The `smgn` Reference Manual

Holger M. Kienle
University of Stuttgart, Germany
`kienle@cs.ucsb.edu`

Technical Report TRCS00–22
Department of Computer Science
University of California
Santa Barbara, CA 93106
`http://www.cs.ucsb.edu/TRs`

November 2000

Abstract

This document explains the SUIF Macro Generator (`smgn`). `smgn` is a 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.

Even though `smgn` has been specifically written for the SUIF system, its design is general enough to be useful for the translation of domain specific languages. Furthermore, since `smgn` is interpretative and easy to grasp, it is well suited for rapid prototyping.

Because the macro language is easiest to grasp with concrete examples, we give at least one for every major construct. Strictly speaking, the nature of the manual is probably rather a mixture of tutorial and reference guide.

Please help to improve this manual by sending corrections and omissions to `kienle@cs.ucsb.edu`.

Acknowledgments

Thanks to the members of the SUIF team at Portland Group, Inc. (PGI) and Stanford University for making `smgn` available—and for making it general enough to be useful outside of the SUIF-context. (The author is not part of the SUIF team and had no part in coding `smgn`; however, he got upset at the lack of decent documentation.)

As a starting point for this manual, the original (3-page) *Smgn Reference Manual* was used.

Contents

1 Introduction

This document explains the SUIF Macro Generator (`smgn`). `smgn` is a 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 with a specific macro language. While navigating the parse tree, output can be conveniently generated.

In the SUIF system, `smgn` is used to automatically translate from the *Hoof* representation into C++ code. (For further information refer to *The Basic SUIF Programming Guide* [2].) The C/C++ back end (`s2c`) makes also use of the macro language processor.

This reference manual introduces `smgn` so that programmers

- can better understand the `smgn`-specific parts of the SUIF system and make changes to them.

  For example, the `s2c` back-end is written with `smgn`. To fix bugs or to extend the back-end requires a fairly good knowledge of `smgn`. Furthermore, development of a new back-end (with textual output) is probably most conveniently done with `smgn`.

- can utilize `smgn` for their own projects (which must not be necessarily done within the context of SUIF).

  For example, the *Bauhaus* project at the University of Stuttgart [3] uses `smgn` to translate code written in a domain-specific specification language to Ada95. (This manual originated as part of this effort.)

2 The Big Picture

Even though `smgn` has been specifically written for the SUIF system, it is also useful in general for the translation of *domain specific languages* (DSLs). In fact, this is the primary motivation to explain it in more detail. Typically, a DSL can be expressed by a comparably small grammar and the output transformations are rather simple and in textual form (e.g., a program in another programming or description language). The *Hoof* specification [2] is such an example.

Unfortunately, DSLs tend to change frequently throughout their lifetime. Especially in the early design and implementation phase, typically frequent changes to the DSL's grammar and the generated output are necessary before a stable point is reached. For example, grammar changes are caused by additional requirements and the discovery that the grammar is too strict or too lax. Thus, the ability of rapid prototyping seems to be an important factor when developing a new DSL.

On the other hand, a typical approach for building a DSL translator is to use standard compiler tools, such as `lex` and `yacc`; but this approach is not well suited for rapid development and involves tedious work, such as a manual construction of the (abstract) syntax tree. There are specific tools available that assist in the design and implementation of DSLs, but they often are rather complex and difficult to learn.

`smgn`, however, is well suited for rapid prototyping. There are no lengthy modify-compile-run cycles. Similar to an interpreted languages, the programmer can make minor modifications, run the macro interpreter, and immediately look at the result. Furthermore, `smgn` is fairly intuitive and thus easy to learn.

To summarize, we believe that `smgn` is useful for rapid prototyping of small DSLs that require a rather simple transformation to textual output. `smgn` does not use fancy formalisms (such as formal semantics), but helps you to get the job done.

3 Overview of `smgn`

As mentioned before, `smgn` is a general tool that is parameterized by

- a *grammar file* (contains the grammar specification).
- a *text file* (contains the actual input).
- *macro files* (contain macro language code).

When `smgn` is invoked, it first reads the grammar file and then parses the text file with the obtained grammar specification. During the parse a parse tree is constructed. If no macro files are present, execution stops; otherwise, the macro files are read in and the macro processors starts to interpret them in sequence. Typically, the macro files contain code that traverse the parse tree and generate output depending on the information obtained from the parse tree. Thus, the same grammar specification and input can yield different outputs, depending on the used macro files.

In the following, the parser (see section 4) and the macro language (see section 6) are explained in detail.

4 The Parser

4.1 The Grammar Specification

The grammar specification is similar to BNF, thus no repetitions, optionals, or other syntactic sugar is allowed. It consists of productions of the form

$$\mathtt{<}\mathit{lhs}\mathtt{>}\ \mathtt{::=}\ \mathit{alternative}_1\ \mathtt{|}\ \mathtt{...|}\ \mathit{alternative}_n$$

An *alternative* is a (possibly empty) sequence of terminals and nonterminals. The names of nonterminals are encapsulated in angle brackets (e.g., `<x>`). Terminal symbols are written literally, but they can be also encapsulated with double quotes (`"`). If special characters are used in terminals (e.g., `<`, `>`, or `|`), they must be encapsulated.

The start nonterminal of the grammar is the left-hand-side of the first production.

4.1.1 Pseudo Nonterminals

There are special predefined pseudo-nonterminals, which may be used in the grammar:

Pseudo Nonterminal	Purpose
`<identifier>`	Used to match identifiers.
`<verbatim>`	Used to match arbitrary text.

The `<verbatim>` nonterminal matches any text up to, but not including, the terminal character that must appear to the right of `<verbatim>` in the alternative. This feature is useful to capture text that is not supposed to be parsed and subsequently analyzed, but written out later in its original form.

The `<identifier>` nonterminal matches anything up to, but not including, the point that still constitutes a valid identifier. A valid identifier can contain digits, letters (upper and lowercase), and underscores in any order.

4.1.2 Comments

Every language should allow comments. Usually comments are not specified by the grammar itself, but handled directly in the scanner. In order to make life easier, the parser has comments already built in. Any line starting with a hash (`#`) is considered a comment. Leading white spaces before a hash are allowed. In fact, comments are identical for grammar, text, and macro files.

4.2 The Parser

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.

For example, the following grammar may cause a lot of backtracking

```
<if_stat> ::=   if <expression> then <statement> endif
              | if <expression> then <statement> else <statement> endif
```

because the parser has to back-track whenever a if-then-else construct is parsed. Parsing a if-then construct without the else part does not result in back-tracking. In practice, it is probably best to try to write a grammar that is close to LL(1).

For further information about the limitations of the current parser (especially the back-tracking algorithm) please consult the following file:

`$NCIHOME/suif/suif2/tools/smgn/README`.

4.3 The Parse-Tree

When a parse is successful, a tree is generated representing the parsed text. In this tree, all terminals are removed.[1] Each node on the tree contains a number of children, which are the names of the nonterminals matched by the production.

Node names—in the following simply called names— are used by the macro language to specify a certain subtree starting from a certain tree node. Since names correspond to nonterminals, all nonterminals in a production must have unique names in order to unambiguously specify the subtrees.[2]

Accessing of children of a certain node is performed by the *dot operator*:

$x.y$

accesses child y of parent x.

Concatenation of names with the dot operator yields an *access path* (refer to section 6.4.1).

5 Toy Grammar

The following grammar example specifies the syntax for a small scripting language for turtle graphics:

```
1   # Toy grammar for turtle graphics (turtle/turtle.grm)
2
3   <start> ::= <turtle>
4   <turtle> ::= turtle <identifier> "{" <command_list> "}"
5
6   <command_list> ::= <commands> |
7
8   <commands> ::= <command> | <command> <commands>
9   <command> ::= <turn> | <up> | <down> | <forward>
10
11  <turn> ::= turn <left_or_right> by <verbatim> degrees ;
12  <left_or_right> ::= left | right
13  <up> ::= up ;
14  <down> ::= down ;
15  <forward> ::= forward <verbatim> ;
```

Note that the right hand side of the `command_list` production (line 6) has a "`|`" at the end, which means that this production can expand to nothing.

Because it is not possible to explicitly express numbers, we use the `<verbatim>` nonterminal (lines 11 and 15) to match them.[3] This means, of course, that our grammar is less restrictive than we would like to have.

We can now write a turtle program—which will be used in the following as a running example—and parse it with `smgn`.

[1] Actually, they are still present in the `text` child node of the enclosing nonterminal, but are not directly accessible.

[2] In practice, if more than one nonterminal with the same name exists in a production, the first is chosen when the subtree is selected. The other subtrees are simply not accessible.

[3] If we only wanted to match positive numbers, we could have also used the `<identifier>` nonterminal.

```
# Toy turtle program (turtle/myturtle.turtle)

turtle myturtle {
        down;
        forward 10;
        turn left by -90 degrees;
        forward 10;
}
```

For now, `smgn` will only parse the program and report syntax errors. Later on, we will show how to output postscript code after a successful parse (see section A.3).

If you run `smgn` with the `-p` option (see section C), `smgn` prints the parse tree. Figure 1 shows an excerpt of the parse tree for the previous example. The output has the following schema:

{`NamedList type =` *ParentNode*
ChildNode$_1$`=>` ...
...
ChildNode$_n$`=>` ...
`text=>` ...
}

The children of every nonterminal (*ParentNode*) are (recursively) printed followed by a special child node (called `text`) that contains the input text parsed by the *ParentNode*. The `text` node exists only for nonterminals that did actually match some input text and it never exists for the pseudo-nonterminals `identifier` and `verbatim` (see section 4.1.1). Immediate right-recursive rules, such as

`<commands> ::= <command> | <command> <commands>` (cf. toy grammar, p. 6, line 8)

are represented as (flat) lists.[4]

6 The Macro Language

The macro processor interprets the macro files. Typically, macro files generate output to multiple files while traversing the parse tree.

The macro language has two contexts:

- The *text context* is used to output text to the current output buffer. Text written in this context is directly (i.e., unchanged) written to the output buffer. Several exceptions to this rule exist, most notably a way to switch to command context.
- In text context, the *command context* is activated with a starting angle bracket (`<`) and terminated with a closing angle bracket (`>`). Most notably, in the command context it is possible to access nodes in the parse tree.

Both contexts are discussed in more detail in section 6.1.

When the interpreter starts to process the macro file, it is in text context. Thus, running the following macro file

```
# My first macro file (turtle/first.mac)

This text is directly written to stdout...</>
```

will output

[4]This flattening is performed because the macro generator allows to iterate over these nodes conveniently with a special `foreach` loop construct.

```
{NamedList type = start
turtle=>
    {NamedList type = turtle
    identifier=>myturtle
    command_list=>
        {NamedList type = command_list
        commands=>
            {MacroListObject type = commands

                {NamedList type = commands
                command=>
                    {NamedList type = command
                    down=>
                        {NamedList type = down
                        text=>down;
                        }

                    text=>down;
                    }

                text=>down;
                }

                [...]

                {NamedList type = commands
                command=>
                    {NamedList type = command
                    forward=>
                        {NamedList type = forward
                        verbatim=>10
                        text=>forward 10;
                        }

                    text=>forward 10;
                    }

                text=>forward 10;
                }

            }

        text=>down;
        forward 10;
        turn left by -90 degrees;
        forward 10;

        }

    text=>turtle myturtle {
        down;
        forward 10;
        turn left by -90 degrees;
        forward 10;
}
    }

text=>

turtle myturtle {
        down;
        forward 10;
        turn left by -90 degrees;
        forward 10;
}
}
```

Figure 1: Example parse tree (generated with `-p` option)

```
expanded file count is 1
This text is directly written to stdout...
```

followed by a newline if you run it, for example, with the turtle grammar and the demo text file give in section 5 (or alternatively sections A.1 and A.3). The first line is output by smgn and should not bother us for now.

The next example makes use of the command context to print the <identifier> nonterminal that follows the turtle token (cf. line 4 of the turtle grammar file):

```
# My second macro file (turtle/second.mac)

Name of the turtle program: <turtle.identifier></>
```

The generated output reads

```
expanded file count is 1
Name of the turtle program: myturtle
```

The construct "turtle.identifier" is used to denote a certain node in the parse tree. The first item (turtle) denotes the *root* of the parse tree. Note that the root is determined by the right side of the start nonterminal and not the start nonterminal itself (which is start, line 3). The second item denotes a child of turtle. The turtle nonterminal (line 4) has two (potential) children: identifier and command_list. Our example selects child identifier.

The following section explain the text and command context in more detail.

6.1 Text versus Command Context

As mentioned before, the text context is used to output raw text uninterpreted to the current output buffer. The following exceptions exist for this rule:

- Newlines are ignored. (This is the default behavior and can be changed; see section 6.4.4.1.)
- Starting and ending whitespaces are ignored. This is useful to indent the output text in the macro file without affecting the produced output.
- Comments (introduced with a #) are not printed.
- Single characters can be escaped with a preceding backslash (e.g., \x). This is especially useful if you want to print a hash or blank at the beginning of a line. Furthermore, to print an opening angle bracket it is necessary to escape it; otherwise smgn interprets it as the beginning of the command context.[5]
- Escape to the command context: Text starting with an angle bracket is interpreted as the beginning of a *command context* (see below) and thus not printed literally.

When the interpreter starts to process the macro file, it is in text context and the current output buffer is stdout. (Refer to section 6.4.3 for how to change the output buffer.)

The command context is entered with an (un-escaped) starting angle bracket (<). One has the following constructs available in command context:

- Commands:
  Certain commands consist of multiple tags (all given between angle brackets). An example of such a command is

  <if> ... <endif>.

  Other commands consist only of a single pair of angle brackets, for example

[5] Strictly speaking, this is not always necessary. For example, an angle bracket followed by a blank need not be escaped.

`<set x to foo>` or `<ignore_linefeeds>`.

- Node accesses:
  The contents of a node can be printed by giving an *access path* (see section 6.4.1) to that node. The path can be absolute, by giving the root node, or relative by starting with a name that represents a node in the parse tree. We have already seen an example for an absolute access path before, namely

  `<turtle.identifier>`

  It is not an error to expand a node that does not exist. If one does this, nothing is output; you will get a warning message instead.

  If a field is expanded that does not end with a printable node then nothing is output. For example, both

  `<turtle>` and `<turtle.command_list>`

  expand to nothing.

6.2 Commands

6.2.1 Control Flow

6.2.1.1 `if`

The `if` macro comes in several flavors:

- `<if (`*bool-expr*`)>` ...`<endif>`
  The text inside the construct is only expanded if the boolean expression is *"true."*

- `<if (`*bool-expr*`)>` ...`<else>` ...`<endif>`
  The text after the `<if>` is expanded if the boolean expression is *"true;"* otherwise the text after the `<else>` is expanded.

- `<if (`*bool-expr*`)>` ...
  `<elseif (`*bool-expr*`)>` ...
  ...
  `<elseif (`*bool-expr*`)>` ...
  `<else>` ...`<endif>`
  The semantic should be intuitively clear without further explanation.

Boolean expressions are explained in section 6.2.5.1.

6.2.1.2 `foreach`

The `foreach` macro allows iteration over a flat list. Such lists are created by immediate right-recursive grammar rules of the following form:[6]

`<`*lhs*`> ::= <`*element*`> | <`*element*`> <`*lhs*`>`

The parser automatically flattens the hierarchical structure into an order-preserving list that is attached to the parent node *lhs*. As a result of the flattening, all list elements are attached as children to *lhs*. All these child nodes have identical names, namely *element* (and thus can only be accessed with the `foreach` macro).

The macro comes in two flavors:

- `<foreach` *id* `in` *tree-node* `>` ...`<endfor>`
  This constructs iterates over a list who is attached at the parent *tree-node*. A reference that points to the current node in the list is assigned to *id*. Note that *id* does not represent the current node, but rather a reference to the current node. Thus, the current node is accessed with

[6] Currently exactly this form is required!

id . *element*

For example, the following turtle macro file iterates over all commands and prints their textual representation:

```
# turtle/foreach.mac

<foreach cmd in turtle.command_list.commands>
        <cmd.command.text></>
<endfor>
```

This yields the following output:

```
expanded file count is 1
down;
forward 10;
turn left by -90 degrees;
forward 10;
```

In the above example, the given *tree-node* was the actual parent of the node of the list (i.e., corresponding to *lhs*). It is also possible to "reach into" the list for the traversal by giving an access path that goes beyond the parent node. If a certain list element does not have the specified access path, it is ignored. The following example uses an access path that only reaches list elements that represent a `turn` command:

```
# turtle/foreach2.mac

<foreach cmd in turtle.command_list.commands.command.turn>
   <cmd.text></>
   <cmd.verbatim></>
<endfor>
```

Hence the following output is generated:

```
expanded file count is 1
turn left by -90 degrees;
-90
```

In this case `cmd` is indeed the node specified by the access path and not a reference to it! Hence our first example can be rewritten more conveniently as:

```
# turtle/foreach3.mac

<foreach cmd in turtle.command_list.commands.command>
   <cmd.text></>
<endfor>
```

- `<foreach` *id* `in` *tree-node* `such that (`*bool-expr*`)>` ... `<endfor>`
  The `such that` clause in this construct can be used to put a further restriction on the selected list elements. The current list element is skipped if *bool-expr* evaluates to *"false."* The boolean expression can contain the current list element (i.e., *id*).

  For example, the following code generates the same output as the second `foreach` example (`foreach2.mac`):

```
# turtle/suchthat.mac

<foreach cmd in turtle.command_list such that (exists cmd.commands.command.turn)>
   <cmd.commands.command.turn.text></>
   <cmd.commands.command.turn.verbatim></>
<endfor>
```

Another example is the selection based on the value of a certain node—in this case the `verbatim` node of the `forward` command:

```
# turtle/suchthat2.mac

<foreach fwd in turtle.command_list.commands.command.forward
 such that (fwd.verbatim == "10")>
   <fwd.text></>
<endfor>
```

The following output is generated:

```
expanded file count is 1
forward 10;
forward 10;
```

To summarize, the `foreach` macro in its simplest form processes all elements in a list. The selective processing of list elements can be achieved with two techniques: (1) giving an access path that goes beyond the parent node and (2) introducing a `such that` clause.

6.2.1.2.1 Nested Recursions The `foreach` constructs can also handle nested lists. For example, the following productions define an outer list that contains an inner list:

```
<start> ::= <outers>
<outers> ::= <outer> | <outer> <outers>
<outer> ::= "{" <inners> "}"
<inners> ::= <inner> | <inner> <inners>
<inner> ::= "*"
```

This grammar allows to parse input such as

```
{ * * * } { * }
```

To iterate over *all* of the innermost nested list elements, you can write:

```
<foreach elem in outers.outer.inners.inner>
  Handle _all_ *'s here...
<endfor>
```

Alternatively, you can use two nested `foreach` constructs:

```
<foreach oelem in outers.outer>
  Handle { here...
  <foreach ielem in oelem.inners.inner>
    Handle *'s of current list here...
  <endfor>
  Handle } here...
<endfor>
```

The latter is more useful, if one wants to distinguish the innermost lists.

6.2.1.2.2 Predicates Inside the `foreach` macros the special predicates `first` and `last` can be used:

- `(first` *id*`)`
  This predicate is *"true"* if the current list element (i.e., *id*) is the first element that is processed in the corresponding `foreach` macro. Note that the first *processed* element is not necessarily the first element in the list.
- `(last` *id*`)`
  This predicate is *"true"* if the current list element (i.e., *id*) is the last element that is processed in the corresponding `foreach` macro. Note that the last *processed* element is not necessarily the last element in the list.

If these commands are used outside a `foreach` macro, they have no effect.

The following example makes use of both predicates:

```
# turtle/firstlast.mac

<foreach fwd in turtle.commands.command.forward>
        <if (first fwd)>
                first:\ <>
        <endif>
        <if (last fwd)>
                last:\ <>
        <endif>

        <fwd.text></>
<endfor>
```

The `foreach` construct restricts the list to two items:

```
expanded file count is 1
first: forward 10;
last: forward 10;
```

6.2.1.2.3 `pos` Inside the `foreach` macros the `pos` macro can be used:

- <pos *id*>
  Output the current number of iterations of the `foreach` macro with the current list element *id*. The first iteration is denoted by 0.

  If `pos` is used outside a `foreach` macro, it has no effect.

For example, the following turtle macro file

```
# turtle/pos.mac

<foreach cmd in turtle.command_list.commands.command.forward>
   <pos cmd>
   : <cmd.text></>
<endfor>
```

outputs

```
expanded file count is 1
0: forward 10;
1: forward 10;
```

6.2.1.2.4 ? Access Path Wildcard

So far, we used the `foreach` construct to iterate over lists. However, the construct can be also used to iterate over a node's children by means of giving a wildcard (`?`) in the *tree-node* access path.

It is only possible to give a single wildcard in the access path and the path must not start with a wildcard.

The wildcard is especially convenient to iterate over tables that have been constructed with the `set` construct (see section 6.2.6.2).

Here is a simple example

```
# mac/wildcard.mac

<set x[a] to a>
<set x[b] to b>
<set x[c][a] to ca>
<set x[c][b] to cb>
<set x[d] to d>

<foreach value in x.?>
   <value></>
<endfor>
---</>
<foreach value in x.?.b>
   <value></>
<endfor>

```

along with the generated output:

```
a
b

d
---
cb
```

Note, that the first `foreach` matches `x[c]` only once (at the third iteration) and does not generate output for this node since it does not denote a string.

6.2.1.3 endmac

The `<endmac>` macro immediately terminates expansion of a macro definition. If the macro is called at the outermost scope (i.e., not during macro expansion) then interpretation of this macro file is terminated.

This macro is most useful in combination with the `if` macro to stop expansion of the current macro definition if a certain condition is met.

6.2.2 Macro Definition

- `<def` *name* $formal_1$ . . . $formal_N$`>` . . . `<enddef>`
  Defines a new macro with the name *name*. The formal parameters are listed after the macro's *name*. (Note that no commas are used to list the formals.) The formal parameter names are known within the macro body and can be used in command context like node names. At the beginning of the macro body, the interpreter is in text context.

  After the new macro is defined, it can be called by using its *name* (refer to section 6.2.3).

Macros are identified by their name and the number of their parameters. Thus, it is possible to define macros that have the same name but differ in the number of their parameters.

If a new macro is defined that has the same name and the same number of formal parameters as a previously defined macro, the new macro replaces the previously defined one. In this case, no warning is given!

It is possible to have *nested* macro definitions (i.e., a macro definition within another macro definition). However, nested macros have global visibility, i.e., once a macro definition has been processed, it can be called from anywhere.

6.2.3 Macro Calls

- <*name* $actual_1$ ... $actual_N$>
  Expands (or calls) a macro with the name *name*. If a macro with such a name does not exist, a warning is given and the call is ignored.

  The actual parameters of the call are mapped in the same order to the macro's formals parameters. Actuals are given in command context. An actual is typically an access path or a string. (The latter must be enclosed in double quotes.)

  If more than one macro with a matching name exist, the macro to be called is determined as follows:

  – If a macro with the same number of parameters exists, it is called.

  – If macros with less parameters exist, the macro with the most parameters is called. The superfluous formal parameters are *valueless*.
    It is possible to check for valueless parameters with the `<?...>` construct (see section 6.2.3.1).

  – If only macros with more parameters exist, a warning is given and the call is ignored.

  Thus, the macro with the least superfluous formal parameters is chosen.

  Inside the macro, aliases defined by a surrounding `let` (see section 6.2.6.3) are accessible.

  Recursive calls are of course possible.

Calls with no actuals introduce an interesting ambiguity:

<*name*>

can be either a call to macro *name* or an access path to node *name*. In such cases, the ambiguity is resolved in favor of the macro call. For example, the following code

```
# mac/ambiguous.mac

<def x>
        macro x
<enddef>

<set x to variable x>

<x></>
```

generates

```
expanded file count is 1
macro x
```

6.2.3.1 Valueless Parameters and <?...>

A macro call must not provide values for all formal parameters. Formal parameters that cannot be bound to an actual parameter are called *valueless*.

If an attempt is made to expand a valueless formal parameter, a warning is given and not value is output. However, it is possible to use a valueless formal parameter as an actual parameter in another macro call.

- <?*param text*>
  Checks if the formal parameter *param* is bound to a value. If this is the case then *text* is output; otherwise *text* is ignored and the whole construct has no effect.

  Note that *text* is not interpreted as text context which means that it is not possible to switch into command context! For example, the following is illegal:

  ```
  <def foo p1>
     # Illegal!
     Value of p1 (if existent): <?p1 <p1>>.
  <enddef>
  ```

  In order to achieve the desired behavior one can write

  ```
  # mac/call_opt.mac

  <def foo p1 p2>
     foo:
     <if ([<?p1 true>])>
        \ <p1>
     <endif>
     <if ([<?p2 true>])>
        \ <p2>
     <endif>
     </>
  <enddef>

  <foo>
  <foo "x">
  <foo "x" "y">
  ```

  which gives the expected output

  ```
  expanded file count is 1
  foo:
  foo: x
  foo: x y
  ```

  This example makes use of text substitution (see section 6.3).

6.2.3.2 Callbacks

Text substitution (see section 6.3) can be employed to implement callbacks.

- <[<*var*>] $actual_1$ ... $actual_N$>
  Expands a macro whose name is given by the access path *var*.

An ordinary macro call directly gives the name of the macro. In the this case, the contents of *var* supplies the macro name.

The following turtle example decouples the iteration over the commands list from the action performed for an individual list element:

```
# turtle/callback.mac

# Iteration of the list
<def iter_commands callback>
   <foreach cmd in turtle.command_list.commands.command>
      <[<callback>] cmd>
   <endfor>
<enddef>

# Callback for each list element
<def handle_command cmd>
   <cmd.text></>
<enddef>

<iter_commands "handle_command">
```

The iteration code in `iter_commands` can be reused with different callbacks.

6.2.3.3 Calls with Return Values

Text substitution (see section 6.3) can be used to "return" a string result back to the caller. If the call is encapsulated with square brackets, the output that the macro call generates is not written to the currently active output buffer; instead, the output is textually substituted for the macro call.

The following example shows how this mechanism can be used to implement a macro call that returns a boolean result. Its output is identical to `call_opt.mac` (see section 6.2.3.1):

```
1  # mac/call_w_return.mac
2
3  <def exists_param param>
4     <if ([<?param true>])>
5        true
6     <else>
7        false
8     <endif>
9  <enddef>
10
11 <def foo p1 p2>
12    foo:
13    <if ([<exists_param p1>])>
14       \ <p1>
15    <endif>
16    <if ([<exists_param p2>])>
17       \ <p2>
18    <endif>
19    </>
20 <enddef>
21
22 <foo>
23 <foo "x">
24 <foo "x" "y">
```

The macro `exists_param` outputs either the string `"true"` or `"false"`. By calling the macro with square brackets at a place that expects a boolean result (lines 13 and 16), the call gets substituted with the output of the call, which then is interpreted as a boolean value.

6.2.4 Expansion of Macro Files

There are two mechanisms that allow the expansion of (ordinary) files as macro files: `include` and `parse`.

6.2.4.1 `include`

- <include *macro-file-name*>
  Processing of the current macro file is suspended and interpretation of the macro file with name *macro-file-name* starts.

  Macro-file-name is treated as text context. Thus, the name can be given literally or built by switching to the command context.

  The current working directory as well as directories given at the `smgn` command line with the `-I` flag (see section C) are searched for the macro file. If the file is not found, a warning is given and interpretation proceeds after the `include` macro.

  Interpretation of the included file affects the state of the including file. After interpretation of the included file has been finished, normal execution resumes at the including file.

  Furthermore, the following is important to know:

  - The included macro file must not execute the `file` construct.[7]
  - Output that is generated while the included file is processed is lost.

  Thus, the included macro file should typically contain only macro definitions and execute commands that manipulate the parse tree.

6.2.4.2 `parse`

- <parse *file-name*>
  Processing of the current macro file is suspended and interpretation of the (internal) buffer with the name *file-name* starts. Note that *file-name* refers to a buffer that is currently (internally) constructed and not to a file that resides on disk.

  File-name is treated as text context. Thus, the name can be given literally or built by switching to the command context.

  This mechanism allows the dynamic generation and interpretation of macro files. For example, the following macro file dynamically creates two macro definitions in the buffer `trash` and then calls these definitions.

```
# mac/parse.mac

<def build_def fname>
   <file trash>
   \<def <fname> txt>
      <fname>: \<txt>\</>
\<enddef>
<enddef>

<build_def "mydef1">
<build_def "mydef2">
```

[7]The currently implementation redirects the output generated by the included macro file into a dummy output file. Switching the output file in the included file confuses `smgn` and results in the generation of a file with the telling name `@@@@@@`.

```

<file>
<parse trash>
<mydef1 "Hello ">
<mydef2 "World!">
```

The generated output is as follows:

```
expanded file count is 2
mydef1: Hello
mydef2: World!
```

As a (possibly unwanted) side-effect, the generated buffer is written to disk after `smgn` terminates:

```
<def mydef1 txt>mydef1: <txt></><enddef><def mydef2 txt>mydef2: <txt></><enddef>
```

However, the file can prove useful for debugging.

6.2.5 Expressions and Data Types

Besides strings, `smgn` has (pseudo) support for booleans and numbers.

6.2.5.1 Booleans

Boolean expressions are mainly used by the `if` macro and the `such that` clause of the of the `foreach` macro.

Constants: The boolean constants "true" and "false" are simply represented as strings `"true"` and `"false"`, respectively.[8] For example:

```
# mac/boolean1.mac

<if ("true")>
   I am expanded!</>
<endif>
```

Since boolean constants are strings, it is also possible to given an access path to a node that contains a "boolean string:"

```
# mac/boolean2.mac

<set x to true>

<if (x)>
   I am expanded!</>
<endif>
```

Predicates: All boolean predicates take non-boolean values and yield a boolean value. The following predicates are available:

- `(exists` *node*`)`
  If the given *node* exists in the parse tree, this construct yields "true." Otherwise it yields "false."
  Node is treated as command context (i.e., it is not surrounded by angle brackets).
  For an example refer to section 6.2.1.2.

[8] `smgn` currently interprets every string different from `"true"` as *"false."*

Operators: All boolean operators take boolean values and yield a boolean value. The following operators are available:

- `(!` *bool-expr*`)`
  Unary "not."
- `(`*bool-expr* `&&` ... `&&` *bool-expr*`)`
  `(`*bool-expr* `&` ... `&` *bool-expr*`)`
  The result is "true" if all *bool-expr*s are "true," otherwise the result is "false."
  Both the `&&` and the `&` operator behave the same. (They are no short-circuit operators.)
- `(`*bool-expr* `|` ... `|` *bool-expr*`)`
  `(`*bool-expr* `||` ... `||` *bool-expr*`)`
  The result is "false" if all *bool-expr*s are "false," otherwise the result is "true."
  Both the `||` and the `|` operator behave the same. (They are no short-circuit operators.)

It is often possible to omit round brackets, but this is discouraged since no precedence or associativity rules are defined.

It is possible to use numbers (see section 6.2.5.2) instead of boolean constants in boolean expressions. However, since all numbers are treated as "false," this does not seem to make much sense and hence will not be discussed any further.

6.2.5.2 Numbers

Numeric expressions are mainly used by the `eval` construct (see section 6.2.5.4).

Constants: A number constant is a decimal integer value. Optionally, a leading "+" or "−" can be written (without whitespaces before the digits). No floating-point numbers are supported.

The numbers are (internally) represented as strings. Because of this, it is possible to give an access path to a node that contains a string that can be interpreted as a number constant.[9]

The constant can be arbitrarily large, but predicates and operators restrict the precision.

Predicates: Predicates yield a boolean value.

- `(`*number* `==` *number*`)`
  `(`*number* `=` *number*`)`
  The above predicates compare two string for equality. Both the `==` and the `=` operator behave the same.
- `(`*number* `!=` *number*`)`
  The `!=` predicate is identical to

  `(! (`*string* `=` *string*`))`.
- `(`*number* `>=` *number*`)`
- `(`*number* `>` *number*`)`
- `(`*number* `<=` *number*`)`
- `(`*number* `<` *number*`)`

The compared *number*s must be small enough to fit in a C `long` type.[10]

Operators: All numerical operators take numerical values and yield a numeric value. The following operators are available:

- `(`*number* `+` *number*`)`

[9] If strings are no integers, the result of numerical operations is (for practical purposes) unpredictable.
[10] Hence the actual size depends upon the compiler and platform.

- (*number* - *number*)
- (*number* * *number*)
- (*number* / *divisor*)
  First, the divisor is converted to an integer value. If the divisor is zero, the result is *"false."* Otherwise, the result is the division converted to an integer value.
  Conversion to an integer values is done by truncating the remainder.

Every computation step must be small enough to fit in a C `int` type.

6.2.5.3 Strings

Strings are mainly used in boolean expressions and as actual parameters in macro calls.

Constants: A strings constant is enclosed within double quotes:

```
"I am a string constant"
```

String constants are only used in command context. It is not possible to switch to command context within a string constant.

Predicates: Predicates yield a boolean value.

- (*string* == *string*)
  (*string* = *string*)
  The above predicates compare two string for equality. Both the `==` and the `=` predicate behave the same. Typically, *string* is a string literal or an access path that results in a node that holds a string.
  String is treated as command context. This means that string literals must be encapsulated by quotation marks. For example:

```
<if (turtle.identifier == "myturtle")>
  Strings are identical.
<endif>
```

- (*string* != *string*)
- (*number* >= *number*)
- (*number* > *number*)
- (*number* <= *number*)
- (*number* < *number*)

String comparison uses the ASCII character set to establish a total ordering of the characters.

6.2.5.4 `eval`

Certain expressions can be only used in certain macros (e.g., the `if` macro expects a boolean expression). In contrast, the `eval` macro expands an arbitrary expression.

- `<eval(`*expr*`)>`
  Expands the expression *expr*. The result of the expression is returned as a string.

The `eval` macro is also useful in combination with the `set` macro. See section 6.2.6.1.

Here are some examples

```
# turtle/eval.mac

<eval ((1+2)*4)></>
<eval (turtle.identifier)></>
<eval ("A string" == turtle.identifier)></>
<eval (! ("false"))></>
```

with the corresponding output:

```
expanded file count is 1
12
myturtle
false
true
```

6.2.6 Definitions and Aliases

This section discusses the possibilities that `smgn` offers to introduce new names and new parse tree nodes.

6.2.6.1 Node Definitions with `set`

- `<set` *node* `to` *str*`>`
  Defines a (new) node in the parse tree with the access path *node*. The contents of the node is the string *str*.

  If the access path does denote an existing node that is not a string, a warning is given and the macro is ignored.

  Str is treated as text context. Thus, the name can be given literally or built by switching to the command context.

It is not possible to delete a node once it has been introduced with a node definition. However, a node can be redefined. For example, the following code first defines a new node `x` and then redefines `x` by using its "old" contents:

```
# mac/set.mac

<set x to 42>
<set x to <eval (x+1)>>
```

It is also possible to redefine an existing node that is part of the original parse tree as long as the node denotes a string. This is the case for leaf nodes whose access paths end with `identifier`, `verbatim`, or `text`.

6.2.6.2 Table Definitions with `set`

The `set` macro in conjunction with text substitution (see section 6.3) yield a powerful construct called *tables*.

When specifying an access path for *node*, you can give parts of the path in square brackets.[11] The text in the square brackets is treated as text context and gets expanded *before* the access path is applied. The following example demonstrates the concept:

```
1   <set x[y] to whatever>
2
3   <set nodename to y>
4   <set x[<nodename>] to whatever>
```

The `set` macros in line 1 and 4 have the same effect. The access path in line 1 is equivalent to `x.y`. The flexibility of tables is demonstrated in line 4. The node `nodename` is expanded (yielding the string `y`) resulting in the access path `x.y`. Hence, tables are similar to *associative arrays*.

Note that it is possible to "concatenate" square brackets:

```
<set x[y][z] to whatever>
```

Nesting them, however, is not allowed.

The following (a bit artificial) example checks if a turtle program contains duplicate turtle commands.

[11] However, the first element of the path must not be a bracket; otherwise `smgn` can get confused.

```
# mac/duplicates.mac

<def check str1 str2>
   <if (exists dup[<str1>][<str2>])>
      Duplicate detected! (<str1>/<str2>)</>
   <endif>

   <set dup[<str1>][<str2>] to exists>
<enddef>

<foreach cmd in turtle.command_list.commands.command>
   <if (exists cmd.turn)>
      <check cmd.turn.left_or_right cmd.turn.verbatim>
   <elseif (exists cmd.up)>
      <check "up" "">
   <elseif (exists cmd.down)>
      <check "down" "">
   <elseif (exists cmd.forward)>
      <check "forward" cmd.forward.verbatim>
   <endif>
<endfor>
```

The generated output is as follows:

```
expanded file count is 1
Duplicate detected! (forward/10)
```

6.2.6.3 Aliases with `let`

- `<let` *id* `be` *node*`>` ... `<endlet>`
  The name *id* is defined to denote the same parse tree node as the access path given by *node* does. Thus, *id* is an alias (or short-hand form) for *node* and can be used instead of the full access path.

  Id is valid only within the body of the construct. More precisely, *id* is known to all constructs that are executed within `let`'s body (*dynamic name binding*). Thus, if a macro is called in the body, the macro can refer to the defined alias.

  The body of `let` is (silently) not expanded if the access path given by *node* is not valid![12]

The following turtle macro file gives an example of the `let` construct:

```
# turtle/let.mac

<def print>
   <t.identifier></>
   <id></>
<enddef>

<let t be turtle>
   <let id be t.identifier>
      <print>
   <endlet>
<endlet>
```

[12]This can be conveniently used to change control flow depending if an access path exists or not. However, it is not intuitive and hard to debug.

The corresponding output is a follows:

```
expanded file count is 1
myturtle
myturtle
```

It is also possible to give a comma-separated list of alias definitions. For example, the above example can be rewritten less verbose:

```
# turtle/let2.mac

<def print>
   <t.identifier></>
   <id></>
<enddef>

<let t be turtle, id be t.identifier>
   <print>
<endlet>
```

6.2.6.4 Aliases with `map`

- `<map` *node1* `to` *node2*`>`
  The access path given by *node2* denotes now the parse tree node given by access path *node1*. Thus, *node2* is an aliased access path.

The `map` construct 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.

It is possible to use square brackets (see section 6.2.6.2) when specifying the access path for *node1* as well as for *node2*.

The following example uses `map` to construct a table, which can then be easily iterated with the `foreach` construct:

```
# turtle/map.mac

<set x[a] to a>
<set x[b] to b>
<set x[c][a] to ca>
<set x[c][b] to cb>
<set x[d] to d>

<map x[a] to l[a]>
<map x[b] to l[b]>
<map x[c][a] to l.c>
<map x[c][b] to l.d>
<map x[d] to l[e]>
<map turtle.identifier to l[f]>
<set l.g to xxx>

<foreach value in l.?>
   <value></>
<endfor>
```

The macro file generates the following output:

```
a
b
ca
cb
d
myturtle
xxx
```

The `set` in line 15 shows that constructs do not differentiate between nodes that have been introduced by `set` or `map`.[13]

6.3 Text Substitution with `[...]`

- `[...]`
  Expands the constructs between the square brackets. The generated output of these constructs is then textually replaced with `[...]`. Note that this means that the output of the expanded constructs is not written to the currently active output buffer.

  The expanded constructs must not change the output buffers (e.g., by calling the `file` or `input` macro).[14]

  This substitution only works in the command context. If used in the text context, the square brackets are simply printed. For example, the following code

  ```
  # mac/sb.mac

  <def m>
     Some output...
  <enddef>

  [<m>]
  ```

  outputs

  ```
  expanded file count is 1
  [Some output...]
  ```

Several concrete applications of this construct have been already discussed:

- Callbacks (section 6.2.3.2).
- Table definitions with `set` (section 6.2.6.2).
- Macro calls with return values (section 6.2.3.3).

6.4 Text Handling

6.4.1 Content of Parse Tree Nodes

When generating output, it is frequently necessary to access textual information from the parse tree—either to output the contents of the node or to change control flow depending on the contents of the node.

[13] Almost true. If the same access path is defined by `set` as well as `map`, then the contents given by `set` is always chosen. Thus, in such a case `map` has no effect!

[14] Debugging note: If this is done, a file with the name `@@@@@@` will be created by `smgn`.

- <*node*>   (text context)
  node   (command context)
  The contents of a *node* in the parse tree can be accessed by giving an access path to that node. (Note that all nodes in the original parse tree represent nonterminals; terminals are lost.) If the access path does not denote a string, the empty string is output (and no warning is given).

  The access path has two specific variants:

  – If the path ends with `verbatim` or `identifier`, the content of that node is output (i.e., the input that has been parsed for the corresponding nonterminal).
  – If the path ends with `text` then the content of the nonterminal that immediately proceeds `text` is output. This does not work if the immediately proceeding nonterminal is `verbatim` or `identifier`.

It is considered a mistake to follow an access path to a node that does not exist. If you do this, no output is generated; you will get a warning message instead.

6.4.2 Text Sections with `use`

It can be awkward to generate sequential output for a file. Sometimes one would like to generate output for a certain position in the output file without disturbing the other. For example, if generating C code you might want to put all include files at the top of the file. One approach would be to process the text file twice: The first pass outputs the required include files and the second pass generates the actual C code.

Fortunately, `smgn` offers a more convenient approach. It is possible to split an output file into several sections. The output can then be redirected into an arbitrary section. The final output is obtained by concatenation of the sections. The sections are ordered sequentially. A new section is appended at the end after all existing ones. Thus, the order of the generated output depends on the creation time of the sections!

- `<use` *section-name*`>`
  Output is switched to the section with the name *section-name*. Furthermore, the previously selected section is pushed on the section stack.

  Section-name is treated as text context. Thus, the name can be given literally or built by switching to the command context.

  If the section does not exist then a new section is created. The output of this section is printed after all previously created sections.

- `<use>`
  Output is switched to the default section. Furthermore, the previously selected section is pushed on the section stack.

- `<use *>`
  The topmost section in the section stack is popped and output is switched to this section.[15]

Each file has its own sections. Thus, `use` depends on the current output file (which has been set with the `file` macro; see section 6.4.3).

Each file has a *default section* (which has no name). If a file is created, the default section is the only existing section and the section stack is empty.

The section stack is useful to switch output temporarily to another section and then switching back to the previously active one without actually knowing which section was the previously active one.

For example, to generate a `lex` file using three sections can be practical:

[15] The current implementation terminates with a segmentation fault if an attempt is made to pop from the empty stack.

```
<use regular_definitions>
<use translation_rules>
%%
<use auxiliary_procedures>
%%
<use *>
<use *>
This output goes to regular_definitions

<use auxiliary_procedures>
  void install_id() { ... }
<use *>
```

The output is divided into four consecutive sections: The default section first, followed by `regular_definitions`, `translation_rules`, and `auxiliary_procedures`. The last three lines demonstrate how to use the section stack.

Here is another example:

```
<use bucket1>
<use bucket2>
<use bucket3>
3</>
<use *>
2</>
<use *>
1</>
<use *>
0</>

<use bucket2>
22</>
<use *>

<use>
00</>
```

It generates the following output:

```
expanded file count is 1
0
00
1
2
22
3
```

6.4.3 Redirecting Output with `file`

When the macro interpreter starts executing, the generated output is written to the standard output buffer (`stdout`). Typically, the generated output will go to a single file or will be split up into several files. The `file` macro provides this functionality:

- <file *file-name*>
  The output buffer is switched to the buffer with the name *file-name*. Right after interpretation is terminated

and before smgn terminates, all buffers are written to their corresponding files. This means that the physical files are not created right away and only written if smgn terminates normally.

File-name is treated as text context. Thus, the name can be given literally or built by switching to the command context.

If this is the first time that output is switched to this buffer, then an empty buffer with *file-name* is created. If the buffer already exists, the output is appended to it.

After macro file execution stops, all buffers are written to disk. If a file with the name *file-name* already exists, it is silently overwritten.

- `<file>`
  The output buffer is switched to `stdout`.

Note that every file can consist of sections and that switching to a different file also restores the state of the sections of that file (refer to section 6.4.2 for further explanation).

A file is written to the current working directory. The file name can also specify a path (e.g., `mydir/myfile`). Under Unix, it is possible to, for example, redirect output to `stdout` by giving `/dev/stdout` for the *file-name*.[16]

6.4.4 Text Formatting

This section discusses constructs that can be used when generating output.

6.4.4.1 Newlines

- `</>`
  Output a newline character to the currently active output buffer.

  (Because of the intermixing of commands and text in macro files, real newlines are not output by default.)

The behavior how smgn treats newlines in the macro files can be changed with the following (pseudo) macro calls:

- `<ignore_linefeeds>`
  Newlines in the macro file are ignored in the text context. This means that newlines can be only generated with `</>`.

  This is smgn's default behavior.

- `<notice_linefeeds>`
  Newlines in the macro file are output as well if they are encountered in a text context.

  An empty line (i.e., the line contains only non-printed whitespaces) does not cause the output of a newline. For example, assuming text context, the second line does output a newline, whereas the third line does not:

  ```
  <notice_linefeeds>
  \␣
  ␣
  ```

  This command is especially useful to output continuous text without cluttering the macro file representation with lots of `</>`s.

[16] We discourage using these features because they are not portable.

6.4.4.2 Indentations The following commands control the indentation (i.e., the number of leading blanks) of the output. After every newline the current indentation is output.

- `</=`*indent*`>`
  Set the indentation to *indent*. A negative value sets the indentation to zero.
- `</+`*indent*`>`
  Increase the current indentation by *indent*.
- `</-`*indent*`>`
  Decrease the current indentation by *indent*. The indentation does not drop below zero.
- `</0>`
  Text within a pair of `</0>`s are output without indentation. The first `</0>` sets indentation to zero and the second `</0>` restores the indentation to its previous setting.

Indent must be a number. The new setting of the indentation comes into effect *after* a newline (in the corresponding buffer) is output.[17]

Here is an example

```
# mac/indent.mac

No indentation so far...
</=4>
Text with indentation of 4...
</+2></>
Now we have two more...
</0></>No indentation here!</0>
</-2></>
Back to 4...
</>
```

with the corresponding output:

```
expanded file count is 1
No indentation so far...
    Text with indentation of 4...
      Now we have two more...
No indentation here!
    Back to 4...
```

6.4.4.3 String Formatting

Because smgn has no mechanism to arbitrary modify strings (except for concatenation of them), it provides several predefined string manipulation commands.

- `<!CAPS` *node*`>`
  The string denoted by the access path *node* is changed to all uppercase.
- `<!CPZ` *node*`>`
  Changes *node* so that underscores are eliminated and characters after underscores are capitalized.

[17] This could be considered a bug.

- `<!LLC` *node*`>`
  Changes the first character to lowercase.
- `<!LOWS` *node*`>`
  Changes *node* to all lowercase.
- `<!NSPC` *node*`>`
  Eliminate spaces in the string.[18]
- `<!SING` *node*`>`
  Changes *node* from plural to singular with the following rules:

  `ies$` → `y` (e.g., `spies` → `spy`)
  `([^s])[sS]$` → `\1` (e.g., `foes` → `foe`)
- `<!UNL` *node*`>`
  Turns text to lowercase and puts an underline between capitalized characters and their preceding character; except for the first character, which is only changed to lowercase.
- `<!"` *node*`>`
  Enclose the string with quotationmarks.
- `<!'` *node*`>`
  Enclose the string with ticks.

Note that the above commands destructively overwrite the string given by the *node* access path. If you do not want that, you can first copy the string to another node with the `set` construct.
The follow tables gives a few examples:

Before	!CAPS	!CPZ	!LLC	!LOWS	!NSPC	!SING	!UNL	!"	!'
SpieS	SPIES	SpieS	spieS	spies	–	Spie	spie_s	"SpieS"	'SpieS'
AaaBC	AAABC	AaaBC	aaaBC	aaabc	–	AaaBC	aaa_b_c	"AaaBC"	'AaaBC'
_Aa_Bb	_AA_BB	AaBb	_Aa_Bb	_aa_bb	–	_Aa_Bb	__aa__bb	"_Aa_Bb"	'_Aa_Bb'

6.5 Debugging Output

The following constructs output information to `stderr` and are meant for debugging output.

6.5.1 echo

- `<echo` *text*`>`
  Print *text* to `stderr`. *text* is treated as text context. Thus, it is possible to switch to the command context.

For example, the following turtle macro file

```
# turtle/echo.mac

<echo Name: <turtle.identifier>>
```

outputs

```
echo.mac:3: Name: myturtle
<echo Name: <turtle.identifier>>
                                ^
expanded file count is 1
```

[18]The current implementation dies with a segmentation fault if the string does not contain blanks.

6.5.2 show

- <show *node*>
  Print the parse tree anchored at access path *node* to stderr. A header is printed as well that shows the command along with its source position.

This macro is useful, for example, inside macros defintions when you want to see exactly the contents of a formal parameter for a specific call.

To print the whole parse tree, one can give the right hand side of the start nonterminal:

```
# turtle/show.mac

<show turtle>
```

The output of the parse tree corresponds to lines 3–66 of Figure 1.

If an access path denotes a terminal, its contents is printed (without a newline). For example

```
# turtle/show2.mac

<show turtle.identifier>
```

outputs

```
show2.mac:3: show called
<show turtle.identifier>
      ^
myturtleexpanded file count is 1
```

7 *Hoof*-Specific Features

The structure of the SUIF intermediate representation (IR) [2] is not that much different from a parse tree. Thus, the SUIF IR graph can be mapped to a corresponding smgn parse tree. This mapped "parse tree" can be traversed and manipulated just like an ordinary one.

The SUIF IR graph consists of the following components:

Ownership links: Every node in the graph has exactly one parent node, which "owns" the node.[19] Thus, the ownership links build an *ownership tree*. This tree is embedded within the SUIF IR graph.

The root of the graph (and of the ownership tree) is a SUIF FileSetBlock. The root is implicity assumed in access paths.

A SUIF node can contain fields that hold ownership links.

Simple fields: For simple fields, you follow the ownership link by using the field name in the access path. For example, the FileBlock node contains a field symbol_table, which holds an ownership link to a SymbolTable node. If the FileBlock node is represented by the access path fb, then the SymbolTable node is denoted by the access path fb.symbol_table.

Collection type fields: All fields with collection types (such as list, searchable_list, indexed_list, and vector) represent essentially a list of children. Over this list can be iterated with the foreach macro. For example, a StatementList has a field statements, which hold a list of Statement nodes. If the StatementList node is represented by the access path sl, then the following construct iterates over all Statement nodes that are owned by StatementList:

[19] This is true except for the root node, which is the SUIF FileSetBlock.

```
<foreach stmt in sl.statements>
   ...
<endfor>
```

Reference links: Reference links are similar to ownership links, with the exception that these references are aliases to other nodes.

Thus, the ownership tree corresponds to the parse tree and the reference links correspond to aliases introduced with the `map` construct (see section 6.2.6.4).

Primitive values: Fields that hold values of primtive types (such as `bool`, `int`, and `LString`) constitute the leafs of the ownership tree. All values are represented as corresponding strings in the `smgn` parse tree. For example, boolean values are translated to the strings `"true"` and `"false"`.

For SUIF parse trees, `smgn` offers additional constructs that make use of the class hierarchy of SUIF nodes. These constructs are discussed in the following two sections.

7.1 Dispatching of Macro Calls

Section 6.2.3 explained how a matching macro definition is determined for a macro call. For SUIF nodes, an additional matching rule can be used: Matching of the first parameter can be restricted to a specific node type. Thus, (single) dispatching can be realized.

To realize dispatching, the name of the first formal parameter is followed by a colon (`:`), followed by the name of the type that the actual must match. The actual will match the exact type or a subtype of the given type. If more than one macro definition matches, the one with the most "precise" type is chosen.

The following example is taken from the `c_text.mac` file of the `s2c` back end:

```
1   ### Output alignment of Type in bytes
2
3   <def put_byte_alignment_type type:Type>
4       /* Cannot compute alignment of <!TYPE type> Type */
5   <enddef>
6
7   <def put_byte_alignment_type type:QualifiedType>
8       <put_byte_alignment_type type.base_type>
9   <enddef>
10
11  <def put_byte_alignment_type type:DataType>
12      <eval ([<type.bit_alignment>]/8)>
13  <enddef>
14
15  <def put_byte_alignment_type type:CProcedureType>
16      <eval ([<type.bit_alignment>]/8)>
17  <enddef>
18
19  ...
20
21  <def handle_Expression expr:ByteAlignmentOfExpression parent_expr>
22      <put_byte_alignment_type expr.ref_type>
23  <enddef>
```

Depending on the type of the first actual in the `put_byte_alignment_type` call (line 22), the macro call dispatches to the most precise type. The topmost macro definition (line 3) implements the "default" behavior.

7.2 Type Tests

In a SUIF IR parse tree, every node (except for leafs, which contain strings) has a certain node type.

- `<!TYPE` *node*`>`
  Output the type of the access path *node*.

  If this construct is called for a node that contains a string, nothing (i.e., the empty string) is output.

  Node is treated as command context (i.e., it is not surrounded by angle brackets).

  This can be used to test for the exact type of a node, such as in the following example:

  ```
  <if ([<!TYPE node>] == "PointerType")>
     ...
  <endif>
  ```

- `<!ISKINDOF` *node node-type*`>`
  Returns the string `"true"` or `"false"`, depending if the access path *node* has the type or subtype given in *node-type*.

  Node is treated as command context (i.e., it is not surrounded by angle brackets).

  For example, this construct can be used for a type test as follows:

  ```
  <if ([<!ISKINDOF node CProcedureType>])>
     ...
  <endif>
  ```

8 Hints

A good way to learn `smgn` is to simply look at the existing macro files that come with the SUIF system. Still, to increase the learning curve, this section provides several hopefully useful hints.

Using terminals to distinguish parses: Sometimes terminal symbols are used in alternatives (e.g., line 12 in the turtle grammar). The `text` node can then be used to retrieve the matching terminal, for example:

```
<if (turn.left_or_right == "left")>
   ...
<elseif (turn.left_or_right == "right")>
   ...
<endif>
```

Epsilon rules: One should try to specify epsilon rules as “early” as possible in the grammer.

For example, consider the following grammer file for our turtle example:

```
# "Wrong" grammar for turtle graphics (turtle/wrong.grm)

<start> ::= <turtle>
<turtle> ::= turtle <identifier> "{" <commands> "}"

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

<turn> ::= turn <left_or_right> by <verbatim> degrees ;
<left_or_right> ::= left | right
<up> ::= up ;
```

```
12   <down> ::= down ;
13   <forward> ::= forward <verbatim> ;
```

Note that the `command` nonterminal (line 7) can now expand to nothing. Compared to the original grammar, this one gets rid of the nonterminal `command_list` and looks simpler; so, why not use this one?

Here is the accompanying `foreach` construct (analogous to `foreach3.mac` shown in section 6.2.1.2):

```
# turtle/foreach3.wrong.mac

<foreach cmd in turtle.commands.command>
        <cmd.text></>
<endfor>
```

Besides making the life of the parser more complicated, this solution has the following nasty drawback: A warning is generated if no turtle commands are given, i.e., if the turtle file looks as follows:

```
# Empty turtle program (empty.turtle)

turtle empty {
}
```

In this case, the `commands` nonterminal expands to nothing and hence the interpreter generates the following warning:

```
foreach3.wrong.mac:4: warning: name cmd.text not found
        <cmd.text></>
                 ^

valid names are empty
expanded file count is 1
```

In order to avoid this problem, you can, for example, modify the macro file as follows:

```
# turtle/foreach3.wrong.fix.mac

<foreach cmd in turtle.commands.command such that
 (exists turtle.commands.command.text)>
        <cmd.text></>
<endfor>
```

Still, you should refrain from using this grammar and make your life more complicated.

Iterators with callbacks: Using an iterator that generates a callback for every element to be processed decouples the concrete parse tree from the action to be performed on a certain node in the parse tree. Section 6.2.3.2 gives an example.

Macros in text context: Several macros expect strings, which are given in text context. This feature can be used to put complex macro instead of simple names, for example:

```
# turtle/set2.mac

<set result to <if (turtle.identifier == "myturtle")>
                  yes
               <else>
                  no
               <endif>>

<result></>
```

Empty versus existent nodes: A node can be existent in the parse tree, yet empty. To check whether a node n exists use

(exists n)

To check whether a node exists and is not empty use

(exists n.text)

This works since the `text` node is only created if the parent node is associated with parsed text.

Execute frequently: It is a good idea while changing a macro file to execute it frequently—even after a small change. In case of a syntax error, `smgn` never complains. Instead, a certain part of the macro file is ignored or `smgn` loops infinitely!

For example, the `if` macro in the following macro file is missing a closing angle bracket (end of line 4):

```
1   # mac/syntax_error.mac
2
3   Before the if</>
4   <if ("true")
5      Syntax Error (missing "<")</>
6   <endif>
7   After the if</>
```

In this case, `smgn` skips the `if` macro (and part of the following output):

```
expanded file count is 1
Before the if
fter the if
```

Here is another example:

```
# turtle/syntax_error.mac

<if turtle.identifier == "myturtle">
   Never gets expanded!</>
<endif>
```

The boolean expression of the `if` macro is not encapuslated by round brackets. Because of this, the body of the macro will never be executed.

Such bugs can be extremely hard to trace, expecially if you made lots of changes in your macro file since the last test run.[20]

Debugging output: Do not accidently generate debug output for macro calls that are supposed to "return" a boolean result...

9 Discussion of smgn's Approach

We already outlined `smgn`'s application domain in section 2. In the following, we try to briefly evaluate `smgn`'s strengths and weaknesses.

[20] Using the `smgn` emacs mode helps somewhat in catching these kind of errors.

9.1 Strengths

- Very well suited for rapid prototyping of domain-specific languages.
- Unified concepts for processing of parse trees generated from a grammar specification and from a SUIF IR graph. Thus, it is not necessary to learn two different concepts when working on SUIF.
- Simple and intuitive; thus easy to learn.[21]

9.2 Weaknesses

- The grammar specification and the macro code that traverses the resulting parse tree are tighlty coupled. Thus, changes in the grammar usually mean changes in the macro code.[22]
- It is not possible to abstract from the concrete syntax given by the grammar by building an abstract syntax tree.
- There is no syntax checking of the macro files. For syntax errors, `smgn` typically fails silently (skipping part of the macro code).
- There is no good support to generate error messages. Most notably, no line number information is present in the parse tree.
- It is not possible to call external programs ("shell escape").
- There is no high-level specification that describes parse tree transformations.

Even though by looking at the lists one could get the impression that `smgn`'s weaknesses outweight its benefits, most weaknesses are missing functionality that is in practice not crucially needed.

A Turtle Example

This example shows how to specify a grammar and macro file for a simple domain specific language, called *turtle*. It is used as a running example in this manual.

The turtle language is a small scripting language to draw a picture with turtle graphics. Its meaning should be intuitive to understand.

A.1 Grammar File (`turtle.grm`)

The following grammar is used to construct the parse tree of a turtle program:

```
# Toy grammar for turtle graphics (turtle/turtle.grm)

<start> ::= <turtle>
<turtle> ::= turtle <identifier> "{" <command_list> "}"

<command_list> ::= <commands> |

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

<turn> ::= turn <left_or_right> by <verbatim> degrees ;
```

[21]This might be an enthusiastic overstatement, but hopfully this statement is true when one uses this manual!

[22]Using callbacks can mitigate this problem.

```
<left_or_right> ::= left | right
<up> ::= up ;
<down> ::= down ;
<forward> ::= forward <verbatim> ;
```

A.2 Sample Input File (`myturtle.turtle`)

The following example program is used by all turtle macros described in this manual:

```
# Toy turtle program (turtle/myturtle.turtle)

turtle myturtle {
        down;
        forward 10;
        turn left by -90 degrees;
        forward 10;
}
```

A.3 Macro File (`ps.mac`)

The following macro file is a bigger and complete example. It generates corresponding PostScript output for the turtle commands.

```
# turtle/ps.mac

# Introduced nodes:
#   angle (current orientation of turtle)
#   pen   ("up" or "down")
# Temporary nodes:
#   newangle

### Macro Definitions

# Iterate over the command list with callback
<def iter_commands callback p1 p2 p3>
   <foreach cmd in turtle.command_list.commands.command>
      <[<callback>] cmd p1 p2 p3>
   <endfor>
<enddef>

# Depending on the actual command, call the corresponding callback
<def handle_command cmd callback>
   <if (exists cmd.turn)>
      <[<callback>]_turn cmd.turn.left_or_right.text cmd.turn.verbatim>
   <elseif (exists cmd.up)>
      <[<callback>]_up>
   <elseif (exists cmd.down)>
      <[<callback>]_down>
   <elseif (exists cmd.forward)>
      <[<callback>]_forward cmd.forward.verbatim>
   <else>
      %%% Error: Unknown command: cmd.text</>
```

```
    <endif>
<enddef>

<def handle__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 handle__up>
   <set pen to up>
<enddef>

<def handle__down>
   <set pen to down>
<enddef>

<def handle__forward length>
   # Compute new position of pen:
   #   sin(angle) * length + cos(angle) * length
   <angle> sin</>
   <length> mul</>
   <angle> cos</>
   <length> mul</>

   <if (pen == "up")>
      # Change position without drawing
      rmoveto</>
   <elseif (pen == "down")>
      # Draw line
      rlineto</>
   <else>
      %%% Error: ‘pen’ neither "up" nor "down".</>
   <endif>
<enddef>

### Start of execution

# Generated postscript output goes to ‘out.ps’.
<file out.ps>

# At startup, the turtle’s pen is up
<set pen to up>

# At startup, the turtle points north.
<set angle to 0>

```

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

<iter_commands "handle_command" "handle_">

stroke</>
showpage</>
```

This macro code makes extensive use of callbacks. If you do not like this style, you can of course "inline" the callbacks and end up with a more monolithic style.

The generated code is written to `out.ps`. You can view the result of the generated code as follows:

```
smgn turtle.grm myturtle.turtle ps.mac ; gs out.ps
```

A.4 Generated PostScript Output

If you run

```
smgn turtle.grm myturtle.turtle ps.mac
```

then the following PostScript code is generated:

```
4 setlinewidth
newpath
200 200 moveto
0 sin
10 mul
0 cos
10 mul
rlineto
90 sin
10 mul
90 cos
10 mul
rlineto
stroke
showpage
```

B Command Summary

- `#` — 4.1.2, p. 5
  A line starting with a `#` is a comment line.

- `<>`
  Allows to continue text on the next line. After `<>`, newlines and whitespaces are ignored.

- *char* — 6.1, p. 9
  In text context, the single character *char* is escaped.

- <*node-access-path*> — 6.4.1, p. 25
  Outputs the contents of a node in the parse tree specified by the access path.

- <*macro-name* $actual_1$ ... $actual_N$> — 6.2.3, p. 15
  Expands (or calls) a macro.

- `</>` — 6.4.4.1, p. 28
  Output a newline character to the currently active output buffer.

- `</=`*indent*> — 6.4.4.2, p. 29
  `</+`*indent*>
  `</-`*indent*>
  `</0>`
  Text indentation.

- `<?`*param text*> — 6.2.3.1, p. 16
  Checks if the formal parameter *param* is bound to a value. If so, *text* is expanded.

- `<!CAPS` *node*> — 6.4.4.3, p. 29
  `<!CPZ`*node*>
  `<!LLC` *node*>
  `<!LOWS` *node*>
  `<!NSPC` *node*>
  `<!SING` *node*>
  `<!UNL` *node*>
  `<!"` *node*>
  `<!'` *node*>
  Destructive text formatting of *node*.

- `<!TYPE` *node*> — 7.2, p. 33
  `<!ISKINDOF` *node node-type*>
  Type tests for SUIF parse trees.

- `<def` *name* $formal_1$ ... $formal_N$`>` ... `<enddef>` — 6.2.2, p. 14 (7.1, p. 32)
  Defines a new macro.

- `<echo` *text*> — 6.5.1, p. 30
  Print (debugging) *text* to `stderr`.

- `<eval(`*expr*`)>` — 6.2.5.4, p. 21
  Outputs the result of the expression *expr*.

- `<endmac>` — 6.2.1.3, p. 14
  Immediately terminates expansion of a macro definition.

- `(first` *id*`)` 6.2.1.2.2, p. 13
  This predicate is *"true"* if the current list element (denoted by *id*) is the first element that is processed in the corresponding `foreach` macro.

- `<foreach` *id* `in` *tree-node* `such that (`*bool-expr*`)>` ... `<endfor>` 6.2.1.2, p. 10
  This constructs iterates over a list. The optional `such that` clause in this construct can be used to put a restriction on the selected list elements.

- `<if (`*bool-expr*`)>` ... 6.2.1.1, p. 10
  `<elseif (`*bool-expr*`)>` ...
  ...
  `<elseif (`*bool-expr*`)>` ...
  `<else>` ... `<endif>`

- `(exists` *node*`)` 6.2.5.1, p. 19
  Predicate that checks whether the given *node* exists in the parse tree.

- `<ignore_linefeeds>` 6.4.4.1, p. 28

- `<include` *macro-file-name*`>` 6.2.4.1, p. 18
  Start interpretation of given macro file.

- `(last` *id*`)` 6.2.1.2.2, p. 13
  This predicate is *"true"* if the current list element (denoted by *id*) is the last element that is processed in the corresponding `foreach` macro.

- `<let` *id* `be` *node*`>` ... `<endlet>` 6.2.6.3, p. 23
  The name *id* is aliased to the access path *node*.

- `<map` *node1* `to` *node2*`>` 6.2.6.4, p. 24
  The existing access path *node1* is aliased with *node2*.

- `<notice_linefeeds>` 6.4.4.1, p. 28

- `<parse` *file-name*`>` 6.2.4.2, p. 18
  Start interpretation of the given (internal) buffer.

- `<pos` *id*`>` 6.2.1.2.3, p. 13
  Ouput the current number of iterations of the corresponding `foreach` macro.

- `<set` *node* `to` *str*`>` 6.2.6.1, p. 22
  Defines a (new) node in the parse tree with the access path *node* and the contents *str*.

- `<show` *node*`>` 6.5.2, p. 31
  Print the parse tree anchored at access path *node* to `stderr`.

- `<use` *section-name*`>` 6.4.2, p. 26
  Output is switched to the section with the name *section-name*.
  `<use>`
  Output is switched to the default section.
  `<use *>`
  Pop section from section stack.

C Command Line

Invocation of `smgn` without any parameters prints the command line options:

```
usage is smgn <options> <grammar> <source> <macro file 1> <macro file 2> ...
options are: -p print result of parsing source
             -d debug macro expansion
             -D<name>=<value> define a name for macro expansion
             -I<directory> add directory for includes
```

The macro files are executed in the order that they appear on the command line.

For example, the turtle macro files are executed as follows:

`smgn turtle.grm myturtle.turtle` *macro-file*.mac

Command line options:

-p: The complete parse tree is printed that has been generated by reading the input file (before the macro files are executed). An example of such a parse tree output is given in Figure 1.

-d: Invokes the interactive, built-in debugger (right before the macro files are executed). Typing "`h`" or "`?`" prints a command summary of the debugger.[23]

-D*name*=*str*: Before execution starts, the node *name* is put at the root of the parse tree with the string value *str*. The *name* must not contain a dot, which means that only child nodes of the (implicit) root can be created. As usual, the specified string can be accessed in a macro file with

<*name*>

If the same *name* is given multiple times on the command line, the value of the first definition is chosen.

-I*dir*: Add the directory *dir* to the list of directories to be searched when looking for a macro file to execute. By default, the current directory is the only one that is searched.

The list of directories is used when searching for macro files given by the command line as well as by the `input` macro.

References

[1] SUIF2 Home Page, `http://suif.stanford.edu/suif/suif2`.

[2] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, Constantine Sapuntzakis, *The Basic SUIF Programming Guide*, November 1999.

[3] Bauhaus Home Page, `http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus`.

[23] The debugger is not very intuitive to use and explaining it in more detail would be a good idea, but this manual is already too long...