smgn: Rapid Prototyping of Small Domain-Specific Languages

Holger M. Kienle[1,3] and David L. Moore[2]

[1]Department of Computer Science, University of Victoria, Canada
[2]Intel Compiler Lab, Hillsboro, U.S.A.

This paper presents smgn, a grammar-based tool that provides support for scanning, parsing, and automatic parse tree construction. The parse tree can be easily navigated and manipulated with a specific macro language while conveniently generating textual output. smgn is easy to learn—even for non-compiler experts—and well suited for rapid prototyping of small domain-specific languages. It is part of the SUIF compiler system, where it has been used for the development of the *Hoof* domain-specific language. Furthermore, smgn was employed successfully for the rapid prototyping of another domain-specific language, called Bauhaus IMDL. We introduce smgn, describe experiences in using it for DSL construction and evaluate its usefulness based on these experiences.

Keywords: domain-specific language, domain-specific processor, language prototyping, rapid prototyping, SUIF compiler system

1. Introduction

Domain-Specific Languages (DSLs) are ubiquitous in the computer domain. This fact is often not realized by programmers and users alike. Typically, HTML documents, shell scripts, and X resource files are not perceived as DSLs. Still, they all have common features that qualify them as such.

This paper defines a DSL as a small formal (programming) language whose expressive power is limited to a certain application domain and that does not include many of the features found in general-purpose programming languages. (Alternative definitions are proposed in [4, 21, 31].) A Domain-Specific Description (DSD) is a program written in a DSL. Such a program is compiled, interpreted, or analyzed by a Domain-Specific Processor (DSP) [30]. When the DSD is compiled, the generated output can be in textual format (e.g., another DSD or a program in a general-purpose programming language) or in binary format.

As pointed out in [14], the design and implementation of a DSL typically is not trivial and evolves over time. Thus, frequent changes in the design and implementation are necessary. In this context, the ability to rapidly prototype a DSL is beneficial because such a prototype can be used to get early feedback from users, for instance, to uncover missing requirements. For example, the designers of the Hancock language [5], a DSL to describe signatures (i.e., evolving customer profiles computed from phone call records), employed an iterative design process. A subset of this process consists of (1) language design, (2) writing of sample DSDs, and (3) DSP implementation. Both the evaluation of the written DSDs and the DSP implementation can lead to changes in the language design. For Hancock, the designer "found that [they] needed to iterate through this process many times." Furthermore, they note: "we also built artifacts at several intermediate stages. These artifacts proved useful even though they are not the final product."

The SUIF macro generator [18], called smgn, is a simple tool well suited for rapid prototyping of a certain class of DSLs. smgn is a grammar-based tool that allows the user to parse an input file according to a grammar specification. The resulting parse tree can then be easily nav-

[3]Part of this research was conducted as a member of the Bauhaus Group, University of Stuttgart, Germany.

igated and manipulated with a specific macro language. While navigating the parse tree, textual output can be conveniently generated. Thus smgn is a tool to write DSL compilers that translate a DSD to textual output. Furthermore, the grammar specification and macro language are easy and intuitive, which allows even DSL implementors that are not compiler experts to familiarize themselves quickly with the tool.

Though smgn was originally written for the SUIF Compiler System [26], its design is general enough to be useful for the translation of DSLs. smgn is used by SUIF to implement a DSL called *Hoof*, which is used to specify nodes for the SUIF intermediate representation. *Hoof* makes it easy for SUIF users to specify additional nodes (by subclassing from existing ones) for their own analyses. In the Bauhaus project, smgn has been employed to implement a DSL, called IMDL, to describe an internal graph representation [2].

1.1. turtle Example

As a running example, we show how to implement a small DSL, called Turtle, with smgn. A Turtle program draws a (static) picture by means of turtle graphics. (The Turtle commands are inspired by Logo turtle graphics [11].)

```
 1  # triangle.ttl
 2
 3  turtle triangle {
 4      down;
 5      turn right by 90 degrees;
 6      forward 50;
 7      turn left by 120 degrees;
 8      forward 50;
 9      turn left by 120 degrees;
10      forward 50;
11  }
```

Fig. 1. A Turtle program (triangle.ttl).

Turtle supports the following commands: (1) raising and lowering of the pen, (2) moving the turtle forward by a certain number of steps and (3) turning the turtle to the left or right by a certain degree. At the beginning, the turtle's pen is up and it points northwards. Using these operations, Figure 1 shows how a triangle can be drawn. The drawing is depicted in Figure 2.

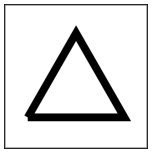

Fig. 2. Drawing of triangle (specified in Figure 1).

Our goal is to write a DSL compiler that translates a Turtle program to PostScript [15] code. Figure 3 shows a possible translation to PostScript of the triangle specified in Figure 1. This code has been generated automatically with the smgn macro file given in Appendix A.

```
 1   4 setlinewidth
 2   newpath
 3   200 200 moveto
 4   90 sin
 5   50 mul
 6   90 cos
 7   50 mul
 8   rlineto
 9   -30 sin
10   50 mul
11   -30 cos
12   50 mul
13   rlineto
14   -150 sin
15   50 mul
16   -150 cos
17   50 mul
18   rlineto
19   stroke
20   showpage
```

Fig. 3. Generated PostScript code for triangle specified in Figure 1.

As argued later, Turtle is not an ideal candidate for an implementation with smgn since it does not have a declarative nature—it is instead a command language. However, it is simple, small, and intuitive to understand and thus fits in the paper.

1.2. Contents

The paper is organized as follows. Section 2 introduces smgn using Turtle as a running example. Furthermore, smgn's historical development is briefly discussed as well as its applicability. Section 3 discusses experiences of three DSL implementations that have been conducted with smgn. Based on these experiences, strengths and weaknesses of smgn are evaluated. Section 4 introduces a (rather limited) taxonomy for DSL implementation techniques and classifies smgn according to it. Section 5 gives related work that is similar in spirit to smgn and Section 6 concludes with future work.

2. smgn's Features and Scope

First, smgn is briefly introduced to give the reader a first impression of its capabilities and features. Then, we identify the properties that make a DSL suitable for implementation in smgn.

2.1. Overview

When smgn is invoked, its actions are parameterized by the following files, which are given on the command line:

- *grammar file* (contains the grammar spec)
- *text file* (contains the actual input, e.g., the Turtle program in Figure 1)
- *macro file* (contains macro language code).

For example, to invoke smgn to generate PostScript output for the example program given in Figure 1, one types

```
smgn turtle.grm triangle.ttl 2ps.mac
```

smgn reads in the grammar file (turtle.grm) and then parses the text file (triangle.ttl) with the obtained grammar specification. During the parse, a parse tree is constructed. If no macro file is present, execution stops; otherwise, the macro file is read in (2ps.mac) and the macro processor starts interpreting. The macro file contains code that traverses the parse tree and generates output depending on the information obtained from the parse tree.

Typically, the grammar and macro files are fixed, whereas text files differ. But this is not necessarily the case. For example, the same grammar specification and text file can yield different outputs, depending on the macro file used.

In the following, the grammar specification, the generated parse tree and the macro processor are introduced in more detail.

2.2. Grammar Specification

Figure 4 shows the grammar specification for Turtle. The smgn grammar specification is similar to BNF, thus no repetitions or optionals are allowed. The names of nonterminals are encapsulated in angle brackets. Terminal symbols are written literally. Special characters in terminals are escaped by enclosing them in double quotes. The start nonterminal of the grammar is the left-hand-side of the first production (i.e., start in the Turtle grammar).

```
1   # turtle.grm
2
3   <start> ::= <turtle>
4
5   <turtle> ::= turtle <identifier>
6                { <cmd_list> }
7
8   <cmd_list> ::= <commands> |
9
10  <commands> ::=
11        <command>
12      | <command> <commands>
13
14  <command> ::=
15        <turn>
16      | <up>
17      | <down>
18      | <forward>
19
20  <turn> ::= turn <left_or_right>
21             by <identifier>
22             degrees ;
23  <left_or_right> ::= left | right
24
25  <up> ::= up ;
26  <down> ::= down ;
27  <forward> ::= forward
28                <identifier> ;
```

Fig. 4. Turtle Grammar (turtle.grm).

Most notably, no scanner specification is required. This scheme is less powerful and flexible than giving an explicit specification of the scanner tokens, but makes the specification easier. (A similar idea has been developed for the lexical source model extractor (LSME) tool [22].) This approach helps DSL implementors that are not compiler experts to develop a first working grammar in a short time. Since scanner tokens cannot be given by the user, two special nonterminals are predefined:

- The <identifier> nonterminal matches input that consists of digits, letters (upper and lowercase), and underscores in any order.
- The <verbatim> nonterminal matches any text up to a certain end-maker. The end-maker is a terminal symbol that must be given immediately after <verbatim>.

The latter is useful, for example, to capture the text that is not supposed to be parsed and subsequently analyzed, but written out later in its original form. (Arie van Deursen pointed out

to us that this construct can be useful to implement a partial parser (e.g., for analyzing a legacy system) that ignores parts of the input. In this respect, its functionality is similar to awk's "/ /,/ /" command.)

With this scheme, the token specification is usually less strict than one would like to have. For example, <identifier> can be used to match numbers, but it will also match other strings.

Typically, comments are not specified by the grammar itself, but handled directly in the scanner. Since scanner support does not exist, the smgn parser has comments already built in. Any line starting with a hash (#) is considered a comment.

The parser is implemented as a straightforward back-tracking parser. This means that grammars that are not LL and not LALR can be handled, which frees the grammar writer from the task to take these constraints into account when developing the grammar. Conflict resolution of grammars (especially LALR) can be tedious and difficult for non-compiler experts. On the other hand, if the grammar and the input force the parser to do a lot of back-tracking, the performance may suffer significantly. In practice, it is beneficial to try to develop a grammar that is close to LL(1).

If an input file cannot be parsed, smgn outputs an error message indicating the line number and the token in the line that caused the parse to fail. This symbol is normally just above the "high water mark" of the text that has been parsed. Sometimes, the syntactic error will actually precede the point of the diagnostic, possibly by a large amount, which can make detecting the actual error difficult.

2.3. Parse Tree

While parsing the input, smgn builds a parse tree. In this tree, all terminals are removed. Figure 5 shows a conceptual representation of the generated parse tree for the input given in Figure 1 parsed with the grammar given in Figure 4. (This textual representation is inspired by the Graph Modelling Language (GML) [13, 10].)

```
start [
  turtle [
    identifier "triangle"
    cmd_list [
      commands [
        command [
          down [
            text "down"
          ]
          text "down;"
        ]
        command [
          turn [
            left_or_right [
              text "right"
            ]
            identifier "90"
            text "turn right by 90 degrees;"
          ]
          text "turn right by 90 degrees;"
        ]
        command [
          ...
        ]
        ...
        command [
          ...
        ]
      ]
      text "down; ... forward 50;"
    ]
    text "turtle triangle { down; ... }"
  ]
  text "turtle triangle { down; ... }"
]
```

Fig. 5. Parse tree for Turtle program.

Conceptually, every node in the parse tree contains an ordered list of key/value pairs; the key being the name of a nonterminal and the value being either another node or a string. In Figure 5, nodes are represented with square brackets and strings are enclosed in double quotation marks. A line starts with the name of the key followed by its value. In the Figure, strings have been abbreviated by using ellipses (...) and the original line breaks (which are preserved by smgn) are removed.

Nonterminals (except for the predefined nonterminals <identifier> and <verbatim>) have a special text key, whose value contains the matched input during the parse for that nonterminal. The text key is only present at nonterminals that did actually match some input text.

Note that a node can have keys with identical names. For example, in Figure 5 the commands

node contains several `command` keys (i.e., child nodes with identical names). This is the case for immediate right-recursive rules, i.e., rules of the following form:

$$<nt> ::= \alpha \mid \alpha <nt>$$

where α denotes an arbitrary sequence of terminals and nonterminals. The `Turtle` grammar in Figure 4 contains such a rule at lines 10–12 that defines the `commands` nonterminal.

Hence, the parse tree for immediate right-recursive rules is "flattened" by `smgn` to an order-preserving list. The macro language has a special `foreach` construct (see Section 2.4.3) to conveniently iterate over such lists.

Since `smgn` builds the parse tree automatically during the parse, the programmer need not worry about programming, maintaining, and documenting explicit tree construction code. Furthermore, the shape of the constructed tree is unambiguously documented/specified by the grammar.

On the other hand, automatic tree construction means that the shape of the resulting tree cannot be controlled; in particular, the direct construction of an abstract syntax tree (AST) is not possible. Even though this sounds like a serious restriction, in practice it is not, since `smgn` grammars are typically fairly simple and hence the resulting (flattened) parse tree is not much more complex than an AST. Furthermore, the initial tree can be augmented and transformed later on with the help of the macro language.

2.4. Macro Language

This section gives an informal (and necessarily incomplete) introduction of the macro language. For a more detailed description refer to [18].

The macro language follows the imperative programming paradigm. It has control flow constructs, macro definitions and calls, and expressions. There are commands for text handling, output management, and parse tree manipulation and traversal. Each of these features are briefly discussed in the following.

Most programmers are familiar with imperative programming. `smgn` builds on this paradigm, offering specific parse tree manipulation and traversal mechanisms on top. These concepts are easier to grasp for non-compiler experts than, for example, attribute grammars and algebraic specifications.

The term "macro" is rather misleading—it bears no resemblance with macro processing in C—since the macro language is in fact rather an interpreted command language. The name is kept for historical reasons.

2.4.1. A First Example

Here is the first simple example of a command sequence:

```
1  Turtle prg name is <turtle.identifier>
2  </>
3  Program text: <text>
4  </>
```

Text that is not escaped with angle brackets is written verbatim to the current output buffer (default is `stdout`). Leading white spaces are ignored. *Path expressions* (such as `turtle.identifier`) given in angle brackets `<...>` are used to output the contents of a node in the parse tree. An *absolute path expression*, such as the two above ones, starts at the root of the parse tree (omitting the leading nonterminal, which is unambiguous). The "`</>`" command outputs a newline.

If a path expression denotes a non-existing node, `smgn` generates a warning message and ignores the expression. (This behavior is helpful for debugging.) If the path expression leads to a node that does not contain a string value (i.e., it is not a leaf node), no warning is given and no output is generated.

The generated output for the above macro file applied to the `Turtle` program given in Figure 1 is as follows:

```
Turtle prg name is triangle
Program text:

turtle triangle {
    down;
    ...
    forward 50;
}
```

The `turtle.identifier` path expression denotes the node in the parse tree that contains the string "`triangle`." The `text` path expression denotes the string that contains the whole parse.

2.4.2. Macro Definitions

The above code can be rewritten by putting it in a macro definition (which corresponds to a subprogram) with two subsequent macro calls:

```
1  <def p txt node>
2    <txt>: <node></>
3  <enddef>
4
5  <p "Turtle prg name is " turtle.identifier>
6  <p "Program text" text>
```

The macro `p` takes two formal parameters, `txt` and `node`. A formal parameter can either represent a string or a node in the parse tree. In the latter case, the formal parameter can be used as the starting node of a *relative path expression*. Recursive macro calls are possible.

2.4.3. Control Flow

For control flow, `smgn` has a

$$< \texttt{if}(\textit{expr}) > \ldots < \texttt{else} > \ldots < \texttt{endif} >$$

construct. Typically, *expr* is a boolean expression that is used for string comparisons of nodes, or existence checks of nodes with the `exists` predicate. For convenient iteration over nodes that have identical path expressions, the `foreach` construct is provided:

```
1  <foreach cmd in
2   turtle.cmd_list.commands.command>
3    <pos cmd>: <cmd.text></>
4  <endfor>
```

All nodes that match the given path are visited, one at each iteration. The `cmd` parameter denotes the current node of the iteration and can be used in `foreach`'s body. The `Turtle` program in Figure 1 consists of seven commands, thus the path expression finds seven matching nodes (see also Figure 5):

```
0: down;
1: turn right by 90 degrees;
...
6: forward 50;
```

The `pos` command outputs the current number of iterations (starting from zero). Besides `pos`, inside `foreach` the predicates `first` and `last` can be used to determine if the current iteration processes the first and last node, respectively.

The `foreach` construct can be extended with a `such that` clause, which takes a boolean expression to put a restriction on the nodes to be iterated. For example, extending the previous example to only iterate over `forward` commands that advance by 50 steps, one can write:

```
1  <foreach cmd in ... such that
2   (cmd.forward.identifier == "50")>
3  ...
```

2.4.4. Parse Tree Manipulations

To introduce new nodes into the parse tree, the `set` construct is used:

```
1  <set brand.new.node
2   to <turtle.identifier>>
3  <set turtle.identifier
4   to A different string>
```

The first `set` creates a new node (actually three nodes are created: `brand`, `new`, and `node`). `brand` is attached as a child of `start`, the implicit root node. The contents of `node` is initialized with the string that the node `turtle.identifier` holds (i.e., "`triangle`"). Thus, the first set has the same effect as writing "`<set brand.new.node to triangle>`."

The second `set` overrides the contents of an existing node. (This is only possible if the node holds a string.) For a node initialization with `set`, only strings can be given.

Since path expression can become rather lengthy, aliases can be given for them with the `let` construct. For example, the above `set` statements can be rewritten as follows:

```
1  <let tid be turtle.identifier>
2    <set brand.new.node to <tid>>
3    <set tid to A different string>
4  <endlet>
```

The introduced alias (`tid`) is visible inside the `let` construct. Dynamic name binding applies.

So far all path expressions were literal. However, it is also possible to build path expressions dynamically. The name of a node in a path expression can be built from the string contents of any node or parameter. This indirection is achieved by giving the node name in square brackets:

```
1  <def insert key val>
2    <set arr[<key>] to <val>>
3  <enddef>
4
5  <insert "dog" "fido">
6  <insert "cat" "mimi">
```

This code inserts two new nodes with the path `arr.dog` and `arr.cat` with the string values `"fido"` and `"mimi"`, respectively. In a path expression, literal nodes and nodes given in square brackets can be freely interchanged. Both nodes `dog` and `cat` have `arr` as their parent node. Hence, this constructs effectively realizes an associate array, the key being the node name. To iterate over all values of `arr`, one can write:

```
1  <foreach val in arr.?>
2     <val></>
3  <endfor>
```

The "`?`" in the path expression is a wildcard that matches a single child node of `arr`. Given in combination with `foreach`, it can be used to iterate through all child nodes of `arr`.

Finally, there is also a `map` construct, which is a combination of `set` and `let`. It is similar to `set` in the sense that it defines a (new) node in the parse tree, and similar to `let` in the sense that an alias to another node is established.

2.4.5. Text Substitutions

Square brackets are also used for in-line text substitution. The constructs written between square brackets are processed and replaced with the *output* that they produce. The contents inside the brackets can be arbitrary complex, but in practice a path expression or a single macro call is used.

Text substitution can be used with a macro call to obtain a string result from a macro definition. For example, the following macro definition performs a string equality test and outputs either "`true`" or "`false`:"

```
1  <def eq s1 s2>
2     <eval (s1==s2)>
3  <enddef>
```

Now, if a call to this macro is done with text substitution, the generated string is not written to `stdout`, but substituted in-line. Thus, the result can be used in a boolean expression such as the following:

```
1  <if (! ([<eq "foo" "bar">]))>
2     Not identical...</>
3  <endif>
```

In this case, "`[<eq "foo" "bar">]`" will be substituted with "`false`" and the negation (`!`) evaluates the whole expression to true. Another example of text substitution is an indirect macro call. Here is an example:

```
1  <set themacro to eq>
2  <[<themacro>] "foo" "bar">
```

2.4.6. Polymorphism

One of the more interesting features is `smgn`'s support for polymorphic macro calls. The left-hand-sides of grammar productions have a type associated with them. In the following production

$$<nt> ::= \ldots$$

the nonterminal <*nt*> is associated with the type *nt*. A parse tree, which is constructed from the typed grammar, is typed as well. The type of a (parent) node that represents the application of a production in the derivation has the same type as the left-hand-side of that production.

Formal parameters of macro calls can be given types to restrict the applicability of the macro. The (optional) type is given after the formal parameter, separated with a colon. For example, the following macros handle the different kinds of `Turtle` commands:

```
 1  <def dp cmd:turn>
 2     I handle turn...</>
 3  <enddef>
 4
 5  <def dp cmd:up> ... <enddef>
 6  <def dp cmd:down> ... <enddef>
 7  <def dp cmd:forward> ... <enddef>
 8
 9  <def dp cmd>
10     I should never be called!
11  <endef>
```

The macro that is actually called during execution time depends on the type of the actual parameter. If no matching type is found, the last macro (which has no type restriction) is chosen. (Such an untyped macro can be used as a fall-back mechanism to implement default behavior or as a "guardian" that raises an error

to indicate that a macro for a certain type has not been implemented.)

Typically, a type restriction is placed only on the first parameter. This corresponds to single dispatch and can be conveniently used when iterating over a list of different types:

```
1  <foreach cmd in
2   turtle.cmd_list.commands.command.?>
3     <dp cmd>
4  <endfor>
```

The `foreach` iterates over the different types of `Turtle commands` and the macro call inside dispatches to the definition that has a matching type during runtime. Note that it is now quite easy to add another `Turtle` command. One simply needs to extend the grammar and write another macro definition.

One can place type restrictions on any number of formal parameters. This corresponds to multiple dispatch.

Types are not organized hierarchically (i.e., `smgn` has no concept of type inheritance). (Except that one can view the absence of a type as actually denoting the common supertype all the other types inherit from.) Because no type hierarchy exists, a polymorphic call has to find a macro definition with precisely matching types. That is why polymorphic calls can always be resolved unambiguously (which is not true of languages with polymorphic calls and type inheritance, such as Cecil [6]).

2.4.7. Output Management

In all of the above examples, the generated output was directed to `stdout`. `smgn` has a `file` command that redirects subsequent output to a given output buffer. This is especially useful if more than one file needs to be generated (e.g., `.h` and `.c` files).

It can be awkward to generate sequential output. Sometimes one would like to insert output at a certain position in the output buffer rather than being restricted to only appending at the end. In such a case, the `use` construct can be used to subdivide the buffer into different sections. Output can then be redirected to a specific section within a buffer. In the following example

```
1  <use sec1>
2  <use sec2>
3  This goes in section 2.</>
4  <use sec1>
5  This goes in section 1.</>
```

the first line defines (and also activates) `sec1`. The second line defines and activates `sec2`, whose output will be appended after `sec1`. This section will be active until a subsequent `use` command activates another one. This happens at the fourth line. Since `sec1` is already known, it is not defined, but only activated. Hence the generated output reads

```
This goes in section 1.
This goes in section 2.
```

2.4.8. Text Formatting

`smgn` has several commands to ease the text formatting of the output. Text indentation can be changed (relatively and absolutely) and names can be transformed before they are output. In the following example the text is output with an additional indentation of two relative to the previous indentation:

```
1  Previous indent
2  </+2></>
3  Additional indent of 2
4  </-2></>
5  Back to the previous indent</>
```

2.4.9. Command Line Invocation

When `smgn` is invoked, information can be passed to the macro file on the command line with

$$-\texttt{D}name{=}str$$

Before the execution starts, the node *name* is put at the root of the parse tree with the string value *str*.

2.4.10. Iterators with Callbacks

As can be seen from the above examples, during execution, the macro file uses path expressions to access the parse tree—path expressions drive the execution. But this means that a change in the grammar must be reflected by changing the corresponding path expressions, which is not desirable. Implementing iterators that take callbacks help to mitigate this problem:

```
1  <def iter callback>
2     <foreach cmd in
3      turtle.cmd_list.commands.command>
4       <[<callback>] cmd>
5    <endfor>
6  <enddef>
```

The iterator traverses the `commands` in a `Turtle` program and calls a `callback` macro for every command. The callback gets the current node (which represent the `command`) as an argument. The iterator can then be used as follows:

```
1  <def p cmd>
2     <pos cmd>: <cmd.text></>
3  <enddef>
4
5  <iter "p">
```

The behavior of this code is identical with the code given in Section 2.4.3, but is more reusable. Thus, if iterators are employed, after a grammar change often only a single path expression in the iterator needs to be changed. This concept has been extensively used at the implementation of Bauhaus IMDL (see Section 3).

It would be beneficial to introduce high-level constructs into the macro language that specify the traversal of the tree structure. These constructs, analogous to the visitor pattern, separate navigation from tree node computation and allow reuse of traversal patterns. Ovlinger and Wand present such a traversal specification language for use with the visitor pattern [25].

2.5. Historical Development

`smgn` started as a small macro processor for generating Web pages from lists. (That is why its syntax is determined by angle brackets.) When, in the context of the SUIF project, the ability to generate C++ from *Hoof* was needed, the second author hooked up a back-tracking parser (which had been developed before for another purpose) and added the tree flattening capability discussed in section 2.4.3 to make the existing `foreach` construct work. Since an already existing back-tracking parser was reused opportunistically, its BNF-style grammar description was kept. In retrospect, the parser could have introduced a more convenient, EBNF-style list construct instead of flattening the right-recursive rules.

Over time, constructs were added as convenient. While we tried to be as consistent as possible in syntax, the language as it exists today was grown rather than designed, and, as a result, is rather idiosyncratic and cumbersome. Also, the text formatting capabilities lack elegance and generality. These shortcomings (and the ones discussed in Section 3.4) should be fixed in a second version of `smgn`.

2.6. Implementation

`smgn` is implemented with about 4500 lines of C++ code. We used the UNIX `wc` to determine the line count, but omitted empty lines. The code is rather sparsely documented. The implementation uses common data structures (such as stacks and strings) that are part of the SUIF base system. These data structures are not included in the above line count.

No compiler construction tools (such as `lex`/`yacc`) have been used. Instead, both scanner and parser are hand-crafted. The macro interpreter directly interprets the macro file; parsing and macro interpretation are intertwined and handled on-the-fly. (That is why `smgn` requires macro definitions to appear textually before their usage.) An alternative implementation could parse the macro file and construct an intermediate representation before interpretation starts. Such an approach would certainly be beneficial from a performance point of view. Our current approach of hand-crafting the parser has the drawback that no formal grammar of the macro language exists. Even though the need for such a grammar did not arise because of our implementation strategy, constructing one would certainly be feasible. Furthermore, current implementation has an additional drawback in that the generated output is kept in-memory and written out after processing. Hence, if a lot of output is generated, the memory footprint can become large. The current implementation is rather a prototype and proof of the concept. Nevertheless, the implementation is stable and has adequate performance to handle typical DSLs (see next Section).

2.7. Scope

Now that the reader is more familiar with smgn, the type of DSLs that are good candidates for an implementation based on smgn are discussed. A suitable DSL should have the following properties:

- The DSL should have be of declarative nature. Examples of declarative DSLs are SCATTER [4], lex, yacc, and ASDL [32]. This requirement follows the philosophy that DSLs should not be designed to describe computation, but rather to express facts (from which computations can be derived). Furthermore, declarative DSLs are well suited for compilation, which is not always the case for DSLs with execution semantics.
- The transformation from the source to the target language should be fairly simple (e.g., every construct in the source language directly corresponds to a construct in the target language). Complex transformations are not well supported because smgn does not offer pattern matching on the parse tree. Instead, pattern matching rules have to be explicitly programmed with the macro language in an imperative style.
- DSDs written in the DSL should be rather small—about up to several thousand lines of code. The reasons for this limitation are lack of syntax checks (which make debugging difficult) and performance issues (as discussed in Section 2.6) of the current implementation. Since a declarative style results in more concise code, this should not pose a serious restriction. For example, the DSD sizes of 15 DSLs discussed in [27] range from 42 to 2490 lines of code (LOC). The *Hoof* DSL has DSDs that range from 400 to 2000 LOC. Bauhaus IMDL has about 2000 LOC.
- The DSL should be expressible with a rather small grammar (i.e., no more than 100 productions). The grammar is kept in a single file. This is adequate for a small grammar; however, for a larger grammar it is desirable to split it up into several files. A declarative DSL usually means a less complex grammar. For example, arithmetic expressions and control structures need not be modeled. *Hoof* has 58 and Bauhaus IMDL has 51 productions.
- The DSL must be compilable and the generated output must have a textual form. This is the case for DSLs that get translated to another DSL or programming language.
- Fast compilation is not a paramount requirement. However, since the input files tend to be small, a reasonable compile time can be expected.

To summarize, we believe that smgn is useful for rapid prototyping of small DSLs that require rather simple transformation to textual output.

3. Experiences

This section describes experiences based on three independent DSL projects that have been conducted with smgn.

3.1. *Hoof*

smgn was designed and developed with the explicit goal to support *Hoof*. *Hoof* is a DSL that is employed in the SUIF compiler system [26] to specify SUIF's intermediate representation (IR), which is used by front ends and optimization passes. The IR consists of node definitions, which are organized in a specialization hierarchy. A node definition introduces a new node type. It can be concrete (i.e., instantiable) or abstract. Nodes can have fields, which are either primitive (e.g., integer or string), or hold a typed reference to an IR node. To realize functionality that is not covered by *Hoof*, it is also possible to specify C++ code in-line, which is verbatimly pasted into the generated output code. Here is an example of a simplified node definition:

```
concrete IfStatement : Statement {
  Expression * condition;
  Statement * then_part;
  Statement * else_part;
};
```

Nodes have common services, such as instantiation, annotation, reading, writing, printing, and cloning. *Hoof* is translated to C++, the above definition causing the generation of about 100 LOC.

One of SUIF's primary goals is extensibility, which means that applications that are not part of the standard SUIF distribution can further refine the IR hierarchy. *Hoof* is easy to grasp and thus makes the introduction of a new IR node very convenient, shielding the intricacies of the underlying C++ implementation from the user.

Since the development of smgn is tied to *Hoof*, it is a justified question whether it is suited in general to implement DSLs. Experiences of the two projects described below give the first answer.

3.2. *Hoof* for Java

The JSUIF project [17], which has been conducted independently from SUIF, ports SUIF from C++ to Java. For this port, the macro files for *Hoof* had to be rewritten to generate Java code instead of C++. (Note that ideally neither the *Hoof* grammar nor the *Hoof* DSD requires change.) The author, Radu Iosif, who ported the macro files, had no prior experience with smgn, but was able to familiarize himself with it in about two to three days. The tree navigation was felt to be intuitive; dynamic path expressions and text substitution are hardest to grasp. On the downside, the constructs that smgn provides for generating debugging output were perceived as rather confusing. JSUIF's author got a positive impression of smgn and would consider using it for other DSL-related projects.[1]

For the port, most of the original structure of the macro code could be retained. The most intrusive change occurred at the output file handling. JSUIF generates a separate Java file for each node definition, whereas SUIF uses a single C++ file. To interface with the Java Native Interface (JNI), it was necessary to generate names mangled according to the JNI convention. Since this is not supported by smgn, a single new construct realizing the mangling had to be introduced. For *Hoof* itself, the grammar had to be extended with a single production that allows to specify Java code in-line (in addition to the already supported C++).

3.3. Bauhaus IMDL

To give further insight into the question whether smgn is suited for DSL construction in general, it was used for the development of another DSL, called IMDL, which is part of the Bauhaus toolset [2].

Bauhaus, developed at the University of Stuttgart, is a reverse engineering toolset that facilitates program understanding of C legacy code. Bauhaus assists a reengineer in deriving the architecture of the system under examination. Before the system is analyzed, it is first compiled with the Bauhaus C front end. The front end targets Bauhaus InterMediate Language (IML), which is essentially a persistent attributed tree representation. The previous implementation modeled the tree data structure manually in Ada95. The introduction of a new tree node was accomplished with copy-and-paste from other existing nodes. Because of the resulting maintenance problems, it was decided to design a DSL called InterMediate Description Language (IMDL), that specifies the tree nodes.

As an experiment, it was decided to implement IMDL with smgn. The architecture of the system is depicted in Figure 6. The IMDL file

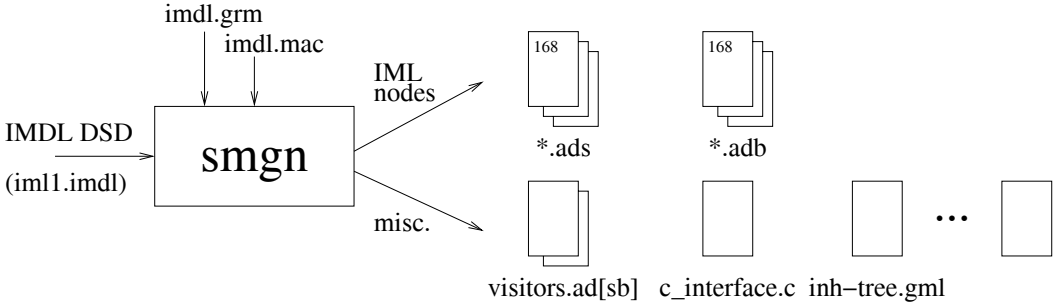

Fig. 6. Architecture of Bauhaus IMDL.

[1] Private email communication with Radu Iosif, Politecnico di Torino, January 2001.

(`iml1.imdl`) describes 168 tree nodes and is about 2000 LOC. From this description several output files are generated. For each specified tree node the corresponding Ada95 code is generated—on average about 300 LOC for both specification (`.ads`) and implementation (`.adb`) file. Other files are generated as well, for example, an Ada95 visitor class for nodes traversal, interface functions for accessing the nodes from C, and, for documentation, a representation of the node inheritance (described in the GML graph format [13, 10]).

Command line options govern which output files are generated. Output generation is split up in different macro files. If certain output needs to be generated, the startup macro file (`imdl.mac`) loads the corresponding macro file on-the-fly and starts to interpret it. Thus, the generation of additional output files can be easily realized by writing a new macro file and registering it with the startup macro file. Macro files use the library of macro definitions that provides support for output string generation and tree traversals with callbacks.

The design process of IMDL turned out to be highly iterative. In fact, our experiences were similar to those of the designers of the Hancock language [5]. Experimental constructs were first introduced in the grammar and used in the node specifications (without output generation) to evaluate their merit. Once the expressiveness of a construct turned out to be appropriate, output for it was generated. Output generation sometimes also caused redesign of the construct. Since IMDL's grammar was incrementally enhanced, it turned out to be beneficial that `smgn` does not pose constraints on the grammar. `smgn`'s limited lexical capabilities did not cause any problems. During the development, other Bauhaus members with no prior knowledge of `smgn` could easily fix bugs and further enhance IMDL's functionality. Even though we implemented a new DSL and generated output for Ada95, `smgn` proved to be well suited for this task. Specifically, it was not necessary to enhance `smgn`'s functionality.

`smgn`'s runtime performance is adequate. A run, that outputs all files shown in Figure 6, takes about 90 seconds on a Sun Ultra-2 with 512MB RAM running Solaris 2.5.1. The files contain about 60000 lines, each line containing on average 28 characters.

3.4. Deficiencies of `smgn`

While working with `smgn`, the following deficiencies were noticed:

- The grammar specification and the path expressions in the macro code are tightly coupled. Thus, a change in the grammar usually means changes in the macro code. (As discussed in Section 2.4, using callbacks can mitigate this problem.)

- It is not possible to abstract from the concrete syntax given by the grammar (e.g., with an AST).

- `smgn` can only parse a single DSD file and does not offer a built-in include mechanism at the language level. Hence, the DSL must be designed such, that each DSD consists of a single file. However, a preprocessing step (e.g., with `m4` or the C compiler preprocessor `cpp`) can be employed that alleviates this restriction. For example, in the SUIF system a Makefile ensures that each *Hoof* file is preprocessed with `cpp`. Thus, other *Hoof* files can be textually included with `cpp`'s `#include` directive.

- There is little syntax checking of the macro files. For syntax errors, `smgn` typically fails unpredictably.

- There is no good support to generate error messages. Most notably, no line number information is preserved in the parse tree.

- It is not possible to call external programs ("shell escape"). An even better approach would be to interface to other (scripting) languages.

- No extensibility mechanisms are offered, for example, to define new constructs for output mangling. (This can be easily achieved with the aid of dynamically loaded shared libraries.)

The last four points are deficiencies of the current implementation rather than conceptual shortcomings.

4. Classification

smgn provides support for scanning, parsing, and construction of the parse tree, which means that its support is limited to standard, well-known generator techniques as they are typically supported by compiler toolkits (e.g., Eli [8] and Cocktail [7]). Thus, smgn is less ambitious than language development systems which employ formal semantics to automatically generate tools [12]. For example, the ASF+SDF language development system uses an algebraic specification with conditional rewrite rules [29]. From such a specification, tools (such as scanner, parser, pretty-printer, type-checker, and interpreter) are generated.

Language development systems are typically complex (sometimes with several different specification languages) and require a fair amount of time to learn. Long-term benefits of these systems can outweigh the effort of learning the language, but potential implementors of a DSL are not always willing to pay the initially high price of familiarizing themselves with such a system—especially if at the beginning of the design and implementation phase it is not clear whether a DSL will prove to be beneficial in the first place. (This paper neither covers the risks of developing a DSL nor discusses alternatives to DSLs. These topics are addressed in [21, 23, 30, 31].)

On the other side of the scale, DSLs are still often built with no or only rudimentary tool support. Typically, a scanner generator (e.g., lex) and a parser generator (e.g., yacc) are employed, but this means that (1) abstract syntax trees (or parse trees) must be implemented by hand, (2) there is no convenient, high-level interface to navigate the tree, and (3) the generation of (textual) output is awkward if the implementation language does not offer high-level string processing. These reasons can cause programmers to refrain from building an experimental DSL because it is hard to design, build and maintain such an implementation.

smgn lies somewhere in the middle of these two approaches. It provides support for easy grammar construction, automatic tree construction, and a high-level macro language for tree traversal/manipulation and convenient output generation. Furthermore, the macro language is easy to grasp for programmers. These features render smgn more suitable for rapid prototyping than the lex/yacc-style. This point is illustrated by Figure 7. In the lex/yacc-style, the programmer has to write two separate specifications for the scanner and parser, has to implement the data structures for the AST, and finally has to insert the appropriate calls for the AST construction in the parser's actions. There is no tool support for tree traversal and code generation. With smgn a single (unrestricted) grammar specification suffices along with a macro file to generate the target output.

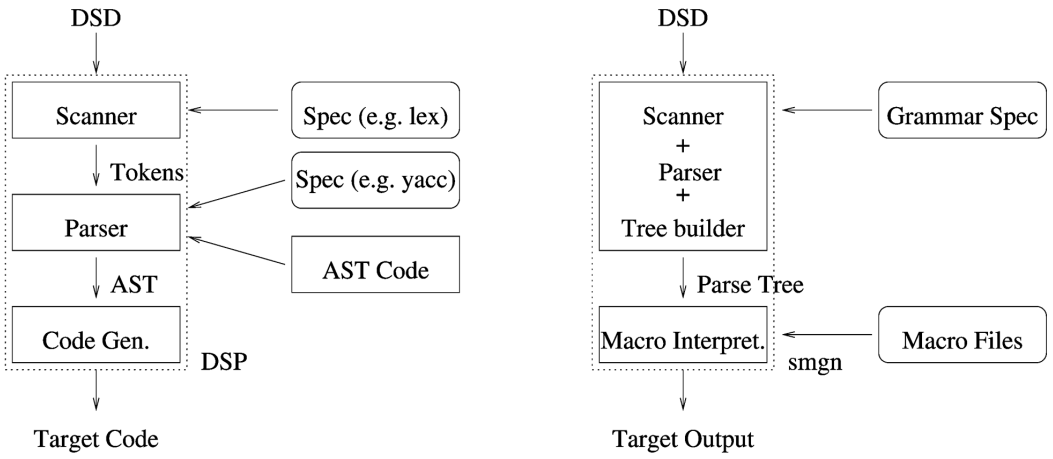

Fig. 7. DSP Architectures: lex/yacc-Style vs. smgn.

5. Related Approaches

This section discusses other approaches that are also suitable for rapid prototyping of DSLs. Considering the classification introduced in Section 4, these approaches are similar to `smgn` because they provide more sophisticated support for DSL construction than the `lex`/`yacc`-style while being less general than fully-fledged language development systems.

The approach of [1] is based on an extensible compiler framework written in Python. The framework provides support for scanning, parsing, and AST construction and traversal. It is customized by means of subclassing from dedicated base classes and can be used for interpreting as well as compiling the DSL. The parser uses the Earley parsing algorithm, which frees the user from intricate grammar design. Tree traversal of the AST can be used for semantic analysis and code generation. This approach is similar to the `lex`/`yacc`-style of developing DSLs, offering mainly support for scanning and parsing. Similar to `smgn` and in contrast to `yacc`, the grammar writer need not worry about grammar constraints. ASTs must be constructed manually and tree navigation is only supported by means of tree traversals. In order to use the framework, a sound knowledge of Python is required.

Jargons [23] are domain-specific extensions implemented on top of a base interpreter. The common base syntax consists of jargon expressions, which are associated with actions. During interpretation of an expression, its corresponding action is executed and the expression's product is appended to an output buffer. Jargons are rapidly developed because neither a scanner nor a parser must be constructed since the syntax is already prescribed. The authors state that the syntax is versatile enough to be applicable to various domains. Because of these features, jargons are well suited for rapid development and prototyping. The authors state: "We have found that developers who cannot make a little language can easily make a jargon in a day or two." Jargons can have multiple semantics customized by their actions. Thus a single jargon can have many different products. `smgn` can be employed similarly by running the macro interpreter with the same grammar and input file, but with different macro files.

The Khepera system [9] translates a DSL by specifying a sequence of tree transformation rules. In fact, it is a transformation system such as TXL [28] and DMS [3]. The rules are given in a transformation language that operates upon an AST. Khepera provides library routines for AST construction. Scanner and parser tools can then be employed to translate a DSD into the corresponding AST. For target output generation and pretty-printing, the user can give a short description in a `printf`-like syntax for every node type in the AST. By decoupling the three stages of the translation process, Khepera "is better able to accommodate changes during the evolution of the DSL syntax and semantics." Its features "facilitate the problem of rapid DSL prototyping and the problem of long-term DSL maintenance." Khepera does not simplify the scanner and parser implementation for a DSL, but rather sits on top of them.

The Depot4 [19] application generator is a grammar-based tool that combines parsing with semantic actions. The grammar (which must not be left-recursive) is described in EBNF. The grammar rules can be augmented with control expressions that, for example, describe the number of repetitions or which alternative has been taken during the parse. These control expressions can also be used as constraints on the grammar constructs, for example, to force a specific number of repetitions. Thus, it is possible to describe a context-sensitive language. For every EBNF production, a translation rule can be given that maps the parsed entities in the grammar rule to the desired textual output format. The translation rules manage to hide the underlying complexity. Depot4's "approach stresses fast and easy usability by non-experts." Its application domain "come[s] from areas where rapid implementation is essential, as in prototyping."

Another implementation strategy is the embedding of DSLs into higher-order, typed (HOT) languages [16, 20, 24]. When embedding a DSL into a host language, the DSL has to adhere to the syntactic limitations imposed by the host language. In [24], the functional decomposition caused "aesthetic problems when reading" the code. If the host language is employed to type-check the DSL (as implemented in [20]) the resulting error messages can be hard to decipher by a programmer not intimately familiar

with the host language. On the positive side, the host language adds to the expressiveness of the DSL, because DSL code can be intermingled with host language code.

6. Future Work

A reimplementation of smgn, which would also provide a cleaner implementation and remedy its current deficiencies (see Section 3.4), could incorporate several design changes as discussed below.

The typing system of the parser needs to be made first class. Only having types associated with the results of parsing (refer to Section 2.4.6) is limiting. This could also be used in making grammars look like macros; one would add right-hand-sides to a left-hand-side (a type) through derivation.

Furthermore, separation of the grammar and macro language should be eliminated. One should be able to use the language for round-trip engineering. For this purpose, one needs to be able to drive the grammar "backwards" (in the thermodynamic sense), to be able to manipulate the tree, and then automatically output the new text. There would have to be no logical distinction between a grammar production and a macro. Similarly, the language should not have a fixed representation. Instead, a round-trip parser working in whatever representation could be employed. Rather than having to translate back into smgn, the smgn parser should be written in smgn and you should be able to replace it.

Finally, one should be able to control the flattening process. An obvious possibility would be to attempt to use attribute grammars for this purpose.

Acknowledgments

Jörg Czeranski and Thomas Eisenbarth were involved in the design and initial prototype implementation of IMDL. Thanks to Radu Iosif for sharing his experiences in developing JSUIF.

Thanks to Arie van Deursen for many suggestions that helped greatly to improve the paper. Thanks to Burkhard Burow, Urs Hölzle, and Erhard Plödereder for recommendations and for proofreading an earlier version of the paper. Last but not least, thanks to the anonymous referees for further recommendations and proofreading.

A. Macro File for PostScript Generation

The following is a larger example in smgn's macro language. It generates (naïve) PostScript output for Turtle programs (see Section 1.1). To keep the code as concise as possible, no iterators with callbacks (refer to Section 2.4.10) are used. A solution that uses iterators—and is thus easier to maintain—is given in [18].

Several new nodes are introduced that act as variables and keep the turtle's state. The angle node holds the current orientation of the turtle and pen contains either the string "up" or "down." For a real implementation, it is better to keep the turtle's state in the target language (e.g., using PostScript variables), but this implementation is better suited to demonstrate smgn's abilities.

```
1   <def turn left_or_right newangle>
2      <if (left_or_right == "left")>
3         # Reverse the sign
4         <set newangle to <eval (0 - newangle)>>
5      <endif>
6
7      # Compute new angle
8      <set angle to <eval (angle + newangle)>>
9   <enddef>
10
11  <def forward length>
12     # Compute new position of pen:
13     #   sin(angle) * length + cos(angle) * length
```

```
14      <angle> sin</>
15      <length> mul</>
16      <angle> cos</>
17      <length> mul</>
18
19      <if (pen == "up")>
20         rmoveto</>
21      <elseif (pen == "down")>
22         rlineto</>
23      <endif>
24   <enddef>
25
26   <file out.ps>
27   <set pen to up>
28   <set angle to 0>
29
30   4 setlinewidth</>newpath</>
31   200 200 moveto</>
32
33   <foreach cmd in turtle.cmd_list.commands.command>
34      <if (exists cmd.turn)>
35         <turn cmd.turn.left_or_right.text cmd.turn.identifier>
36      <elseif (exists cmd.up)>
37         <set pen to up>
38      <elseif (exists cmd.down)>
39         <set pen to down>
40      <elseif (exists cmd.forward)>
41         <forward cmd.forward.identifier>
42      <endif>
43   <endfor>
44
45   stroke</>showpage</>
```

References

[1] JOHN AYCOCK, Compiling Little Languages in Python, *Seventh International Python Conference*, pages 69–77, November 1998.

[2] Projekt Bauhaus, `http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus`.

[3] IRA D. BAXTER, Design Maintenance Systems, *Communications of the ACM*, 35(4):73–89, April 1992.

[4] JON BENTLEY, Little Languages, *Communications of the ACM*, 29(8):711–721, August 1986.

[5] DAN BONACHEA, KATHLEEN FISHER, ANNE ROGERS AND FREDERICK SMITH, Hanckock: A Language for Processing Very Large-Scale Data, *2nd Conference on Domain-Specific Languages (DSL '99)*, pages 163–176, October 1999.

[6] CHRAIG CHAMBERS AND THE CECIL GROUP, *The Cecil Language: Specification and Rationale*, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, USA, December 1998.

[7] Cocktail Toolbox Home Page, `http://www.-cocolab.de/html/cocktail.html`.

[8] Eli Home Page, `http://www.cs.colorado.edu-/~eliuser/`.

[9] RICKARD E. FAITH, LARS S. NYLAND AND JAN F. PRINS, Khepera: A System for Rapid Implementation of Domain Specific Languages, *Conference on Domain Specific Languages*, pages 243–255, October 1997.

[10] Graphlet Homepage, `http://fmi.uni-passau.-de/Graphlet`.

[11] BRIAN HARVEY, *Berkley Logo User Manual*, University of California Berkley, 1993.

[12] JAN HEERING AND PAUL KLINT, SEMANTICS OF PROGRAMMING LANGUAGES: A TOOL-ORIENTED APPROACH, *ACM SIGPLAN Notices*, 35(3):39–48, MARCH 2000.

[13] MICHAEL HIMSOLT, GML: GRAPH MODELLING LANGUAGE, UNIVERSITY OF PASSAU, GERMANY, UNPUBLISHED, DECEMBER 1996.

[14] PAUL HUDAK, BUILDING DOMAIN-SPECIFIC EMBEDDED LANGUAGES, *ACM Computing Surveys*, 28(4ES):ARTICLE NO. 196, DECEMBER 1996.

[15] ADOBE SYSTEMS INCORPORATED, *PostScript* LANGUAGE: TUTORIAL AND COOKBOOK, Addison-Wesley, 1986.

[16] JAMES JENNINGS AND ERIC BEUSCHLER, Verischemelog: Verilog embedded in Scheme, *2nd Conference on Domain-Specific Languages (DSL '99)*, pages 123–134, October 1999.

[17] JSUIF Home Page, `http://www.dai-arc.polito.it/dai-arc/manual/tools/yav-/jsuif`.

[18] HOLGER M. KIENLE, The smgn Reference Manual, Technical Report TRCS00–22, Department of Computer Science, University of California Santa Barbara, November 2000.

[19] JÜRGEN LAMPE, Depot4 – A generator for dynamically extensible translators, Software – Concepts & Tools, 19(2):97–108, 1998.

[20] DAAN LEIJEN AND ERIK MEIJER, Domain Specific Embedded Compilers, *2nd Conference on Domain-Specific Languages (DSL '99)*, pages 109–122, October 1999.

[21] MARJAN MERNIK, UROS NOVAK, ENIS AVDICAUEVIC, MITJA LENIC AND VILJEM ZUMER, Design and Implementation of Simple Object Description Language, In *16th ACM SAC2001 symposium on Applied computing*, pages 590–595, 2001.

[22] GAIL C. MURPHY AND DAVID NOTKIN, Lightweight Lexical Source Model Extraction, ACM Transactions on Software Engineering and Methology, 5(3):262–292, July 1996.

[23] LLOYD H. NAKATANI, MARK A. ARDIS, ROBERT G. OLSEN AND PAUL M. PONTRELLI, Jargons for Domain Engineering, *2nd Conference on Domain-Specific Languages (DSL '99)*, pages 15–24, October 1999.

[24] KURT NØRMARK, Programming World Wide Web Pages in Scheme, *ACM SIGPLAN Notices*, 34(12):37–46, December 1999.

[25] JOHAN OVLINGER AND MITCHELL WAND, A Language for Specifying Recursive Traversals of Object Structures, *Conference on Object Oriented Programming Systems Languages and Aplications (OOPSLA '99)*, pages 70–81, November 1999.

[26] The SUIF2 Compiler System, http://suif.stanford.edu/suif/suif2

[27] DIOMIDIS SPINELLIS AND V. GURUPRASAD, Lightweight Languages as Software Engineering Tools, *Conference on Domain Specific Languages*, pages 67–76, October 1997.

[28] TXL Home Page, http://www.txl.ca.

[29] A. VAN DEURSEN, J. HEERING AND P. KLINT, editors, *Language Prototyping*, volume 5 of AMAST Series in Computing, World Scientific, 1996.

[30] ARIE VAN DEURSEN AND PAUL KLINT, Little Languages: Little Maintenance?, *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, March/April 1998.

[31] ARIE VAN DEURSEN AND PAUL KLINT AND JOOST VISSER, Domain-Specific Languages: An Annotated Bibliography, *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[32] DANIEL C. WANG, ANDREW W. APPEL, JEFF L. KORN AND CHRISTOPHER S. SERRA, The Zephyr Abstract Syntax Description Language, *Conference on Domain-Specific Languages*, pages 213–228, October 1997.

Received: July, 2001
Revised: October, 2001
Accepted: November, 2001

Contact address:
Holger M. Kienle
Department of Computer Science
University of Victoria
Room ELW 342, Engineering Lab Wing
Canada
Phone: +1 250 721 7294
e-mail: kienle@csr.uvic.ca

David L. Moore
Intel Compiler Lab
Hillsboro, OR 97124
U.S.A.
e-mail: David.Moore@intel.com

HOLGER M. KIENLE received his Master of Science degree in Computer Science from the University of Massachusetts Dartmouth (1995) and his Diploma in Computer Science from the University of Stuttgart, Germany (1999). He received a two year Post Graduate Research Fellowship (1997–1998) from the University of California Santa Barbara to work as a member of Professor Hölzle's Object-Oriented Compilers group. He is currently a Ph. D. candidate in Computer Science at the University of Victoria, Canada, where he is a member of Professor Müller's Rigi group. His interests include software reverse engineering, programming languages, program analyses, and domain-specific languages.

DAVID MOORE received his B. Sc. (1973) in Mathematics and Physics from the University of Queensland and M. Sc. (1983) in Computer Science from Queensland Institute of Technology. He is involved in the National Compiler Infrastructure (NCI) Project, working as project manager at PGI. Smgn, of which he is the chief designer, is a part of the NCI project. He is currently a software engineer with Intel. He has been a long time student of type safe programming languages, an early evangelist for Pascal and the implementor of a Modula-2 compiler and environment for CP/M-80 and MSDOS machines.