A combinator-based parser generator for Scala

Ivan Motyashov
Department of Computer Science
University of California, San Diego
imotyash@eng.ucsd.edu

December 6, 2013

Adviser: Ranjit Jhala

Abstract

This project addresses the problem of writing parsers in the Scala language, specifically using a technique known as "combinatory parsing". We give an overview of Scala's standard parser combinators library, its perceived shortcomings, and our solution, which consists of a rewrite of the said library and a standalone parser generator that accepts an abstract description of a language and generates a combinator-based parser implementation in Scala.

1 Introduction

1.1 Background

In computer science, "parsing" refers to the act of matching a string of characters against a description of a language and deducing whether the given string is in the language or not, possibly transforming the string into a different representation in the process. Some would argue that this is a solved problem [1] That is debatable. However, *parsing tools* targeting a given programming language are often far from perfect.

How frequently are parsers needed in practice? Besides being necessary for specialized tools like compilers and compiler suites for well-established general-purpose programming languages, parsers are commonly found in implementations of Domain-specific languages (DSLs). DSLs are languages designed to solve a particular set of problems and are, as such, limited in scope (and usually in size). Unix applications, for example, have a long-running tradition of "mini-languages" (e.g. shell scripting, sed, awk, emacs, and even parser generators, like yacc). There even exists an approach to software engineering and development that essentially decomposes the task of solving a problem using software into the subtasks of creating a DSL for that problem, and solving the problem in it, known as "Language Oriented Programming" [2].

Certain kinds of languages, e.g. regular, context-free, and even context-dependent languages can be described by means of a formal grammar (e.g. in an Extended Backus-Naur Form or EBNF), and thus yield themselves to parser generation. There exists a multitude of parsing techniques that are used in parser generators. The most familiar of these are the table-based LR, LL, and LALR, commonly employed in generators that target imperative languages, like yacc/bison (LALR, GLR), or ANTLR (LL).

Functional languages (or, generally, languages which have first-class functions) allow for another technique, known as "combinatory parsing". In this context, a *parser* is defined as a parameterized function that accepts a representation of the input (e.g. a string), and returns a result (e.g. a parse tree) and the remainder of the input. A parser combinator is a higher-order function that accepts one or more parsers and transforms

[1] See `http://tratt.net/laurie/blog/entries/parsing_the_solved_problem_that_isnt` for a dissenting opinion, which I find persuasive.

[2] `http://martinfowler.com/articles/languageWorkbench.html`

them into a single parser, which typically has a different behavior. For example, a sequence combinator takes in a list of parsers and applies them to the input sequentially, combining the results. Combinatory parsing facilitates the construction of human-readable, modular, and thus maintanable recusive-descent parsers for a large set of formal languages (not limited to context-free languages, in fact). Many functional languages provide an implementation of this technique; for example, *parsec*[2].

Consider Scala: a modern, purely object-oriented, multi-paradigm language, which includes features like first-class functions, a flexible syntax (enabling, for example, infix notation for method calls), and algebraic data types (realized via inheritance and so-called "case classes"). [1] Scala has an extensive standard library, which includes an internal DSL for combinatory parsing. In the remainder of this paper, we shall explore some of the perceived shortcomings of this library in practical applications, and propose a solution in the form of a revised version of the Scala parser combinator library and a parser generator, which transforms an EBNF grammar into a combinatory parser in Scala.

To avoid confusion, we shall henceforth use "parser" to refer to a generic function that takes an input and returns a parse result and the remainder of the input (i.e. the parser combinator sense), and "Parser" to refer to a program or module that parses an entire language.

Throughout the rest of this paper, we will use as an example a small, artificial language, called $\mathfrak{I}$, for which we shall construct several Parsers. Consult Appendix B for its formal grammar.

1.2 Motivation

Started as an academic project in 2001, Scala has since grown to be a powerful, scalable (hence the name), and performant general-purpose programming language, increasingly used in the industry, by companies like Twitter [3]. Given the prevalence of DSLs and the consequent demand for parsing tools, a simple and usable parser generator would be an important asset to the language. Since Scala runs on the JVM and boasts interoperability with Java as one of its selling features, it naturally has access to the mature tools available on that platform. These tools, e.g. ANTLR [7], or Parboiled [8], use either an external (ANTLR) or internal (Parboiled) DSL to describe a language (usually limited to context-free languages or a subset thereof) and generate code that parses it. The resulting code is often virtually unreadable (and hence unmaintanable) and, in the case of tools inherited from Java, not written in the target language (Scala). Parser combinators offer a viable alternative: parsers that are modular, readable, and more easily maintanable due to their design — a very simple core, consisting of basic primitives and rules for their composition. However, this approach, as implemented in the Scala standard library, is not without its own drawbacks. For example:

- Error handling – a very essential function of Parsers, increasingly important for complex languages – is very poorly realized. There is no error recovery (and thus the Parser can only report one syntax error per run), and error reporting suffers from imprecision and incompleteness. Consider, for example, the following $\mathfrak{I}$ sentence:

  ```
  if(5) { foo } else { 10 }
  ```

  This snippet contains a syntax error: a meaningless token "foo" is found in place of an expression. A Parser hand-written using the provided library may yield the following error on this input:

  ```
  [1.9] error: '(' expected but 'f' found

  if(5) { foo } else { 10 }
          ^
  ```

  Not only does the error reporting mechanism offer no context for the error (what was the Parser *trying* to parse?) but the list of "expected" tokens is incomplete (in this case, one could also reasonably expect a number, for example), and thus more confusing than illuminating.

[3] http://blog.redfin.com/devblog/2010/05/how_and_why_twitter_uses_scala.html

- Some other useful features, like Parser debugging (in the form of tracing the execution of its various components, for example), are either missing or implemented in a non-user friendly way.
- The parser combinators library suffers from its design as an *internal* DSL. Although it intends to make the Parser code visually resemble a BNF representation of the underlying grammar (e.g. by using the '|' symbol for the combinator that selects amongst alternatives), this conceals the fact that this code is not actually a grammar. There is a certain "locality" property — the code is merely a collection of "parser" functions, each unaware of each other's existence and not expressing the kind of information that is contained in a grammar (e.g. conceptual separation of productions and actions). It makes it impossible to automatically reason about the grammar as a whole: does it make use of left recursion? indirect left recursion? Transformations and optimizations are likewise impossible. A notable example of where this limitation becomes crippling is error recovery; even the naive approach taken in this project requires the computation of the FOLLOW set for each nonterminal, which requires analyzing the entire grammar *qua* grammar.
- Even without error recovery (which, as we will see, introduces extra code to each production that uses it), there is quite a bit of boilerplate [4], which tends to make the parser code tedious to write and harder to read, somewhat taking away from this advantage offered by parser combinators (see Appendix D for Parsers hand-written and generated for $\mathfrak{J}$).

This project attempts to improve the usability of parser combinators in Scala via a set of tools that provide better error handling and minimize the need for boilerplate code.

2 Solution

2.1 Overview

Our solution to the stated problem is two-fold: an internal *and* an external DSL. Firstly, we propose a redesign and an extension of the original parser combinators library, to provide the missing functionality and improve error handling. Secondly, we propose an external DSL to describe the language to be parsed (via a formal grammar in EBNF notation) and auxiliary data structures (i.e. the description of the AST), understood by a compiler, which produces Scala code to parse the given language using our extended library. Using an external DSL allows us to reason about the grammar in ways forbidden by the nature of an internal DSL. By compiling into Scala, we retain the benefits of combinatory parsing (modularity, readability, etc.), while relieving the user of the need to write repetitive boilerplate code, which can be automatically generated from the grammar.

2.2 Design of the parser combinators library

The original Scala parser combinators library is designed as follows: It is a *trait*, which is mixed in with the module or class that implements the actual Parser. This trait defines a *ParseResult* data structure, which represents the result of a parse, and can be one of:

- **Success**, which representes a successful parse and encapsulates a *value*, and the remainder of the *input*.
- **Failure**, which represents a failed parse and triggers a backtracking whenever possible.
- **Error**, which represents a failed parse that is known to be a syntax error and is propagated straight to the root, without backtracking.

The trait also defines a set of basic *parsers* and means for their generation (e.g. a *literal* function which takes a string and returns a parser that recognizes this string and only this string.), and a set of *combinators*, which take one or more parsers and combine them into one, with some specified behavior. Two notable examples are:

[4] Such as type annotations, pattern matching to deconstruct values returned by parsers before they can be used, etc.

- The sequence or '~' combinator, which takes two parsers and returns a parser, which attempts the first parser on the given input, and, in case of success, attempts the second parser on the remainder of the input. The value returned by the resulting parser will be a pair consisting of the results of the underlying parsers.

- The alternative or '|' combinator, which takes two parsers and returns a parser, which attempts the first parser on the given input, and, in case of failure, attempts the second parser on the same input.

- The commit combinator, which takes a parser and transforms its *Failures* into *Errors*. This has a combined effect of improving performance and error reporting, by giving the user the ability to prevent backtracking when we are sure we are in the right production (e.g. in $\mathfrak{I}$, after we've seen the "while" keyword, we are sure to be parsing a *while* statement, and thus know not to backtrack on failure to attempt to parse an *if* statement instead.)

Our redesigned library attempts to retain the "look and feel" of the original library, maintaining the same names for basic parsers and combinators whenever possible.

To improve error reporting, we introduce the following changes (heavily influenced by the mechanisms used in *parsec*; see [2] for details):

- For each error, we construct a limited "backtrace", which describes the state of the parser at the time the error was encountered. This is done by means of the rule combinator (see 3.1.2 for details).

- Rather than displaying the error message contained in the *Error* or *Failure* object that gets propagated to the top level, we reconstruct the *Errors* and *Failures* at each intermediate parsing step and update their messages as necessary. For example, if the parser generated by the '|' combinator attempts both of its alternatives, and both fail without consuming any input, their error messages are merged. This amounts to a dynamic calculation of each rule's FIRST set.

- Since the aforementioned FIRST set of a rule will typically consist of *terminals* that will be the first tokens recognized by this rule, error messages using only the FIRST set can become cryptic. For example, if the rule is for parsing a number, and its FIRST set is a collection of regular expressions that recognize different representations of numbers, the error message may be confusing. To counter this, we introduce a "label" combinator, which replaces the error messages returned by the underlying parser with a provided label (something like "number" or "identifier" to replace "[0-9]+" or "[A-Za-z_][A-Za-z0-9_]*", respectively).

Local error recovery, conceptually, can use one of two mechanisms [3]. Either the expected, but missing, token is "inserted" into the text so parsing can continue, or unexpected, but present, tokens are discarded until the Parser reaches a state from which it can resume normal operation. We provide both mechanisms as the following parser combinators:

- recoverInsert takes a parser p, a replacement value v, and an *expected* set E, and returns a parser which tries p on the input and, in case of failure, successfully returns v while recording that an error has occurred (because one of E was expected but another token found).

- recoverSkip takes a parser p and a set of FOLLOW parsers T, and returns a parser which tries p on the input and, in case of failure, skips ahead until one of T parses successfully or until the end of input is reached, and successfully returns some default value (like NULL) while recording that an error has occurred.

In order to allow backtracking to function properly, error recovery mechanisms can only be invoked when we are sure not to backtrack (i.e. in a *committed* parser).

2.3 Design of the grammar compiler

The grammar processor and compiler follows a familiar pipeline compiler structure, consisting of a parsing phase, a sequence of analysis phases, and a target code generation phase.

The input files to the compiler describe the language to be parsed, the auxiliary data structures (e.g. AST), as well as various options that direct the parser generation (See 3.2.1 for a detailed description of the source file structure). The language description is an EBNF, augmented with several new constructs that provide additional information to the parser generator (like the notion of "committing", which becomes especially important in the light of error recovery).

The analysis phases transform and collect information about the grammar, as well as performing a best-effort verification of its consistency (primarily through type-checking).

The code generation phase uses all the information collected by the analysis phases to generate the Scala code for the resulting Parser.

3 Implementation

3.1 Combinator library

3.1.1 Data structures

In our redesigned combinator library, the data structures representing parse results had to be modified to accommodate the added error handling mechanisms. We maintain the conceptual distinction between *Success*, *Failure*, and *Error*, but all three now carry more information than their vanilla Scala prototypes:

- All *ParseResults* have a field *errors* which carries the descriptions of errors that were recovered from the process of producing this Result.
- All *ParseResults* have a field *expected* which contains the set of "things" expected at this position (either terminals or labels). This is used in error message generation (e.g. "Expected x or y but found z at position ...")
- *Error* and *Failure* additionally have a field *backtrace* which is a stack of "rules" (as Strings) and positions, which describe the Parser context. This is used in error message generation (e.g. "While parsing 'expression' at position ...").

The error descriptions carried by *ParseResults* in the *errors* field are tuples consisting of: the set of *expected* things, the position of the error, and a *backtrace*.

3.1.2 Fundamental combinators

The fundamental combinators that were introduced in the description of the vanilla Scala library remain, but their internals have to modified to account for the changes in data structures and error handling[5].

The sequence combinator ('~') needs to handle the case when the first parser succeeds without consuming any input. In this case, the *expected* set it returns must be merged with the *expected* set of the result of the second parser to correctly determine the FIRST set of the resulting parser. The sets of recovered errors returned by both parsers are also merged. In that event, if the second parser does not succeed, the backtrace of the result is cleared, to reflect a fact that a potentially *new* error has been generated.

The alternative combinator ('|') suffers similar modifications. If both underlying parsers consume no input, their *expected* and *errors* sets are merged. Indeed, even if the first alternative succeeds, but does not consume any input, the second alternative must still be attemped (and its result discarded, with the exception of its *expected* set). The *errors* sets are merged. If both alternatives result in a *Failure*, the

[5]The changes that have to do with error *reporting* reflect some of the mechanisms used in *parsec*. [2] provides a more detailed rationale for them.

resulting *backtrace* becomes empty, to reflect the fact that a *new* error has been potentially generated (by merging the *expected* sets).

The newly added rule combinator takes a parser p and a name n, and returns a parser which pushes n and the position of the input onto the *backtrace* of the result of p (this includes the backtraces of the recovered errors carried in the result).

3.1.3 Error handling mechanisms

The error reporting mechanism is implemented in the sequence and alternative combinators described above. The combinators dynamically compute the set of *expected* terminals or labels at the position of the error by appropriately merging the *expected* sets returned by the underlying parsers.

Error recovery is realized by means of two combinators, recoverInsert and recoverSkip.

recoverInsert takes three parameters: a parser p, a replacement value v, and an expected set E. It returns a parser, which converts p's *Failures* and *Errors* into *Successes* that carry the value v, whose *expected* set is E, and whose *errors* set contains the corresponding error (expected E but found ...).

recoverSkip takes a parser p and a set of "terminator" parsers T. The resulting parser tries p on the given input, and, if unsuccessful, skips ahead in the input until one of the "terminator" parsers succeeds. Then it returns a *Success* with a default value (e.g. NULL or 0, depending on the result type), the advanced input, the *expected* set of the *Failure* or *Error* that prompted the recovery, and a set of recovered *errors* that contains the corresponding entry.

3.2 Grammar compiler

3.2.1 Structure of the input file

The source files accepted by the compiler admit the following structure. A *file* consists of some number of *sections*, each of which has its own syntax and carries its own kind of information:

- A "settings" section contains key-value pairs that alter the configuration of the compiler. For example, "parserName" changes the name of the file/module that will contain/define the generated parser. The "positional" setting, if enabled, modifies the parsers to store the positions of the parsed entity in the resulting object, whenever appropriate (useful when parsing into ASTs for further analysis).

- A "tree" section contains a hierarchical definition of a tree-like structure (e.g. an AST) that can be used to define parser *actions*.

- A "grammar" section contains a chunk of the grammar of the language to be parsed, in EBNF notation. It is augmented with additional constructs for expressing, for example, the notion of committing parsers. A '!' character following a production element (like a terminal) in a sequence "commits" and triggers error recovery mechanisms in the remainder of the sequence (e.g. `‘‘while’’! ‘(’ expr ‘)’ ...`). A grammar is a sequence of *rules*, each of which has a name, a (possibly empty) set of options (like "label"), a list of *productions*. Each *production* consists of a sequence of *production elements* (terminals, non-terminals, repeated production elements, etc.) – its body, and an optional *action*. The *action* is a reference to a function that will be applied to the result of successfully parsing the current production body (which will be either a single value or a tuple, depending on the number of elements in the body). This function will typically be a constructor of an AST node, defined in the "tree" section. An action can also be an arbitrary piece of Scala code, in which case it has to be explicitly annotated with a type to aid the type checker. The default action simply returns the result of parsing the body of the production (i.e. it is equivalent to an identity function).

- A "declarations" section contains a number of function signatures, which the user will have to separately implement to complete the Parser. The names of these functions can be supplied in the "grammar" section as *actions*.

3.2.2 Initial processing

The initial analysis stage performs several semantic sanity checks (such as ensuring that a repeated term is not decorated with the '!' modifier) and rewrites the internal grammar representation accordingly. It also collects some basic information, and compiles the lists of rules, types, functions, etc. into internal data structures.

3.2.3 First and follow sets

The second phase uses well-known algorithms to compute the FIRST and FOLLOW sets of each non-terminal (rule). See [7] for details. The FIRST sets are currently unused in the compiler, except as intermediary steps in the calculation of FOLLOW sets, which are necessary for error recovery (specifically, in generating applications of recoverSkip).

3.2.4 Type checker

The type checker phase performs a best-effort sanity check, issuing warnings when it encounters types it does not know about (e.g. a Scala type that is not hardcoded as a basic type, like Int, and that isn't defined in the "tree" section).

First, a dependency resolver phase is run to establish the order in which the rules need to be type checked. Each rule is scanned to compile a set of other rules that it directly references. The resulting dependency graph is analyzed and sets of mutually recursive rules are compiled using Tarjan's algorithm for finding strongly connected components [6]. A new dependency graph is constructed, with these sets as nodes, and dependencies between them as edges. Since this is now guaranteed to be a DAG, it is topologically sorted to provide an order in which these sets will be type checked.

The type checker implements a simplified Hindley-Milner algorithm([9]), extended with a naive notion of subtyping. When type checking a set of mutually-dependent rules, each rule is temporarily assigned a fresh type variable. Then the rules are type checked in some arbitrary sequence, producing substitutions that map the assigned type variables to actual types. Applying a composition of these substitutions to the original type variables assigns types to the rules.

Each rule is type checked in a bottom-up way, by recursively inferring the types of all production elements, then the type of each production body, then the type of each action application. Finally, the types inferred for each production of a given rule are unified to yield the type of the rule.

When the type checker encounters a type it does not recognize, it treats it as a wildcard type, compatible with any other type. This is required to allow for the use and declaration of potentially any Scala type, and makes it so the type checker can only perform on a best-effort basis.

3.2.5 Code generation

After the analysis phases have completed, code generation is straightforward. We use pretty-printing combinators supplied by the *kiama* language processing library ([5]). The tree structure defined in the "tree" section is translated into a set of Scala class definitions, with tree branches being translated into *traits* and leaves into *case classes*. A separate trait containing all tree definitions is generated, along with a trait that contains an abstract description of the corresponding types and their subtyping relationships (to decouple the structure from the implementation, which the user may want to manually alter). In a similar way, the "declarations" section is compiled into two separate traits, one of which contains the abstract function signatures, whereas the other contains stub implementations, intended to be filled in by the user.

The "grammar" is compiled into a Scala module ("object"), which mixes in all of the aforementioned traits, along with the "Parsers" trait that defines our parser combinators library. Each rule is translated into a "lazy val", whose type is a "Parser" parameterized by the type of the rule, and whose value is a straightforward translation of the rule's productions and actions. The necessary boilerplate (such as

[6] `http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm`

using pattern matching to deconstruct the value returned by the production body before feeding it into its corresponding action, or the error handling mechanisms) is inserted accordingly. The parser for each rule is wrapped in an application of the rule combinator, which ensures correct backtrace computation.

4 Evaluation

Recall that the stated goals of the project are to:

- Improve the *functionality* of the Scala parser combinators library, most notably by improving error handling.
- Simplify the definition of Parsers by removing the need for unneccessary boilerplate code and automatically compute certain properties of the grammar (like FOLLOW sets, directly used for error recovery).

Consider our example language ℑ. Appendix C contains a sample source file that describes a Parser for ℑ in the language of our parser generator. Compare this with the generated Scala code for the same (Appendix D).

For example, the rule for parsing an *if statement*, defined as follows:

```
ifStatement[label]
  ::= ["if"]! ["("] expr [")"] ["{"] expr ["}"]
        (["else"]! ["{"] expr ["}"])? -> IfStmt
```

is translated into:

```
    lazy val ifStatement: Parser[IfStmt] = lrule("ifStatement") {
      "if" ~!> (recoverInsert("(") ~>
        (recoverSkip(expr, Seq(")", "}", ";")) ~
        (recoverInsert(")") ~> (recoverInsert("{") ~>
          (recoverSkip(expr, Seq(")", "}", ";")) ~
        (recoverInsert("}") ~> opt("else" ~!>
          (recoverInsert("{") ~> (recoverSkip(expr
          , Seq(")", "}", ";")
          ) <~ recoverInsert("}")))
        ))))))) ^^ { case x1 ~ (x2 ~ x3) => IfStmt.apply(x1, x2, x3) }
    }
```

The original code is concise, readable, and easily recognized as (a slightly modified) EBNF notation. The resulting Scala code is cluttered with things like type annotation, many sets of parantheses to correct precedence, information that was implicit in the grammar (the FOLLOW set of *expr*), different varieties of the *seq* combinator (“~”, “~>”, “~!>”, etc.). In essence, we get the best of both worlds: the common case is simplified by only requiring from the user the bare essentials needed to generate a Parser, but the resulting code is still modular and maintanable by an “advanced” user, since it uses combinatory parsing.

To review error handling, let us return to our motivating example. Consider the following ℑ code:

```
  if(5) { foo } else { 10 }
```

An implementation of a ℑ Parser using the Scala-provided library gives the following error:

```
[1.9] error: ‘(’ expected but ‘f’ found

if(5) { foo } else { 10 }
        ^
```

A hand-written Parser written using our reimplementation of the library, run on the same input, yields:

```
[error]
While reading 'multiply expression' @ line 1, col 9:
While reading 'simple expression' @ line 1, col 9:
  Expected one of: number, "("; but found 'f' @ line 1, col 9
if(5) { foo } else { 10 }
        ^
```

The provided backtrace gives some insights into the state of the Parser at the time the error was encountered; the user now knows that the error occur while parsing a "simple expression" as part of a possible multiplication.

Now, let us consider a program with multiple syntax errors (the same unexpected token "foo", a missing opening brace after "if", and a missing opening brace after the "else"):

```
if(5) foo } else 10 }
```

The original Parser complains of the first error, ignoring the second:

```
[1.7] error: '{' expected but 'f' found

if(5) foo } else 10 }
      ^
```

A hand-written Parser using the new library uses the recoverInsert but not the recoverSkip mechanism (which would require calculating the FOLLOW set), so it successfully recovers from the first error, but cannot recover from the second:

```
[error]
While reading 'multiply expression' @ line 1, col 7:
While reading 'simple expression' @ line 1, col 7:
  Expected one of: number, "("; but found 'f' @ line 1, col 7
if(5) foo } else 10 }
      ^

While reading 'statement' @ line 1, col 1:
While reading 'if statement' @ line 1, col 1:
  Expected "{"; but found 'f' @ line 1, col 7
if(5) foo } else 10 }
      ^
```

Finally, the generated Parser successfully recovers from all three errors, and can thus report all of them:

```
[error]
While reading 'statement' @ line 1, col 1:
While reading 'ifStatement' @ line 1, col 1:
  Expected "{"; but found 'f' @ line 1, col 7
if(5) foo } else 10 }
      ^

While reading 'multiply expression' @ line 1, col 7:
While reading 'simple expression' @ line 1, col 7:
  Expected one of: number, "(", "true", "false"; but found 'f' @ line 1, col 7
if(5) foo } else 10 }
      ^
```

```
While reading 'statement' @ line 1, col 1:
While reading 'ifStatement' @ line 1, col 1:
  Expected "{"; but found '1' @ line 1, col 18
if(5) foo } else 10 }
                 ^
```

5 Limitations and future work

5.1 Limitations

The project, as it stands today, struggles with the following limitations:

- Certain significant Scala features are impossible to express in our language. For example, we assume that functions have no lazy (call-by-name) or repeated parameters. We do not allow name overloading. We do not allow for the use of almost arbitrary characters in identifiers and names (Scala allows unicode characters and special characters like '+' in method and field names).
- The type checker only knows about a set of basic hardcoded types, and the types explicitly defined by the user. The vast universe of Scala types remains unknown and is treated transparently as wildcard types, greatly limiting the scope of the type checker's usefulness.

5.2 Future work

- Given the extensible (by design) architecture of our compiler, one could suggest the addition of new analysis and rewriting phases, serving various error-correcting and optimization purposes. One could, for instance, conceive of a phase that identifies the presence of left recursion (direct and indirect) and optionally rewrites the grammar to avoid it.
- One common functionality of parser generators that we left out in the prototype is dealing with operator precedence. As of yet, parsing infix arithmetic expressions requires the use of the '<+>' or '<*>' operators (which repeatedly parse a given production element with the given separator) and an externally-defined function for combining the resulting sequences (with the desired associativity). We could implement an equivalent of the standard "chainl1" and "chainr1" combinators that simplify this task.
- Knowledge of all existing Scala types could be gained, without bothering the user, by using Scala's powerful reflection API. This would greatly extend the scope of our type checker.
- A thorough evaluation of our proposed solution's usability (with a large user base) and performance has yet to be done. Although we can anticipate that the performance of the resulting Parsers will suffer as a result of the additional overhead incurred by the newly-added features (like error recovery), the performance is likely to be satisfactory for most uses in small, non-performance-critical DSLs that typically have small source files. Naturally, if performance becomes critical, the user is probably better off using mature, optimized tools, like yacc or bison to satisfy their parsing needs.

6 Conclusion

We have presented a suite of parsing tools – namely an extended library, and a compiler for an external DSL – that address the perceived shortcomings of Scala's standard parser combinators library. In particular, we tackle the issue of poor error handling by providing primitives that collect more details for error reporting (backtraces, more complete sets of expected tokens, labels), and enable some local error recovery (by inserting tokens that are expected and not found, and skipping tokens that are found but not expected). To reduce the

burden of excessive boilerplate on the user and enable more advanced grammar analysis and transformations than permitted by the existing framework, we introduce an EBNF-based language for concisely describing formal grammars, and a compiler that generates the corresponding Scala code after performing the necessary analysis and sanity checks.

The small examples we have studied so far (like $\mathfrak{I}$) suggest that this combined approach indeed allows to define Parsers in Scala in a more efficient and concise manner, while improving the quality of the produced Parsers (via enhanced error handling) and retaining the advantages offered by combinatory parsing over alternative methods (modularity and maintanability).

References

[1] `http://scala-lang.org/`

[2] Leijen, Daan, Erik Meijer, and W. A. Redmond. "Parsec: Direct Style Monadic Parser Combinators For The Real World." (2001).

[3] Swierstra, S. Doaitse, and Pablo R. Azero Alcocer. "Fast, error correcting parser combinators: A short tutorial." SOFSEM99: Theory and Practice of Informatics. Springer Berlin Heidelberg, 1999.

[4] Frost, Richard A., Rahmatullah Hafiz, and Paul Callaghan. "Parser combinators for ambiguous left-recursive grammars." Practical Aspects of Declarative Languages. Springer Berlin Heidelberg, 2008. 167-181.

[5] Sloane, Anthony M. "Lightweight language processing in Kiama." Generative and Transformational Techniques in Software Engineering III. Springer Berlin Heidelberg, 2011. 408-425.

[6] `http://code.google.com/p/kiama/`

[7] Parr, Terence J., and Russell W. Quong. "ANTLR: A predicated LL(k) parser generator." Software: Practice and Experience 25.7 (1995): 789-810.

[8] `http://github.com/sirthias/parboiled/wiki`

[9] Damas, Luis, and Robin Milner. "Principal type-schemes for functional programs." Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1982.

A Language grammar

Notation:
x ⟨+⟩ y means x repeated at least once, separated by y.
x ⟨*⟩ y means x repeated zero or more times, separated by y.

⟨*file*⟩ ::= ⟨*section*⟩*

⟨*section*⟩ ::= settings
| ⟨*trees*⟩
| ⟨*grammar*⟩
| ⟨*definition*⟩

A.1 Settings

⟨*settings*⟩ ::= '`#settings`' ⟨*setting*⟩+

⟨*setting*⟩ ::= ⟨*identifier*⟩ '`=`' ⟨*string*⟩

A.2 Trees

⟨*trees*⟩ ::= '`#trees`' ⟨*treeDef*⟩+

⟨*treeDef*⟩ ::= ⟨*treeBranch*⟩ | ⟨*treeLeaf*⟩

⟨*treeBranch*⟩ ::= ⟨*identifier*⟩ '`{`' ⟨*treeDef*⟩* '`}`'

⟨*treeLeaf*⟩ ::= ⟨*identifier*⟩ '`(`' ⟨*leafParam*⟩ ⟨*+*⟩ '`,`' '`)`'

⟨*leafParam*⟩ ::= ⟨*identifier*⟩ '`:`' ⟨*paramType*⟩

⟨*declarations*⟩ ::= '`#declarations`' ⟨*declaration*⟩+

⟨*declaration*⟩ ::= ⟨*identifier*⟩ '`:`' ⟨*scalaType*⟩
| ⟨*identifier*⟩ '`(`' ⟨*paramType*⟩ ⟨***⟩ '`,`' '`)`' '`:`' ⟨*scalaType*⟩

A.3 Grammar

⟨*grammar*⟩ ::= '`#grammar`' ⟨*rule*⟩*

⟨*rule*⟩ ::= ⟨*identifier*⟩ ⟨*ruleOptions*⟩? ⟨*production*⟩ ⟨*+*⟩ '`|`' '`;`'

⟨*ruleOptions*⟩ ::= '`[`' ⟨*ruleOption*⟩ ⟨*+*⟩ '`,`' '`]`'

⟨*ruleOption*⟩ ::= ⟨*identifier*⟩ ('`=`' ⟨*string*⟩)?

⟨*production*⟩ ::= ⟨*conjunction*⟩ ('`->`' ⟨*action*⟩)?

⟨*action*⟩ ::= ⟨*identifier*⟩
| ⟨*customAction*⟩

⟨*customAction*⟩ ::= r'`(\{%([^%]|[\r\n]|(%+([^%\}]|[\r\n])))*%\})`' '`:`' ⟨*scalaType*⟩

⟨*conjunction*⟩ ::= ⟨*repSepTerm*⟩+

⟨*disjunction*⟩ ::= ⟨*conjunction*⟩ ⟨*+*⟩ '`|`'

⟨*repSepTerm*⟩ ::= ⟨*optionalTerm*⟩ ('`<+>`' | '`<*>`') ⟨*optionalTerm*⟩
| ⟨*optionalTerm*⟩

⟨*optionalTerm*⟩ ::= ⟨*repTerm*⟩ '`?`'?

⟨*repTerm*⟩ ::= ⟨*labeledTerm*⟩ ('`+`' | '`*`') ⟨*labeledTerm*⟩
| ⟨*labeledTerm*⟩

⟨*labeledTerm*⟩ ::= ⟨*committedTerm*⟩ '`<?>`' ⟨*string*⟩
| ⟨*committedTerm*⟩

⟨*committedTerm*⟩ ::= ⟨*simpleTerm*⟩ '`!`'?

⟨*simpleTerm*⟩ ::= '`[`' ⟨*disjunction*⟩ '`]`'
| '`(`' ⟨*disjunction*⟩ '`)`'
| ⟨*string*⟩
| ⟨*regex*⟩
| ⟨*number*⟩
| ⟨*identifier*⟩

A.4 Types

⟨*scalaType*⟩ ::= ⟨*functionType*⟩ | ⟨*infixType*⟩

⟨*functionType*⟩ ::= ⟨*functionArgTypes*⟩ '`=>`' ⟨*scalaType*⟩

⟨*functionArgTypes*⟩ ::= ⟨*paramType*⟩ ⟨***⟩ '`,`'

⟨*paramType*⟩ ::= '`=>`'? ⟨*scalaType*⟩ '`*`'?

⟨*infixType*⟩ ::= ⟨*simpleType*⟩ ⟨*typeMods*⟩

⟨*typeMods*⟩ ::= '`[`' ⟨*typeArgs*⟩ '`]`'
| ⟨*typeProjection*⟩

⟨*typeArgs*⟩ ::= ⟨*scalaType*⟩ ⟨*+*⟩ '`,`'

⟨*typeProjection*⟩ ::= '`#`' ⟨*identifier*⟩

⟨*simpleType*⟩ ::= '`(`' ⟨*scalaType*⟩ ⟨*+*⟩ '`,`' '`)`'
| ⟨*stableId*⟩

A.5 Tokens

⟨*identifier*⟩ ::= r'`[a-zA-Z_][a-zA-Z0-9_]*`'

⟨*stableId*⟩ = ⟨*identifier*⟩ ⟨***⟩ '`.`'

⟨*string*⟩ ::= r'`"(\.|[^"])*"`'

⟨*regex*⟩ ::= r'`r"(\.|[^"])*"`'

⟨*number*⟩ ::= real | integer

B ℑ grammar

The following is the formal EBNF grammar for the example language ℑ used in this report.

⟨*program*⟩ ::= ⟨*statement*⟩ ⟨*+*⟩ '`;`'

⟨*statement*⟩ ::= ⟨*whileStatement*⟩
| ⟨*ifStatement*⟩
| ⟨*exprStatement*⟩

⟨*whileStatement*⟩ ::= '`while`' '`(`' ⟨*expr*⟩ '`)`' '`{`' ⟨*expr*⟩ '`}`'

⟨*ifStatement*⟩ ::= '`if`' '`(`' ⟨*expr*⟩ '`)`' '`{`' ⟨*expr*⟩ '`}`' ('`else`' '`{`' ⟨*expr*⟩ '`}`')?

⟨*exprStatement*⟩ ::= ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*mulExpr*⟩ ⟨*+*⟩ '`+`'

⟨*mulExpr*⟩ ::= ⟨*simpleExpr*⟩ ⟨*+*⟩ '`*`'

⟨*simpleExpr*⟩ ::= ⟨*number*⟩
  | ‘(’ ⟨*expr*⟩ ‘)’

⟨*number*⟩ ::= r‘[0-9]+’

C Description used to generate a Parser for ℑ

This is the .grm file used to generate a Parser for ℑ.

```
#settings
positional = "true"
packageName = "imperative"
//packrat = "true"

#tree
Term {
  Program(stmts: Seq[Stmt])

  Expr {
    Num(n: Int)
    Add(op1: Expr, op2: Expr)
    Times(op1: Expr, op2: Expr)
    Boolean(repr: String)
  }

  Stmt {
    WhileStmt(cond: Expr, body: Expr)
    IfStmt(cond: Expr, ifBranch: Expr, elseBranch: Option[Expr])
    ExprStmt(e: Expr)
  }
}

#grammar

program ::= statement <+> ";" -> Program
          ;

statement ::= whileStatement
            | ifStatement
            | exprStatement
            ;

whileStatement[label]
  ::= ["while"! "("] expr [")" "{"] expr ["}"] -> WhileStmt
    ;

ifStatement[label]
  ::= ["if"! "("] expr [")" "{"] expr ["}"]
        (["else"! "{"] expr ["}"])? -> IfStmt
    ;
```

```
exprStatement ::= expr -> ExprStmt
           ;

expr[name = "expression"]
  ::= mulExpr <+> "+"! -> leftAssocAdd
    ;

mulExpr[name = "multiply expression"]
  ::= simpleExpr <+> "*"! -> leftAssocMultiply
    ;

simpleExpr[name = "simple expression"]
  ::= number | ["("!] expr [")"]
    ;

number[label]
  ::= r"[0-9]+" -> strToInt
    ;

#declarations

leftAssocAdd(Seq[Expr]): Expr
leftAssocMultiply(Seq[Expr]): Expr
strToInt(String): Num
```

D ℑ Parsers

The following is the code for three versions of a combinator-based Parser for the ℑ language: one hand-written, utilizing the parser combinator library shipped with Scala, one hand-written, using our proposed alternative to that library, and one generated by our grammar compiler.

D.1 Hand-written, using Scala library

```
object HandOldParser extends RegexParsers {
  override protected val whiteSpace = """[\n\t␣]+""".r

  def apply(prgm: String) =
    parseAll(program, prgm)

  lazy val program: Parser[Program] =
    rep1sep(statement, ";") ^^ Program.apply

  lazy val statement: Parser[Stmt] =
    whileStmt | ifStmt | exprStmt

  lazy val whileStmt: Parser[Stmt] =
    "while" ~! ("(" ~> expr <~ ")") ~ ("{" ~> expr <~ "}") ^^ {
      case _ ~ cond ~ body => WhileStmt(cond, body)
    }

  lazy val ifStmt: Parser[Stmt] =
```

```
    "if" ~! ("(" ~> expr <~ ")") ~ ("{" ~> expr <~ "}") ~
    opt("else" ~> ("{" ~> expr <~ "}")) ^^ {
      case _ ~ cond ~ ifBranch ~ elseBranch =>
        IfStmt(cond, ifBranch, elseBranch)
    }

  lazy val exprStmt: Parser[Stmt] =
    expr ^^ ExprStmt.apply

  lazy val expr: Parser[Expr] =
    rep1sep(mulExpr, "+") ^^ leftAssocAdd

  lazy val mulExpr: Parser[Expr] =
    rep1sep(simpleExpr, "*") ^^ leftAssocMultiply

  lazy val simpleExpr: Parser[Expr] =
    num | ("(" ~> expr <~ ")")

  lazy val num: Parser[Expr] =
    "[0-9]+".r ^^ { n => Num(Integer.parseInt(n)) }

  // utils

  def leftAssoc(es: Seq[Expr], op: (Expr, Expr) => Expr) =
    es reduceLeft op

  def leftAssocAdd(es: Seq[Expr]) =
    leftAssoc(es, Add.apply)

  def leftAssocMultiply(es: Seq[Expr]) =
    leftAssoc(es, Multiply.apply)
}
```

D.2 Hand-written, using our library

```
object HandParser extends Parsers {
  override protected val whiteSpace = """[\n\t␣]+""".r

  def apply(prgm: String) =
    parseAll(program, prgm)

  lazy val program: Parser[Program] = rule("program") {
    rep1sep(statement, ";") ^^ Program.apply
  }

  lazy val statement: Parser[Stmt] = rule("statement") {
    whileStmt | trace(ifStmt)("ifStmt") | exprStmt
  }

  lazy val whileStmt: Parser[Stmt] = lrule("while␣statement") {
    "while" ~!> ((recoverInsert("(") ~> expr <~ recoverInsert(")")) ~
    (recoverInsert("{") ~> expr <~ recoverInsert("}"))) ^^ {
      case cond ~ body => WhileStmt(cond, body)
    }
```

```
  }

  lazy val ifStmt: Parser[Stmt] = lrule("if␣statement") {
    "if" ~!> ((recoverInsert("(") ~> expr <~ recoverInsert(")")) ~
      (recoverInsert("{") ~> expr <~ recoverInsert("}")) ~
      opt("else" ~!> (recoverInsert("{") ~> expr <~ recoverInsert("}")))) ^^ {
        case cond ~ ifBranch ~ elseBranch => IfStmt(cond, ifBranch, elseBranch)
      }
  }

  lazy val exprStmt: Parser[Stmt] = rule("expression␣statement") {
    trace(expr ^^ ExprStmt.apply)("exprStmt")
  }

  lazy val expr: Parser[Expr] = lrule("expression") {
    rep1sepc(mulExpr, "+") ^^ leftAssocAdd
  }

  lazy val mulExpr: Parser[Expr] = rule("multiply␣expression") {
    rep1sepc(trace(simpleExpr)("simpleExpr"), "*") ^^ leftAssocMultiply
  }

  lazy val simpleExpr: Parser[Expr] = rule("simple␣expression") {
    trace(num)("num") | ("(" ~> expr <~ ")")
  }

  lazy val num: Parser[Expr] = lrule("number") {
    "[0-9]+".r ^^ { n => Num(Integer.parseInt(n)) }
  }

  // utils

  def leftAssoc(es: Seq[Expr], op: (Expr, Expr) => Expr) =
    es reduceLeft op

  def leftAssocAdd(es: Seq[Expr]) =
    leftAssoc(es, Add.apply)

  def leftAssocMultiply(es: Seq[Expr]) =
    leftAssoc(es, Multiply.apply)
}
```

D.3 Generated

```
object SampleParser extends Parsers
with SampleParserTrees
with SampleParserTreesImpl
with SampleParserUtils
with SampleParserUtilsImpl {
  lazy val exprStatement: Parser[ExprStmt] = rule("exprStatement") {
    expr ^^ { x => ExprStmt.apply(x) }
  }

  lazy val simpleExpr: Parser[Expr] = rule("simple␣expression") {
```

```
  number ^^ { x => (x) } |
  "(" ~!> (recoverSkip(expr, Seq(")", "}", ";")) <~ recoverInsert(")")) ^^ {
    x => (x)
  }
}

lazy val statement: Parser[Stmt] = rule("statement") {
  whileStatement ^^ { x => (x) } |
  ifStatement ^^ { x => (x) } |
  exprStatement ^^ { x => (x) }
}

lazy val whileStatement: Parser[WhileStmt] = lrule("whileStatement") {
  "while" ~!> (recoverInsert("(") ~> (recoverSkip(expr
    , Seq(")", "}", ";")
    ) ~ (recoverInsert(")") ~> (recoverInsert("{") ~> (recoverSkip(expr
    , Seq(")", "}", ";")
    ) <~ recoverInsert("}")))))) ^^ {
    case x1 ~ x2 => WhileStmt.apply(x1, x2)
  }
}

lazy val number: Parser[Num] = lrule("number") {
  "[0-9]+".r ^^ { x => strToInt.apply(x) }
}

lazy val ifStatement: Parser[IfStmt] = lrule("ifStatement") {
  "if" ~!> (recoverInsert("(") ~> (recoverSkip(expr, Seq(")", "}", ";")) ~
    (recoverInsert(")") ~> (recoverInsert("{") ~>
      (recoverSkip(expr, Seq(")", "}", ";")) ~ (recoverInsert("}") ~>
        opt("else" ~!> (recoverInsert("{") ~> (recoverSkip(expr
      , Seq(")", "}", ";")
      ) <~ recoverInsert("}")))
    ))))))) ^^ { case x1 ~ (x2 ~ x3) => IfStmt.apply(x1, x2, x3) }
}

lazy val program: Parser[Program] = rule("program") {
  rep1sep(statement, ";") ^^ { x => Program.apply(x) }
}

lazy val expr: Parser[Expr] = rule("expression") {
  rep1sepc(recoverSkip(mulExpr, Seq(")", "}", "+", ";"))
    , "+"
    , Some(mulExpr)
    ) ^^ { x => leftAssocAdd.apply(x) }
}

lazy val mulExpr: Parser[Expr] = rule("multiply␣expression") {
  rep1sepc(recoverSkip(simpleExpr, Seq("*", ";", "}", ")", "+"))
    , "*"
    , Some(simpleExpr)
    ) ^^ { x => leftAssocMultiply.apply(x) }
}
```

```
  override protected val whiteSpace = """[\n\t␣]+""".r

  def apply(in: String) = {
    parseAll(program, in) match {
      case Left(e) => println(e); scala.sys.error("Syntax␣error.")
      case Right(v) => v
    }
  }
}
```