Using smgn for Rapid Protoptyping of Small Domain-Specific Languages

Holger M. Kienle
Department of Computer Science
University of Stuttgart
Breitwiesenstr. 20–22
D-70565 Stuttgart
Germany
kienle@informatik.uni-stuttgart.de

Abstract

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 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 rapid development of the *Hoof* domain-specific language. Furthermore, smgn was recently employed for the rapid development of another domain-specific language, called Bauhaus IMDL. These successful experiences motivate the promotion of smgn in the hope that other researchers that face the task of implementing a domain-specific language will find smgn equally helpful.

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 qualifies them as such.

This paper defines a DSL as a small formal (programming) language whose expressive power covers precisely a certain application domain and does not include many of the features found in general-purpose programming languages. (Alternative definitions are proposed in [3, 17].) 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) [16]. If 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 [8], 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 rapid prototype a DSL is beneficial because such a prototype can be used to get feedback from users early on, for example, to uncover missing requirements. For example, the designers of the Hancock language [4], 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 [11], called smgn, is a simple tool well suited for rapid prototyping of a certain class of DSLs. In a nutshell, smgn is a grammar-based tool that allows to parse an input file according to a grammar specification. The resulting parse tree can then be easily navigated 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.

Even though smgn has been specifically written for the SUIF2 Compiler System [14], 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 [2], smgn has been employed to implement a DSL, called IMDL, to describe an internal graph representation.

1.1 Turtle Example

As a running example, it is shown how to implement a small DSL, called Turtle, with smgn. A Turtle program draws a (static) picture by means of turtle graphics.[1]

```
# triangle.ttl

turtle triangle {
    down;
    turn right by 90 degrees;
    forward 50;
    turn left by 120 degrees;
    forward 50;
    turn left by 120 degrees;
    forward 50;
}
```

Figure 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 simple operations, Figure 1 shows how a triangle can be drawn. The drawing is depicted in Figure 2.

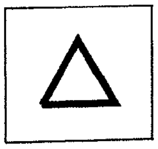

Figure 2: Drawing of triangle (specified in Figure 1)

[1]The Turtle commands are inspired by Logo turtle graphics [6].

Our goal is to write a DSL compiler that translates a Turtle program to PostScript [9] 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.

```
4 setlinewidth
newpath
200 200 moveto
90 sin
50 mul
90 cos
50 mul
rlineto
-30 sin
50 mul
-30 cos
50 mul
rlineto
-150 sin
50 mul
-150 cos
50 mul
rlineto
stroke
showpage
```

Figure 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.

2 smgn's Features and Scope

First, smgn is briefly introduced to give the reader a first impression of its capabilities and features. Based on these, we identify the properties that DSLs should preferably posses to be applicable for an implementation with smgn.

2.1 Overview

When smgn is invoked, its actions are parameterized by the following files, which are given at 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
```

First, **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.

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).

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. 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 text that is not supposed to be parsed and subsequently analyzed, but written out later in its original form.

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.

```
# turtle.grm

<start> ::= <turtle>

<turtle> ::= turtle <identifier>
             { <cmd_list> }

<cmd_list> ::= <commands> |

<commands> ::=
      <command>
    | <command> <commands>

<command> ::=
      <turn>
    | <up>
    | <down>
    | <forward>

<turn> ::= turn <left_or_right>
           by <identifier>
           degrees ;
<left_or_right> ::= left | right

<up> ::= up ;
<down> ::= down ;
<forward> ::= forward
              <identifier> ;
```

Figure 4: `Turtle` Grammar (`turtle.grm`)

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. 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 probably best to try to write 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.

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) [7].)

```
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 { ... }"
  ]
  text "turtle triangle { ... }"
]
```

Figure 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 omitted.

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 8–10 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 a list.

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 tree cannot be controlled; in particular, the 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.

2.4 Macro Language

This section gives an informal (and necessarily incomplete) introduction of the macro language. For a more detailed description refer to [11].

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.

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 a first simple example of a command sequence:

```
Turtle prg name is <turtle.identifier>
</>
Program text: <text>
</>
```

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 looks 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 subsequently macro calls:

```
<def p txt node>
  <txt>: <node></>
<enddef>

<p "Turtle prg name is " turtle.identifier>
<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

<if (*expr*)> ...<else> ...<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:

```
<foreach cmd in
 turtle.cmd_list.commands.command>
  <pos cmd>: <cmd.text></>
<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, to only iterate over forward commands that advance by 50 steps, one can write

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

2.4.4 Parse Tree Manipulations

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

```
<set brand.new.node
 to <turtle.identifier>>
<set turtle.identifier
 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. 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:

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

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

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.

So far all path expression 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.

```
<def insert key val>
  <set arr[<key>] to <val>>
<enddef>

<insert "dog" "fido">
<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

```
<foreach val in arr.?>
   <val></>
<endfor>
```

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

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

performs a string equality test and outputs either "true" or "false." 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

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

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

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

2.4.6 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 section construct can be used to subdivide the buffer into different sections. Output can than be redirected to a specific section within a buffer.

2.4.7 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.

2.4.8 Command Line Invocation

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

```
-Dname=str
```

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

2.4.9 Iterators with Callbacks

As can be seen by 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.

```
<def iter callback>
   <foreach cmd in
    turtle.cmd_list.commands.command>
     <[<callback>] cmd>
  <endfor>
<enddef>
```

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

```
<def p cmd>
   <pos cmd>: <cmd.text></>
<enddef>

<iter "p">
```

The behavior of this code is identical with the code given in Section 2.4.3, but 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).

2.5 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 a declarative nature. Examples of declarative DSLs are SCATTER [3], lex, yacc, and ASDL [18]. 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.
- DSDs written in the DSL should be rather small—about up to several thousand lines of code. 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 [15] range from 42 to 2490 lines of code (LOC). The *Hoof* DSL has DSDs that range from 400 to 2000 LOC.
- The DSL should be expressible with a rather small grammar (no more than 100 productions). 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 a rather simple transformation to textual output.

3 Experiences

This section gives some experiences with smgn that have been gained during the development of two DSLs, namely *Hoof* and Bauhaus IMDL.

smgn was designed and developed with the explicit goal to support *Hoof*. *Hoof* is a DSL that is employed in the SUIF2 compiler system [14] to specify SUIF's intermediate representation (IR), which is used by front ends and optimization passes. 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. To answer this question, smgn was used for the development of another DSL, called IMDL, which is part of the Bauhaus toolset.

Bauhaus [2], developed at the University of Stuttgart, is a reverse engineering toolset that facilitates program understanding of C legacy code. 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-past 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 implemented IMDL with smgn. The design process of IMDL turned out to be highly iterative. 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. Currently 168 tree nodes are described with IMDL (in about 2000 LOC) and the generated Ada95 code for a single node is on average about 330 LOC.

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).
- There is no 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").
- No extensibility mechanisms are offered, for example, to define new constructs for output mangling.

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

4 Related Approaches

This section discusses other approaches that are also suitable for rapid prototyping of DSLs.

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. 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 [13] 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.

The Khepera system [5] translates a DSL by specifying a sequence of tree transformation rules. 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. Khepera's features "facilitate the problem of rapid DSL prototyping and the problem of long-term DSL maintenance."

The Depot4 [12] application generator is a grammar-based tool that combines parsing with semantic actions. 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. 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."

5 Resources

- smgn is available as part of the SUIF Compiler System [14].
- An (informal) smgn reference manual is available as technical report [11].
- The Java-port of the SUIF system is a available at the JSUIF home page [10].
- More information about Bauhaus is available at the Bauhaus home page [2].

Acknowledgments

Jörg Czeranski and Thomas Eisenbarth were involved in the design and initial prototype implementation of IMDL.

Thanks to Burkhard Burow, Urs Hölzle, and Erhard Plödereder for proofreading and recommendations that helped to improve the paper.

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] Jon Bentley. Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[4] 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.

[5] 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.

[6] Brian Harvey. *Berkley Logo User Manual.* University of California Berkley, 1993.

[7] Michael Himsolt. GML: Graph modelling language. Draft, Unpublished, December 1996.

[8] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):Article No. 196, December 1996.

[9] Adobe Systems Incorporated. *PostScript Language: Tutorial and Cookbook.* Addison-Wesley, 1986.

[10] JSUIF home page. http://www.dai-arc.polito.-it/dai-arc/manual/tools/yav/jsuif.

[11] Holger M. Kienle. The smgn reference manual. Technical Report TRCS00-22, Department of Computer Science, University of California Santa Barbara, November 2000.

[12] Jürgen Lampe. Depot4 – a generator for dynamically extensible translators. *Software – Concepts & Tools*, 19(2):97–108, 1998.

[13] 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.

[14] The SUIF2 compiler system. http://suif.-stanford.edu/suif/suif2.

[15] Diomidis Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. *Conference on Domain Specific Languages*, pages 67–76, October 1997.

[16] Arie van Deursen and Paul Klint. Little languages: Little maintenance? Technical Report SEN-R9704, Centrum voor Wiskunde en Informatica (CWI), 1997.

[17] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[18] 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.

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 2.4.9) are used. A solution that uses iterators—and is thus easier to maintain—is given in [11].

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.

```
# 2ps.mac

<def turn left_or_right newangle>
   <if (left_or_right == "left")>
      # Reverse the sign
      <set newangle to <eval (0 - newangle)>>
   <endif>

   # Compute new angle
   <set angle to <eval (angle + newangle)>>
<enddef>

<def forward length>
   # Compute new position of pen:
   #   x: sin(angle) * length  y: cos(angle) * length
   <angle> sin</><length> mul</>
   <angle> cos</><length> mul</>

   <if (pen == "up")>
      rmoveto</>
   <elseif (pen == "down")>
      rlineto</>
   <endif>
<enddef>

<file out.ps>
<set pen to up>
<set angle to 0>

4 setlinewidth</>newpath</>
200 200 moveto</>

<foreach cmd in turtle.cmd_list.commands.command>
   <if (exists cmd.turn)>
      <turn cmd.turn.left_or_right.text cmd.turn.identifier>
   <elseif (exists cmd.up)>
      <set pen to up>
   <elseif (exists cmd.down)>
      <set pen to down>
   <elseif (exists cmd.forward)>
      <forward cmd.forward.identifier>
   <endif>
<endfor>

stroke</>showpage</>
```