Modular Language Implementation in Rascal
—*Experience Report*—

Bas Basten, Jeroen van den Bos[f], Mark Hills[d], Paul Klint[a,c], Arnold Lankamp[e], Bert Lisser[a], Atze van der Ploeg[b], Tijs van der Storm[a,c], Jurgen Vinju[a,c]

[a] *Centrum Wiskunde & Informatica, Amsterdam, The Netherlands*
[b] *Chalmers University of Technology, Gothenborg, Sweden*
[c] *INRIA Lille - Nord Europe, France*
[d] *East Carolina University, NC, United States*
[e] *Xebia, The Netherlands*
[f] *Netherlands Forensic Institute (NFI), The Netherlands*

Abstract

All software evolves, and programming languages and programming language tools are no exception. And just like in ordinary software construction, modular implementations can help ease the process of changing a language implementation and its dependent tools. However, the syntactic and semantic dependencies between language features make this a challenging problem. In this paper we detail how programming languages can be implemented in a modular fashion using the RASCAL meta-programming language. RASCAL supports extensible definition of concrete syntax, abstract syntax and operations on concrete and abstract syntax trees like matching, traversal and transformation. As a result, new language features can be added without having to change existing code. As a case study, we detail our solution of the LDTA'11 Tool Challenge: a modular implementation of OBERON-0, a relatively simple imperative programming language. The approach we sketch can be applied equally well to the implementation of domain-specific languages.

Keywords: Language implementation, language engineering, language workbench, meta-programming, modularity, compiler generators

---

1. Introduction

Like all software, programming languages and programming language tools evolve. A case in point is the regular releases of new versions of mainstream programming languages like Java, Javascript, Python, and C#. Just like ordinary software, the implementation of such languages needs to evolve, to respond to new language versions, to changes in the environment, or to changes in the user requirements. Modular implementations can help ease the process of changing a language implementation and its dependent tools. However, the syntactic and semantic dependencies between language features make this a challenging problem.

An important trait of evolvable software is when its artifacts are "open for extension, but closed for modification" [1]. This entails that a software system can be extended with new features without having to change existing code.

In this paper we present an extended experiment in applying the open-closed principle to the implementation of a relatively simple, imperative language of moderate size used in compiler construction courses [2]. The case study involves implementing the concrete syntax, name analysis, type checking, code generation and formatting components for different variants of the language in a modular fashion.

The main contributions of this paper are:

- A case study of implementing OBERON-0 in RASCAL, which illustrates how to implement many aspects of a programming language in this language workbench.
- A demonstration how to structure these implementations in a modular and extensible fashion.

Organization. This paper is further organized as follows. First, we introduce the RASCAL language and its module system in Section 2. This provides the necessary background to the presentation of the OBERON-0 case study in Section 3. Based on the case study, we present results on the artifacts of the case study in Section 4 and discuss observations and directions for future work in Section 5. We then position RASCAL in the broader context of modular semantics, static analysis and language implementation in Section 6. The paper is concluded in Section 7.

2. Rascal, a DSL for meta-programming

2.1. Introduction

RASCAL[1] is a domain-specific language for meta-programming [3, 4] that has the ambition to be a one-stop-shop for all aspects of language design and implementation. It aims to be a platform that integrates and extends many existing approaches for language implementation (cf. Section 6). RASCAL has been designed to deal with software languages in the broadest sense of the word. Its application areas include not only the development of DSLs, compilers and IDEs, but also reverse engineering and reengineering of (legacy) software systems and software repository mining. RASCAL features powerful language constructs to make such meta-programming tasks more effective:

- Integrated syntax definitions: context-free grammars can be declared as part of ordinary RASCAL code; the concrete syntax trees defined by such grammars are first-class values, and non-terminals are first-class types.

- Built-in data types: apart from the standard data types (int, bool, string etc.), RASCAL supports sets, relations, maps, lists, algebraic data types (ADTs), concrete syntax trees, and source locations. Every value in RASCAL can be used in pattern matching. This includes concrete syntax trees produced by a grammar defined in RASCAL.

- Constructs for analysis and transformation: sets, relations, maps and lists can be used in comprehensions. The visit-statement allows for strategic traversal and rewriting of abstract or concrete syntax trees.

- A rich and growing ecosystem of libraries that provide facilities for visualization, statistical analysis, reading and writing existing data formats (JSON, XML, CSV, and others) and data repositories, and analysis and transformation of existing programming languages (Java, PHP, C#).

We have aimed to make RASCAL easy to learn and debug by adhering to well-known C/Java/C#-like syntax and familiar programming idioms (mostly standard and explicit control-flow, functional programming, and pattern-matching) and by providing a gradual learning path to more advanced features.

[1] `http://www.rascal-mpl.org`.

2.2. Rascal's Module System and Extensibility

Modularity in programming language descriptions has a long and rich history that we will sketch later in Section 6.

RASCAL's module system aims at providing modular and extensible definitions of syntax and semantics of a language of interest. It is easiest explained using an example. Consider the following module that defines the abstract syntax of a simple expression language and an interpretation function `eval`:

```
module Exp
data Exp = lit(int n) | add(Exp l, Exp r);

int eval(lit(n)) = n;
int eval(add(l, r)) = eval(l) + eval(r);
```

This module employs *pattern-based dispatch*: function definitions can be defined using signatures containing arbitrary patterns (e.g., `add(l, r)`) and functions are called based on the pattern that they match. Formal parameters of functions are thus generalized to arbitrary patterns, supporting a powerful way of overloading function with additional cases.

Rascal modules such as these are lexically closed: all function references in it are resolved within the current module or any of its imports. For instance, the recursive call to `eval` for addition, only sees the two cases defined in this module.

Consider writing another module to add expressions involving multiplication:

```
module ExpMul
import Exp;
data Exp = mul(Exp l, Exp r);

int eval(mul(l, r)) = eval(l) * eval(r);
```

In this module, the `Exp` data type is extended with a constructor for multiplication expressions. Within `ExpMul` it is now possible to create both additive, literal and multiplication expressions. In other words: the data type `Exp` in the context of `ExpMul` can be used as if the alternatives of `Exp` are merged with the alternatives defined here (i.e. `mul`).

Accordingly, the `eval` function is extended with the semantics of multiplication. However, because modules are lexically closed regarding function names, evaluating an addition expression with a nested multiplication in the context of this module leads to a dynamic error:

```
eval(add(lit(1), mul(lit(2), lit(3))));
// ⇒ ERROR: Called signature ‘eval(mul(...,...))‘ does not match
```

Although from the perspective of the current module, eval is defined for both addition and multiplication, when the definition for add recursively calls eval, the only available cases are for literals and addition.

To address this situation, RASCAL features another import directive, **extend**, to "open up" recursively defined functions defined in a module. As a result, the additional cases will participate in the recursion. So changing **import** to **extend** in the ExpMul module above will give the correct answer when evaluating nested multiplications.

Another RASCAL feature to make language definition functions like eval complete, even for arbitrary extensions to the Exp data type, are default definitions: `default int eval(Exp _) = -1;`. They act as a catch all rule when no others are applicable.

The module extension pattern is similar to how inheritance in object-oriented language works, and is similarly based on open recursion. Each module can be roughly considered to be a class and name resolution of (recursive) function calls is implicitly parameterized by *self*, which is the (extending) module where the initial call originated. The visibility of declarations can be controlled in a similar way.

The example above served to illustrate extension of algebraic data types and functions specified in a case-based fashion. The same pattern also applies to extension of concrete syntax definitions.

Modularization in RASCAL is tailored towards open extensibility but still lacks certain features (e.g., renaming of sorts and functions on import) or provides different mechanisms (e.g., parameterized types versus parameterized modules) compared to its ancestor language ASF+SDF [5]. Providing a formal account of Rascal's module system, however, remains an important direction for future work.

3. Case study: OBERON-0 implemented in RASCAL

3.1. Introduction

OBERON-0 is a relatively simple, imperative language designed by Niklaus Wirth and used in his book on compiler construction [2]. The LDTA'11 Tool Challenge consisted of implementing this language. The challenge distinguished four language levels; in each level five tasks needed to be completed. The four language levels are:

- L_1: Basic control flow statements, constant, type and variable declarations, assignments and expressions.
- L_2: Extension with FOR-statements and CASE-statements.
- L_3: Definition of (nested) procedures and procedure call statements.
- L_4: Support for arrays and records, including subscript and field selection expressions and assignment statements.

The five tasks are:

- T_1 (SYNTAX): Syntax analysis: (a) mapping source text into a parse tree (parsing); and (b) mapping a parse tree back to text (pretty printing, formatting).
- T_2 (BIND): Name analysis: bind each use of a name to its definition.
- T_3 (CHECK): Type checking: checking that all language constructs are used in a type-correct manner.
- T_4 (DESUGAR): Desugaring: mapping language extensions to previous language layers.
- T_5 (COMPILE): Compilation: compile an OBERON-0 program into C code.

In addition to these five given tasks, however, we have implemented the following additional tasks:

- T_6 (EVAL): An interpreter for OBERON-0.
- T_7 (TOJAVA): Compilation to Java source code (defined for L_4).
- T_8 (TOJVM): Compilation to byte code (defined for L_4).
- T_9 (CFLOW): Control-flow graph extraction and visualization.

Finally, OBERON-0 programs can be edited in an automatically generated editor with basic IDE features, including syntax highlighting and code folding. The source code of our complete OBERON-0 implementation can be found at `https://github.com/cwi-swat/oberon0/tree/ldta-only`.

3.2. The Challenge in RASCAL

RASCAL allows programming languages to be implemented in a modular fashion, through the simultaneous extension of syntax definition (grammars), algebraic data types (ADTs) and functions operating on data conforming to these data types. The OBERON-0 implementation consists of four layers each corresponding to a language level. Each new layer extends the previous layer.

Level L_1 represents the base language: all relevant syntax, data types and functions are introduced here. In the subsequent levels any of these types and functions may have to be extended. For instance, in L_2, where the FOR and CASE statements are introduced, the grammar and AST definition of L_1 are extended with new alternatives to deal with these constructs. Similarly, the functions for name analysis, type checking and formatting are extended. The compiler to C, however, does not require extension; instead FOR and CASE statements are desugared to L_1 constructs.

Extension in RASCAL works through the **extend** construct (explained earlier in Section 2.2). For grammars and ADTs the definitions of the extended module and the extending module are simply merged. Function extension, on the other hand, requires that functions are defined using pattern-based dispatch, for each case in an algebraic data type. Consider, for instance, the following rule implementing the check function for type-checking WHILE statements:

```
set[Message] check(whileDo(c, b)) = checkCond(c) + checkBody(b);
```

where checkCond and checkBody return a set of messages and + represents set union. The complete implementation of the check function consists of multiple such rules, one for each AST constructor. Such definitions may be distributed over different modules to construct extension hierarchies using the **extend** mechanism. For instance, in L_3 where procedure definitions and procedure calls are introduced, the check function is extended by adding an implementation rule dispatching on the AST constructor for procedure calls. In the OBERON-0 implementation, this pattern was used to modularize name analysis, type checking, formatting, and compilation.

For syntax and abstract syntax the mechanism is similar. In higher layers (i.e., L_2, L_3, and L_4), extension modules add alternatives to both grammars and ADTs. In L_3 this mechanism is also used to extend the symbol table data type to deal with nested scopes.

```
syntax Statement
  = assign: Ident ":=" Expression
  | ifThen: "IF" Expression "THEN" {Statement ";"}+  ElsIfPart* ElsePart? "END"
  | whileDo: "WHILE" Expression "DO" {Statement ";"}+ "END"
  | skip: ;

syntax ElsIfPart = "ELSIF" Expression condition "THEN" {Statement ";"}+ body ;
syntax ElsePart = "ELSE" {Statement ";"}+ body;
```

Listing 1: Concrete syntax definition of OBERON-0 statements in L_1

3.3. Scanning and parsing

RASCAL's syntax definitions are backed by a scannerless variant of the GLL parsing algorithm [6]. Scannerless parsing solves the problem of modular lexical syntax. To deal with (lexical) ambiguities, RASCAL has built-in support for longest-match and keyword reservation. Each language level may introduce additional reserved keywords, that are not reserved in previous levels. For instance, the "FOR" keyword, introduced in L_2, is not reserved in L_1.

Grammars are annotated with constructor names to obtain automatic mapping of concrete syntax trees to ASTs. The RASCAL standard library function implode converts a concrete syntax tree into an AST, given an ADT definition of the abstract syntax. Implode uses the *reified ADT* as a recipe. Reified types are reflective value representations of RASCAL types. The information in the reified type is used by implode to decide how a concrete tree must be mapped to an AST. For instance, some lexical concrete syntax nodes are mapped to (native) integers if this is dictated by the abstract syntax ADT.

The statement syntax of OBERON-0 L_1 is defined using the grammar shown in Listing 1. Alternatives can be labeled with a constructor name that corresponds to a constructor in the abstract syntax, which (for statements) is defined as shown in Listing 2. In the abstract syntax ordinary RASCAL lists are used to represented optional or repeated entities defined in the concrete syntax by regular operators: N?, N+, N*, {N sep}+, {N sep}* for any nonterminal N and literal separator sep.

If a syntax production is not labeled (like ElsIfPart and ElsePart), implode will assume that these concrete nodes do not represent explicitly typed AST nodes, and are inlined. For instance, the optional ElsePart becomes a **list**[Statement] in the abstract syntax. Alternatively, implode can map con-

```
data Statement
  = assign(Ident var, Expression exp)
  | ifThen(Expression condition, list[Statement] body, list[ElsIf] elseIfs, list[Statement] elsePart)
  | whileDo(Expression condition, list[Statement] body)
  | skip();

alias ElseIf = tuple[Expression condition, list[Statement] body];
```

Listing 2: Abstract syntax definition of OBERON-0 statements in L_1

crete nodes to anonymous tuple nodes. This is illustrated in the case of ElsIfPart, which is mapped to a tuple containing a condition expression and list of statements, defined by the alias ElseIf (Listing 2).

The RASCAL parser annotates concrete syntax trees with source locations (a native datatype in RASCAL). The implode function propagates these locations to the AST so that meaningful error messages can be given during later processing. Additionally, implode annotates the AST with comment nodes so that a pretty printer can reinsert them.

For instance, imploding the parse tree of the expression "`a (* this is A *) + b`", produces the following AST:

```
Expression: add(
  lookup(id("a")[
      @location=|file:///exp.ob0|(0,1,<1,0>,<1,1>)
    ])[
    @location=|file:///exp.ob0|(0,1,<1,0>,<1,1>),
    @comments=()
  ],
  lookup(id("b")[
      @location=|file:///exp.ob0|(20,1,<1,20>,<1,21>)
    ])[
    @location=|file:///exp.ob0|(20,1,<1,20>,<1,21>),
    @comments=()
  ])[
  @location=|file:///exp.ob0|(0,21,<1,0>,<1,21>),
  @comments=(0:["(* this is A *)"])
]
```

The location annotations provide exact source locations of each AST node, listing filename, offset, length, begin and end line, and begin and end column. The comments annotation contains a map from positions inbetween production

```
lexical Comment = "(*" CommentElt* "*)";     lexical CommentChar
lexical CommentElt                              = ![*(]
  = CommentChar+ >> "*)"                        | [*] !>> [)]
  | CommentChar+ >> "(*"                        | [(] !>> [*];
  | Comment;
```

Listing 3: Grammar describing OBERON-0's nested comments

elements to comment strings. In the example, the comment "(* this is A *)" is positioned at 0, because it is the first layout position according to the syntax production for addition, right after the first operand.

Scannerless parsing means that there is no separate tokenization phase: both lexical and context-free syntax are described using the same grammar formalism. This makes it, for instance, trivial to support OBERON-0's nested comments, the grammar for which is shown in Listing 3. In essence, this is just context-free syntax, except that the **lexical** keyword indicates that no layout (i.e. spaces, tabs, comments and newlines) is allowed between symbols. The CommentChar non-terminal captures all characters which are not "*" or "(" using the negated character class ![*(]. The characters "*" and "(" are allowed, however, if they are not followed by ")" and "*", respectively. This look-ahead restriction is expressed using the follow restriction "!>>", which reads "must not be followed by". The positive follow restrictions ("must be followed by") on CommentChar+ ensures longest match on non-recursive CommentElts.

3.4. Formatting

Formatting in RASCAL consists of mapping (abstract) syntax trees to Box constructs [7]. The resulting Box expressions describe how elements should be laid out, e.g., horizontally, vertically, indented or aligned, which font-style to use, and how adjacent elements should be spaced relative to one another.

As an example, consider the formatting of the WHILE statement in Listing 4. Again this function uses pattern-based dispatch to match the AST constructor for WHILE statements. The result of the function is a Box expression (which is just an ADT in RASCAL). The result dictates that the WHILE and DO are both keywords (KW), and the condition c is formatted using the function exp2box. The L constructor injects string values into the Box data type. All three sub-boxes are laid out horizontally (H). Finally,

```
Box stat2box(whileDo(c, b)) = V([
  H([KW(L("WHILE")), exp2box(c), KW(L("DO"))])[@hs=1],
    I([V(hsepList(b, ";", stat2box))]),
  KW(L("END"))
]);
```

Listing 4: Formatting WHILE-statements

the horizontal spacing between the elements should be 1, as indicated by the annotation `@hs=1`. The body `b` of the WHILE statement should be indented one level (indicated by the `I` construct), and the statements should be placed vertically. The helper function `hsepList` is used to correctly combine separated lists where the separator (";") should be horizontally combined with an element; it is parameterized by a function to convert each element of the list to a Box expression (i.e. `stat2box`). Finally, the three sub-boxes are wrapped in a `V` box so that the header, body and END keyword are placed vertically.

A common problem with pretty-printing is the (re)insertion of parentheses in binary expressions according to the precedence and associativity rules of the grammar. In Rascal this can be solved using reified types. In Section 3.3, we described how `implode` used the reified type of the abstract syntax ADT to guide the mapping of concrete syntax trees to ASTs. In this case we proceed the other way around: we use the reified type of the *grammar* to obtain precedence and associativity information. Since all non-terminals are first-class types in Rascal, obtaining the reified type of a non-terminal gives us the complete grammar.

To illustrate how this works, consider the following formatting rule for multiplication expressions:

```
Box exp2box(p:mul(lhs, rhs)) =
  H([exp2box(p, lhs), L("*"), exp2box(p, rhs)])[@hs=1];
```

This rule lays out expressions horizontally. But instead of calling directly the unary function `exp2box` on `lhs` and `rhs`, there is an intermediate call to `exp2box` with two arguments; in both cases the first argument is `p` which is bound to the current expression using the capturing colon (:) in `p:mul(lhs, rhs)`. This parent expression is used to decide whether to insert parentheses or not:

```
Box exp2box(Expression parent, Expression kid) =
  parens(PRIOS, parent, kid, exp2box(kid), parenizer);

Box parenizer(Box box) = H([L("("), box, L(")")])[@hs=0];
```

In this example, the global constant PRIOS contains the precedence information obtained from the grammar. The parens function checks if the occurrence of kid directly below parent requires parentheses, and if so, it calls parenizer on the Box expression resulting from exp2box(kid). The function parenizer simply horizontally wraps the argument with parentheses. Different parenizer functions can be defined and called where appropriate, if different kinds of parentheses are required.

As described in Section 3.3, AST nodes are annotated with comments. Our current pretty printer, however, does not use these annotations to reinsert comments into the formatted output.

3.5. Name analysis

Name analysis consists of a traversal of the AST that performs two tasks at the same time:

1. Annotate use sites of variables, types and constants with their declarations.
2. Maintain a set of error messages, indicating problems such as undeclared identifiers.

These two concerns are implemented in functions bindModule, bindStat, bindExp etc. They carry around a NEnv environment which contains the names that are currently in scope. It is defined as an algebraic data type to allow extension later on: **data** NEnv = scope(**map**[Ident, Decl] env). If an identifier is encountered, it is looked up in the environment and, if found, the identifier is annotated using a Rascal annotation representing the declaration. Annotations can be attached to values of type **node**, which includes ADT values as well as concrete syntax trees. It has to be declared for a specific ADT type and can contain itself a value of an arbitrary type. The annotation of parse trees with source location information mentioned in 3.3 uses this same mechanism. Since all data is immutable, annotating a value returns a new value.

In order to both annotate ASTs and return a set of error messages the bind family of functions returns a tuple containing an AST node and a set of error messsages. As an example, consider the binding of the IF-statement, shown in Listing 5. The function heavily uses Rascal's destructuring assignment to simultaneously replace parts of the AST and update the errs variable. For instance, the first statement invokes bindExp to bind the condition of the IF-statement; the resulting annotated expression is inserted in place of the

```
tuple[Statement, set[Messages]] bindStat(s:ifThen(c, b, eis, e), NEnv nenv, set[Message] errs) {
    <s.condition, errs> = bindExp(c, nenv, errs);
    <s.body, errs> = bindStats(b, nenv, errs);
    s.elseIfs = for (ei ← eis) {
      <ei.condition, errs> = bindExp(ei.condition, nenv, errs);
      <ei.body, errs> = bindStats(ei.body, nenv, errs);
      append ei;
    }
    <s.elsePart, errs> = bindStats(e, nenv, errs);
    return <s, errs>;
}
```

Listing 5: Binding analysis of the Oberon-0 IF-statement

```
default tuple[Ident, set[Message]] bindId(Ident x, NEnv nenv, set[Message] errs) {
  if (isVisible(nenv, x)) {
    return <x[@decl=getDef(nenv, x)], errs>;
  }
  if (x.name in {"TRUE", "FALSE"}) {
    return <x[@decl=trueOrFalse(x.name == "TRUE")], errs>;
  }
  return <x, errs + { undefIdErr(x@location) }>;
}
```

Listing 6: Binding analysis for identifiers

original condition (s.condition). The for-loop folds over the list of ELSIF's while at the same time (possibly) updating the set errs. The final result is a tuple containing the annotated IF-node (s) and the set of errors errs. Note that, although the (destructuring) assignments seem to suggest that parse trees are updated in-place, this is not the case. The bind functions thus return new, annotated ASTs. Persistent data structures used under the hood ensure that this is not inefficient.

The annotation of identifiers is implemented using the function in Listing 6. This function checks the name environment (nenv) to determine whether the identifier x is currently visible. If so, x is annotated with its definition (x[@decl=...]). If x represents one of the constants TRUE or FALSE, the reference is annotated accordingly. Otherwise, the identifier is undeclared and an error is produced, containing the source location of x. Note that the function is marked **default**, to support overriding this function in language level 3, when

```
set[Message] check(s:call(f, as)) {
  errs = {};
  if (!(f@decl is proc)) errs += { notAProcErr(f@location) };
  else {
    fs = (f@decl).formals;
    arity = ( 0 | it + size(ns) | formal(_, ns, _) ← fs );
    if (size(as) ≠ arity) errs += { argNumErr(s@location) };
    else {
      i = 0;
      for (frm ← fs, n ← frm.names) {
        errs += checkFormal(n, as[i], frm.hasVar);
        i += 1;
      }
    }
  }
  return ( errs | it + check(a) | a ← as);
}
```

Listing 7: Checking Oberon-0 procedure calls

(nested) procedures are introduced and the lookup semantics change.

3.6. Type checking

Similar to name analysis, type checking is a traversal of the AST, computing a set of error messages. Unlike in the case of name analysis, however, the AST is not annotated with further information; all required annotations are assumed to be set during name analysis. This simplifies type checking considerably, since all required information is local to a certain AST node.

As an example, consider the code to check procedure calls in Listing 7. There are three cases to consider. First, if the called name `f` is not declared as a procedure, an error message is added to the set `errs`. Second, if `f` is a procedure, but there is an arity mismatch, an error is produced as well. Third, if there's no arity mismatch, the actual arguments are checked against the formal parameters of the declaration of the procedure `f` using the function `checkFormal`. Finally, the actual arguments themselves are checked. The notation `( x | ... it | y )` is a reducing comprehension.

3.7. Source-to-source transformation

Rascal features built-in support for structure-shy traversal of data structures using the `visit` statement. Visit works like a traditional case state-

```
list[Statement] for2while(list[Statement] stats) {
  return innermost visit (stats) {
    case forDo(n, f, t, [], b) ⇒ forDo(n, f, t, [nat(1)], b)
    case forDo(n, f, t, [by], b) ⇒
          begin([assign(n, f), whileDo(leq(lookup(n), t),
            [∗b, assign(n, add(lookup(n), by))])])
  }
}
```

Listing 8: Desugaring OBERON-0 FOR-loops to WHILE-loops

ment, with the difference that cases are matched at arbitrary depth in a data structure. There are 6 builtin strategies to control the traversal order. Visited nodes maybe replaced, thereby rewriting the tree, similar to the traversal functions of ASF+SDF [8] and rewrite strategies of Stratego [9].

The desugaring of FOR-loops and CASE-statements both heavily depend on the visit-statement. For instance, the desugaring of the for-statements introduced in L_2 is shown in Listing 8. Note that the function for2while takes an arbitrary list of statements as argument, but the cases in the **visit** only mention cases of interest. Traversal inside arbitrarily nested statements is taken care of by **visit**. The **innermost** strategy furthermore applies the rewrite rules repeatedly until the argument tree stats does not change anymore.

The desugaring works by first normalizing FOR-loops without a BY-clause to FOR-loops with a BY-clause of 1 (nat(1)). Then, in the second case, FOR-loops are replaced by equivalent L_1 OBERON-0 nodes. Since a single FOR-loop is desugared to a sequence of statements, we use a temporary AST constructor, begin, to allow inserting multiple statements in place of one. The begin nodes are spliced (using the prefix ∗ operator) into their surrounding context in a later phase:

```
list[Statement] flattenBegin(list[Statement] stats)
  = visit (stats) { case [∗s1, begin(b), ∗s2] ⇒ [∗s1, ∗b, ∗s2] };
```

Note the use of associative list matching and splicing to flatten the list of statements b into the surrounding list.

The desugaring of the CASE-statement to nested IF-statements is implemented in a similar way.

3.8. Code generation

Code generation to C works in two phases. The first phase consists of a source-to-source transformation on OBERON-0 to make all identifiers unique and to lift all nested procedures to the top level [10]. The second phase prints such normalized OBERON-0 programs to flat C code.

Although it would have been possible to define an abstract syntax for C and then use RASCAL's Box formatting to obtain a textual representation suitable for compiling to machine code, we have instead opted for a simpler, more pragmatic approach using *string templates*. In RASCAL string literals can be interpolated with expressions, conditional statements (if) and loops (for, while, do-while). This makes string templates very convenient for generating code. Moreover, using the explicit margin markers (single quote '), such templates are convenient to write and the result will be automatically indented. For instance, Listing 9 shows the code to generate C code for OBERON-0's WHILE loops and IF statements. In both string templates the

```
str stat2c(whileDo(c, b)) = "while (<exp2c(c)>) {
                            '  <stats2c(b)>
                            '}";

str stat2c(ifThen(c, b, ei, ep)) = "if (<exp2c(c)>) {
                                   '  <stats2c(b)>
                                   '}<for (<ec, eb> ← ei){>
                                   'else if (<exp2c(ec)>) {
                                   '  <stats2c(eb)>
                                   '}<}>
                                   '<if (ep ≠ []){>
                                   'else {
                                   '  <stats2c(ep)>
                                   '}<}>";
```

Listing 9: Using auto-indenting string templates to generate C-code

statements within statement blocks will be indented relative to the margin. All whitespace to the left of the margin is discarded.

3.9. Interpretation

Interpretation is implemented using pattern-based dispatch to support extension. A fragment of the interpreter for statements is shown in Listing 10.

The function evalStat returns a State value which initially contains just a representation of the heap. The State data type is extended in L_3 to support input/output as well. The Env argument represents bindings of identifiers to memory addresses, types, or constant values. In L_3 environments are extended to support bindings to procedures.

```
State evalStat(assign(v, exp), Env env, State state) {
  state.mem = update(lookupAddress(v, env, state.mem), eval(exp, env, state.mem), state.mem);
  return state;
}

State evalStat(ifThen(c, b, eis, ep), Env env, State state) {
  if (evalCond(c, env, state.mem))  return evalStats(b, env, state);
  if (<ec, eb> ← eis, evalCond(ec, env, state.mem)) return evalStats(eb, env, state);
  return evalStats(ep, env, state);
}

State evalStat(whileDo(c, b), Env env, State state) {
  while (evalCond(c, env, state.mem)) state = evalStats(b, env, state);
  return state;
}
```

Listing 10: Fragment of the Oberon-0 interpreter in Rascal (L_1).

3.10. Java and JVM byte code generation

The compilation of Oberon-0 to Java source code uses string templates similar to the compilation to C (cf. Listing 9). Since Java does not support call-by-reference (needed for VAR parameters), an Oberon-0 program is first transformed to use an explicit stack for all variables (global and local) and procedure parameters. To support composite assignment (i.e. record to record and array to array) the program is transformed so that such assignments are represented using multiple atomic assignments. This program is then directly pretty printed to equivalent Java.

The compilation to JVM byte code also requires the normalization of Oberon-0 programs. It then constructs a Rascal value representing the JVM byte code. Using the JVM API included in Rascal's standard library, the code can be executed directly from within Rascal.

3.11. Control-flow extraction

Control-flow extraction consists of analyzing the syntactic structure of a program to obtain a graph that represents the flow of control between statements. Graphs are represented as (binary) relations between nodes; each node in a control-flow graph is identified by the source location (**loc**) of a statement or expression. The extraction function for WHILE-loops is shown in Listing 11.

First a choice node is created based on the condition of the loop. Then the entry and exit edges are created, going from the while-statement itself to the condition. If the body of the loop contains statements, edges are created from the choice node to the first statement in the loop-body, and from the last statement back to the choice node. If there are no statements, the choice node loops back to itself.

```
CFlow statementCFlow(w:whileDo(cond, body), CFlow cfl) {
  cfl.nodes[cond@location] = choice(cond@location, cond);

  cfl.entry += {<w@location, cond@location>};
  cfl.exit += {<w@location, cond@location>};

  if (body ≠ []) {
    cfl.succ += {<cond@location, head(body)@location>};
    cfl.succ += {<last(body)@location, cond@location>};
    cfl = statementListCFlow(body, cfl);
  }
  else {
    cfl.succ += {<cond@location, cond@location>};
  }
  return cfl;
}
```

Listing 11: Control-flow extraction for WHILE-loops

Extracted control-flow graphs can visualized by mapping them to a Figure, a data type included in Rascal's standard library for modeling interactive visualization [11]. An example of this is shown in Figure 1 on page 22.

4. Artifacts

The Oberon-0 implementation in Rascal consists of the modular implementation of each task for each language level (when appropriate). An

	T_1 Syntax	T_2 Bind	T_3 Check	T_4 Desugar	T_5 Compile	**Total**
L_1	244	162	146	–	60	612
L_2	80	44	26	39	–	189
L_3	73	117	39	–	106	335
L_4	90	67	53	–	48	258
Total	487	390	264	39	214	**1394**

Table 1: Overview of the structure and size (#SLOC) of the Rascal implementation of each task T_i, $i = 1, ..., 5$ for each Oberon-0 language level L_j, $j = 1, ..., 4$.

overview is shown in Table 1. Task T_1 includes the code for the grammar, AST data type, pretty printing rules and AST normalization. The code for T_2 consists of the scope data type and the bind functions. T_3 is covered by the check function. T_4 only applies to L_2 and performs the desugaring of FOR and CASE statements. Finally, T_5 contains the code for compilation to C and the lambda-lifting transformation to lift nested procedures and renaming identifiers [10].

The columns in Table 1 capture artifact dependencies, i.e., T_i depends on tasks $T_0, ..., T_{i-1}$. For instance, T_3 (Check) requires that name analysis has been applied to the AST. The row dimension indicates extension: each higher level (lower in the table) extends the previous level. The extension applies to syntax definition, algebraic data types, and functions.

As described in Section 3, we have implemented additional tasks T_6, ..., T_9. An overview of the structure and size of these tasks is shown in Table 2. The evaluator component includes data types to model the heap and environment. The special task $T'_{7,8}$ represents a normalization operation on Oberon-0 programs required for the compilation to Java and JVM bytecode (see Section 3.10). This normalization component transforms Oberon-0 programs to use an explicit stack and eliminates type and constant references. The control-flow visualization (CFlow) is divided in two parts: extracting a control-flow graph and transforming this graph to a Figure object, which can be rendered on screen [11].

5. Discussion

The experience developing Oberon-0 in Rascal has been generally positive. Nevertheless, there are some areas where we think Rascal could still

	T_6 EVAL	$T'_{7,8}$ NORMALIZE	T_7 TOJAVA	T_8 TOJVM	T_9 CFLOW	**Total**
L_1	211	–	–	–	121	332
L_2	–	–	–	–	71	71
L_3	167	–	–	–	36	203
L_4	117	743	78	116	–	1054
Total	495	743	78	116	228	**1660**

Table 2: Overview of the structure and size (#SLOC) of additional OBERON-0 tasks T_i, $i = 6, ..., 9$ for each language level L_j, $j = 1, ..., 4$.

be improved. Below we discuss what we think contributed to our positive experience and identify areas for further improvement.

Rascal's grammar formalism is very powerful and expressive. As a result the grammar can be written with an eye to the desired abstract syntax: essentially the grammar does not have to be factored in awkward ways to satisfy the parser. This makes defining the syntax of a language very declarative and modular. The Eclipse-based IDE, read-eval-print-loop (REPL) and built-in testing framework are very helpful in developing and debugging language implementations. Furthermore, RASCAL includes the tools AMBIDEXTER [12] and DRAMBIGUITY [13] for (conservatively) checking (non-)ambiguity, and for diagnosing ambiguous parse forests respectively. These tools have been instrumental in creating correct grammars for OBERON-0.

Although the AST could easily be automatically derived from concrete syntax trees, the current scheme of using the abstract ADT as a recipe to guide the "implosion" process allows additional flexibility. This allows, for instance, the implosion of lexical tokens (identifiers, integers etc.) to different RASCAL primitive types. It also supports skipping over irrelevant chain rules (injections) and flattening of nested lists. The feature that makes this kind of guided implosion work, is that RASCAL types can be inspected at runtime. This was also essential for letting the pretty printer insert parentheses when expressions need them. Since a RASCAL grammar defines types (nonterminals and productions), we were able to inspect the OBERON-0 grammar itself to derive the precedence relation.

The name and type analysis tasks are implemented by programming rather than by way of a declarative specification. The powerful pattern matching and traversal features and built-in data structures of RASCAL make

this quite easy. The approach of programming instead of declarative specification gives greater expressivity and flexibility but makes formal analysis of these tasks more difficult. By breaking a function into separate "rules" that dispatch on the relevant AST constructors both analyses as well as code generation could be modularized and extended in later language levels.

With respect to extension, however, we sometimes had to anticipate later extension of the code. The RASCAL **extend** mechanism is currently not powerful enough to *override* a previous definition and hence it is currently impossible to reinterpret an existing language construct. One example is making function definitions **default**, to allow later overriding. (e.g., bindId of Listing 6).

The only extensibility RASCAL currently caters for is *syntactic*: you may add another function alternative (e.g., bind, check) for the new construct, but you cannot revise an existing case. We are currently investigating how function definitions can override a previous definition handling the same case. This would allow calling the previous definition using a special keyword (similar to **super** in OO languages).

Another direction for improvement is concerned with how state and context information currently has to be manually threaded through recursive functions. A restricted form of dynamically scoped variables could eliminate a significant amount of boilerplate and scaffolding code. Such variables would work like function parameters, except that they are not explicitly passed around [14]. For instance, the bind function currently receives the current set of error messages and has to return tuples of an annotated AST and the new set of error messages. With dynamic variables, the set of error messages would be initialized as a local, but dynamically scoped, variable in the first call to bind. Each recursive invocation of bind would update that very same variable.

An open problem regarding the modularity of our implementation is that operations, such as type checking or compilation, might require that other operations, such as name analysis or desugaring, have been performed on the AST. Currently, this is not enforced by the type system for two reasons. First, annotations are not part of a data type. It is therefore impossible to require certain annotations to be present. Second, transformations of ASTs are often considered to be type preserving. Desugaring, for instance, transforms an AST of a certain type to another AST of the same type. As a result, whether such a transformation has been applied cannot be enforced.

The code generation to C is currently implemented using RASCAL's string templates. While very convenient and flexible, there is some overlap in func-

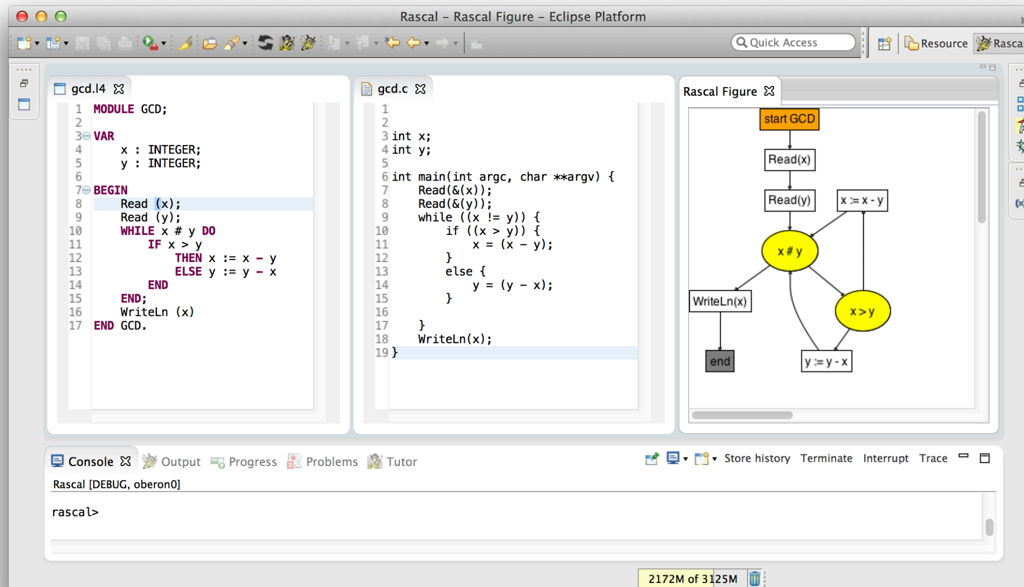

Figure 1: Screen shot of the Rascal Oberon-0 IDE showing editors with an Oberon-0 program (left), the generated C code (middle) and a control-flow visualization.

tionality with pretty printing using Box. It would be interesting to see if some of the formatting features of Box could be integrated into the string templates. More generally, we think a better integration of Box into the language as a whole would make the development of pretty printers much easier. This would involve a much closer integration with the concrete syntax features of Rascal and enabling, for instance, automatic handling of parenthesis insertion.

To conclude, our experience implementing Oberon-0 shows that Rascal is a suitable language for prototyping languages in a modular fashion, with relatively little effort. All five tasks across the four language levels have been implemented in under 1500 source lines-of-code. Finally, we have implemented an Oberon-0 IDE using Rascal's lightweight hooks into Eclipse. A screen shot of the Oberon-0 IDE is shown in Fig. 1

6. Related Work

Modularity in programming language descriptions has a long and rich research history. Modular language engineering has also received ample attention from researchers from diverse backgrounds. In this section we give a brief overview of related work that directly touches upon the case study and the proposed solutions we report on in this paper. The key observation is that various aspects of language syntax and semantics are addressed by different formalisms and techniques that cannot be easily combined and integrated. We also indicate how RASCAL fits into this picture.

Modular syntax. The Syntax Definition Formalism SDF [15] was one of the first formalisms to propose modular syntax definitions. The desire to compose grammars implies the need for modular lexers [16, 17] and parsers [18, 19]. Modular syntax has been applied, amongst others, in ASF+SDF [5], SDF2 [20], and Stratego [9]. With the same goals, but using different techniques, modular grammars were also introduced in TXL [21]. It is noteworthy that TXL provides an override mechanism for syntax rules that has never been introduced in SDF. Recent work on the semantics of modular grammar specification can be found in [22]. RASCAL builds upon the SDF tradition but extends it regarding notation, disambiguation mechanisms, and tight integration of abstract and concrete syntax trees. Outside the realm of general parsing, modularity of syntax has received attention in the context of Parsing Expression Grammars (PEGs) (e.g., [23]) and LALR parsing [24, 25].

Modular static semantics. Attribute grammars [26] are the classical method for expressing static semantics and various extensions have been proposed to make them modular; see for instance [27, 28, 29, 30, 31]. Integration of attribute grammars and functional languages has been proposed and applied with success [32, 33, 34]. Finally, Ruler is a modular system dedicated to programming type rules [35].

Modular dynamic semantics. For decades, dynamic semantics has been a focus of classical research on programming language semantics. Early approaches to semantics, such as denotational semantics [36], did not consider modularity. In subsequent approaches, e.g., monadic semantics [37], Action Semantics [38], an attempt was made to tackle modular aspects of dynamic semantics. Another effort along these lines is Modular Structural Operational Semantics [39]. Modular algebraic approaches to language definition are described in [40] and [41].

Tools and Language Workbenches. Combinations of the above techniques have been included in various tools, IDEs and language workbenches. See [42] for a recent survey of the state-of-the-art of language workbenches. Some notable examples are ASF+SDF Meta-Environment [43, 44], MPS [45, 46], and Spoofax [47]. Other relevant approaches to modular language implementation are described in [48], [49], [50] and [51]. RASCAL falls in this category of tools: it aims to be a one-stop-shop for language implementations, including support for syntax definition, static analysis, dynamic semantics and code generation. All language aspects, including tools and IDE extensions, can be described within a single linguistic framework. An earlier experiment exercised modular language implementation using RASCAL in the context of the Language Workbench Challenge in 2011 (LWC'11) [4].

7. Conclusion

Modular language implementations facilitate language evolution by promoting modular language extension. In this paper we have elaborated an extensive case study in modular language implementation. The implementation of four language levels of OBERON-0 in RASCAL shows that it is indeed possible to realize each next level as an extension of the previous level. RASCAL's modularity features contributing to this feat are: modular definition of concrete, lexical and abstract syntax, the module system's **extend** feature, together with extensible, case-based function definitions. Finally, we have identified directions of further improvement to support open language modules, and implicit passing of context information. The complete language implementation required less than 1500 SLOC; this includes parsing, name analysis, type checking, desugaring, lambda-lifting and compilation to C. Additional tasks – interpretation, compilation to Java and control-flow extraction and visualization – were realized with minimal effort as well.

References

[1] B. Meyer, Object-Oriented Software Construction, 1st Edition, Prentice-Hall, 1988.

[2] N. Wirth, Compiler Construction, Addison-Wesley, 1996.

[3] P. Klint, T. van der Storm, J. Vinju, Rascal: A Domain Specific Language for Source Code Analysis and Manipulation, in: Proc. of the 10th

Working Conf. on Source Code Analysis and Manipulation (SCAM), IEEE, 2009, pp. 168–177.

[4] T. van der Storm, The Rascal Language Workbench, CWI Technical Report SEN-1111, CWI (May 2011).

[5] M. v. d. Brand, J. Heering, P. Klint, P. Olivier, Compiling language definitions: The ASF+SDF compiler, TOPLAS 24 (4) (2002) 334–368.

[6] E. Scott, A. Johnstone, GLL parsing, Electr. Notes Theor. Comput. Sci. 253 (7) (2010) 177–189.

[7] M. G. J. van den Brand, E. Visser, Generation of formatters for context-free languages, ACM Trans. Softw. Eng. Methodol. 5 (1) (1996) 1–41.

[8] M. van den Brand, P. Klint, J. J. Vinju, Term rewriting with traversal functions, ACM Trans. Softw. Eng. Methodol. 12 (2) (2003) 152–190.

[9] E. Visser, Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9, in: Domain-Specific Program Generation, Vol. 3016 of LNCS, Springer, 2004, pp. 216–238.

[10] T. Johnsson, Lambda lifting: transforming programs to recursive equations, in: Proc. of the Conf. on Functional programming languages and computer architecture, Springer, 1985, pp. 190–203.

[11] P. Klint, B. Lisser, A. van der Ploeg, Towards a one-stop-shop for analysis, transformation and visualization of software, in: Proc. of the 5th international conference on Software Language Engineering (SLE), Vol. 6940 of LNCS, Springer, 2012, pp. 1–18.

[12] B. Basten, T. van der Storm, AmbiDexter: Practical ambiguity detection, in: Proc. of the 10th Working Conf. on Source Code Analysis and Manipulation (SCAM), IEEE, 2010, pp. 101–102.

[13] H. J. S. Basten, J. J. Vinju, Parse forest diagnostics with Dr. Ambiguity, in: Proc. of the 4th international conference on Software Language Engineering (SLE), Vol. 6940 of LNCS, Springer, 2011, pp. 283–302.

[14] J. R. Lewis, J. Launchbury, E. Meijer, M. B. Shields, Implicit parameters: Dynamic scoping with static types, in: Proceedings of the 27th

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), ACM, 2000, pp. 108–118.

[15] J. Heering, P. R. H. Hendriks, P. Klint, J. Rekers, The syntax definition formalism SDF. Reference manual, SIGPLAN Not. 24 (11) (1989) 43–75.

[16] A. Casey, L. Hendren, MetaLexer: a modular lexical specification language, in: Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11), ACM, 2011, pp. 7–18.

[17] L. Renggli, M. Denker, O. Nierstrasz, Language boxes: Bending the host language with modular language changes, in: Proceedings of the Second International Conference on Software Language Engineering (SLE'09), Vol. 5969 of LNCS, Springer, 2009, pp. 274–293.

[18] J. Rekers, Parser generation for interactive environments, Ph.D. thesis, University of Amsterdam (January 1992).

[19] G. Economopoulos, P. Klint, J. Vinju, Faster scannerless GLR parsing, in: Proceedings of the 18th International Conference on Compiler Construction (CC'09), Vol. 5501 of LNCS, 2009, pp. 126–141.

[20] E. Visser, Syntax definition for language prototyping, Ph.D. thesis, University of Amsterdam (September 1997).

[21] J. R. Cordy, C. D. Halpern-Hamu, E. Promislow, TXL: A rapid prototyping system for programming language dialects, Comput. Lang. 16 (1) (1991) 97–107.

[22] A. Johnstone, E. Scott, M. van den Brand, Modular grammar specification, Science of Computer Programming 87 (0) (2014) 23 – 43.

[23] R. Grimm, Better extensibility through modular syntax, in: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06), ACM, 2006, pp. 38–51.

[24] E. Van Wyk, A. Schwerdfeger, Context-aware scanning for parsing extensible languages, in: Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE'07), 2007, pp. 63–72.

[25] A. Schwerdfeger, E. Van Wyk, Verifiable composition of deterministic grammars, in: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09), ACM Press, 2009, pp. 199–210.

[26] D. E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145, corrections in **5**(1971) pp. 95–96.

[27] E. Van Wyk, O. de Moor, K. Backhouse, P. Kwiatkowski, Forwarding in attribute grammars for modular language design, in: Proceedings of the 11th International Conference on Compiler Construction (CC'02), Vol. 2304 of LNCS, Springer, 2002, pp. 128–142.

[28] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an extensible attribute grammar system, Science of Computer Programming 75 (1–2) (2010) 39–54.

[29] U. Kastens, W. Waite, Modularity and reusability in attribute grammars, Acta Informatica 31 (7) (1994) 601–627.

[30] T. Ekman, G. Hedin, Modular name analysis for Java using JastAdd, in: Revised papers of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05), Vol. 4143 of LNCS, Springer, 2006, pp. 422–436.

[31] T. Ekman, G. Hedin, The JastAdd system—modular extensible compiler construction, Science of Computer Programming 69 (1-3) (2007) 14–26.

[32] S. D. Swierstra, P. R. A. Alcocer, Attribute grammars in the functional style, in: R. N. Horspool (Ed.), Systems Implementation 2000, Vol. 117 of IFIP Conference Proceedings, Chapman & Hall, 1998, pp. 180–193.

[33] T. Kaminski, E. Van Wyk, Integrating attribute grammar and functional programming language features, in: Proc. of 4th the International Conference on Software Language Engineering (SLE), Vol. 6940 of LNCS, Springer, 2011, pp. 263–282.

[34] A. M. Sloane, L. C. Kats, E. Visser, A pure embedding of attribute grammars, Science of Computer Programming 78 (10) (2013) 1752–1769.

[35] A. Dijkstra, S. D. Swierstra, Ruler: Programming type rules, in: Functional and Logic Programming, Springer, 2006, pp. 30–46.

[36] D. Scott, C. Strachey, Toward a mathematical semantics for computer languages, in: Proc. of the Symposium on Computers and Automata, 1971, pp. 19–46.

[37] E. Moggi, Notions of computation and monads, Information and computation 93 (1) (1991) 55–92.

[38] P. D. Mosses, Theory and practice of action semantics, in: Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science (MFCS'96), Springer, 1996, pp. 37–61.

[39] P. D. Mosses, Modular structural operational semantics, Logic and Algebraic Programming 60-61 (2004) 195–228.

[40] J. A. Bergstra, J. Heering, P. Klint (Eds.), Algebraic Specification, ACM Press/Addison-Wesley, 1989.

[41] P. D. Mosses, Semantics of programming languages: Using ASF+SDF, Science of Computer Programming 97 (2015) 2–10.

[42] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, The state of the art in language workbenches, in: Proceedings of the 6th International Conference on Software Language Engineering (SLE'13), Vol. 8225 of LNCS, Springer, 2013, pp. 197–217.

[43] P. Klint, A meta-environment for generating programming environments, TOSEM 2 (2) (1993) 176–201.

[44] M. van den Brand, M. Bruntink, G. Economopoulos, H. de Jong, P. Klint, T. Kooiker, T. van der Storm, J. Vinju, Using The Meta-environment for Maintenance and Renovation, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07), IEEE, 2007, pp. 331–332.

[45] M. Völter, Language and IDE modularization and composition with MPS, in: Revised Papers of the International Summer School on Generative and Transformational Techniques in Software Engineering IV (GTTSE'11), Vol. 7680 of LNCS, Springer, 2013, pp. 383–430.

[46] M. Völter, E. Visser, Language extension and composition with language workbenches, in: Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10), ACM, 2010, pp. 301–304.

[47] L. C. L. Kats, E. Visser, The Spoofax language workbench: Rules for declarative specification of languages and IDEs, in: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10), ACM, 2010, pp. 444–463.

[48] P. Hudak, Modular domain specific languages and tools, in: 5th international conference on software reuse (ICSR'5), IEEE, 1998, pp. 134–142.

[49] A. M. Sloane, Lightweight language processing in Kiama, in: Revised Papers of the International Summer School on Generative and Transformational Techniques in Software Engineering III (GTTSE'09), Vol. 6491 of LNCS, Springer, 2011, pp. 408–425.

[50] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, M. Felleisen, Languages as libraries, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11), ACM, 2011, pp. 132–141.

[51] S. Liang, P. Hudak, M. P. Jones, Monad transformers and modular interpreters, in: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), ACM, 1995, pp. 333–343.