is being parsed.

For future work, further rounds of improvement could be undertaken for the grammar, through the current process of parsing Swift code with the grammar, identifying the sources of errors or ambiguity, and applying modifications to change the rules responsible. It is possible, to some degree, to automatically detect ambiguity in a grammar[7], by investigating the relationships between rules. In this case, detecting and removing these ambiguous relationships in the grammar, and then checking the result, could possibly remove a large amount of ambiguity.

It would be possible to use the Swift grammar to converge with other sources of Swift grammar. Since Swift is open source, its compiler[8] is available for grammar extraction. Although the compiler is less geared towards readability, and therefore would be harder to extract a grammar from, the grammar would hopefully be very close to absolutely correct in its representation of Swift. Also, as has been briefly mentioned throughout this paper, there are other manuals for Swift which have been produced by Apple, such as the language guide[21], or Apple's own external documentation on Swift[6]. Both of these, and possibly others almost certainly contain information which is not present in the language reference manual, such as on operator precedence, and would be very useful sources of improvements to the grammar.

Convergence could be done either with the grammar after the modifications have been applied to it, or the grammar purely as specified in the manual. Although the modifications made do improve the grammar, in that the grammar causes fewer parse errors and less ambiguity when parsing, the changes made have purely been based on the outcome of parsing Swift code. It is possible that these modifications diverge the grammar further away from how the grammar is defined in other artefacts, and would complicate the process of convergence. In that case, using the grammar purely as defined in the manual, without the modifications

would be more useful.

Overall, although the structure of Swift is clearly specified in the manual in such a way that it can easily be extracted to create a grammar for parsing Swift, many mutations then have to be applied to the grammar in order that it can parse Swift code, and even then, the ability of the grammar to parse Swift code is inconsistent. However, the process of testing and then modifying the grammar has given insight into the flaws and design choices of the Swift manual, and the grammar in its current state can be used for either future rounds of improvement, or convergence with other grammars.

Bibliography

[1] *A zoo containing the grammars of many different languages.* URL: https://slebok.github.io/zoo/.

[2] *Antlr Swift grammar.* URL: https://github.com/antlr/grammars-v4/tree/master/swift/swift3md.

[3] Apple. *Swift language reference manual.* URL: https://docs.swift.org/swift-book/ReferenceManual/zzSummaryOfTheGrammar.html.

[4] Apple. *Swift language reference manual lexical structure section.* URL: https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html.

[5] Apple. *Swift package manager.* URL: https://github.com/apple/swift-package-manager.

[6] *Apple's external documentation for Swift.* URL: https://developer.apple.com/documentation/swift/swift_standard_library/operator_declarations.

[7] Bas Basten. "Ambiguity Detection for Programming Language Grammars". In: *Computation and Language* ().

[8] *Documentation and source code for the compiler of Swift.* URL: https://swift.org/swift-compiler/#compiler-architecture.

[9] *HTML parsing functionality for Rascal.* URL: https://github.com/usethesource/rascal/tree/master/src/org/rascalmpl/library/lang/html.

[10] Paul Klint, Ralf Lämmel, and Chris Verhoef. "Towards an Engineering Discipline for GRAMMARWARE". In: *ACM Trans. Softw. Eng. Methodol.* 14 (July 2005), pp. 331–380. DOI: 10.1145/1072997.1073000.

[11] R. Lämmel and C. Verhoef. "Semi-automatic grammar recovery". In: *Software: Practice and Experience* 31.15 (2001), p. 1395. DOI: 10.1002/spe.423.

[12] Ralf Lämmel and Vadim Zaytsev. "An Introduction to Grammar Convergence". In: *Proceedings of the Seventh International Conference on Integrated Formal Methods (iFM 2009)*. Ed. by Michael Leuschel and Heike Wehrheim. Vol. 5423. LNCS. Springer, Feb. 2009, pp. 246–260. DOI: 10.1007/978-3-642-00255-7_17.

[13] Ralf Lämmel and Vadim Zaytsev. "Recovering Grammar Relationships for the Java Language Specification". In: *Software Quality Journal (SQJ); Section on Source Code Analysis and Manipulation* 19.2 (Mar. 2011), pp. 333–378. ISSN: 0963-9314. DOI: 10.1007/s11219-010-9116-5.

[14] *Layout.* URL: http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Declarations/SyntaxDefinition/SyntaxDefinition.html.

[15] *Pattern.* URL: https://docs.swift.org/swift-book/ReferenceManual/Patterns.html.

[16] Chris Verhoef Ralf Lämmel. *Cobol grammar.* URL: https://www.cs.vu.nl/grammarware/browsable/cobol/.

[17] *Rascal Grammar.* URL: https://github.com/usethesource/rascal/tree/master/src/org/rascalmpl/library/Grammar.rsc.

[18] *Rascal Node.* URL: http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Expressions/Values/Node/Node.html.

[19] *Swift parsing testbed.* URL: https://github.com/apple/swift/tree/main/test/Parse.

[20] *Symbol.* URL: http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Declarations/SyntaxDefinition/Symbol/Symbol.html.

[21] *The Swift language guide.* URL: https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html.

[22] Vadim Zaytsev. "Correct C♯ Grammar too Sharp for ISO". In: *Participants Workshop, Part II of the Pre-proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*. Extended abstract. Braga, Portugal: Technical Report, TR-CCTC/DI-36, Universidade do Minho, July 2005, pp. 154–155.