Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 193 (2007) 3–18
www.elsevier.com/locate/entcs

Executable Grammars in Newspeak

Gilad Bracha [1]

Cadence Design Systems
San Jose, California, USA

Abstract

We describe the design and implementation of a parser combinator library in Newspeak, a new language in the Smalltalk family. Parsers written using our library are remarkably similar to BNF; they are almost entirely free of solution-space (i.e., programming language) artifacts. Our system allows the grammar to be specified as a separate class or mixin, independent of tools that rely upon it such as parsers, syntax colorizers etc. Thus, our grammars serve as a shared executable specification for a variety of language processing tools. This motivates our use of the term *executable grammar*. We discuss the language features that enable these pleasing results, and, in contrast, the challenge our library poses for static type systems.

Keywords: Parser combinators, dynamic programming languages, Smalltalk, reflection

1 Introduction

Newspeak is a programming language derived from Smalltalk, currently under development. We have developed a parser combinator library in Newspeak, and find that Smalltalk and Newspeak have properties that make them very well suited for the implementation and use of such a library.

The concept of parser combinators has a long history in functional programming [11,13,21]. From an object-oriented perspective, the basic idea is to view the operators of BNF as methods, known as combinators, that operate on objects representing productions of a grammar. Each such object is a parser that accepts the language generated by a particular production. The results of the combinator invocations are also such parsers.

To make this concrete, lets look at a fairly standard rule for identifiers:

[1] Email: gilad@cadence.com

doi:10.1016/j.entcs.2007.10.004

id = letter (letter | digit)*

We can express this in Newspeak as:

id = letter, (letter | digit) star

Assume letter is a parser that accepts a single letter and digit is a parser that accepts a single digit. The subexpression letter | digit invokes the method | on the parser that accepts a letter, with an argument that accepts a digit. The result will be a parser that accepts either a letter or a digit.

We then invoke the method star on the result

(letter | digit) star

which yields a parser that accepts zero or more occurrences of either a letter or a digit.

We pass this parser as an argument to the method "," which we invoke on letter. The "," method is the sequencing combinator (which is implicit in BNF). It returns a parser that first accepts the language of the receiver and then accepts the language of its argument. In our example the result accepts a single letter, followed by zero or more occurrences of either a letter or a digit, just as we'd expect. Finally, we bind this result to id.

You should now have a basic intuition for the way parser combinators work, and a feel for what executable grammars look like in our framework. Here is a roadmap to the rest of this paper:

Section 2 shows a complete example of an executable grammar. Then, sections 3 through 6 discuss a number of design issues and features of our framework. In particular, section 4 showcases the modular separation between grammar and processing. The paper presents ideas for future improvements (section 7); In section 7.1, we analyze the type safety of grammars and parsers constructed using the library, and what mechanisms are needed to statically check them. We discuss related work (section 8), and conclude with section 9.

2 A Complete Example

We now consider a small grammar, and show how to represent it in our framework. In doing so, we'll highlight some interesting design issues. We will use the following grammar

expression = id
returnStatement = ^ expression.

This grammar relies on terminal symbols id, letter and digit defined as:

```
digit = 0 - 9
letter = (a - z) | (A - Z)
id = letter (letter | digit)*
```

We represent this in Newspeak as follows:

```
class ExampleGrammar1 = ExecutableGrammar (
  |
    digit = charBetween: $0 and: $9.
    letter =  (charBetween: $a and: $z) | (charBetween: $A and: $Z).
    id = letter, (letter | digit) star.
    identifier = tokenFor: id.
    hat = tokenFromChar: $^.
    expression = identifier.
    returnStatement = hat, expression.
  |
 )()
```

We define a class ExampleGrammar1 which is a subclass of ExecutableGrammar, the main class in our framework.

Every production in our grammar has a corresponding instance variable, or *slot*. All access to slots in Newspeak is via method invocations. Whenever a slot is declared, the language automatically defines accessors for it. Any use of a slot's name denotes a call to its accessor. This means that we can freely alter the representation of an object, and as long as we maintain its method interface, no code needs to change - not even code within the object's own class. This notion of representation-independent code was pioneered in Emerald [3], Trellis/Owl [18] and Self [20], and is also supported in Scala [16].

An object's slots are initialized when the object is created. If the slot has an explicit initializer, the initializer is evaluated and the slot is set to the result. Otherwise, the value of a slot is initially **nil**.

Here slots are initialized to their corresponding parser, so each slot definition gives a production rule of the grammar. The rules for digit and letter are in direct correspondence to the grammar. They use the inherited utility method charBetween:and: that produces a parser that accepts characters within a given range.

The rule for id is the same one we saw before. At this point, however, it becomes evident that the lack of distinction between lexing and parsing is a bit of a problem. Traditionally, we rely on a lexer to tokenize the input. As it does that, it does away with whitespace (ignoring languages that make whitespace significant) and comments. This is dealt with by the operator, tokenFor:, that takes a parser p and returns a new parser that skips any leading whitespace

and comments and then accepts whatever p accepts. This parser will also attach start and end source indices to the result, which is very handy when integrating a parser into an IDE. It is useful to define a production identifier that produces such tokenized results.

The rule, hat, for the terminal ^ is similar, but relies on a convenience method tokenFromChar: that parses a given character and tokenizes it. We can now define the syntactical elements of the grammar without concern for whitespace or comments, just as we would in traditional BNF.

3 Mutual Recursion

Grammar rules are often mutually recursive. As an example, let's extend our grammar with these two productions:

```
statement = ifStatement  | returnStatement
ifStatement = if expression then statement else statement
```

We'll need to deal with the new keywords, and add the new productions to the grammar:

```
class ExampleGrammar2 = ExampleGrammar1 (
 |
    if = tokenFromSymbol:#if.
    then = tokenFromSymbol:#then.
    else = tokenFromSymbol:#else.
    ifStatement = if, expression, then,  statement,  else, statement.
    statement = ifStatement | returnStatement.
 |
)()
```

Because our grammar is embedded in a Newspeak class, we can leverage Newspeak's support for inheritance and define the grammar extension as a subclass, literally extending the original grammar.

However, there seems to be a problem: when initializing ifStatement, we make use of statement, which is not yet initialized. Reversing the order between the two won't help, because the productions are mutually recursive. In a lazy language like Haskell [17] this is not an issue - which is one key reason Haskell is very good at defining domain specific languages. Newspeak is not a lazy language however.

One approach to this problem would have our parser combinators take closures as arguments rather than parsers. Indeed, an earlier version of our framework did exactly that. Instead of writing

```
returnStatement = hat, expression
```

We would write

```
returnStatement = hat, [expression]
```

One complaint about using closures as arguments for parser combinators is that programmers may mistakenly pass parsers as arguments directly. In fact, this is a non-issue. Such mistakes were dynamically detected by our earlier framework. Even though detection was dynamic, it occurred during the construction of the grammar, not during actual parsing. The effect was essentially the same as with static checking. Moreover, it would be trivial to allow combinators to accept either parsers or closures.

Nevertheless, the use of closures detracts from the clarity of the grammar. We are fortunate that the closure syntax in Smalltalk (and hence Newspeak) is very lightweight, but we still prefer to avoid closures altogether. The use of closures introduces a disturbing asymmetry, where the receiver of a combinator is a parser, but its argument is a closure that evaluates to a parser.

Instead of relying on closures or laziness, our framework uses reflection to bind the slots for each production to instances of class ForwardReferenceParser before the slot initializers are executed.

Instances of ForwardReferenceParser support the usual parser combinators and are initially *unbound*. When a production such as ifStatement, that is mutually dependent on another (e.g., statement) is computed, the slot for the other production is accessed. In our example, statement will return a ForwardReferenceParser and the construction of the grammar proceeds. Forward references are lazily *bound* to the appropriate parser, to which they forward all subsequent calls, the first time they are called upon to parse.

4 Grammar Processing

The ability to define a parser by writing a program that so closely corresponds to the grammar is attractive. However, just accepting a language is not all that useful. Typically, some processing needs to be performed (e.g., to produce an AST as a result).

To address the need for processing, we introduce a new operator on parsers, wrapper:. The result of this operator is a parser that accepts the same language as the receiver. However, the result it produces from parsing differs; during parsing, the results produced by the receiver are passed to a closure which wrapper: takes as its sole parameter. The overall result is the the result returned by the closure - typically an abstract syntax tree.

```
class ExampleGrammar3 = ExecutableGrammar (
  |
    digit = charBetween: $0 and: $9.
    letter =  (charBetween: $a and:$z) | (charBetween: $A and:$Z).
    id = letter, (letter | digit) star
              wrapper:[:fst   :snd  | fst asString, (String withAll: snd)].
    identifier = tokenFor: id
                                    wrapper:[:v | VariableAST new name: v to-
ken; start: v start; end: v end].
    hat = tokenFromChar: $^.
    expression = identifier.
    returnStatement = hat, expression
                                                wrapper:[:r  :e    | Return-
StatAST new expr:e; start: r start; end: e end].
  |
  )()
```

This is illustrated above. The grammar productions for id, identifier and returnStatement have been augmented with processing using wrapper:. The grammar remains clearly distinguishable, with the AST generation on separate lines. However, it is preferable to keep the grammar completely separated. We will therefore move the AST generation code to a subclass, where the grammar production accessors will be overridden as shown below.

```
class ExampleParser1 = ExampleGrammar1 ()
(
  id = (
    ^super id
          wrapper:[:fst   :snd  | fst asString, (String withAll: snd)]
  )
  identifier = (
    ^super identifier
                                    wrapper:[:v | VariableAST new name: v to-
ken; start: v start; end: v end].
  )
  returnStatement = (
    ^super returnStatement
        wrapper:[:r :e  | ReturnStatAST new expr:e; start: r start; end: e end].
         )
   )
```

This organization is useful, for example, if one wishes to parse the same

language and feed it to different back ends that each accept their own AST; or if one needs to use the parser for a different purpose, such as syntax coloring, but wants to share the grammar.

We can do something similar with our extended grammar. However, here we find that single inheritance confronts us with a problem. Should our parser extend ExampleGrammar2 or ExampleParser1? Obviously, we need to extend both.

The appropriate superclass for our extended parser is

```
ExampleParser1 mixin | > ExampleGrammar2
```

the application of the mixin [6,7,1] implicitly defined by class ExampleParser1 to the class ExampleGrammar2.

```
class ExampleParser2 = ExampleParser1 mixin | > ExampleGrammar2 ()
(
  ifStatement = (
    ^super ifStatement
           wrapper:[:ifKw :e :thenKw :s1  :elseKw :s2 |
             IfStatAST new cond: e; trueBranch: s1; falseBranch: s2;
                                    start: ifKw start; end: s2 end
           ]
  )
)
```

This modular structure, in which grammars are separate, reusable executable components is the most distinctive feature of executable grammars.

5 Naming Parsers

Some parser combinator libraries offer support for naming parsers. This makes it easier to debug errors in the grammar. It can also be useful to print out the name of the production when reporting errors. Unfortunately, naming the productions in this way adds clutter and obscures the grammar. It also forces the programmer to repeat information already given in the specification, for example:

```
id = letter, (letter | digit) star.
id name: #id.
```

Here, the name *id* has already been given to the production in the context of the surrounding grammar. However, the parser itself is not aware of that until it is explicitly told its name in the second line above. Instead, our

framework reflectively traverses the grammar and names the various parsers based on the name used for the production in the surrounding context. No redundant and distracting naming combinator is necessary.

6 Error Handling

Care must be taken in order to support good error reporting in a parser combinator framework. Consider the grammar

```
def = identifier { stmt*}
stmt =  (ifStmt  | expr) dot
ifStmt = if expr then stmt else stmt | if expr then stmt
expr = identifier := expr |
        number |
        identifier
```

Given the input

```
foo {
    v := 3.
    if true then s := 5 else s := 7.
   }
```

In a naive implementation of parser combinators, parsing fails

```
foo {
    v := 3.
    '}' expected! -> if
```

The real error is that there is no '.' terminating the true branch of the if statement. The situation would be clearer if the parser failed as follows:

```
foo {
    v := 3.
    if true then s := 5 '.' expected! -> else
```

What actually happens is that when parsing the if statement, parsing fails on both variants (with and without else) of the ifStmt production, due to the missing '.' after 5.

The parser then uses the expr production. The first two options of expr fail, but if is parsed successfully as an identifier. However, there is no '.' after if, and so the production stmt fails. At this point control returns to stmt*, which, having already parsed one statement successfully, is happy to stop trying and return successfully, rolling the input back to just after the first statement (v

:= 3.) Now, the production for def seeks a '}', but doesn't find one, and the parse fails at the token if with the message given.

To alleviate this problem, our framework records the deepest (rightmost in the input) error detected during the parse. If the parse is successful, any such interim failures are irrelevant. However, if the parse fails, it is the deepest error, rather than the latest one, which needs to be reported. That would be the error complaining about the missing '.' after the 5, as we would like.

7 Future Developments

7.1 Typechecking

The pattern we have seen, whereby the grammar is separated from its processing, is not easy to maintain while insisting on fully static typechecking. To explore the issues, we will recast some of our examples in a hypothetical statically typed language similar in style to current mainstream programming languages.

Consider that ExampleParser1 is not a subtype of ExampleGrammar1. For example, in ExampleParser1, returnStatement returns a parser that returns ReturnStatAST, whereas in ExampleGrammar1 the returnStatement method returns a parser that returns a collection. These types are unrelated. In general, every production in the grammar has a corresponding method, and in most cases, the type of the subclass method is not a subtype of the type in the superclass. In most statically typed object-oriented programming languages, this situation is illegal. Despite this, our design is correct and causes no type errors during execution.

All uses of the methods that corrrespond to productions within ExampleGrammar1 rely only on the fact that these methods return an ExecutableGrammar; they do not rely on the type of object that a parser returns when parsing. Suppose we defined an interface type S that included all the methods of type ExampleGrammar, except that they all returned type ExecutableGrammar⟨?⟩ [2]. We could successfully typecheck ExampleGrammar1 under the assumption that the type of self was a subtype of S. This shows that it is sufficient that subclasses of ExampleGrammar1 be subtypes of S for the type safety of ExampleGrammar1 itself to be preserved. In particular, ExampleParser1 is a subtype of S; every production method in it returns an ExecutableGrammar⟨X⟩, for some X.

One possibility is to avoid making ExecutableGrammar a generic type. This inevitably leads to a reliance on dynamic typechecking, since the results of a

[2] Our notation here is similar to that of Java wildcard types [19]

parse would be of type Object, and casting and/or pattern matching would be required to recover the type information for further processing. We will not pursue this direction further, as our interest here lies in exploring how to *statically* typecheck executable grammars.

```
interface AbstractGrammar<R, E> extends ExecutableGrammar <?> {
  ExecutableGrammar<E> expression();
  ExecutableGrammar<R> returnStat();
  ExecutableGrammar<Token> hat();
}
class ExampleGrammar1<R, E> extends ExecutableGrammar<R> {
  type This = AbstractGrammar<R, E>;
   public ExecutableGrammar<Token> hat() {...};
   public ExecutableGrammar<List<Token>> expression(){...}
   public ExecutableGrammar<(Token, E)> returnStat() {
          return hat().seq(expression());
}
class ExampleParser1 extends ExampleGrammar1<ReturnStat, ExpressionAST>
                implements AbstractGrammar<ReturnStat, ExpressionAST> {
   public ExecutableGrammar<ExpressionAST> expression(){...}
   public ExecutableGrammar<ReturnStat> returnStat() {
     return super.returnStat().wrapper(
       {Token hat, ExpressionAST expr =>
            ReturnStat(expr)}
       )
  }
}
class ReturnStat {
   public ReturnStat(ExpressionAST e) {...}
}
```

In our first formulation, the grammar is represented by class ExampleGrammar1, which has one type parameter for each production in the grammar. To keep our example short, we have only two non-lexical productions, and two type variables. A realistic grammar would include dozens of productions, with dozens of type variables.

We also define an interface, AbstractGrammar, which declares methods corresponding to each production. It too declares a type parameter for every syntactic production.

The type of **this**, This in ExampleGrammar1 is declared to be an instance of AbstractGrammar. The type variables are passed through; ExampleGrammar1 does not depend on the return types of the parsers it creates, so these type variables play the role of wildcards. Naming them explicitly will allow us to typecheck the subclass that creates the AST, ExampleParser1.

The subclass instantiates ExampleGrammar1 with actual type parameters corresponding to the AST class created for each production. The actual return type for each production in ExampleGrammar1 is a function of the type parameters to the class. Consequently, we can derive an accurate type for **super** within ExampleParser1. Using this type, one can accurately typecheck the subclass. The subclass is still not a subtype of the superclass, but it is a subtype of the declared self type of its superclass; such subclasses are legal in

our hypothetical language.

The burden of declaring methods corresponding to every production in a separate interface, explicitly declaring a type variable for every production in that interface and in the class implementing the grammar, and instantiating both these types is prohibitive. A grammar for even a small language like Smalltalk can have over 30 productions, requiring more than 30 type variables to be introduced. Virtual types mitigate this only slightly.

Below we illustrate an approach that eases the notational burden somewhat. We can typecheck the superclass against a self type that returns parsers with an unknown return type, using the wildcards construct of Java 5 [19]. We use type selection on the type **this** in the return type ExampleGrammar1.returnStat() to make that type vary according to the return type actually declared by the expression() method in subclasses. This enables us to obtain an accurate type for **super** in ExampleParser1. Selection on **this** provides us implicitly with the necessary type parameterization, allowing this approach to scale to real grammars where the previous one does not.

```
interface AbstractGrammar extends ExecutableGrammar<?> {
  ExecutableGrammar<?> expression();
  ExecutableGrammar<?> returnStat();
  ExecutableGrammar<Token> hat();
}
class ExampleGrammar1 extends ExecutableGrammar<this.expression.returnType> {
  type This = AbstractGrammar;
  public ExecutableGrammar<Token> hat() {...};
  public ExecutableGrammar<List<Token>> expression(){...}
   public ExecutableGrammar<(Token, this.expression.returnType)> returnStat() {
            return hat().seq(expression());
}
class ExampleParser1 extends ExampleGrammar1
                  implements AbstractGrammar {
   public ExecutableGrammar<ExpressionAST> expression(){...}
   public ExecutableGrammar<ReturnStat> returnStat() {
      return super.returnStat().wrapper(
        {Token hat, ExpressionAST expr =>
              ReturnStat(expr)}
        )
  }
}
```

It is clear that we cannot express the separation of grammar and processing properly without relying on dynamic typing in the absence of a very sophisticated type system. Incorporating such type systems in a general purpose language without undue complexity is a difficult challenge. Pluggable type systems [5] may be a good fit for such situations. A well designed pluggable type system should enable one to tailor a type system that could be used to typecheck executable grammars, without burdening the language as a whole.

7.2 Performance

Parser combinators are not noted for high speed. Nevertheless, our experience has been positive. Our very first implementation was adequate in most circumstances. We work in an incremental programming environment, so code is usually compiled one method at a time, and methods tend to be short. Thus, even though each method is parsed on every keystroke (for syntax coloring), response was usually immediate. However, in methods larger than 25 lines, one could detect a delay. Methods containing long strings or comments also exhibited performance pathologies. Small adjustments resulted in an improvement of a factor of 3 in performance on average, and eliminated pathologies.

We have chosen to use the PEG alternation combinator [9], which commits to the first successful choice, reducing backtracking. This limitation has very rarely been a problem. We could easily add another combinator that implemented true alternation as in BNF.

For PEGs, we should be able to produce memoizing packrat parsers [8] that parse in linear time. However, we have not yet felt a need to do so.

The reflective facilities of Smalltalk and Newspeak make it possible to dynamically optimize grammars, either at the time they are constructed or during parse time. Nor is there a need to commit to a single optimization strategy. For example, grammars could support a method packrat which would produce an equivalent memoizing parser if possible, and fail if the grammar was unsuitable (perhaps because it used context sensitive or non-deterministic combinators). In cases where memory consumption is a concern, we might apply other optimizations, e.g., those used by Tedir [10].

7.3 Left Recursion

Left recursive grammars are a problem for classical parser combinator based approaches. For example

```
expr = expr + expr | expr * expr | id
```

In such cases one has to refactor the grammar. In practice this is not a significant issue. The problem can be addressed by any of several design choices, including an explicit fix combinator, or support for higher level operator definitions such as

```
expression =  identifier |
              (binaryOperators:{ {#+. #-.}{ #*. #/} {^. !}}  on: expression)
```

where the binaryOperators:on: method takes a list of lists of operators as its first parameter. Each sublist gives a set of operators at a given precedence

level, from lowest to highest. The base case is given as the second parameter.

Another possible solution is for the parser to refactor itself dynamically to eliminate left recursion. Dynamic refactoring is relatively easy in Smalltalk-like languages, thanks to the rich reflective system and the absence of mandatory typing.

7.4 Context-sensitive Grammars

In principle, parser combinators can be used to parse context-sensitive grammars as well. In a functional setting, monadic parser combinators [11] are necessary to achieve such behavior, but in an imperative language, parsers can communicate shared state. Hence we see no serious difficulty supporting context sensitive parsing, though we have not yet had a need to implement one.

8 Related Work

A great deal of work on parser combinators has been done in the context of functional programming. A comprehensive survey is beyond the scope of this paper. Parsec [13] is one of the most highly evolved parser combinator libraries. Parsec's differs from our framework in that it returns multiple results for the BNF alternation operator. Correspondingly, its error reporting strategy seeks to collect all errors on all possible parsing paths. Parsec provides excellent performance. However, it is not clear how modular separation of grammar and processing would be achieved using Parsec or any other functional parser combinator library we are aware of.

Object oriented frameworks for parser combinators have been implemented in Java (e.g., [12]) and in Scala [15,14]. Like Newspeak, Scala supports operators as methods, mixins and representation independence. It is thus unsurprising that the Sparsec framework is is quite similar to ours in its overall structure. Scala addresses the problem of delayed evaluation of mutually recursive productions using call-by-name. The call-by-name mechanism relies on mandatory typing to coerce expressions into closures. Scala's type system cannot fully statically typecheck examples where grammar and processing are separated.

Andrew Black developed a parser combinator framework in Smalltalk [2]. That framework was designed to be as similar as possible to a monadic parser combinator framework in Haskell for pedagogical reasons. It does not support the separation of the grammar from processing.

Tedir [10] is a system designed for high performance, dynamically modifiable parsing. Packrat parsing [8,9] is a technique for high performance parsing

of deterministic, context free grammars.

Another dimension for comparing related work is programming language design. The language features we have capitalized upon to express executable grammars are:

(i) Dynamic typing
(ii) Binary selectors
(iii) Closures
(iv) Introspection
(v) Inheritance
(vi) Mixins
(vii) Representation independence

As far as we are aware, the only language apart from Newspeak that support all these features is Self. Scala has all but (i), but typechecking can be relaxed through pattern matching. Smalltalk has all but (vi) and (vii), and can represent executable grammars a bit more verbosely than Newspeak, with occasional code duplication. In mainstream languages such as Java, only (iv) and (v) are present, and one cannot express the design shown here without falling back on dynamic checking.

9 Conclusions

We have identified the notion of *executable grammars* and presented a framework for their construction. Executable grammars are defined by the following characteristics:

- An executable notation that is almost entirely free of solution-space artifacts; it should be very close to BNF.
- The ability to modularly separate the grammar from any processing, enabling the grammar to serve as a shared executable specification used by a variety of language processing tools.

Our framework supports these attributes by implementing parser combinators in Newspeak, a new language of the Smalltalk family, leveraging key features of the language: dynamic typing, methods with operator syntax, closures, reflection, mixin based inheritance and representation independence.

Dynamic typing is an attractive alternative to the extremely sophisticated and verbose type discipline needed to encode executable grammars in the presence of mandatory static typing.

Acknowledgements

Martin Odersky suggested the error handling strategy described in section 6. Special thanks also to Eliot Miranda, Christian Plesner-Hansen and Alex Buckley for detailed discussions on some of the ideas in this document. Others who have contributed insights include Peter von der Ahé, Andrew Black, Erik Ernst, Neel Krishnaswami, Christophe Grand, Philip Milne, Adriaan Moors, Stephen Pair, Doaitse Swierstra, Philip Wadler, Jim White and semi-anonymous people who posted comments on my blog [4].

References

[1] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in Strongtalk, 2002. Invited paper, ECOOP Workshop on Inheritance.

[2] Andrew Black. Personal communication.

[3] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the emerald system. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 78–86, November 1986. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[4] Gilad Bracha. Room 101. Blog at gbracha.blogspot.com.

[5] Gilad Bracha. Pluggable types, March 2003. Colloquium at Aarhus University. Slides available at http://bracha.org/pluggable-types.pdf.

[6] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, October 1990.

[7] Gilad Bracha and David Griswold. Extending Smalltalk with mixins, September 1996. OOPSLA Workshop on Extending the Smalltalk Language.

[8] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*, pages 36–47, September 2002.

[9] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 111–122, January 2004.

[10] Christian Plesner Hansen. An efficient, dynamically extensible ELL parser library, May 2004. Masters Thesis.

[11] Graham Hutton and Erik Meijer. Monadic parser combinators. 8:437–444, July 1998.

[12] Jparsec library. Available at http://jparsec.codehaus.org/.

[13] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[14] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser Combinators in Scala. Technical Report CW491, Department of Computer Science, K.U. Leuven, 2007. Under preparation. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.abs.html.

[15] Martin Odersky. Scala by example, January 2007.

[16] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[17] Simon Peyton Jones (editor). Haskell 98 language and libraries, the revised report, December 2002. Available at http://www.haskell.org/onlinereport/.

[18] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Killian, and Carrie Wilpolt. An introduction to trellis/owl. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 9–16, November 1986. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[19] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97116, December 2004. Special issue: OOPS track at SAC 2004, Nicosia/Cyprus, http://www.jot.fm/issues/issue_2004_12/article5.

[20] David Ungar and Randall Smith. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1987.

[21] Philip Wadler. How to replace failure by a list of successes. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.