A New Life for Legacy Language Definition Approaches?[*]

Mikhail Barash

Bergen Language Design Laboratory, University of Bergen,
N-5020 Bergen, Norway
mikhail.barash@uib.no

Abstract. When syntax of software languages is communicated, context-free grammars are a *lingua franca*. They define structure of syntax, but cannot express static semantics. The paper gives an overview of other successful models (attribute, two-level, parsing expression, conjunctive, Boolean grammars) in relation to defining software languages. Author's model—grammars with contexts—can naturally express that "a valid identifier is an identifier that was declared before", and was used to define static semantics of a small typed language. This paper discusses what prevents the practical use of the model in SLE and states open problems for further research.

Keywords: Boolean grammars · Grammars with contexts · Parsing expression grammars · Syntax definition · Language workbenches

1 Attempts to Specify Algol

After Chomsky introduced phrase structure grammars in 1950s, this model quickly became a standard for definition of syntax of programming languages. Grammar rules of the form $A \to \alpha$ convey the idea of defining a construct A as a sequence of strings represented by α. Indeed, consider a rule of the form `VariableDecl` $\to$ "`var`" `ident` "`;`". This rule states that whenever there is a keyword `var` followed by an identifier that, in its turn, is followed by a semicolon, then this sequence of strings represents a variable declaration. Though being surprisingly simple, this mechanism allows defining syntax of modern programming languages [23] (in a form of BNF notation [46,49] that is equivalent to context-free grammars) and is still used as a *lingua franca* when describing syntax [48].

Soon after context-free grammars had been introduced, it became apparent that their expressive power is limited. In 1962, Floyd has shown [16] that Algol is not a context-free language and thus cannot be defined by a context-free grammar. Floyd considered the following Algol program, in which the only identifier

[*] Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

is symbol `x` repeated n times.

$$\texttt{begin real } \underbrace{\texttt{x}\cdots\texttt{x}}_{n}\texttt{;}\underbrace{\texttt{x}\cdots\texttt{x}}_{n}\texttt{:=}\underbrace{\texttt{x}\cdots\texttt{x}}_{n}\texttt{; end}$$

For this program to be valid, number n should be the same in all its three occurrences. Under certain encoding, such programs can be represented as strings of the form $a^n b^n c^n$, for some symbols a, b, c; these strings form a well-known non-context-free language. Nevertheless, context-free grammars are capable of defining the structure of such programs: indeed, as required by Algol's syntax, statements are separated by semicolons, there is an identifier in the left-hand side of the assignment operator, sequence of statements is framed with correctly spelled keywords `begin`/`end`, etc. Below is an example of a program that satisfies these conditions.

```
begin real x; xx:=xxx; end
```

This program will be considered correct despite containing undeclared identifiers.

Context-free grammars define *structural syntax* of languages, but cannot express any facts about their static semantics (what is called *context conditions*). It is impossible to specify in a context-free grammar that every identifier should be declared before use, or that the number of formal parameters and actual arguments in a function call should agree. Context-sensitive grammars (initially proposed by Chomsky to describe syntax of natural languages) are powerful enough to define these conditions, but they are equivalent to nondeterministic linear-bounded Turing machines, in which the "nonterminal symbols", meant to represent syntactic categories, could be freely manipulated as tape symbols. No parsing algorithms with polynomial time complexity exist for such grammars[1].

Subsequent research was done in two main directions: to embed "checking actions" into rules of a context-free grammar, and to come up with an entirely new grammatical model. The first approach led to attribute grammars [25] and syntax-directed translation. In a grammar, every rule is associated with semantic actions, which, for example, may use symbol tables to look up whether a certain variable has been previously declared.

```
VariableDecl  ::=  "var" ident ⟦symbTable.safeAdd($1);⟧ ";"
```

In this example, method `safeAdd` could check whether an identifier with the same name (the name of the "current" identifier is referred to by `$1`) has been declared and, if so, raise an exception. Otherwise, the new identifier is added to the symbol table. Such essentially *ad hoc* techniques are still used in industry-level compiler compilers [18,40].

The other direction of research was to develop a reasonable (that is, efficiently parsable) model, which would be able to specify the desired context conditions. Of many such attempts, two-level grammars by van Wijngaarden received some

[1] It is worth noting here context-sensitive parsing algorithms by Kuno [27] and Woods [47] that avoid producing several equivalent derivations of the input string—which is an issue with other parsing algorithms for context-sensitive grammars.

attention: they were used to formally specify syntax and static semantics of Algol 68 [44,43]. Unfortunately, two-level grammars soon turned out to be Turing-complete: this makes impossible any practical parsing algorithms for them.

2 Logics and Order for Static Semantics

Parsing expression grammars by Ford [17] introduce ordering of rules in a context-free grammar[2]: the choices are tried in order and the first one to succeed is used to parse the string. This is useful in disambiguating constructs like *if-then-else* statements.

```
IfStmt ← “if” Expr “then” Stmts “else” Stmts /
         “if” Expr “then” Stmts
```

This rule matches a conditional statement in a way that the optional *else* clause always binds to the innermost *if*. Without priority of rules, the rule would lead to dangling else ambiguity.

Parsing expression grammars also provide predicates for Boolean conjunction (`&`) and negation (`!`). The following rules define nested comments in Pascal [20, p. 509].

```
      Comment ← “(*” CommentedText* “*)”
CommentedText ← Comment / !“*)” .
```

A comment consists of an opening token “`(*`”, some commented text, and a closing token “`*)`”. This construct is non-trivial to recognize because the commented text may contain symbols “`*`” and “`)`”, but not the sequence “`*)`”. Moreover, comments may again contain comments: `(* a:=b; (* comment *) *)` is a correct statement. This construct can be recognized by the given rules: the idea is that `CommentedText` is either a complete comment or any symbol (expressed as “`.`” in parsing expression grammars) provided that it does not start with the string “`*)`” (this is expressed by the negation predicate `!`“`*)`”). Similarly to negation, conjunction in parsing expression grammars is used as a mechanism of lookahead.

```
BulletList ← &BulletSymb List
      List ← Item* !BulletSymb
```

These rules define the structure of bullet lists in Markdown. Rule for `BulletList` first verifies whether the first element of a list starts with a bullet symbol and only then it tries to recognize the string according to the rule for `List`. In the second rule, `!BulletSymb` expresses that the list cannot end at a position where there is still a bullet symbol to be consumed.

[2] Recent work on extending parsing expression grammars towards context-sensitive parsing includes, for example, principled stateful parsing [29] and parser combinators [28].

A systematic study of Boolean operations in grammars led to conjunctive [37] and Boolean grammars [34,42,26,31]. In such grammars, a rule $S \to A \ \& \ B$ defines all strings produced both by A and B, and a rule $S \to A \ \& \ \neg B$ defines all strings produced by A but not produced by B at the same time.

$$\texttt{ValidIdentifier} \to \texttt{ident} \ \ \& \ \ \neg\,\texttt{Keyword}$$
$$\texttt{Keyword} \to \texttt{var} \mid \texttt{if} \mid \ldots$$

These rules specify in a most natural way that an identifier cannot coincide with a keyword.

Conjunctive grammars can define sophisticated non-context-free languages [32], for example, copy language with central marker $\{wcw \mid w \in \Sigma^*\}$, $c \notin \Sigma$. This language abstracts the condition of having each identifier declared before use, as shown below.

$$\texttt{int}\ \underbrace{\texttt{amount}\overbrace{\texttt{; if (hasBonus)}}^{c}\texttt{amount}}_{wcw\ (w=\text{“amount”})}\ \texttt{+= 100;}$$

The main parsing algorithms for context-free grammars, including cubic-time general parsing algorithm, Earley's algorithm, recursive descent and Generalized LR, have been extended to the case of conjunctive and Boolean grammars [36,35,34], have the same time complexity, and are implemented in a prototype parser generator [33].

A Boolean grammar was constructed to specify syntax and static semantics (including scoping rules) of a programming language [38]. This was apparently the first such specification by an efficiently parsable grammatical model. Because conjunction and negation operators work on entire strings, rather than merely being a lookahead mechanism, the mentioned grammar is quite knotty [38].

$$\begin{aligned}&\texttt{this-func-not-declared-here} \to \\ &\quad \texttt{different-func-name} \mid \\ &\quad \texttt{same-func-name} \ \& \ \texttt{different-number-of-args}\end{aligned}$$

Moreover, the programming language only had one data type and no approach of how to implement type checking was suggested.

3 A New Life for Cross-references and Scoping

Boolean predicates `&` and `!` in parsing expressions grammars are, in fact, *positive* and *negative lookahead* predicates, respectively. In the rules for bullet lists in Markdown, predicate `&BulletSymb` succeeds if the next symbol of the input is a bullet, and predicate `!BulletSymb` succeeds if the next symbol is not a bullet. Neither of the predicates consume any input: they are only used to check the lookahead symbols in the input, and those lookahead symbols can be regarded as the *right context* of a string [10,13,22,24]. Drawing upon both parsing expression grammars and Boolean grammars, *grammars with contexts* [5] provide a built-in mechanism to specify what left and right contexts should be.

$$\texttt{ValidIdentifier} \to \texttt{ident} \ \& \ \lhd \textit{it was declared before}$$
$$\texttt{ValidIdentifier} \to \texttt{ident} \ \& \ \rhd \textit{it will be declared later}$$

These two informal rules state that whenever an identifier is used in a program, its declaration should appear either to its left ($\lhd$) or to its right ($\rhd$).

Consider the following fragment of a program in an assumed C-like language.

$$\underbrace{\texttt{... int f()\{}\boxed{\texttt{int}}\texttt{ms,sec,}\overbrace{\boxed{\texttt{min}}\texttt{; ... return 60*}\boxed{\underline{\texttt{min}}}}^{\text{copied string } wcw\text{, with } w=\texttt{min}}}_{\text{extended left context } (\trianglelefteq) \text{ of use of identifier } \texttt{min} \text{ (underlined)}}\texttt{;\}}$$

To ensure that identifier `min` used in the return statement (underlined) is declared, one can verify whether its left context contains a function header (`int f {}`), keyword "`int`", other identifiers (`ms, sec`), a comma, and the *declaration* of identifier `min`, followed by any other constructs (`; ... return 60 *`), all the way up to the *use* of `min` itself. This can be expressed in a grammar with contexts almost verbatim [6].

$$\begin{aligned}\texttt{ValidIdentfier} \to \texttt{ident} \ \& \ \trianglelefteq\, &\texttt{Functions} \\ &\texttt{FuncHeader "int" Identifiers CopiedString}\end{aligned}$$

This rule finds the substring between two positions in the input: before the declaration of an identifier and after its use. To include the use of the identifier into this substring, a so called *extended* left context $\trianglelefteq$ is used (that is, extended context of an identifier is its left context concatenated with that very identifier). After the desired substring has been found by the rule, it remains to check whether it forms a copy language wcw (copy language can be defined by a conjunctive grammar [37]).

The standard restriction that forbids redeclaration of identifiers can be now expressed by the following rules [6].

$$\texttt{IntegerDeclaration} \to \texttt{"int" InvalidIdentifier}$$
$$\texttt{InvalidIdentifier} \to \texttt{ident} \ \& \ \neg\, \texttt{ValidIdentifier}$$

It also becomes possible to distinguish between types of identifiers: the rule for a valid identifier breaks up into several rules, one for each type in the language.

$$\begin{aligned}\texttt{Valid}\underline{\texttt{Int}}\texttt{Identfier} \to \texttt{ident} \ \& \ \trianglelefteq\, &\texttt{Functions} \\ &\texttt{FuncHeader } \underline{\texttt{"int"}} \texttt{ Identifiers CopiedString}\end{aligned}$$

The only difference between these rules is in the keyword that should occur in the left context of an identifier use.

$$\begin{aligned}\texttt{Valid}\underline{\texttt{Bool}}\texttt{Identfier} \to \texttt{ident} \ \& \ \trianglelefteq\, &\texttt{Functions} \\ &\texttt{FuncHeader } \underline{\texttt{"bool"}} \texttt{ Identifiers CopiedString}\end{aligned}$$

Because identifiers are now distinguished according to their type, it makes sense to embed type checking into a grammar with contexts.

$$\begin{aligned}\texttt{Assignment} &\to \texttt{ValidIntIdentifier}\ \text{“=”}\ \texttt{IntExpr}\\ &\ \texttt{ValidBoolIdentifier}\ \text{“=”}\ \texttt{BoolExpr}\end{aligned}$$

These rules state that a variable of a certain type can only be assigned an expression of the same type.

Syntax and static semantics of a small typed programming language has been defined by a grammar with contexts [6]. That grammar specifies standard context conditions, such as: conditional expression of `while` loop is of type `bool`; number and types of formal parameters and actual arguments in function calls agree; type of expression in a `return` statement is the same as the returning type of a function; and others. By a set of rather sophisticated rules, the grammar also specified scopes of visibility: identifiers are visible in the block where they are defined, and in all its inner blocks.

Grammars with right contexts can be used in a way similar to parsing expression grammars—as a lookahead mechanism.

$$\texttt{BulletList} \to \trianglerighteq\ \texttt{BulletSymb anything}\ \ \&\ \ \texttt{List}$$

The extended right context $\trianglerighteq$ `BulletSymb anything` corresponds to the positive lookahead predicate `& BulletSymb` in a parsing expression grammar discussed earlier.

Several parsing algorithms, including cubic-time general parsing algorithm [5,41], (linear time) recursive descent [7], and Generalized LR [3], have been successfully extended to grammars with contexts. These algorithms have the same time complexity as their prototypes, and have been implemented in a parser generator [9,3].

4 Can Grammars with Contexts Be Used in Practice?

Grammars with contexts can define arbitrary cross-references within a string [4]. They have been used to specify an abstract programming language where identifiers can be declared before or after their use (example motivated by function prototypes in C) [4].

Ability to define cross-references is what makes grammars with contexts applicable to software languages. On a practical side, this ability is also distinctive in parser-based language workbenches [14], such as Eclipse Xtext [15,11]. Definition of a language in Xtext starts with writing a grammar in a metalanguage that is similar to that of ANTLR [39,40] and supports cross-references.

```
  Variable : 'var' name=ID ';' ;
Assignment : [Variable] '=' Expression ';' ;
```

The first rule defines what a variable declaration is: it is keyword `var` followed by an identifier and a semicolon. The identifier is to be “stored” in feature `name` associated with this rule[3]. In the second rule, `[Variable]` specifies a reference to an existing `Variable`, in particular, to its feature `name` (in the default setting, references are associated with the feature called `name`)[4].

This is conceptually very similar to grammars with contexts, where `ValidVariable` would be used to specify the idea of what `[Variable]` expresses in Xtext. There is, however, an important difference between the two approaches. To perform static checking, Xtext bases on high-level programming language code, which is either generated automatically by Xtext or is written later by a user. In contrast, grammars with contexts are purely a syntactic formalism, and after a language is defined by a grammar, no additional code is required to further define its static semantics. This clearly has its advantages: a language is defined solely by a very formal mechanism, and correctness of this definition can be proven in a formal way.

Definition of a typed programming language by a grammar with contexts [6] is heavily based on the copy language *wcw* that is used to check cross-references. This language is defined in a non-trivial manner, and would require a substantial modification to accommodate possible changes to how identifiers are defined in a language. However, for small sublanguages of larger programming languages (for example, a sublanguage of arithmetical expressions), it might make sense to use pure grammatical models to achieve higher degree of certainty about the correctness of the definition [30].

Can the formalism of grammars with contexts be made more approachable? Answering this question might involve implementing an Xtext-like language workbench based on these grammars (all necessary parsing algorithms exist [8,3], they were proven to have decent time complexity, and are implemented in a prototype parser generator [9]) and, most probably, introducing a simplified "front-end" formalism [43,21] that would be then translated to a grammar with contexts. For example, in such a formalism, concepts of a language can be annotated with tags ("suffix numbers" [43, p. 37] [2]) to define cross-references.

```
     NewIdent : ident   &  ! DeclaredIdent
DeclaredIdent : ident.1  &  < 'var' ident.1
```

In the rule for `DeclaredIdent`, both the ordinary (`ident.1`) and contextual (`<` ... `ident.1`) occurrenes of `ident` have the same tag to express that these identifiers should be identical. That is, this rule expresses that "a declared identifier is any identifier that was declared before".

[3] Xtext creates a model from a grammar using Eclipse Modeling Framework [19]. In a simplified setting, rules become classes (instances of `EClass`) and features of rules become fields of those classes. The model is then populated during the parsing, essentially resulting in an AST that can be further analyzed or transformed by the user.

[4] The reference is made to an instance of `EClass Variable`; if such an instance does not exist, an error is reported. If the instance exists, the reference is associated with the value of its field `name`.

Devising an adequate and convenient formalism on top of grammars with contexts is a challenging task. If this task is solved then implementing a language workbench based on these grammars would be a matter of technique.

Acknowledgements

The author is thankful to Alexander Okhotin and anonymous reviewers for their comments on earlier versions of the paper.

References

1. A. V. Aho, *Indexed Grammars – An Extension of Context-Free Grammars*, J. ACM 15(4). 647–671. 1968.
2. *American National Standard COBOL*. American National Standards Institute. 1975.
3. M. Barash, A. Okhotin, *Generalized LR Parsing Algorithm for Grammars with One-Sided Contexts*, Theory Comput. Syst. 61(2). 581–605. 2017.
4. M. Barash, A. Okhotin, *Two-sided context specifications in formal grammars*, Theor. Comput. Sci. 591. 134–153. 2015.
5. M. Barash, A. Okhotin, *An extension of context-free grammars with one-sided context specifications*, Inf. Comput. 237. 268–293. 2014.
6. M. Barash, *Programming language specification by a grammar with contexts*, NCMA 2013. 51–67.
7. M. Barash, *Recursive descent parsing for grammars with contexts*, SOFSEM 2013. Local Proceedings II. 10–21. 2013.
8. M. Barash, *Contexts in Recursive Descent Parsing*, TUCS Technical Reports 1151. 2015.
9. M. Barash, *Parser generator with grammars with right contexts*. 2012. Available at: `http://users.utu.fi/mikbar/gwrc/`.
10. M. E. Bermudez, K. M. Schimpf, *Practical Arbitrary Lookahead LR Parsing*, J. Comput. Syst. Sci. 41:2. 1990. 230–250.
11. L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, Packt Publishing. 2016.
12. N. Chomsky, *A Note on Phrase Structure Grammars*, Information and Control 2(4). 393–395. 1959.
13. K. Culik II, R. S. Cohen, *LR-Regular Grammars - an Extension of LR(k) Grammars*, J. Comput. Syst. Sci. 7:1. 1973. 66–96.
14. S. Erdweg, T. van der Storm, M. Voelter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, *Evaluating and comparing language workbenches: Existing results and benchmarks for the future*. Comput. Lang. Syst. Struct. 44. 2015. 24–47.
15. M. Eysholdt, J. Rupprecht, *Migrating a large modeling environment from XML/UML to Xtext/GMF*, SPLASH/OOPSLA 2010. 97–104. 2010.
16. R. W. Floyd, *On the nonexistence of a phrase structure grammar for ALGOL 60*, Commun. ACM 5(9). 483–484. 1962.

17. B. Ford, *Parsing expression grammars: a recognition-based syntactic foundation*, POPL 2004. 111–122. 2004.
18. *GNU Bison*. Available at: `https://www.gnu.org/software/bison/`.
19. R. C. Gronback, *Eclipse Modeling Project – A Domain-Specific Language (DSL) Toolkit*, Addison-Wesley. 2009.
20. D. Grune, C. J. H. Jacobs, *Parsing Techniques - A Practical Guide*. Springer. 2008.
21. Q. T. Jackson, *Efficient formalism-only parsing of XML/HTML using the s-calculus*, SIGPLAN Notices 38(2). 29–35. 2003.
22. S. Jarzabek, T. Krawczyk, *LL-Regular Grammars*, Inf. Process. Lett. 4:2. 1975. 31–37.
23. L. C. L. Kats, E. Visser, G. Wachsmuth, *Pure and Declarative Syntax Definition: Paradise Lost and Regained*, SIGPLAN Not. 45(10). 2010.
24. S. M. Kearns, *Extending Regular Expressions with Context Operators and Parse Extraction*, Softw., Pract. Exper. 21(8). 787–804. 1991.
25. D. E. Knuth, *The Genesis of Attribute Grammars*, Attribute Grammars and their Applications. 1990. 1–12.
26. V. Kountouriotis, Ch. Nomikos, P. Rondogiannis, *Well-founded semantics for Boolean grammars*, Inf. Comput. 207(9). 945–967. 2009.
27. S. Kuno, *A context sensitive recognition procedure*, In: Mathematical linguistics and automatic translation, Rep. NSF-18, The Computation Lab., Harvard U., Cambridge, Mass. 1967.
28. J. Kurs, J. Vraný, M. Ghafari, M. Lungu, O. Nierstrasz, *Efficient parsing with parser combinators*. Sci. Comput. Program. 161. 57–88. 2018.
29. N. Laurent, K. Mens, *Taming context-sensitive languages with principled stateful parsing*. SLE 2016. 15–27.
30. R. Lämmel, *Grammar Testing*, ETAPS 2001. 201–216.
31. A. Megacz, *Scannerless Boolean Parsing*, Electr. Notes Theor. Comput. Sci. 164(2). 97–102. 2006.
32. A. Okhotin, *Conjunctive and Boolean grammars: The true general case of the context-free grammars*, Computer Science Review 9. 2013. 27–59.
33. A. Okhotin, *Whale Calf, a Parser Generator for Conjunctive Grammars*, CIAA 2002. 213–220.
34. A. Okhotin, *Boolean grammars*, Inf. Comput. 194(1), 19–48. 2004.
35. A. Okhotin, *Generalized LR Parsing Algorithm for Boolean Grammars*, Int. J. Found. Comput. Sci. 17(3). 629–664. 2006.
36. A. Okhotin, *Recursive descent parsing for Boolean grammars*, Acta Inf. 44(3-4). 167–189. 2007.
37. A. Okhotin, *Conjunctive Grammars*, J. of Aut., Lang. and Comb. 6(4). 519–535. 2001.
38. A. Okhotin, *On the existence of a Boolean grammar for a simple programming language*, AFL 2015.
39. T. J. Parr, R. W. Quong, *ANTLR: A Predicated LL(k) Parser Generator*, Softw., Pract. Exper. 25:7. 1995. 789–810.
40. T. Parr, K. Fisher, *LL(*): the foundation of the ANTLR parser generator*, PLDI 2011. 425–436. 2011.
41. M. Rabkin, *Recognizing Two-Sided Contexts in Cubic Time*, CSR 2014. 314–324. 2014.
42. A. Stevenson, J. R. Cordy, *Parse views with Boolean grammars*, Sci. Comput. Program. 97. 59–63. 2015.
43. J. C. Cleaveland, R. C. Uzgalis, *Grammars for Programming Languages*, Elsevier. 1977.

44. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, R. G. Fisker, *Revised Report on the Algorithmic Language ALGOL 68*, Acta Inf. 5. 1–236. 1975.
45. A. van Wijngaarden, *The Generative Power of Two-Level Grammars*, ICALP 1974. 9–16.
46. M. H. Williams, *A Flexible Notation for Syntactic Definitions*, ACM Trans. Program. Lang. Syst. 1982. 113–119.
47. W. A. Woods, *Context-sensitive parsing*, Commun. ACM 13:7. 1970. 437–445.
48. V. Zaytsev, *Grammar Zoo: A corpus of experimental grammarware*, Sci. Comput. Program. 98. 28–51. 2015.
49. V. Zaytsev, *BNF Was Here: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions*, SAC 2012.