The sbt-rats Parser Generator Plugin for Scala (Tool Paper)

Anthony M. Sloane Franck Cassez Scott Buckley

Programming Languages and Verification Group
Department of Computing
Macquarie University, Australia
Anthony.Sloane@mq.edu.au, Franck.Cassez@mq.edu.au, Scott.Buckley@mq.edu.au

Abstract

Tools for creating parsers are a key part of a mature language eco-system. Scala has traditionally relied on combinator libraries for defining parsers but being libraries they come with fundamental implementation limitations. An alternative is to use a Java-based parser generator such as ANTLR or Rats! but these tools are quite verbose and not ideal to use with Scala code. We describe our experiences with Scala-focused parser generation that is embodied in our sbt-rats plugin for the Scala Build Tool. At its simplest, sbt-rats provides a bridge to the Rats! parser generator for Java. On top of this bridge, we have a simple grammar definition notation that incorporates annotations for tree construction and pretty-printing. As well as generating a Rats! grammar, sbt-rats can optionally generate case class definitions for the tree structure and a pretty-printer defined using our Kiama language processing library. We explain the sbt-rats grammar notation and describe our positive experiences using it to define grammars for LLVM assembly notation and the SMTLIB input/output language for SMT solvers.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Parsing

Keywords Parsing expression grammars, Scala build tool.

1. Introduction

Scala has traditionally relied on a parser combinator library that directly encodes grammars and semantic actions.[1] The standard library is convenient but builds inefficient parsers that essentially interpret the grammar. The libraries fastparse[2] and parboiled[3] gain performance by using more optimised implementations and macros, respectively.

sbt-rats is a Scala Build Tool (sbt) plugin that brings parser generation to Scala. Instead of analysing and translating individual parser definitions, a generation approach can apply more holistic optimisation techniques that rely on analysing the whole grammar. Code and documentation can be found at the project site.[4]

sbt-rats is based on the Rats! parser generator [4]. Rats! transforms parsing expression grammars [3] into Java implementations of packrat parsers [2]. Parsing expression grammars work at a character level so they incorporate both lexical and phrase structure analysis rather than defining these components separately.

At it simplest, sbt-rats allows Rats! grammars to be included in sbt projects. It takes care of automatically running Rats! and including the generated Java in the build.

Unfortunately, Rats! grammars are verbose and Rats! does not have built-in support for common tasks: tree representation and pretty-printing. In this paper we focus on a new high-level syntax formalism that incorporates all of these aspects in a single definition. These syntax definitions are much simpler than equivalent Rats! grammars since details such as semantic action code are not necessary.

sbt-rats translates a syntax definition into a Rats! parser specification which is further translated by Rats! into a Java implementation. sbt-rats optionally applies some post-processing to make the Java implementation more useful from Scala, such as replacing the Rats! implementation of lists with standard Scala ones. sbt-rats can generate Scala implementations of the tree representation and pretty-printer.

Figure 1 illustrates a typical use of an sbt-rats syntax definition and the components that are generated from it. We've been using sbt-rats in the implementation of a verification tool for C programs via LLVM IR code. LLVM IR assembly code is parsed, represented and pretty-printed by components generated by sbt-rats. Using sbt-rats achieves con-

[1] https://github.com/scala/scala-parser-combinators

[2] http://www.lihaoyi.com/fastparse

[3] https://github.com/sirthias/parboiled

[4] https://bitbucket.org/inkytonik/sbt-rats

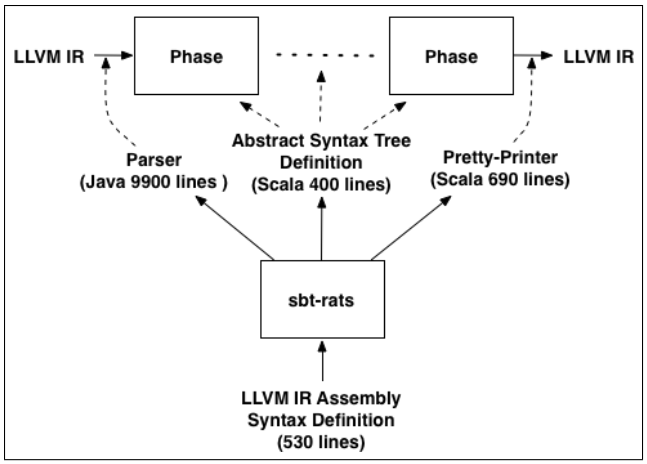

Figure 1. sbt-rats for processing LLVM IR assembly code.

siderable economy of expression since roughly 500 lines[5] of syntax definition generate over 10,000 lines of Java and Scala code. The generated Rats! grammar is more than 1500 lines long. Our LLVM parser is not precisely equivalent to the assembly parser in the LLVM code base but, for comparison, the latter consists of more than 5000 lines of C++ code in the lexical analyser and parser.

2. JSON Grammar Example

Figure 2 shows an sbt-rats syntax definition for JSON that illustrates most of the available notations. Figure 3 shows the syntax tree classes that are used by the generated parser to represent JSON values.

We assume the reader is familiar with extended BNF-style grammars and regular expression operators. Literals in an sbt-rats syntax definition can be delimited with either single or double quotes. This distinction is not meaningful for parsing but affects pretty-printing (Section 3).

sbt-rats uses lookahead operators that are inherited from Rats!. The `!` operator specifies a negative lookahead, consuming no input if its argument does not match. (The `&` operator specifies a positive lookahead.) For example, in the definition of `StrChars` (line 19 of Figure 2) negative lookahead is used with a wildcard operator `_` to match one or more characters that are not a double quote. This definition is similar to the regular expression `[^"]+` but in general the lookahead operator is more powerful since its argument can be an arbitrary expression. These operators are also used by sbt-rats to define the end of file symbol `EOF` to simply be `!_`.

sbt-rats also adds operators to specify separated lists. The binary `**` operator in the definitions of `JArray` and `JObject` (lines 13 and 14) specifies possibly empty separated sequences. (There is also the `++` operator for non-empty separated sequences.) E.g., a `JArray` is a possibly empty comma-separated sequence of values inside square brackets (line 11).

sbt-rats distinguishes between grammar symbols that produce a syntax tree node, those that produce a string and void symbols that produce no value. Apart from the type of value produced, the main difference between these kinds of symbol is the handling of spacing.

```
1  JSON = Spacing JValue EOF.
2
3  JValue = 'true'     {JTrue}
4         | 'false'    {JFalse}
5         | 'null'     {JNull}
6         | JObject
7         | JArray
8         | StringLit  {JStr}
9         | Number.
10
11 JArray : JValue = '[' JValue ** ',' ']'.
12 JObject : JValue = '{' JPair ** ',' '}'.
13
14 JPair = StringLit ":" JValue.
15
16 StringLit : Token =
17     '"' (StrChars / Escape)* '"'.
18
19 StrChars : String = (!'"' _)+.
20
21 Escape : String =
22     '\\' [\"/\\bfnrt]
23   | '\\u' Hexit Hexit Hexit Hexit.
24
25 Hexit : String = [0-9a-fA-F].
26
27 Number : JValue =
28     NumberToken {
29       JNumber,
30       1: Double.parseDouble : Double
31     }.
32
33 NumberToken : Token =
34     [+\-]? Integral Fractional? Exponent?.
35
36 Integral   : String = '0' | [1-9] Digits?.
37 Fractional : String = '.' Digits.
38 Exponent   : String = [eE] [+\-]? Digits.
39 Digits     : String = [0-9]+.
40
41 Spacing : Void = Space*.
```

Figure 2. sbt-rat syntax definition for JSON.

[5] Reported line counts include only non-commented, non-blank lines.

```
sealed abstract class ASTNode extends Product
case class JSON(jValue : JValue) extends ASTNode
sealed abstract class JValue extends ASTNode
case class JTrue() extends JValue
case class JFalse() extends JValue
case class JNull() extends JValue
case class JObject(optJPairs : Vector[JPair]) extends JValue
case class JArray(optJValues : Vector[JValue]) extends JValue
case class JStr(stringLit : String) extends JValue
case class JNumber(numberToken : Double) extends JValue
case class JPair (stringLit : String, jValue : JValue) extends ASTNode
```

Figure 3. Classes for tree representation of JSON.

Tree-valued symbols implicitly allow arbitrary spacing between their constituents. In the example, `JSON`, `JValue`, `JArray`, `JObject`, `JPAir` and `Number` are tree symbols.

Since the `JSON` symbol is defined by a single alternative, its tree nodes will be built using a `JSON` constructor. `JValue` has many alternatives (lines 3–9) so `JValue` will be an abstract class with sub-classes for each alternative. The constructor name is given in braces after the alternative. For example, a `true` literal is represented by the `JTrue` sub-class of `JValue`. The definitions of `JArray`, `JObject` and `Number` specify their type to be `JValue` (lines 11, 12 and 27) so these constructors will also be sub-classes.

`Number` illustrates a more complex tree symbol definition (lines 27–31). The constructor is `JNumber`. The rest of the definition specifies processing that is to be performed on the recognised input text. In this case the first argument of the `JNumber` constructor is to be produced by processing the input using `parseDouble` to obtain a `Double` value.

String symbols do not implicitly allow spacing within their constituents. Regular expression notation can be used to define string symbols (lines 22, 25, 34, 36–39). A token symbol is simply a string symbol that is immediately followed by spacing (lines 16 and 33).

Void symbols are defined as for string symbols but their values are discarded. E.g., `Spacing` is defined to be zero or more spaces (line 41). The definition of `Space` is provided by default by sbt-rats, including tabs and similar characters that play the same role as actual space characters.

Comments can be defined using the full notation. If comments are a spacing alternative they can appear between any pair of symbols. For example, the following rules specify spacing similar to Java but include nesting of multi-line comments. (The symbol `EOL` matches end of line.)

```
Spacing   : Void = (Space / Comment)*.
Comment   : Void = SLComment / MLComment.
SLComment : Void = "//" (!EOL _)* (EOL / EOF).
MLComment : Void = "/*"
                      (MLComment / !"*/" _)*
                   "*/".
```

An option controls whether sbt-rats generates syntax tree classes or not. Figure 3 shows the classes that are generated by sbt-rats for the JSON example. If the default definitions are not sufficient, perhaps because methods need to be included, a developer can turn the option off and add definitions of classes with the same names to the project.

3. Pretty-Printing

In applications of tree-structured data it is common to output that data as text. This might be for debugging or maybe as user-level output. sbt-rats can generate pretty-printers from syntax definitions. The pretty-printers are implemented using functional pretty-printing combinators that are part of our Kiama language processing library [5, 6].

sbt-rats has a convention-based pretty-printing approach. Most of the information about how to pretty-print is determined automatically from the syntax definition. E.g., lexical symbols pretty-print as the text that they matched. Literals `'foo'` pretty-print as themselves. A sequence of tree symbols pretty-prints as a sequence of each symbol pretty-printed separately.

These conventions are enough to get a long way toward acceptable pretty-printed output. sbt-rats augments the syntax definition with formatting directives to get the rest of the way. (A similar approach was developed independently for object grammars in the Ensō project [7].)

The directives `sp` and `\n` can be used to show where an extra space or possible line break should be inserted. (A double-quoted literal `"bar"` is short-hand for `'bar' sp` which avoids many formatting directives in practice.) The directive `nest(e)` specifies that the pretty-printed form of `e` should be indented one level more than its surroundings. The line width which determines where line breaks have to appear can be configured by a run-time option. Similarly, the size of an indentation level can be configured.

Using formatting directives we can specify that JSON array values should print with each value or bracket on a separate line and with the values indented with respect to the brackets.

```
[
    {
        "name": "Joe Bloggs",
        "age": 27.4,
        "keywords": [
            "cycling",
            "jazz"
        ]
    }
]
```

Figure 4. A pretty-printed JSON value.

```
'[' nest((\n JValue) ** ',') \n ']'
```

An analogous rule suffices to pretty-print objects. It is enough to replace the right-hand sides of the two rules on lines 11 and 12 of Figure 2 with the new versions and leave the other definitions unchanged. Figure 4 shows an example of JSON output from this generated pretty-printer.

4. Application: MQ-SCALA-SMTLIB

We have been using sbt-rats to aid in the interaction between our verification tool and SMT solvers via the SMTLIB standard [1].[6] sbt-rats syntax definitions describe the form of input/output offered by SMTLIB compliant solvers. The communication between the solvers and the tool sends pretty-printed SMTLIB terms and parses the responses.

A similar project SCALA-SMTLIB[7] used a hand-coding approach. The SCALA-SMTLIB code base includes lexer, parser and pretty-printer components that are 378, 1221 and 1073 lines, respectively, for a total of over 2500 lines.

In our implementation, MQ-SCALA-SMTLIB, we replaced the earlier hand-coded lexical analysis, parsing, and tree construction components with code generated by sbt-rats. Our sbt-rats syntax definition in MQ-SCALA-SMTLIB has 468 lines, with much simpler and more maintainable definitions than the equivalent hand-coded counterparts in SCALA-SMTLIB. The automatically generated Java parser has 11017 lines, the pretty-printer 372 lines and the syntax-tree classes 250 lines.

5. Other Features

The earlier example did not use some features of sbt-rats which we briefly mention here.

Rats! calculates positions for parsed structures. If Kiama and Scala trees are being used, sbt-rats adds these positions to Kiama's position store from which subsequent processing can access them.

Block symbols allow data to be parsed whose length is determined by another symbol. We can use this feature to write a syntax definition for PNG image file chunks where the length of the `Data` field is the value of the `Length` field.

```
Chunk = Length Type Data[1] CRC
  {1: strToInt : Int}.
```

The generated Rats! semantic action for this rule calls out to a method that reads the bytes to build the `Data` field.

Rats! has limited support for left recursion. sbt-rats supports precedence and associativity annotations. For example, in a rule for expressions (`Exp`), the following specifies a left recursive addition operator at precedence level 2.

```
Exp "+" Exp     {Add, left, 2}
```

The generated Rats! rule iterates the tail (i.e., `"+" Exp`) to build a right recursive action structure and executes those actions to build the left recursive structure. The annotations are also used to restore precedence and associativity in pretty-printing, and to eliminate unnecessary parentheses.

6. Conclusion

Our experience so far shows that sbt-rats provides considerable help when building tools that process tree-structured data. The new syntax definition notation is much less verbose than the Rats! grammar notation to which it is translated. Performance seems to be good compared to the library approach, but we have not conducted detailed benchmarking. Future plans include supporting the modularity features of Rats!, retaining `Void` symbols in parsed structures so that things like comments can be preserved, and generating grammar-based string interpolators.

References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[2] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ACM SIGPLAN Notices*, volume 37, pages 36–47. ACM, 2002.

[3] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[4] Robert Grimm. Better extensibility through modular syntax. In *ACM SIGPLAN Notices*, volume 41, pages 38–51. ACM, 2006.

[5] Anthony M. Sloane. Lightweight Language Processing in Kiama. In *Generative and Transformational Techniques in Software Engineering III*, number 6491 in Lecture Notes in Computer Science, pages 408–425. Springer Berlin Heidelberg, January 2011.

[6] Anthony M. Sloane and Matthew Roberts. Oberon-0 in Kiama. *Science of Computer Programming*, 114:20–32, December 2015.

[7] Tijs van Der Storm, William R Cook, and Alex Loh. Object grammars. In *International Conference on Software Language Engineering*, pages 4–23. Springer, 2012.

[6] http://smtlib.cs.uiowa.edu

[7] https://github.com/regb/scala-smtlib