CYBERNETICA

Institute of Information Security

Implementing Oberon0 Language with Simpl DSL Tool

Margus Freudenthal

T-4-18 / 2013

The research reported here was supported by:

1. the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS
2. Estonian Doctoral School in Information and Communication Technology

Implementing Oberon0 Language with Simpl DSL Tool

Margus Freudenthal

23rd February 2013

Abstract

This report describes submission to the LDTA 2011 tool challenge, created with Simpl DSL tool.

1 Introduction

In 2011, the Workshop on Language Descriptions, Tools and Applications (LDTA) issued a tool challenge [LDT11] that aimed to compare different language tools. The participants completed the same tasks and reported on the results. This technical report describes the tool challenge entry that was implemented using the Simpl DSL toolkit [FP11].

1.1 Description of the Task

Detailed description of the implementation task can be found at the challenge home page [LDT11]. In short, the task was to create an implementation of Oberon0 as described by Niklaus Wirth in his compiler construction book [Wir96] (see Section 1.2 for notes on the Oberon0 language). The task was divided into several modules, according to language levels and implemented functionality. The reason for this was that one of the goals of the challenge was to see if the language tools can produce modular implementations that support language evolution.

Table 1 lists the language levels and table 2 lists the functional tasks. To tie the two aspects together, a set of implementation artifacts (see table 3 for list) was defined. Each artifact represented a module that was implemented based on previous modules and that could be run separately.

Table 1: Language levels used in task description

Level	Description
L1	Oberon0 with primitive types, simple expressions, and assignment statements
L2	L1 with Pascal-style *for* loop and *case* statement
L3	L2 with support for procedures
L4	L3 with support for arrays and records

Table 2: Functional tasks

Task	Description
T1	Parsing and pretty-printing the Oberon0 program
T2	Name analysis – binding the name uses to their declarations and reporting the errors
T3	Type analysis – checking type correctness of the program and reporting the errors
T4	Source-to-source transformation – lifting the nested procedures to top level and performing other transformations, such as expressing complex language constructs in terms of simpler ones
T5	C code generation – translating the Oberon0 program to ANSI C

1.2 Notes on the Oberon0 Language

The task language for the challenge was Oberon0, a simplified version of the Oberon language described in Niklaus Wirth's book "Compiler Construction" [Wir96]. In addition, language level L2 amended the basic Oberon0 with *for* and *case* constructs taken from the main Oberon language [Wir88].

Table 3: The challenge artifacts

Artifact	Language	Tasks	Comments
A1	L2	T1-2	Core language with pretty-printing and name analysis
A2a	L3	T1-2	A1 with added support for procedures
A2b	L2	T1-3	A1 with type checking
A3	L3	T1-3	Composition of A2a and A2b
A4	L4	T1-5	A3 with support for arrays and records, source-to-source transformation, and code generation

Oberon0 is a simple imperative language belonging to Modula-2 and Pascal family. It contains all the basic building blocks, such as integer, boolean, array and record types; variables; expressions; assignment, conditional and iteration statements; and procedures that support both by-value and by-reference parameter passing. Figure 1 shows an example Oberon0 program.

During the challenge, it was discovered that the book's description of Oberon0 and the reference implementation are incomplete and/or contradictory. The reference implementation imposed constraints that were not present in language description. For example, boolean constants can only be valued with constants *TRUE* or *FALSE*. It is not possible to use other boolean expressions as constant values. Additionally, the nested procedures did not use nested scope as initially expected, but instead only variables defined inside procedure were visible inside it. This also reduced the value of the procedure lifting task as there was no need to handle variables that are defined in outer scope. In the end, some open issues in the language definition were were simply decided by the participants. This definition was encoded in test suite consisting of 355 Oberon0 programs and the expected results. The test suite contained both positive and negative (e.g., parse errors, type errors) examples.

2 Simpl DSL Toolkit

Simpl is a toolkit mainly targeted at implementing domain-specific languages (DSLs) in an enterprise setting. The aim is to use DSLs in systems that are

```
MODULE Factorial;

VAR
    n, fact: INTEGER;

PROCEDURE Fact(n: INTEGER; VAR result: INTEGER);
    VAR
        i: INTEGER:
    BEGIN
        result := 1;
        FOR i := 1 TO n DO
            result := result * i
        END
    END Fact;

BEGIN
    Read(n);
    Fact(n, fact);
    Write(fact);
    WriteLn
END Factorial.
```

Figure 1: Example Oberon0 program

built in a popular language (Java, C#) using a framework that dictates the overall architecture of a system and where the DSL program is just one module in the large system. In particular, the ability to embed DSL programs and DSL implementations into a larger system is one of the main design goals of Simpl. For a more thorough analysis of technical requirements for embeddable DSL tools and review of existing DSL tools based on these requirements, see [Fre10].

In order to be embeddable, a DSL toolkit should consist of two separate parts. One, "non-visual" part contains the core of the DSL implementation: parser, program checker, code generator, etc. that can be embedded into a larger system. The other, "visual" part contains (possibly integrated) environment for editing and managing DSL programs. Secondly, the non-visual part of the DSL implementation should not make any assumptions on how the system is implemented. In particular, the non-visual part must not have dependencies on the visual part and must not assume that the DSL implementation is a top-level program or function in the system.

Simpl is designed to follow these non-functional requirements while providing maximal usability. The current design philosophy of Simpl is to reuse existing tools to minimize the amount of new tools and programming languages the developer must learn. For example, instead of developing a new language for expressing program transformations, Simpl relies on the programming language Scala. In addition to Scala, Simpl builds on the ANTLR parser generator [PQ94], Eclipse IDE platform, and IDE Meta-Tooling Platform (IMP) [CFS07]. The main rationale for selecting these particular tools is that they are mature, have good quality and are distributed under open source licenses. Tools that make up the non-visual part of the DSL implementation have few dependencies and can easily be embedded (and can coexist with other DSL tools). From the integration point of view, the main restricting choice is using Eclipse as the IDE platform – if the DSL user wants to use IDE developed via Simpl, she must install Eclipse. In this case, we chose the most popular platform.

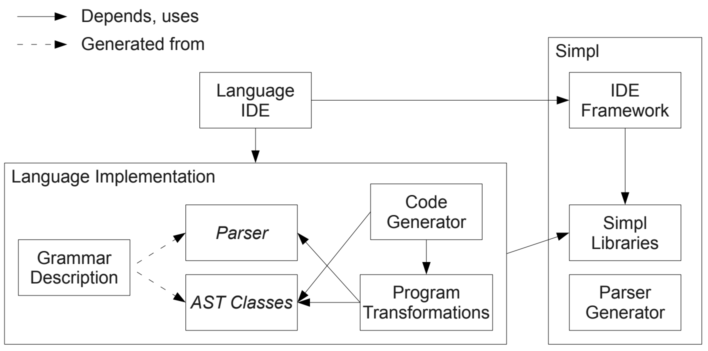


Figure 2: Architecture of a DSL implemented with Simpl. Components with captions in italic are automatically generated.

For the DSL developer, Simpl provides a parser generator, libraries for pretty-printing and code generation, and an IDE framework. Figure 2 shows the main components of a DSL implementation created with Simpl. The first part is the non-visual language implementation that can be embedded into a bigger system. Development of a new DSL starts with grammar description that specifies both the context-free grammar of the DSL and the classes for representing the abstract syntax tree (AST) of a DSL program. The Simpl parser generator takes the grammar description as input and produces a parser and the AST classes. The (optional) program transformation component takes as input an AST and checks or transforms it. The code generator converts the preprocessed AST to text. The second part of the DSL implementation is the language IDE. It builds on the Simpl IDE framework and the non-visual part of the language implementation.

3 Scanning and Parsing

In Simpl, the grammar description is used to generate both the parser and Scala case classes that are used to express the abstract syntax tree (AST) of the DSL program. By default, the AST class and attribute names are derived from rule names. The developer can add annotations in the context-free grammar to modify the generated AST classes. As an example, Figure 3 shows set of rules for parsing the Oberon0 *CASE* statement. Identifier before an equals sign names the attribute in the case class that is used for representing the AST of the child. Figure 4 shows the Scala case classes that were generated from the example rules. The attribute types are automatically derived from the types of called rules. If an attribute refers to rule(s) that can be called multiple times, then the type of the attribute will be a list. For example the *clauses* attribute in the *CaseStatement* class is typed as list because there can be more than one clause in one case statement. Using the same attribute name several times in a rule is allowed if the rule calls assigned to this attribute have compatible types.

```
CaseStatement:
    "CASE" expr=CompExpr "OF"
        clauses=CaseClause ("|" clauses=CaseClause)*
        ("ELSE" elseClause=StatementSequence)?
    "END";

CaseClause:
    items=CaseConstant ("," items=CaseConstant)* ":"
        stmt=StatementSequence;

CaseConstant: begin=SimpleExpr (".." end=SimpleExpr)?;
```

Figure 3: Grammar rules for parsing the Oberon0 *CASE* statement.

```
case class CaseStatement(
    var expr: Expression,
    var clauses: List[CaseClause],
    var elseClause: StatementSequence) extends Statement

case class CaseClause(
    var items: List[CaseConstant],
    var stmt: StatementSequence)

case class CaseConstant(
    var begin: Expression,
    var end: Expression)
```

Figure 4: AST classes for expressing the Oberon0 *CASE* statement.

Simpl allows the developer to modify the AST nodes by using return expressions. Return expressions can specify the return type of a rule and/or a Scala expression that is used to compute the actual AST node returned by the rule. In the Oberon0 implementation, return expressions were used to make the AST more regular and uniform. Figure 5 shows two examples. The first rule ensures that the use of parentheses does not introduce additional wrapping of the AST nodes. The second rule makes all the unary expressions use a common AST class *Unary(operation, expression)* so that they can be uniformly treated in the processing code.

```
ParenExpr returns Expression {expr}: "(" expr=CompExpr ")";

NotExpr
    returns Expression {Unary(UnaryOp.Not, expr)}
    : "~" expr=Factor;
```

Figure 5: Example rules using return expressions. Rule *ParenExpr* does not wrap the inner AST node. Rule *NotExpr* returns a manually created AST class.

In addition to return expressions, Simpl supports adding attributes and methods to AST classes. For example, the code for parsing identifiers is shown in Figure 6. This code modifies the normally generated *Id* class by adding attributes for storing reference to the definition of this identifier, pre-calculated constant value if this identifier evaluates to a constant, and information whether the identifier is a by-ref procedure parameter. These attributes are filled out and used during name, type checking and code generation.

```
terminal Id {
    // What does this identifier point to?
    var ref: Id = null
    // If this is constant, then what is its value?
    var constVal: Option[Int] = None
    // Used for "VAR" parameters.
    var byRef: Boolean = false
    def isByRef = byRef || ((ref ne null) && (ref.byRef))
} : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9')*;
```

Figure 6: Rule for parsing identifiers. The generated class *Id* is amended by adding properties and methods to it.

4 Name Analysis

Simpl itself has no direct support for name analysis, therefore the name analysis for the Oberon0 language was written in Scala. The implementation is quite straightforward: it walks the AST (using depth-first traversal) and for each identifier fills out its *ref* attribute (points from identifier, such as variable reference, to declaration of this identifier. See also Figure 6). During the walk, we maintain an environment: a mapping from identifier names to declarations.

5 Type Checking

Like name analysis, type checking was implemented in Scala. Since name analysis is run before type checking, there is no need to use environments (mappings from names to types). Instead, the type checker first attaches type information to all the declared identifiers (variables, constants, procedure parameters). It then propagates the type information to all expressions and procedure calls and checks the types of operation arguments and procedure call parameters. According to the task definition, checking for negative array sizes is also part of type checking, and thus the type checker computes values of all constant expressions and stores the values in the AST.

6 Source-to-Source Transformation

In order to simplify C code generation, the Simpl implementation transforms Oberon0 programs to simplify Oberon0 language constructs that have no direct equivalent in the C programming language. The first transformation lifts nested procedures to top level. The second transformation replaces *CASE* statements with a sequence of *IF* statements.

6.1 Procedure Lifting

Procedure lifting uses links created during the name analysis step and performs in-place modifications of the AST. The task is accomplished in three steps. The first step consists of scanning the AST and locating all the nested procedures. Figure 7a shows an example Oberon0 program with the inner procedure highlighted. In the second step, all the nested procedures are lifted to top level. The new name is formed by concatenating names of outer and inner procedures. If the concatenation does not produce unique name, a number is added to the name. Figure 7b shows the results of this step with the lifted procedure highlighted. Finally, the AST is scanned and all identifiers that reference the lifted procedures are renamed to reflect the new names. Since the name analysis links identifiers to concrete objects (procedures, variable definitions, etc.), renaming of the procedures leaves the reference information intact. Figure 7c shows the final result with the renamed identifier highlighted.

6.2 Simplifying *CASE* Statements

The Oberon0 *CASE* statement is more powerful than the *switch* statement in C as it has support for ranges. In order to simplify code generation, we transform Oberon0 *CASE* statements to a series of *IF-ELSE* statements (see Figure 8 for an example transformation). For each *CASE* statement, we generate new variable for storing the *CASE* condition. Because the results of

```
PROGRAM Foo;
    PROCEDURE Bar;
        PROCEDURE Baz;
        END Baz;
    BEGIN
        Baz
    END Bar;
BEGIN
    Bar
END Foo.
```

(a)

```
PROGRAM Foo;
    PROCEDURE BarBaz;
    END BarBaz;

    PROCEDURE Bar;
    BEGIN
        Baz
    END Bar;
BEGIN
    Bar
END Foo.
```

(b)

```
PROGRAM Foo;
    PROCEDURE BarBaz;
    END BarBaz;

    PROCEDURE Bar;
    BEGIN
        BarBaz
    END Bar;
BEGIN
    Bar
END Foo.
```

(c)

Figure 7: Three stages of procedure lifting: locating nested procedures (a), lifting the procedures (b), and renaming the procedure references (c).

CASE simplification will go straight to C code generation, we can optimize by not making the generated identifier *gen_1* a legal Oberon0 identifier. In this way we do not have to ensure that it does not clash with any identifiers in this scope.

```
CASE foo + bar OF
      1 : Write(1)
    | 3..4 : Write(34)
ELSE
    Write(-1)
END;
```

(a)

```
VAR gen_1: INTEGER;
...
gen_1 := foo + bar;
IF gen_1 = 1 THEN
    Write(1)
ELSE IF gen_1 >= 3 AND gen_1 <= 4 THEN
    Write(34)
ELSE
    Write(-1)
END;
```

(b)

Figure 8: *Oberon0 CASE* statement before (a) and after transformation (b).

7 Code Generation

When implementing C code generation, we decided to decouple translating Oberon0 language constructs to C from outputting properly formatted C code. This allowed us to concisely express the transformation between the Oberon0 AST to a C AST without concerning ourselves with pretty-printing of C code. First, we created Scala case classes for expressing the abstract syntax of C. Next, the Oberon0 AST was transformed to a C AST. Because the more complicated Oberon0 constructs were previously simplified (see the previous subsection), the translation was quite straightforward. In the last step, the C AST was transformed to a string using the pretty-printing library included in Simpl (it is based on Philip Wadler's Haskell library [Wad98]).

Figure 9 illustrates the code generation process with an example procedure for calculating factorials. Since Oberon0 has no functions, the result is returned in a *VAR* parameter. Whereas AST of the Oberon0 *FOR* statement (see Figure 9b) corresponds to Oberon0 syntax, the translated *for* statement (see Figure 9c) corresponds to C syntax (initialization and increment are statements, guard is an arbitrary expression). During the translation, all the identifiers are prefixed with underscore to prevent clashes with existing keywords.

8 Overview of the Results

8.1 Code Sizes

The Oberon0 implementation is composed of five different artifacts (A1, A2a, A2b, A3, A4). Each artifact either adds additional constructs to the language or adds additional features, such as type checking, procedure lifting or code generation. Table 4 shows the code sizes for the various artifacts and tasks. Code size is measured in lines of code; empty lines and comments were not counted. For counting lines of code, we used the program *cloc*[1] that was extended with support for Simpl grammar files. In the table, each row represents the size of a particular component:

[1]see http://cloc.sourceforge.net/

```
PROCEDURE Fact(
    n: INTEGER;
    VAR Res: INTEGER);
  VAR i: INTEGER;
BEGIN
  Res := 1;
  FOR i := 1 TO n DO
    Res := Res * i;
  END
END Fact;
```

(a) Oberon0 source before code generation.

```
ForStatement(
 Id(i),
 NumberLit(1),
 Id(n),
 null,
 StatementSequence(
  List(
   Assignment(
   Id(Res),
   Binary(*,Id(Res),Id(i))))))
```

(b) Oberon0 AST corrensponding to the highlighted *FOR* statement.

```
For(
 Assign(
  Id(_i,false),
  NumberLit(1)),
 Binary(
  <=,
  Id(_i,false),
  Id(_n,false)),
 Inc(_i,NumberLit(1)),
 Sequence(
  List(
   Assign(
    Id(_Res,true),
    Binary(
     *,
     Id(_Res,true),
     Id(_i,false))))))
```

(c) *FOR* statement translated to AST representing a C program.

```
void _Fact(int _n,int *_Res) {
  int _i;
  (*_Res) = 1;
  for (_i = 1; _i <= _n; _i += 1)
{
    (*_Res) = (*_Res) * _i;
  }
}
```

(d) C source generated from the function in sub-figure (a). The highlighted *for* statement corresponds to AST from sub-figure (c).

Figure 9: Code generation example: translating the procedure from Oberon0 to C.

- Parse – Simpl grammar file and any additional Scala code (such as custom classes for expressing binary operators and checks that numerical constants fit into 32 bits);
- Name – code for name analysis;
- Type – code for type checking and and constant inlining[2];
- Lift – code for lifting nested procedures to top level;
- Gen – code generator, including simplification of *CASE* statements, transforming Oberon0 AST to C AST, and pretty-printing C AST;
- Pretty – pretty-printing of Oberon0 code;
- Other – other supporting code, such as error handling, main functions, etc.

Table 4: Code sizes for different artifacts and components. Sizes are expressed as non-blank, non-comment lines of code.

Artifact	Parse	Name	Type	Lift	Gen	Pretty	Other	**Total**
A1	193	181					44	**418**
A2a	38	135					16	**189**
A2b			301				17	**318**
A3			92				17	**109**
A4	48	44	89	72	463	198	39	**953**
Total	**279**	**360**	**482**	**72**	**463**	**198**	**133**	**1987**

Except for A1, the artifacts are not self-contained in the terms of code. The artifacts reuse grammar files and Scala code. Figure 10 shows the dependency graph between the artifacts. The grammar files are reused by using include directives. For example, L3 grammar includes L2 grammar and overwrites production rules for declarations (by adding procedure declarations) and statements (by adding procedure calls). Services written in Scala, such as name analysis and type checking, were extended using inheritance. In general, the extension consisted in overriding *processStatement*, *processExpr* etc. methods and adding new case clauses to process additional kinds of statements or expressions.

[2]Constant inlining is used to check for negative array sizes.

$$\begin{tikzcd}
 & A4 \arrow[d] & \\
 & A3 \arrow[dl] \arrow[dr] & \\
A2a \arrow[dr] & & A2b \arrow[dl] \\
 & A1 &
\end{tikzcd}$$

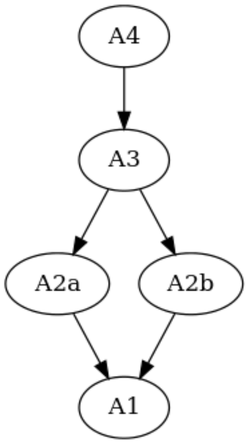
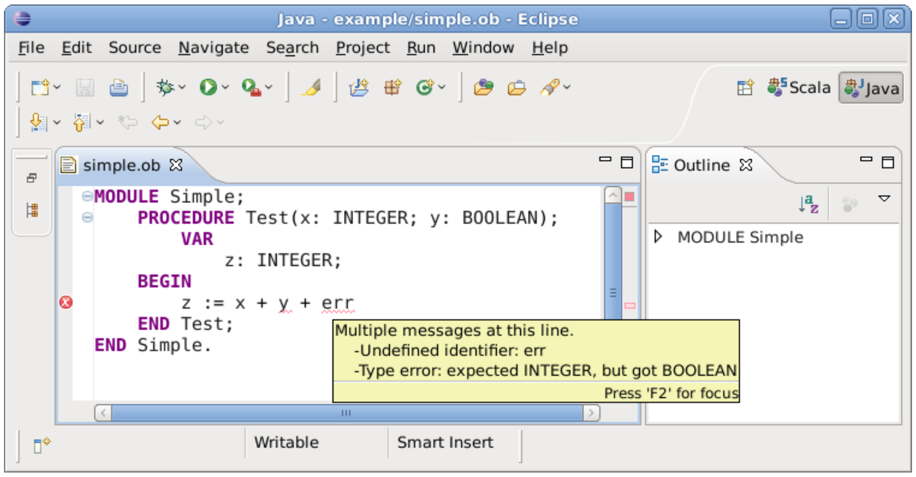


Figure 10: Dependencies between artifacts in the Oberon0 implementation.

In addition to artifacts mandated in the challenge, we used Simpl to implement a basic IDE for the Oberon0 language (see Figure 11 for example screenshot). The IDE provided syntax highlighting, error highlighting, outline view, hyperlinking, code folding, and occurrence marking. The total code size for the IDE module was 106 lines, some of which was filler. Most of the Oberon0-related functionality was encapsulated in a 33-line class that contained IDE services. The IDE code was based on the existing code checker (artifact A4) and did not involve references to Eclipse APIs.


Java - example/simple.ob - Eclipse
File Edit Source Navigate Search Project Run Window Help
Scala
Java
simple.ob
Outline
MODULE Simple
Writable
Smart Insert

```
MODULE Simple;
    PROCEDURE Test(x: INTEGER; y: BOOLEAN);
        VAR
            z: INTEGER;
    BEGIN
        z := x + y + err
    END Test;
END Simple.
```

Multiple messages at this line.
-Undefined identifier: err
-Type error: expected INTEGER, but got BOOLEAN
Press 'F2' for focus

Figure 11: Screenshot of Oberon0 IDE

8.2 Observations

First, it should be noted that Simpl is not directly targeted at creating full-featured implementations of typical programming languages. Currently the focus is on domain-specific languages and quite simple code generators. Therefore, Simpl provided direct support for a subset of all the tasks contained in this challenge. In particular, we used Simpl to create a parser and class model for expressing AST of Oberon0 programs, and for pretty-printing the AST. Although not included in the challenge, we also used Simpl to create an IDE for the Oberon0 language.

Simpl currently uses ANTLR as a parser backend and therefore inherits the use of the *LL(k)* parsing algorithm. The *LL(k)* algorithm has difficulties expressing left-recursive grammar rules and thus parsing left-associative operators. This limitation means that operator precedence must be encoded in grammar rules[3] and it also results in cumbersome AST. The cumbersome AST can be worked around by using return expressions that reshape the AST nodes returned by the grammar rules. In the future, we plan to use a parser backend that does not have this restriction.

The Simpl grammar system is not specifically targeted at implementing modular grammars. It only supports simple inclusion mechanism with the ability to overwrite rules in the included grammars. In the challenge, this introduced minor code duplication. For example, in order to introduce a new kind of statement, one has to repeat the *Statement* rule with all the other preexisting statements. Overall, the modularity features were adequate for the current situation where we were dealing with language levels with increasing complexity and each following level only extended the language. However, for more complicated situations the current modularity features likely would not have sufficed.

9 Conclusion

In general, implementing the challenge with Simpl was a straightforward exercise. The grammar description was legible and the automatically generated AST classes worked well with processing code written in Scala. Simpl does

[3]In this challenge the grammar was already specified in this form.

not have support for implementing program checkers and program transformations, therefore these functions were written in straight Scala. In addition to the required tasks, we implemented an IDE for Oberon0. The IDE was based on the code of the challenge tasks and required a minimal amount of effort to create.

References

[CFS07] Philippe Charles, Robert M. Fuhrer, and Jr. Stanley M. Sutton. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 485–488, New York, NY, USA, 2007. ACM.

[FP11] Margus Freudenthal and David Pugal. Simpl: a Toolkit for Rapid DSL Implementation. In *Proceedings of 12th Symposium of Programming Languages and Software Tools*, Tallinn, Estonia, October 2011.

[Fre10] Margus Freudenthal. Using DSLs for developing enterprise systems. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, Paphos, Cyprus, 2010.

[LDT11] LDTA 2011 Tool Challenge. Available online at `http://ldta.info/2011/tool.html`, 2011.

[PQ94] Terence J. Parr and Russell W. Quong. ANTLR: a predicated-LL(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[Wad98] Philip Wadler. A Prettier Printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.

[Wir88] Niklaus Wirth. The Programming Language Oberon. *Softw., Pract. Exper.*, 18(7):671–690, 1988.

[Wir96] N. Wirth. *Compiler construction.* International computer science series. Addison-Wesley, 1996.