

Prological Language Processing

Ralf Lämmel¹

*CWI & Vrije Universiteit
Amsterdam, The Netherlands*

Günter Riedewald²

*Fachbereich Informatik
Universität Rostock
Rostock, Germany*

Abstract

We describe a Prolog-based approach to the development of language processors (such as preprocessors, frontends, evaluators, tools for software modification and analysis). The design of the corresponding environment *Laptob* for prological language processing is outlined. Language processor definitions in *Laptob* are basically Prolog programs. The programs might contain grammars, that is, we consider logic grammars. The programs can be typed, and they can be higher-order. The adaptation and composition of the logic programs themselves is supported by meta-programming. The environment offers tool support for efficient scanning, testing, and application development based on a make-system. We report on recent and ongoing applications of the Prolog-based approach.

1 Introduction

Prolog and language processing Prolog has been proposed and used for language-processing tools in abundance, e.g., for parsing and compilation [60,10,61], for interpretation [8,55], and other aspects of language processing. Despite Prolog's simplicity, it is extremely expressive, mainly due to its meta-logical features (I/O, negation, univ-operator, assert/retract etc.) and because it is basically untyped. Prolog is generally suited for prototyping. Prolog's scalability has been found to be acceptable [43,46]. There are mature implementations of Prolog. Well-designed foreign language interfaces allow us to resort to C for critical or external components, or to connect to a library for

¹ Email: Ralf.Laemmel@cwi.nl

² Email: gri@informatik.uni-rostock.de

GUI development. Prolog is standardised [49]. Prolog is a reliable platform for a language processing environment.

Premise and form of the paper We report on a prological approach to language processor development based on the formula *logic programming + grammars + types + higher-order program schemes + infrastructure*. We outline a corresponding environment *Laptob*.³ This environment is the successor of the *LDL system* [22] which was developed in the *LDL project* [53] (LDL—Language Development Laboratory). Besides the exposition of the *Laptob* formula, we also discuss recent and ongoing applications of *Laptob* or LDL, respectively. The main proposition of the paper is that Prolog-based language processing is convenient and scalable, and that, on the other side, a corresponding environment can be of remarkable simplicity. To prove that proposition, the article is written in the style of a survey paper to a large extent.

Structure of the paper In Section 2, we present logic programs, and in particular an enriched form, that is, logic grammars—the basic notation for developing frontends and backends of *Laptob* language processors. The programs can be optionally typed. Parsing and unparsing is enabled. In Section 3, we discuss higher-order programs to deal with program schemes and generic functionality as it is useful to raise the level of abstraction, and to improve reusability. In Section 4, several infrastructural facilities for the prological environment are discussed, namely support for efficient scanning, for meta-programming, for building applications, and for testing language processors. In Section 5, recent and ongoing applications of *Laptob* or the predecessor LDL resp. are described. The paper is concluded in Section 6.

Lightweights vs. heavyweights Language processors developed according to the *Laptob* formula are basically Prolog programs. It is well-known that several formalisms used for the definition of language processors can be faithfully encoded in Prolog, e.g., context-free grammars [47], natural semantics [15], and attribute grammars [54,14,44]. We rely on Prolog’s concepts and its expressiveness to supply features like parsing, typing, modularity, adaptiveness, genericity. *Laptob* is a lightweight compared to compiler compilers like Eli [19], Cocktail [20], FNC-2 [27], or Lisa [62]. Such compiler compilers are complex pieces of software which usually favour a certain class of attribute grammars and their efficient implementation. We should also compare *Laptob* to other environments for language design, program transformation, or language processing, e.g., Centaur [4], ASF+SDF [30], Mjølner [31], ELAN [3], Stratego [59]. Like compiler compilers, these environments are not lightweights. The development effort for these systems usually amounts to several man years. Those systems offer one or several specific formalisms which then need to be interpreted and/or compiled. Debugging support and other tool support has to be offered. Considerable effort is spent to gain efficiency, or certain spe-

³ *Laptob*—Language processor toolbox; home page: <http://www.cs.vu.nl/Laptob>

cific properties like incremental parsing or evaluation, generalised LR parsing, interfaces to general-purpose languages. The basic architecture of *Laptob* is deliberately kept simple by using Prolog as description language (for language processors) and implementation language (for the environment itself). *Laptob* provides an experimental framework for language design, language processing and program transformation. *Laptob* scales up for some desired applications. The Prolog-orientation is certainly an impediment for highly efficient language implementations, or for implementation models which are based a-priori on other formalisms or implementation languages.

Acknowledgement The authors are grateful for discussions with Anke Dittmar, Jan Echternach, Paul Klint, Wolfgang Lohmann, Marjan Mernik, Elke Tetzner. We also thank the users of LDL—the predecessor of *Laptob*—who provided us with invaluable feedback. This work was supported, in part, by *NWO*, in the project 612.014.006 “*Generation of Program Transformation Systems*”, and by the Slovenian and German governments in the bilateral project SVN 99/028 “*Generic, adaptive and aspect-oriented language definitions*”.

2 Language processor = Prolog program + sugar

Language processors usually consist of phases such as frontends, analyses and transformations, translations, backends. Besides, language processors employ helper modules such as abstract data types, e.g., for error handling or symbol table management. In this section, we argue that all these components can be implemented as logic programs. To facilitate parsing and unparsing, we complete logic programs into a suitable logic grammar formalism. Finally, we will argue that the components of language processing should be typed. Language processors are basically modular, optionally typed, enriched first-order logic programs. In Section 3, we accomplish higher-order style. In the present paper, we follow a pragmatic approach regarding the semantics of our language processors: We rely on impure features of Prolog as opposed to pure definite clauses. Sometimes we also rely on the operational semantics of Prolog. As for the basic parser model, for example, we rely on the standard computation rule, and on the order of clauses.

2.1 Logic programs

Prolog is a prominent language for symbolic computation. The language is particularly suited for the phases of language processing which can be performed at the level of abstract representations. It is well-established that logic programs are well-suited to describe analysis, evaluation and transformation of abstract representations.

We will use a simple program transformation problem as the running example in the paper. More challenging applications of *Laptob* or LDL resp. are discussed in Section 6. The running example is concerned with the elimination of dead let-expressions for a simple language of arithmetic expressions.

```

ed1(const(Val),const(Val)).
ed1(var(Var),var(Var)).
ed1(uop(Uops,Exp0),uop(Uops,Exp1)) :- ed1(Exp0,Exp1).
ed1(bop(Exp1,Bops,Exp2),bop(Exp3,Bops,Exp4))
  :- ed1(Exp1,Exp3), ed1(Exp2,Exp4).
ed1(let(Var,Exp1,Exp2),Exp0)
  :- ed1(Exp1,Exp3), ed1(Exp2,Exp4), freevars(Exp4,Vars),
     (member(Var,Vars) -> Exp0 = let(Var,Exp3,Exp4); Exp0 = Exp4).

```

Fig. 1. `ed1.pl`: A logic program for elimination of *dead lets*

Consider, for example, the following expression:

```

let a = 1 in
  let b = 2 in
    a + 3

```

The inner `let` is dead because the corresponding variable `b` is not used. Thus, we can eliminate the inner `let`. The input and the output of the corresponding transformation is shown below using an abstract representation based on Prolog terms:

```

Input: let(a,const(1),let(b,const(2),bop(var(a),plus,const(3))))
Output: let(a,const(1),bop(var(a),plus,const(3)))

```

The logic program in Figure 1 describes the transformation for elimination of dead lets. The program performs a simple traversal on expressions—one clause for each form of expression. Encountering a `let`-expression of the form `let(Var,Exp1,Exp2)`, the following actions are performed. Firstly, the two subexpressions are simplified. Then, the free variables of the body `Exp2` are computed. Finally, it is checked if the variable `Var` is among these free variables. If this is the case, the `let` needs to be retained. Otherwise, the `let` can be eliminated.

2.2 Logic grammars and parsing

Several approaches have been proposed to enable parsing in logic programs. One can distinguish approaches where logic programming and grammars are amalgamated [47,1,11], or where parsers are explicitly implemented [10,24]. A very well-known approach belonging to the former class is based on the DCG formalism [47] (DCG—definite clause grammars). While the body of a pure Horn clause is basically a conjunction of literals, the body of a DCG clause can also contain terminals. DCGs are supported in Prolog environments by a compilation model based on accumulators [56], that is, the token sequence to be analysed is modelled with additional parameter positions in the logic program derived from the logic grammar. This standard model exposes certain insufficiencies as identified, for example, in [45], namely non-determinism of parsing, lack of error handling, termination problems with left-recursion, and complete separation of scanning and parsing.

```

exp(const(Val))           :- nat(Val).
exp(var(Var))             :- id(Var).
exp(uop(Uops,Exp1))      :- uops(Uops), exp(Exp1).
exp(let(Var,Exp1,Exp2))  :- @"let", id(Var), @"=", exp(Exp1), @"in", exp(Exp2).
exp(bop(Exp1,Bops,Exp2)) :- exp(Exp1), bops(Bops), exp(Exp2).

```

Fig. 2. `frontend.pl`: A logic grammar for parsing an expression language

As for *Laptob*, we use a different model which—in contrast to DCGs—does not require any extension of logic programming. Instead we rely on impure capabilities of Prolog for file I/O. The model for *Laptob* can be best conceived by the following equations:

Nonterminals	=	Predicates
Morpheme classes	=	Predicates
Terminals t	=	Literals of the form @ t
(Top-Down) Parsing	=	Prolog computation rule + file processing

We prefix keywords, separators and that alike by a special predicate symbol @/1 which takes the terminal as a string parameter. Terminals corresponding to morpheme classes are covered by corresponding predicates. @/1 and the predicates for morpheme classes are supposed to perform scanning on the input stream. The operational semantics of Prolog with its left-to-right depth-first computation immediately leads to (top-down) parsing as in the case of DCGs. This model was first suggested in [54].

In Figure 2, we show the logic grammar defining a frontend which is suitable to parse the simple expression language of our running example. The parameters of the predicates from the frontend definition serve for the synthesis of an abstract representation. For brevity, we do not list the clauses for `uops/1` and `bops/1` covering unary or binary operation symbols, respectively. Note the occurrences of @/1, `id/1`, and `nat/1`. The predicate `id/1` serves for scanning variable identifiers, whereas the predicate `nat/1` serves for scanning natural numbers. It is trivial to define these predicates for scanning. For convenience, in Figure 3, we show the context-free grammar underlying the logic grammar from Figure 2.

```

exp → nat
exp → id
exp → uops exp
exp → exp bops exp
exp → "let" id "=" exp "in" exp

```

Fig. 3. The context-free grammar underlying Figure 2

We should point out the properties of this approach, and we should also indicate opportunities for improvement of the basic scheme. Logic grammars

in the sense of *Laptop* are logic programs. No other formalism is involved. A translation is not required. Logic grammars are immediately executable in Prolog. The operational interpretation of logic grammars in *Laptop* relies on the impure interpretation of predicates for terminals to perform scanning. Scanning and parsing can be interleaved, that is, when a scanner predicate is encountered, scanning can be performed, or a previously scanned token can be looked up from a buffer. These are the main improvements over DCGs. Otherwise, the basic scheme exposes some insufficiencies also present for DCGs, namely nondeterminism, lack of error handling, and termination problems with left-recursion. For these problems, we can adopt the techniques proposed for DCGs elsewhere [10,24,45]. We can, for example, transform the logic grammar to perform deterministic parsing, if the underlying context-free grammar falls into a corresponding category. We also experimented with the option to employ production quality parser generators. For that purpose, we extract the parser generator input from the logic grammar. A transformed version of the original logic grammar will then be used to interact with the generated efficient parser. Error handling can be accomplished by the additional specification of error rules, and by standard methods such as the panic method. An important concept is that improvements of the parsing behaviour, e.g., making the parser deterministic, or handling errors, can be *easily* achieved by *meta-programming*.

2.3 Parsing vs. unparsing

Language processing involves parsing of programs, analysis, transformation, and evaluation of the obtained abstract representations, and unparsing of the results. Our model of logic grammars facilitates parsing and unparsing in a uniform manner. If a logic grammar in our sense is executed for parsing, the terminal predicates are interpreted to scan text. If a logic grammar is executed for unparsing, the terminal predicates are interpreted to generate text. In Figure 4, a logic grammar suitable for unparsing the expression language of our running example is defined.

```

exp(const(Val))           :- nat(Val).
exp(var(Var))             :- id(Var).
exp(uop(Uops,Exp1))      :- uops(Uops), sp, exp(Exp1).
exp(let(Var,Exp1,Exp2))
  :- @"let", sp, id(Var), @ "=", exp(Exp1), sp, @"in", sp, exp(Exp2).
exp(bop(Exp1,Bops,Exp2)) :- exp(Exp1), sp, bops(Bops), sp, exp(Exp2).

```

Fig. 4. `backend.pl`: A logic grammar for text-generation

The backend completes our running example. The dead-let elimination consists of three phases, namely the frontend from Figure 2, the transformation phase defined in Figure 1, and the backend from Figure 4. For brevity, we do not show the glue to put these three phases together in a pipeline.

Note the similarity of the frontend and the backend. The frontend synthesizes abstract representations. The execution of the logic grammar is controlled by parsing the input. The backend generates text as the concrete representation for the input term. Thus, the execution of this logic grammar is controlled by the input term. Because of this duality, we can, in principle, use the same logic grammar for both parsing and unparsing. Usually, unparsing has to adhere to some scheme of pretty-printing. Therefore, the logic grammar for unparsing is usually a refined variant of the logic grammar for parsing. As for the running example, the backend differs from the frontend only in that some occurrences of the terminal `sp` facilitate (very modest) pretty-printing. The predicate `sp` is meant to generate a single space.

2.4 Typeful programming

Prolog is basically untyped. This is useful for many applications, e.g., for meta-programming. On the other hand, most problems in language processing benefit from typing. Types are useful for program comprehension, to prevent programs from going “wrong”, to optimise programs and others. For these and other reasons, type systems have been proposed for logic programming [41,25], and they have been integrated with some logic languages such as Gödel and Mercury. Also, when specification formalisms are mapped to Prolog, we often have types available anyway, e.g., in the case of Typol [15]. As for *Laptob*, we adopt a standard many-sorted type system based on certain basic types, and extended with polymorphism. We will later consider expressiveness to cover higher-order predicates and generic term traversals. The *Laptob* model for using types is very flexible and rather soft:

- Types are optional.
- Types can be used as an interface to untyped (say untypable) functionality.
- We can enforce well-typedness w.r.t. given types.
- We can also infer types from given logic programs.
- Explicit dynamic type tests can be performed.

There are the basic types `atom` and `integer`. Data type declarations are given by goal clauses of the following form:

$$:- \text{data } t = f_1(t_{1,1}, \dots, t_{1,k_1}) \mid \dots \mid f_n(t_{n,1}, \dots, t_{n,k_n}).$$

This declaration means that there are n functors of type t . The types $t_{i,1}, \dots, t_{i,k_i}$ are the argument types of the functor f_i . We can also define the types of predicates. The corresponding type declarations are goal clauses of the following form:

$$:- \text{profile } p(t_1, \dots, t_n).$$

This declaration says that the predicate symbol p is n -ary. The (many-sorted) types of the parameter positions are t_1, \dots, t_n . The types for the logic grammar from Figure 2 are shown in Figure 5. The predicate type declarations are

also reusable for the backend from Figure 4. The functor type declarations apply to all phases of our running example.

Predicate type declarations for Figure 2+Figure 4	Functor type declarations for abstract expressions
<pre>:- profile exp(exp). :- profile nat(integer). :- profile id(atom). :- profile uops(uops). :- profile bops(bops).</pre>	<pre>:- data exp = const(integer) var(atom) uop(uops,exp) bop(exp,bops,exp) let(atom,exp,exp).</pre>

Fig. 5. Interface for `frontend.pl`

The type system of *Laptob* goes beyond first-order many-sorted expressiveness in several ways as we will see. The most obviously needed extension is parametric polymorphism, or polymorphism for short. The common list operations, for example, do not put any constraints on the element type. Thus, the list type might be conceived as a type which is parametric in the element type. In the definition of polymorphic types via `data`-clauses, we use logic variables as type parameters. The polymorphic data type `list(X)` is defined in Figure 6 including two standard predicates for list manipulation. We should mention that overloading (*ad-hoc* polymorphism) is also enabled in *Laptob*.

```
:- data list(X) = [] | [X|list(X)].
:- profile member(X,list(X)).
:- profile append(list(X),list(X),list(X)).
```

Fig. 6. Homogeneous lists

Note that type checking is not always feasible or desirable. Think of, for example, the common applications of the `univ`-operator (“`=..`”) of Prolog. The list of parameters of a term obtained via the `univ`-operator is inherently heterogeneous. For that kind of reason, types are optional in *Laptob*. In glueing together modules to assemble language processors, one can disable type checking within certain modules if necessary.

2.5 Modes

Many-sorted or even polymorphic types provide just one dimension of typing in logic programming. We also employ modes or directions [6] for parameter positions of predicates. As for *Laptob*, two basic modes are relevant, namely “+” and “-” for input or output positions, respectively. In Figure 7, the modes for the predicates involved in the various phases of our running example are shown. We use again goal clauses based on `profile/1`. In a sense, the modes illustrate the data-flow in the language processor. As for notation, we might also combine the type and the mode declarations for parameter positions of predicates. Then, the mode “+” and “-” prefixes the type.

Modes for frontend.pl	Modes for edl.pl	Modes for backend.pl
<code>:- profile exp(+).</code>	<code>:- profile edl(+,-).</code>	<code>:- profile exp(-).</code>
<code>:- profile nat(+).</code>	<code>:- profile freevars(+,-) .</code>	<code>:- profile nat(-).</code>
<code>:- profile id(+).</code>	<code>:- profile member(+,+).</code>	<code>:- profile id(-).</code>
<code>:- profile uops(+).</code>		<code>:- profile uops(-).</code>
<code>:- profile bops(+).</code>		<code>:- profile bops(-).</code>

Fig. 7. Modes for the phases of dead-let elimination

We should explain the meaning of modes in more detail. In general, a mode is meant to regulate whether certain positions of predicates are required to be instantiated or ground on invocation, or whether they are instantiated or ground on return. Furthermore, modes might also regulate the backtracking behaviour of predicates, that is, if backtracking might yield more answer substitutions. The two basic modes “+” and “-” favoured for *Laptob* are interpreted as follows. The mode “+” means that the corresponding position has to be ground when a corresponding goal is called. The mode “-” means that the corresponding position will be ground on return. We omit the discussion of further forms of mode annotations.

3 Higher-order programs

Higher-order style in logic programming adds opportunities for reuse. It also encourages abstraction. Using higher-order programs, we can encode program schemes as opposed to concrete programs. In [42], it was shown that higher-order style is available to the native Prolog programmer. In this section, we explain a typed model of the higher-order style. This style is then shown to be beneficial if not essential for the definition of language processors in *Laptob*. We present some prominent program schemes such as abstractions for list processing, extended BNF notation, and traversal strategies.

3.1 Incomplete goals

There is a distinguished form of type for terms which are meant to be goals in the Prolog sense. Actually, we are concerned with potentially incomplete goals, that is, goals which need to be completed by some more parameters before they can be invoked. For such goal-like terms the type is of the form `goal(L)`, where the parameter L is a list of types of arguments to complete the goal. If $L == []$, then we deal with a complete goal. Let us first consider complete goals. In Figure 8, we define how complete goals can be formed in Prolog (note that “,” “;”, “->” are infix symbols, and “not” is a prefix symbol). The body of a clause is a complete goal defined in terms of the listed forms.

Incomplete goals provide a simple and appealing model for higher-order style in Prolog. We assume a predicate `apply/2` for completion. This predicate is usually provided by Prolog systems, and it is intentionally defined as follows:

```

:- profile true.
:- profile fail.
:- profile goal([],goal([])).
:- profile goal([]);goal([]).
:- profile goal([])->goal([]).
:- profile call(goal([])).
:- profile not goal([]).
:- profile !.

```

Fig. 8. Formation of goals, rule bodies

```

apply(G0, Ts0)
  :- G0=..[P|Ts1], append(Ts1,Ts0,Ts2), G1=..[P|Ts2], call(G1).

```

We construct incomplete goals like terms, that is, given a predicate symbol s and some parameters p_1, \dots, p_i , the term $s(p_1, \dots, p_i)$ denotes an incomplete goal. We cannot provide a standard type declaration for `apply/2` because the predicate is somewhat special. Our type system needs to be aware of the predicate, and also of the idea of construction of incomplete goals. A term $s(p_1, \dots, p_i)$ is of type `goal([ti+1, ..., tn])`, if there is a predicate s of type $s(t_1, \dots, t_n)$ and the p_1, \dots, p_i are of type t_1, \dots, t_i . Well-typedness of the goal `apply(G,Ps)` simply means that G is an incomplete goal, and the formal types for the remaining parameters coincide with the actual types of Ps .

```

:- profile and(goal([+X]),+list(X)).
:- profile fold(+Y,goal([+X,+Y,-Y]),+list(X),-Y).
:- profile map(goal([+X,-Y]),+list(X),-list(Y)).
:- profile and(goal([+X,+A,-A]),+list(X),+A,-A).
:- profile map(goal([+X,-Y,+A,-A]),+list(X),-list(Y),+A,-A).

and(_, []).
and(G, [H|T]) :- apply(G, [H]), and(G, T).

fold(N, _, [], N).
fold(N, C, [H|T], V1) :- fold(N, C, T, V0), apply(C, [H, V0, V1]).

...

```

Fig. 9. List processing

3.2 List processing

Let us present some prime examples of higher-order predicates. In Figure 9, we show list-processing predicates. The predicate `and(G,L)` applies the incomplete goal G to all the elements in L . The predicate `fold(N,C,L,V)` performs a straightforward fold on the list L computing V by “replacing” `[]` by N and `[...|...]` by C . The predicate `map(G,L0,L1)` applies G to all successive pairs of elements from $L0$ and $L1$. There are elaborated variants `and/4` and `map/5` where the additional parameter positions facilitate accumulation [56]. Note that the modes stated in Figure 9 provide a guideline for how to use the predicates for list processing.

Logic grammar	Extracted EBNF
<code>vardecs(Decs) :-</code>	<code>vardecs → "var" vardec+</code>
<code> @"var",</code>	<code>vardec → id ":" type ";"</code>
<code> plus(vardec,true,Decs).</code>	<code>statements → (statement (";" statement)*)?</code>
	<code>statement → id "!=" exp</code>
<code>vardec(dec(Var,Type)) :-</code>	
<code> id(Var), @":", type(Type), @";".</code>	
<code>statements(Stms) :-</code>	
<code> star(statement,@";",Stms).</code>	
<code>statement(assign(Var,Exp)) :-</code>	
<code> id(Var), @":=", exp(Exp).</code>	

Fig. 10. Excerpt of a syntax definition with EBNF

3.3 Extended BNF

Let us consider another class of examples for program schemes. Up to now we can basically deal with simple Backus-Naur-Form in logic grammars. One can also encode nested alternatives with “;”. Higher-order schemes provide an elegant way to introduce star- and plus-notation for logic grammars. In Figure 10, we illustrate this kind of extended BNF expressiveness by an excerpt from a frontend definition for an imperative language. The parameterisation deals again with the synthesis of abstract representations, namely sequences of declarations and statements.

```

:- profile star(goal([],goal([])).
:- profile plus(goal([],goal([])).
:- profile star(goal([-X]),goal([],-list(X)).
:- profile plus(goal([-X]),goal([],-list(X)).
:- profile star(goal([+A,-A]),goal([],+A,-A).
:- profile plus(goal([+A,-A]),goal([],+A,-A).
:- profile star(goal([-X,+A,-A]),goal([],-list(X),+A,-A).
:- profile plus(goal([-X,+A,-A]),goal([],-list(X),+A,-A).

star(E,S)      :- plus(E,S);true.
plus(E,S)     :- call(E),(call(S),plus(E,S);true).
...

```

Fig. 11. Schemes for extended BNF

In Figure 11, we overload the predicate symbol `star` and `plus` to support star- and plus-notation in combination with some other behaviour, namely synthesis of a list, and/or accumulation. `star/2` and `plus/2` are useful for pure parsing, that is, if no abstract representation is to be derived, and also no accumulation is performed. The first goal parameter models the elements to be parsed. The second goal parameter can be used for a separator. If no separator is needed, the parameter should be instantiated to `true`. The variants `star/3` and `plus/3` append a list from the parsed elements. `star/4` and `plus/4` also maintain an accumulator while parsing the lists. `star/5` and

`plus/5` combine list construction and accumulation.

3.4 Generic traversals

Prolog offers the `univ` operator (“`=..`”) to destruct terms. This operator can be used to perform traversals which are applicable to any term. However, typing programs which involve the `univ`-operator is hardly possible. We already encountered that problem when we introduced the predicate `apply/2`. Moreover, the `univ`-operator is at a too low level of abstraction. We would rather prefer to encode traversals in terms of operators modelling standard traversal schemes such as reduction. This idea carries over from the concepts of generalised folds in functional programming [39,37], and generic traversal strategies in strategic rewriting [59].

```
edl(Exp0,Exp1) :- bu(edlRule,Exp0,Exp1).
edlRule(let(Var,_,Exp2),Exp2)
  :- freevars(Exp2,Vars), \+ member(Var,Vars), !.
edlRule(Exp,Exp).
```

Fig. 12. `edl.pl`: A variant using generic traversals

In Figure 12, elimination of dead lets is revisited. We define the transformation more compactly by resorting to a bottom-up traversal scheme (cf. predicate `bu/3`). The scheme is parameterised by the auxiliary predicate `edlRule` which performs the actual transformation of let-expressions. This new formulation is very concise. No syntax-specific traversal is needed anymore.

In Figure 13, various generic traversal schemes are defined. The predicate `accumulate(G,Y,A0,A1)` traverses `Y` for accumulation. At each node, `G` is applied. Descent into children stops if `G` succeeds. Otherwise, the children are traversed relying on `and/4` defined earlier (with accumulation involved). The predicate `reduce(G,Zero,Op,Y,Z)` performs reduction of `Y` returning `Z`. The ingredients `Zero` and `Op` encode the monoid for reduction. The goal `G` is applied at each node. If it succeeds it means that reduction stops, and the data computed from the node by `G` is returned. Otherwise, reduction is performed for the children using `fold/4` for lists. The predicates `td/3`, `bu/3` perform top-down and bottom-up traversals. Using the terminology from [37], accumulation and reduction are inherently type-changing whereas top-down and bottom-up traversals are necessarily type-preserving. There are elaborations `td/5`, and `bu/5`, which combine type-preserving traversal behaviour with accumulation. As an aside, all these traversal predicates are defined in a way that they faithfully deal with non-ground terms (cf. the occurrences of `var/1`).

Note that `and/4` and also the other list traversal predicates used in Figure 13 are applied to inhomogeneous lists of children. The definitions of the generic traversal schemes do not type-check for this reason. The listed type declarations are just assumed as interface. The traversal schemes illustrate

```

:- profile accumulate(goal([+X,+A,-A]),+Y,+A,-A).
:- profile reduce(goal([+X,-Z]),+Z,goal([+Z,+Z,-Z]),+Y,-Z).
:- profile td(goal([+X,-X]),+Y,-Y).
:- profile bu(goal([+X,-X]),+Y,-Y).
:- profile td(goal([+X,-X,+A,-A]),+Y,-Y,+A,-A).
:- profile bu(goal([+X,-X,+A,-A]),+Y,-Y,+A,-A).

accumulate(G,Y,A0,A1) :- apply(G,[Y,A0,A1]), !.
accumulate(_,Y,A0,A0) :- var(Y), !.
accumulate(G,Y,A0,A1) :- Y =.. [_|Ps], and(accumulate(G),Ps,A0,A1).

reduce(G,_,_,Y,Z)      :- apply(G,[Y,Z]), !.
reduce(G,Zero,_,Y,Zero) :- var(Y), !.
reduce(G,Zero,Op,Y,Z)  :- Y =.. [_|Ps], fold(Zero,op2cons(G,Zero,Op),Ps,Z).

% Helper predicate for reduce/5
op2cons(G,Zero,Op,Y,Z0,Z2) :-
    reduce(G,Zero,Op,Y,Z1), apply(Op,[Z0,Z1,Z2]).

td(_,Y0,Y1) :- var(Y0), !, Y1=Y0.
td(G,Y0,Y2) :-
    apply(G,[Y0,Y1]), Y1 =.. [S|Ps0], map(td(G),Ps0,Ps1), Y2 =.. [S|Ps1].

...

```

Fig. 13. Traversal schemes

the flexibility of our framework: Functionality using the traversal schemes can be type-checked although the schemes themselves cannot.

4 Infrastructure

Infrastructural facilities need to be added to logic programming in order to enable convenient development of scalable language processors. In the present section, we discuss facilities for efficient scanning, meta-programming, testing, and assembly of applications.

4.1 Efficient scanning

As for performance, the bottleneck of the LDL system was scanning. We used a scanner generator developed in LDL itself. However, production-quality scanners can easily be integrated into a Prolog-based environment through the Prolog-C interface. The efficient scanner is then generated with *lex* (or a similar tool). As for *Laptob*, we decided to integrate efficient scanning in a fully transparent manner. A programmer only writes down a lexical grammar. Otherwise, the make- and the run-time system automate the make and the link process which is necessary to compose an efficient scanner and a logic grammar.

In Figure 14, the lexical grammar for our running example is shown. We use *lex* notation. Between the optional braces, one can define a name for

```

/* White spaces and comments */
[ \t\n]+
"/*" ( [^*] | "*" [^/*] )* "*" "/"
"%" [^\n]*

/* Proper morpheme classes */
[a-zA-Z] [a-zA-Z0-9]* { id }
[0-9]+ { nat }

```

Fig. 14. `scanner.lg`: Lexical grammar for `frontend.pl`

the morpheme class. If no such name is declared, the corresponding tokens are simply skipped. If language processors are composed from logic grammars and lexical grammars, the corresponding *lex* definitions, and C- and Prolog-stubs for integration are generated automatically. At run time, parsing and scanning is performed in interleaved manner. As for non-deterministic parsing, the scanned tokens are buffered (for backtracking) in the generated Prolog-stub. We already mentioned the option to go further by accomplishing efficient external parsers. For most applications, however, and especially for language prototyping, the speed of the Prolog parser is sufficient. Note also that Prolog-based parsing immediately enables semantics-directed parsing which is beneficial if not obligatory for various applications.

4.2 Meta-Programming

Meta-programming is a vital tool in logic programming (cf. [5]). We will show how meta-programs can be used for the development of language processors, and for the underlying framework itself. In the setting of *Laptob*, meta-programs are (as usual in Prolog) encoded as ordinary predicates which process object programs in their term representation. In Prolog, it is extremely simple to obtain a term representation of a logic program at hand. Either we query the Prolog data base with `clause/2`, or we just read in the program from the file with the basic `read/1` predicate.

As an introductory example, we revisit the duality between frontend and backend specifications of the same language. It turns out that we can derive backends from frontends in a systematic manner by program transformation. The difference between the frontend in Figure 2 and the backend in Figure 4 is that there are additional occurrences of the terminal `sp` in the backend which accomplish pretty-print functionality. The fact that the frontend is assumed to synthesize abstract representations whereas the backend is supposed to pretty-print is just a matter of usage. It does not affect the specifications themselves. In Figure 15, we show clauses for meta-programming which can be *applied* to the frontend to actually *derive* the backend from it. We assume a simple meta-programming predicate `replace(G,01,02)` which applies the incomplete goal `G` to all bodies of `01` to derive `02`. As for our example, we apply `replace/3` in the following instance:

```
replace(frontend2backend,F,B)
```

Here, `F` is bound to the frontend serving as input, and the computed backend will be bound to `B` on return. The clauses for `frontend2backend/2` catch the terminals of the grammar for our expression language, and then, occurrences of `sp/0` are added as prefixes and/or postfixes.

```
frontend2backend(uops(Uops), (uops(Uops), sp)).
frontend2backend(bops(Bops), (sp, bops(Bops), sp)).
frontend2backend(@"let", (@"let", sp)).
frontend2backend(@"in", (sp, @"in", sp)).
```

Fig. 15. Derivation of `backend.pl` from `frontend.pl`

The aforementioned predicate `replace/3` is debatable. In general there is no guarantee that the semantics of the object program is preserved. Of course, there is some way to restrict rewriting of clause bodies (and clause heads, too) in a suitable way. For brevity, we do not discuss this opportunity. There are also many other useful operators than just `replace/3`. In [36], for example, we define semantics-preserving roles of program extension inspired by stepwise enhancement [33,29]. These roles comprise addition of parameters, premises and predicates, renaming, and substitution. Further common concepts are folding and unfolding. The development of the *Laptop* operator suite for meta-programming is ongoing work.

Environment propagation

```
exp(const(Val), Vars) :-
  nat(Val).
exp(var(Var), Vars) :-
  id(Var).
exp(uop(Uops, Exp1), Vars) :-
  uops(Uops),
  exp(Exp1, Vars).
exp(bop(Exp1, Bops, Exp2), Vars) :-
  exp(Exp1, Vars),
  bops(Bops),
  exp(Exp2, Vars).
exp(let(Var, Exp1, Exp2), Vars) :-
  @"let",
  id(Var),
  @ "=",
  exp(Exp1, Vars),
  @"in",
  exp(Exp2, Vars).
```

Environment access

```
exp(const(Val), Vars) :-
  nat(Val).
exp(var(Var), Vars) :-
  id(Var),
  member(Var, Vars).
exp(uop(Uops, Exp1), Vars) :-
  uops(Uops),
  exp(Exp1, Vars).
exp(bop(Exp1, Bops, Exp2), Vars) :-
  exp(Exp1, Vars),
  bops(Bops),
  exp(Exp2, Vars).
exp(let(Var, Exp1, Exp2), Vars) :-
  @"let",
  id(Var),
  @ "=",
  exp(Exp1, Vars),
  @"in",
  exp(Exp2, [Var|Vars]).
```

Fig. 16. Extending the frontend with static semantics

In Figure 16, we illustrate the idea of adapting language processors by a simple example. The original frontend from Figure 2 is merely a parser constructing an intermediate representation. The extension in Figure 16 is meant to check context-conditions in addition. For the simple language at

hand, we only want to check that used variables are actually defined. We show two steps of evolution in Figure 16. The program on the left is an intermediate result with additional parameters to propagate the environment of defined variables. The program on the right also performs a membership test for variables, and it extends the environment for let-expressions. These steps are automated with the transformations presented, for example, in [36]. Adaptation by meta-programming is vital for reuse in the development of non-trivial language processors.

Let us finally illustrate how meta-programming helps in the development of *Laptob* itself. To perform type checking, interaction with external scanners, modular composition based on a compilation-based semantics [7] etc., program analyses and program transformations have to be performed. The following trivial analysis traverses an object-program to accumulate all keywords:

```
keywords(Ts,Kws) :- reduce(keyword, [], union, Ts, Kws).
keyword(@S, [S]) :- ground(S).
```

The analysis `keywords(Ts, Ss)` extracts all keywords from the given logic program, that is, it looks for goals of the form `@S`. This functionality is ultimately useful for the completion of a lexical grammar to take keywords from a logic grammar into account. Note how simple this analysis is because of the application of the traversal scheme for reduction.

4.3 Application development

Complete language processors are derived by modular composition from the phases and helper modules which contribute to the language processor at hand. A make system takes care of preprocessing and integrating user-defined modules, e.g., in order to interact with external scanners, or to enforce type-declarations formulated by the user, or to derive stand-alone programs. Modular composition (cf. [22] for the LDL module system) and application assembly (in the sense of make) is essentially based on meta-programming. Because of space constraints, we omit a thorough discussion of these concepts.

4.4 Testing

Considerable effort in the development of language processors is spent on testing. Also, as a language processor evolves, regression tests are due. Furthermore, testing notions, e.g., to characterise coverage, are useful in the design of language processors itself. In the context of software renovation, for example, coverage analysis helps to determine the language patterns to be covered by a desired transformation. Testing as we favour it for *Laptob*, comprises test case characterisation, coverage analysis, and test set generation.

There are different ways conceivable to add testing support to a prological language processing environment. One basic idea is to use the frontend in one way or another [52,22]. As for test set generation, we can interpret terminals

Regular expression	Term
<i>Some uop 2 const Some</i>	$uop(neg, const(0))$
<i>Some uop 2 var Some</i>	$uop(neg, var(a))$
<i>Some uop 2 uop Some</i>	$uop(neg, uop(neg, const(0)))$
<i>Some uop 2 bop Some</i>	$uop(neg, bop(const(0), plus, const(0)))$
<i>Some uop 2 let Some</i>	$uop(neg, let(a, const(0), const(0)))$
...	

Fig. 17. Context-dependent rule coverage (excerpt)

in the logic grammar to generate tokens. If the frontend program contains a specification of context conditions, the generated programs will immediately satisfy these conditions. Also, if we want to enforce structural properties of the generated programs, e.g., a certain level of nesting of lets, we can extend the frontend program with further constraints. These constraints are like special-purpose context conditions. Usually, they access the abstract representations.

There are other ways to add testing support. We refer the reader to [12,26] for some Prolog-oriented testing approaches which are not biased to language processing. We currently work on another approach which is biased to language processing. The basic idea underlying the characterisation of test cases and the description of coverage criteria is to use regular expressions over constructor symbols (from the abstract syntax) and parameter indices. The following expression, for example, characterises terms with certain kinds of nested lets:

$$(Some\ let\ 3\ Some)^2$$

More precisely, the expression characterises terms with nested occurrences of lets where the level of nesting is at least 2 as pointed out by the exponent for iteration. The nonterminal *Some* is supposed to generate all terminal strings. The parameter index 3 restricts the kind of nesting, i.e., we are concerned with nesting w.r.t. the third parameter position of *let*, that is, the body of a *let*. We call such expressions regular path expressions because they describe paths in terms. Test case generation can commence as follows. Given such a regular path expression, one can generate a concrete path from it based on simple regular language theory. This path in turn can be completed to a full term. Finally, the term can be passed to the pretty-printer. Test set generation according to a coverage criterion is an elaboration of this idea.

A well-known coverage notion is rule coverage [50]. One can also think of more challenging criteria, e.g., an elaboration of rule coverage, where all rules (say constructors) are covered in all occurrences of the corresponding sort. In Figure 17, we list all regular expressions modelling the coverage of all constructors in the context of the second parameter position of *uop*. In the figure, we also show small test cases experiencing the associated path expression. There are some ways to take the semantic part of a language

processor into account. In [21], we consider a two-dimensional coverage notion where one dimension corresponds to the syntax (given by the skeleton of a logic program), and another to the types involved in the computations.

5 Applications

The system LDL has been extensively used in research projects, and in teaching. We will report on a few recent applications which are well-documented. The new *Laptob* is more flexible and expressive than LDL. We will mention a few challenging applications we have in mind for the emerging framework. We conclude the section with the exposition of a few lessons learned from applying a prological approach to language processing.

5.1 Recent projects

Programming with patterns In [17], we describe an approach to object-oriented programming where special language support is provided to program with (design) patterns. Essentially, language constructs are added to the Eiffel language in order to be able to describe class structures (underlying patterns) and their superimposition (application and combination of patterns). The resulting language PaL (*Pattern Language*) is compiled to Eiffel. The design of PaL and its implementation with LDL is described in [9].

Generation of user interfaces In [16], a model-based approach to the semi-automatic generation of prototypes of user-interfaces is described. The prototypes are derived from formal task and domain models. The actual generation tools were developed in LDL. The generation approach, the design of the tools, and the implementation are described in [28]. The generation comprises several phases, and it is controlled by a knowledge base. In a first phase, the models are transformed to a set of abstract interaction objects. In the second phase, concrete interaction objects are derived. In the third and last phase, a concrete user interface management system is targeted.

Symbolic simulation In [57], a domain-specific language for the description of modular hybrid systems and queries for the analysis of the systems is presented. Hybrid systems are used to represent models for time- and safety-critical applications. The behaviour of such systems may be analysed by symbolic simulation in constraint-logic programming languages. A corresponding domain-specific language was implemented with LDL as a translator to Prolog IV. The language implementation also involves the static semantics of the domain-specific language. The mapping to Prolog IV is not trivial because the hybrid systems are based on timed automata which have to be descriptively superimposed for the executable representation in the constraint-logic program.

Integrity monitoring In [58], a concept for the implementation of integrity conditions in an object-oriented database programming language PLEX is

developed and implemented. The implementation is based on LDL. The conditions are stated as formulae in temporal logic. The implementation model in PLEX is based on transition graphs induced by the formulae. Essentially, the implementation is meant to monitor the integrity conditions. The code generated from the temporal logical formulae is attached as pre- and postconditions to the PLEX program. The implementation derives the final PLEX program from the temporal logical formulae and the original PLEX program in a number of phases.

5.2 *Active research*

Workbench for language design The original idea of the LDL project [53] was, inspired by [48], to provide a workbench for language design. Two specific objectives of the LDL project were to provide a library of language constructs, and an expert system-like tool to help the language designer in selecting, adapting, and completing components from the library. Expert systems have also been advocated by others [2] to help in this respect. The aforementioned objectives are still subject to research. The most challenging problems in this context are probably to identify a suitable formalism and format for the representation of language concepts, and to work out sensible methods for querying and adaptation in the language design process. In our previous work, we used meta-programs to adapt natural semantics definitions and attribute grammars [34,35]. In the semantics and functional programming communities, other approaches are favoured, e.g., monads [38,18] and action semantics [40]. Regardless of the formalism to be used, more work is needed to develop useful *language design methods*. Similar objectives are pursued in the domain-specific language community (cf. [23]).

Format migration Based on *Laptop*, we investigate suitable transformation frameworks, and language constructs to formulate and implement format migration procedures. The general problem can be stated as follows. Given are two format definitions A and B , e.g., in the form of many-sorted signatures. We assume that B has been derived from A by a number of well-defined transformation steps. One challenge is to derive the transformation at the term level, from the transformation at the signature level. This kind of abstract problem is relevant in the context of XML document migration, or evolution of persistent objects. One can also go one step further by taking clients depending on the evolving format into account, e.g., programs depending on an API, or a rewrite system over a signature. Then, the format change should lead to a program transformation to make clients compliant with the new format.

5.3 *Lessons learned*

Language processors are declarative programs. It has been a notorious criticism of common practice that only the simplest tools (say *lex/yacc*-like tools) are employed for the development of language processors, and the bulk

of functionality is encoded in a general purpose language. This criticism even applies to academia, and to prototyping projects. We learned that if people are encouraged to use a tool like LDL, they can develop relatively involved language processors with little effort. The resulting processors are compact and comprehensible. Declarative languages like Prolog offer a number of concepts immediately useful for language processing, e.g., pattern matching, symbolic computation, constraint resolution, and backtracking.

Types and order do matter. Language processors are often large pieces of software, and they also tend to evolve, e.g., because the underlying language needs to be extended, or additional phases need to be accomplished. Types for the abstract representations, and for the predicates pay off. Without types, the development of language processors gets very tedious. The resulting Prolog programs will just fail too often, and too much development time is spent on debugging. *Laptob* supports polymorphism to facilitate reuse of functionality. We also learned that first-order logic programs are not sufficiently expressive since the same (rather trivial) program schemes have to be encoded again and again. *Laptob* supports higher-order style. On the other hand, such additional expressiveness makes types even more desirable.

Recipes and usability are appreciated. One reason why people might resort to simple tools like *lex* and *yacc*, or even to general-purpose languages like C and Perl, is that they might better understand how to use these tools and languages. The prological approach, and more general, the declarative approach needs to be pushed by suitable catalogs of recipes, and by environments with reasonable usability. Programmers need to get equipped with a set of programming techniques to develop their language processors. These techniques should not only cover the actual encoding of recurring problems like parsing, intermediate representation, mappings, well-formedness checking, etc., but also infrastructural and process details. A programmer needs to know how to test and to debug a language processor, how to adapt it, and how to integrate external components. We envision catalogs which are written in a pragmatic style. Programmers of language processors should not need to be experts in logic programming or formal methods.

Prolog helps with acceptance and lightweightness. LDL offered a number of specification formalisms with a straight translation to Prolog, e.g., a notation for grammars of syntactical functions [51] was used instead of the logic grammars in *Laptob*. These formalisms had to be defined, explained, translated and others. Thereby, LDL became a complex system with several poorly supported formalisms, various poorly integrated experiments for meta-programming, GUIs, test set generation, literate programming and others. LDL became inflexible and unmaintainable. Since the translation of the formalisms to Prolog was so straight, it is a sensible question whether one should not bypass these extra formalisms. This is the basic model favoured for *Laptob*. Language processors are essentially logic programs. A thoroughly

Prolog-based framework for language processing makes it easier to develop the framework itself. Furthermore, the developed language processors are comprehensible for everyone who knows the well-established Prolog language. This decision does not exclude the option to introduce special notations on top of Prolog at some point in the future.

6 Concluding remarks

Many systems have been designed to implement programming languages. Some of these systems are also useful for the development of more general *language processors* (cf. [32] for the term). There is no sane answer to the question what the best system, architecture, or formalism is. Considerable effort has been spent to implement particular designs, to derive production quality systems and/or very user-friendly systems, and to complete specific formalisms into useful specification languages. Language design and language implementation is one of the oldest subjects in computer science. For that reason, our proposal for a prological environment for language processing is less of a technical contribution. We have described an approach which is effective, scalable, lightweight, and flexible. It is *effective* because we have applied it to a wide range of applications. The approach is *scalable* because by a few design decisions, namely types, higher-order style, efficient scanning, the prological environment is also applicable to non-trivial problems. The approach is *lightweight* because the corresponding framework is merely a cap on top of Prolog. This makes sense because Prolog is already a very high-level language close to the problem domain of language processing. The approach is *flexible* because the framework itself is a declarative program which can easily be extended and adapted using higher-order style and meta-programs.

References

- [1] H. Abramson. Definite Clause Translation Grammars. In *Proc. International Symposium on Logic Programming*, pages 233–241, Atlantic City, 1984. IEEE, Computer Society Press.
- [2] H. Abramson. Towards an Expert System for Compiler Writing. Technical Report TR 87-33, University of British Columbia, Department of Computer Science, 1987.
- [3] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proc. International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, Pont-à-Mousson, France, September 1998. Elsevier Science.
- [4] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proc. SIGSOFT'88, Boston, USA*, 1988.

- [5] A. Bowers. *Effective Meta-programming in Declarative Languages*. PhD thesis, Department of Computer Science, University of Bristol, January 1998.
- [6] J. Boye and J. Maluszynski. Directional Types and the Annotation Method. *Journal of Logic Programming*, 33(3):179–220, December 1997.
- [7] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(3):225–237, 1994.
- [8] B.R. Bryant and A. Pan. Rapid Prototyping of Programming Language Semantics Using Prolog. In *Proc. IEEE COMPSAC '89*, pages 439–446, Orlando, Florida, 1989.
- [9] S. Bünnig. Entwicklung einer Sprache zur Unterstützung von Design Patterns und Implementierung eines zugehörigen Compilers. Master's thesis, University of Rostock, Department of Computer Science, July 1999.
- [10] J. Cohen and T.J. Hickey. Parsing and Compiling Using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, April 1987.
- [11] V. Dahl and H. Abramson. *Logic Grammars*. Springer-Verlag, 1989.
- [12] R. Denney. Test-case generation from Prolog-based specifications. *IEEE Software*, 8(2):49–57, March 1991.
- [13] P. Deransart, B. Lorho, and J. Maluszyński, editors. *Programming Languages Implementation and Logic Programming, Proc. International Workshop PLILP '88, Orleans, France*, number 348 in LNCS. Springer-Verlag, May 1989.
- [14] P. Deransart and J. Maluszyński. *A Grammatical View of Logic Programming*. The MIT Press, 1993.
- [15] T. Despeyroux. Typol: A formalism to implement natural semantics. Technical report 94, INRIA, March 1988.
- [16] A. Dittmar and P. Forbrig. Methodological and Tool Support for a Task-Oriented Development of Interactive Systems. In J. Vanderdonckt and A. Puerta, editors, *Proc. Third International Conference on Computer-Aided Design of User Interfaces, Louvain-la-Neuve*, pages 271–274. Kluwer Academic Publishers, 1999.
- [17] P. Forbrig and R. Lämmel. Programming Design Patterns. In *Proc. TOOLS-USA 2000*. IEEE, 2000.
- [18] J.E.L. Gayo. LPS—a Language Prototyping System, 2000. <http://lsi.uniovi.es/~labra/LPS/LPS.html>.
- [19] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM* 35, pages 121–131, February 1992.

- [20] J. Grosch and H. Emmelmann. A Tool Box for Compiler Construction. In *Proc. CC'90*, 1990.
- [21] J. Harm and R. Lämmel. Two-dimensional Approximation Coverage. *Informatica*, 24(3), 2000.
- [22] J. Harm, R. Lämmel, and G. Riedewald. The Language Development Laboratory ($\Lambda\Delta\Lambda$). In M. Haveraaen and O. Owe, editors, *Selected papers from the 8th Nordic Workshop on Programming Theory, December 4–6, Oslo, Norway, Research Report 248, ISBN 82-7368-163-7*, pages 77–86, May 1997.
- [23] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, 35(3):39–48, March 2000.
- [24] P.R. Henriques. A semantic evaluator generating system in prolog. In Deransart et al. [13], pages 201–218.
- [25] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
- [26] O. Jack. *Software Testing for Conventional and Logic Programming*. Number 10 in Programming Complex Systems. Walter de Gruyter, Berlin, 1996.
- [27] M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6).
- [28] T. Kiaupat. Automatisierte Generierung von Benutzungsschnittstellen. Master's thesis, University of Rostock, Department of Computer Science, 1999.
- [29] M. Kirschenbaum, S. Michaylov, and L.S. Sterling. Skeletons and Techniques as a Normative Approach to Program Development in Logic-Based Languages. In *Proc. ACSC'96, Australian Computer Science Communications*, 18(1), pages 516–524, 1996.
- [30] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2), pages 176–201, 1993.
- [31] J.L. Knudsen, M. Lofgren, O. Lehrmann-Madsen, and B. Magnusson. *Object-Oriented Environments: The Mjølner Approach*. Prentice Hall, New York, 1994.
- [32] K. Koskimies, O. Nurmi, J. Paakki, and S. Sippu. The Design of a Language Processor Generator. *Software—Practice and Experience*, 18(2):107–135, February 1988.
- [33] A. Lakhotia. *A Workbench for Developing Logic Programs by Stepwise Enhancement*. PhD thesis, Case Western Reserve University, 1989.
- [34] R. Lämmel. Declarative aspect-oriented programming. In Olivier Danvy, editor, *Proc. PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1*, pages 131–146, January 1999.

- [35] R. Lämmel and G. Riedewald. Reconstruction of paradigm shifts. In *Second Workshop on Attribute Grammars and their Applications*, pages 37–56, March 1999. INRIA, ISBN 2-7261-1138-6.
- [36] R. Lämmel, G. Riedewald, and W. Lohmann. Roles of Program Extension. In *Post-workshop proceedings LOPSTR '99*, volume 1817 of *LNCS*. Springer-Verlag, 2000.
- [37] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In Johan Jeuring, editor, *Proc. WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000. available at <http://www.cwi.nl/~ralf/>.
- [38] S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.
- [39] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. FPCA '91*, volume 523 of *LNCS*. Springer-Verlag, 1991.
- [40] P.D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
- [41] A. Mycroft and R.A. O'Keefe. A polymorphic type system for PROLOG. *ARTIF. INTELL. (NETHERLANDS) ISSN: 0004-3702*, 23(3):295–307, August 1984.
- [42] L. Naish. Higher Order Logic Programming in Prolog. In *Proc. Workshop on Multi-Paradigm Logic Programming, JICSLP'96, Bonn*, 1996.
- [43] J. Paakki. A note on the speed of prolog. *SIGPLAN Notices*, 23(8):73–82, August 1988.
- [44] J. Paakki. A Logic-Based Modification of Attribute Grammars for Practical Compiler Writing. In David H. D. Warren and Peter Szeredi, editors, *Proc. 7th International Conference on Logic Programming*, pages 203–217. The MIT Press, 1990.
- [45] J. Paakki. A practical implementation of DCGs (abstract). In Dieter Hammer, editor, *Compiler Compilers, Third International Workshop on Compiler Construction*, volume 477 of *LNCS*, pages 224–225, Schwerin, Germany, 22–26 October 1990. Springer-Verlag.
- [46] J. Paakki. Prolog in Practical Compiler Writing. *The Computer Journal*, 34(1):64–72, 1991.
- [47] F.C.N. Pereira and D.H.D. Warren. Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13(3):231–278, 1980.
- [48] U.F. Pleban. Compiler prototyping using formal semantics. In *Proc. SIGPLAN '84 Symposium on Compiler Construction*, pages 94–105. ACM, ACM, 1984.

- [49] ISO Prolog Standard, 1995.
http://www.logic-programming.org/prolog_std.html
- [50] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972.
- [51] G. Riedewald. *Compilerkonstruktion und Grammatiken syntaktischer Funktionen*. Dissertation B, Rechenzentrum der Universität Rostock, 1979.
- [52] G. Riedewald. The LDL – Language Development Laboratory. Preprint CS-01-91, University of Rostock, Department of Computer Science, December 1991.
- [53] G. Riedewald. The LDL—Language Development Laboratory. In U. Kastens and P. Pfahler, editors, *Compiler Construction, 4th International Conference, CC'92, Paderborn, Germany*, number 641 in LNCS, pages 88–94, October 1992.
- [54] G. Riedewald and U. Lämmel. Using an attribute grammar as a logic program. In Deransart et al. [13], pages 161–179.
- [55] K. Slonneger and B.L. Kurtz. *Formal Syntax and Semantics — A Laboratory Based Approach*. Addison-Wesley Publishing Company, 1995.
- [56] L.S. Sterling and E.Y. Shapiro. *The Art of Prolog*. MIT Press, 1994. 2nd edition.
- [57] E. Tetzner and R. Riedewald. MODEL-HS - An Approach to Specify Hybrid Systems for Symbolic Simulation. In A. Poetsch-Heffter and W. Goerigk, editors, *ATPS 99*, number 258 in Informatik-Berichte, pages 201–224. University Hagen, 1999.
- [58] E. Tetzner. Umsetzung allgemeiner Integritätsbedingungen in Vor- und Nachbedingungen, November 1997. Studienarbeit, Universität Rostock, FB Informatik.
- [59] E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *International Conference on Functional Programming (ICFP'98), Baltimore, Maryland. ACM SIGPLAN*, pages 13–26, September 1998.
- [60] D.H.D. Warren. Logic programming and compiler writing. *Software—Practice and Experience*, 10(2):97–125, February 1980.
- [61] R. Wilhelm and D. Maurer. *Compiler Design*. Addison Wesley, 1995.
- [62] V. Zumer, N. Korbar, and M. Mernik. Automatic implementation of programming languages using object-oriented approach. *Journal of Systems Architecture*, 43:203—210, 1997.