GTL,
a Grammar Transformation Language for SDF specifications

E. P. Schatborn

September 2005

Master's Thesis in Computer Science
Programming Research Group
Faculty of Science
University of Amsterdam
Supervisor: prof. dr. P. Klint

Acknowledgements

When I first started work on this thesis, I still thought that writing one or two pages a day was very little. I was very wrong.

At first, what precious pages that were written, were labeled 'very concise' by my supervisor prof. dr. Paul Klint. He might have used the word 'extremely', I'm not sure. Of course, I did not agree, but I worked on explaining things a bit more. The next time I handed in those pages, they had grown in number, but in my opinion gave no more information than they previously had.

"very good, stretch to 10 pages", read the comment on what I was handed back. I was horrified. I had gone from three to six pages, giving no extra information whatsoever, and now I had to stretch it to ten? So, I gave more explanations, and restructured the text even more, ending up with I think sixteen pages.

None of them are part of the thesis anymore.

I'd like to think I got the message in the end. Eventually I understood that you must take the reader by the hand, and guide him or her through the text. I don't think I've succeeded in that respect, but I have learned a lot. I'd like to thank Paul for his patience with me, and for leaving his door open.

Whenever I had a problem with the meta environment, I would email Jurgen Vinju. Somehow he always found time to answer, even though he is at the final stages of his promotion. For this I am grateful.

Working at the WCW was fun, there were a lot of friends there, working on their own theses. Jette Bunders, Martijn Brekhof, Marius Olsthoorn, Nadeem de Vree, Bouke Huurnink are but a few. Apart from going to lunch, there were always one or two of them ready to go get some coffee, or have a game of pingpong. I saw most of them graduate, although some are still there.

Thoughout my studies, there were always my parents Wouter and Thida, helping me where they could, both with money and the occasional Kick in the Ass. Both were very much needed, although the latter was always less appealing than the former. I thank them both for their enormous patience and support.

Every once in a while I still meet up with three friends from way back. Eugene Tuinstra, Jelle Kok and Remco de Boer. They graduated years ago, and every time I see them they ask me: "Don't you think its about time . . . ".

Well, my friends . . . the time has come.

Contents

Chapter 1

Introduction

SDF, or Syntax Definition Formalism, provides a solid method for creating specifications of languages, which can be used to create a parser for that language. In combination with ASF, or Algebraic Specification Formalism, an interpreter or a compiler can be created with little effort, giving the user the ability to create a (programming) language easily.

SDF has been around for a while now. In that time, a growing number of languages, data-representations, protocols and other entities that are regulated by some sort of grammar have been implemented in SDF. Most of these grammars are built from scratch or based on earlier versions using traditional editors like *emacs*, *nedit* or *vi*. Adapting these grammars, whether for re-engineering, recovery or other purposes, is done using the same tools.

The traditional editors treat an SDF grammar as text. They are not aware of the semantic properties of the language, and have no functionality to aid in an adaptation on that level. They provide functions like search/replace, or undo/redo, or even a syntax highlighter.

An SDF grammar is more than text. A simple SDF parser knows the difference between sorts and unquoted literals, or between productions and aliases, which largely have the same syntax. It would be beneficial to the adaptation process if this informations can be used. We would then, for instance, be able to search/replace only constructs of a certain type, like sorts. If the programmer wants to change the name of a sort in the grammar, it would be nice not to have the unquoted literals with the same text, if any, be replaced also.

What is needed is an editor that pays full attention to the engineering aspects of grammars. The editor needs to be *grammar-aware* [9].

Beyond Editors

Adapting grammars that are based on earlier versions would also benefit from grammar aware software. Using traditional editors, the programmer would copy the original text and adapt it manually. The digital link between versions is lost, since there is no way to get from one version to the other, other than to re-adapt the grammars. If, for instance, a problem is found in the specification of an early version that is also present in the later

ones, it would be nessesary to update all versions manually, making long-term maintenance difficult.

An example of a language where many versions have to be maintained is COBOL, which has many different dialects with the same language base. The basics of the COBOL language stay the same between dialects, but each dialect introduces differences.

A grammar aware adaptation, or transformation, language would enable a programmer to automate the process of creating a new version of a grammar. If a change is made in the original, the programmer will, in most cases, only have to re-run the transformation process to get the updated, newer version.

There are many more functions that can be envisioned to be useful in adapting SDF grammars. What is needed, is a grammar aware transformation language that can address these issues and serve as a powerful tool for adapting SDF grammars.

Adaptation of Grammars

Grammar adaptation was previously described in [7], in which a number of operators for grammar development, grammar maintenance, grammar reengineering and grammar recovery were given. The paper describes these operators independant of a specification language.

An existing transformation language called FST, a Framework for SDF Transformation, implements the operators presented in [7] for the transformation of SDF grammars. It was described by Ralph Lämmel and Guido Wachsmuth in [5]. A prototype implementation was created that is based on an older version of SDF. It is no longer up to date, and cannot be used with the current version of SDF.

There is another implementation of the framework also called FST which is part of the GDK or Grammar Deployment Kit. The GDK introduces a new specification language with a BNF-like notation called LLL, which is also pronounced as “L-Cube”. The GDK contains tools to transform SDF to LLL and back. The language FST that is part of the GDK can be used to perform transformations on LLL specifications. The GDK is more up to date, but SDF specific notations and semantics are lost in the translation from SDF to LLL, since SDF is more expressive than LLL.

In both versions of FST, each adaptation has to be implemented literally, by removing productions and then adding new ones which have to be provided by the programmer. The resulting script can then be used to adapt the grammar, and grammars derived from it, assuming the productions referenced are still part of the derived grammars. This means that scripts created with either version of FST are very much grammar dependant in that they can only be used with the grammar they were intended to transform, possibly including derived grammars.

Although useful, both versions of FST lack the functionality needed for a more generic approach in grammar adaptation. Creating tranformations based on information gathered from the specification for instance, is not possible in either language. Also, creating generic transformations that can be used to transform *any* grammar is not possible.

An SDF Specific Grammar Adaptation Language

In this thesis, we will attempt to create a scripting language for the adaptation of SDF grammars that adresses the issues described, as wel as others. The goal is to create a language that is flexible enough to enable a programmer to fully adapt an SDF specification, whether for re-engineering, recovery or other purposes. The language should allow for the creation of scripts that are *grammar-independant* in the sense that they can be applied to any grammar, given the correct input. Examples range from renaming all dash-separated sorts to camelcase (see section 2.4.3, p.19), to comparing two versions of the same grammar and compiling a (sub)grammar of the parts they have in common.

The Approach of this Thesis

To discover the issues that need to be addressed in creating an SDF specific grammar adaptation language, we will re-implement an adaptation previously done without the use of adaptation tools (chapter 2, p.15) In this case study each of the steps taken will be analyzed and where possible implemented in FST to discover what functionality is needed in a grammar adaptation language, both for the direct adaptation of the case and for a more generic approach to allow for scripts that can be used on different and not just derived grammars.

Using the results of the case study, we will design a new language for adapting SDF grammars (chapter 3, p.35). This language should be extensible so that it can be adapted to work with future versions of the SDF language. It should also be flexible enough to allow the easy recreation of the case study (chapter 5, p.95), and more (appendix C, p.125).

Our aim is to provide a working interpreter of a language (chapter 4, p.79) implementing the designed framework.

Appendices

There are several appendices:

- appendix A, p.111 provides an overview of the diagrams presented in chapter 4, p.79, as well as a depandancy graph of the implementation of the interpreter in its current state.
- appendix B, p.115 provides an implementation of the functions of FST in GTL.
- appendix C, p.125 shows an example program in GTL.
- appendix D, p.127 gives an index for the thesis, mostly for chapter 3, p.35.
- appendix E, p.131 lists the abbreviations used in this thesis, providing a small description for each of them.

Chapter 2

Case Study

2.1 Introduction

Before we can create a new transformation language for SDF, we need to know what functionality is useful and what functionality is required to perform transformations on SDF grammars. We will do this by retracing the individual transformations steps taken in an earlier, manual adaptation of a grammar. The chosen grammar transformation is the adaptation of a C grammar in LEX+YACC to an SDF grammar. This specification was originally converted and adapted to an SDF specification by Jurgen Vinju and Hayco de Jong. The transformation steps that were taken were listed in comments at the top of the resulting SDF specification.

To retrace the original adaptation of the C grammar, each transformation step as listed in the resulting original SDF specification will be copied and analyzed. In some cases, the transformation may lend itself to be implemented in FST[1], but since the interpreter for the language works with an older version of SDF we cannot test the scripts. Implementing the scripts in FST will still provide feedback as to the approach of the language and how the functionality it provides could be used to create a new grammar transformation language for SDF. For some steps, an implementation will also be given in a *hypothetical* version of FST. This hypothetical version of FST will be called *h*FST in the remainder of this thesis.

Implementing some of the adaptations in *h*FST will show functionality that is nonexistant in FST but that is needed to automate the transformation. The resulting *h*FST scripts can provide us with additional functionality that could be useful in the design of the new transformation language for SDF.

In analyzing the transformation steps in this case study, we will be looking at repetition and order of statements. We hope to end up with a primitive set of transformations or underlying transformation steps which will serve as a basis for the design of the grammar transformation language. The concepts that result from this case study will be used to create a new SDF transformation language called GTL. With this language, we should then be able to define scripts comparable to the *h*FST scripts presented here, and execute

[1] In this chapter, FST implies the SDF specific version [5], not the version part of the GDK.

them.

2.2 FST

In this chapter, FST implies the version as implemented in SDF [5]. It implements most of the operators presented in [7], and only works on sorts and sort definitions, as well as modules. SDF specific notations and semantics like *aliases* or *priorities* can only be added or removed as syntax that is part of an encompassing module. This also needs to be addressed in the design of the new transformation language.

The operators present in FST work within a *focus*. This focus can be a module or list of modules, or a sort or list of sorts. Any operator used in a focus will only change the productions within that focus.

FST implements a number of *conditions* for checking the state of the grammar. For example, one can find out if a module is empty, or if a sort is defined or not. The language provides operators for comparing two (sub)grammars. The *covers* operator for instance, will check whether two SDF grammars are *structurally equal*. There are also operators for checking whether a grammar is a subgrammar of another.

In FST, comments start with '`%%`', just like in SDF. Variables or user-defined functions are not part of the language.

2.3 Approach

In this case study we will use the ANSI-c specification in LEX+YACC from the April 30, 1985 draft of the proposed standard[2]. This specification has been converted to an SDF specification by Jurgen Vinju and Hayco de Jong. We will retrace their steps both manually and by implementing some of the transformations in FST, as well as *h*FST, a hypothetical version of FST.

The parts of the case study that can be done by a computer will be executed on a system running RedHat 9 or Fedora Core 3, both GNU/linux based operating systems. The XT toolkit is a bundle of tools for for building program transformation systems. It contains a tool named '`yacc2sdf`' which can transform a YACC specification to a primitive SDF specification. This is the only tool from the XT toolkit that is used in this case study.

The source code of the prototype implementation of FST in ASF+SDF will be used to recover the semantics specific to SDF as expressed by the various operators and functions part of FST.

2.4 Retracing the Manual Steps

The original SDF specification listed in its header the various steps taken. They are reproduced here one by one. For each step, an explanation of the transformation is presented, and a recount is given on their reproduction.

[2] http://www.lysator.liu.se/c/ANSI-C-grammar-y.html

2.4.1 "first a one-to-one translation from YACC to SDF"

The first step we take is to use use the '`yacc2sdf`' transformer provided with XT for the trivial translation. The various commands are not explained here. They are listed to help the reader recreate the case study should that be desired.

- ```
  parse -l yacc \
        -I -i ANSI-C-grammar.yacc \
        -o ANSI-C-grammar.yacc.ast
  ```

- ```
  yacc2sdf -b -i ANSI-C-grammar.yacc.ast \
           -o ANSI-C-grammar.def.ast
  ```

- ```
  ast2abox -i ANSI-C-grammar.def.ast \
           -o ANSI-C-grammar.def.abox \
           -p ~/local/share/gb/sdf.2.1/sdf.cons.pp
  ```

- ```
  abox2text -i ANSI-C-grammar.def.abox \
            -o ANSI-C-grammar.def
  ```

The following command takes an SDF definition file ('`*.def`') and extracts the SDF module files ('`*.sdf`') from it. It is not strictly nessesary for the translation process.

- ```
  unpack-sdf ANSI-C-grammar.def
  ```

These commands do not take care of the '`.l`' lexical file, which defines the various keywords and operators of ANSI-c.

2.4.2 "replaced terminal identifiers ('`RETURN`') by literal symbols ('`"return"`')"

The terminal identifiers and operators (tokens) present at the beginning of the YACC grammar file were translated to empty SDF productions and put in a separate module named '`Lexical`' within the resulting SDF specification.

Most of the tokens represent keywords in ANSI-c, some represent operators. Their literal representations are listed in the lexical file ('`*.l`'). They were not included in the '`yacc2sdf`' transformer simply because LEX is not the same as YACC. A different tool should be created (say '`lex2sdf`') to transform the LEX file to SDF '`lexical syntax`'.

Since FST can only parse SDF the literal representations will have to be taken from the LEX file and then added. By hand, we would copy the literals from the LEX file to the representing production in the '`Lexicals`' module. Specifying these transformations in FST would, in the case of the '`RETURN`' identifier, amount to applying the following script:
  ```

FST
```
focus on module Lexical
do
(
 include "return" in RETURN;  %% add production:    "return" -> RETURN
 exclude  from RETURN;        %% remove production:          -> RETURN
 unfold RETURN;               %% substitute all sorts RETURN with its
                              %%  single defining term ("return")
 eliminate RETURN;            %% remove all definitions of RETURN
);
```

To repeat this for all the identifiers we will have to copy/paste this script for each, replacing the literal and the sort name. For instance, transforming the 'CONTINUE' and 'BREAK' identifiers:

FST
```
focus on module Lexical
do
(
 include "continue" in CONTINUE; %% add production:    "continue" -> CONTINUE
 exclude  from CONTINUE;         %% remove production:            -> CONTINUE
 unfold CONTINUE;                %% substitute all sorts CONTINUE with its
                                 %%  single defining term ("continue")
 eliminate CONTINUE;             %% remove all definitions of CONTINUE

 include "break" in BREAK; %% add production:    "break" -> BREAK
 exclude  from BREAK;      %% remove production:         -> BREAK
 unfold BREAK;             %% substitute all sorts BREAK with its
                           %%  single defining term ("break")
 eliminate BREAK;          %% remove all definitions of BREAK

 %% ...
);
```

As can be seen, each identifier can be transformed by calling four operators (include, exclude, unfold and eliminate) in order. If we had variables and functions with arguments, we could write something like the following function, where variable names start with a dollar-sign ('$') and function names with an ampersand ('&'). Note that the declaration of the arguments to the function contains type references (Sort, Literal).

*h*FST
```
function &replace_id(Sort $S, Literal $L) (
 include $L in $S;  %% add production:    "return" -> $S
 exclude  from $S;  %% remove production:          -> $S
 unfold $S;         %% substitute all sorts $S with its
                    %%  single defining term ($L)
 eliminate $S;      %% remove all definitions of $S
);
```

The previous scripts could then be implemented as follows:

*h*FST
```
focus on module Lexical
do
(
 &replace_id(RETURN,"return");
 &replace_id(CONTINUE,"continue");
 &replace_id(BREAK,"break");
 ...
);
```

2.4.3 "renamed non-terminals from dash separated to CamelCase"

The '`yacc2sdf`' script already took care of this. Apparently we are working with a different version than the one used for the original transformation. It is not known which version-, or even if the same script- was used.

This item does show however, that in transforming an SDF specification it might be useful to be able to (partially) rename sorts, using the old name as a basis. For instance from '`This-sort-name`' to '`ThisSortName`'. A mechanism would be needed to match the relevant parts of a name, and then insert the match into a new name.

2.4.4 "imported lexical definition of C comments from the CPP grammar"

The definition of C comments is missing from the SDF grammar generated by the '`yacc2sdf`' script, since it is part of the LEX file but has no tokens in the YACC file. We will need to add it.

Fortunately, all productions defining comments are in the '`Comments`' SDF module in the CPP specification. The modules '`CommentsAsLayout`' and '`LineCommentsAsLayout`' which define comments as part of the layout of the CPP language, can be directly included without alteration into the ANSI-c specification.

Although not strictly nessesary in this case, this item shows that it can be useful to be able to select parts of one specification and insert them into another.

2.4.5 "then added follow restrictions on identifiers and some unary operators"

Follow restrictions in SDF are used to disambiguate lexical syntax. For instance, the sort '`Identifier`' in ANSI-c is defined as follows:

SDF
```
lexical syntax
  ...
  [0-9]        -> D
  [a-zA-Z\_]   -> L
  L ( L | D )* -> Identifier
  ...
```

This makes the string '`Apple`' a valid identifier. However, cutting up the string into '`App`' and '`le`' does not mean that either part is not a valid identifier. To prevent '`Apple`' from being parsed as five adjacent, single character identifiers, a follow restriction is added which states that an identifier cannot be followed by a character that is allowed to be part of that identifier. The follow restriction for '`Identifier`':

SDF

```
1 lexical restrictions
2   Identifier -/- [0-9a-zA-Z\_]
```

Since the SDF specification generated by the '`yacc2sdf`' program does not include follow restrictions, they need to be added where nessesary.

The following restrictions, in which the regular expression '`[0-9a-zA-Z_]`' represents the original definition of the sort '`Identifier`' ('`(L |D)`'), were added to the original specification:

SDF

```
159 lexical restrictions
160
161   Identifier -/- [0-9a-zA-Z\_] %% (L | D)
162   D+ -/- [0-9]
```

SDF

```
179 context-free restrictions
180
181   LAYOUT? -/- [\ \t\011\n\r\012]
```

SDF

```
220 context-free restrictions
221   "&" -/- [\&]
222   "-" -/- [\-]
223   "+" -/- [\+]
```

Performing this transformation by hand means identifying and adding the restrictions. This can be a difficult task, since some problems stemming from the lack of restrictions can be complex, occurring only in specific situations.

FST does not provide a means for manipulating restrictions, but an extension to the '`include`' operator can be added to *h*FST that would allow it to take other SDF constructs than productions. It could look like this:

FST

```
1 focus on module Generated
2 do
3 (
4  include lexical restriction Identifier -/- [0-9a-zA-Z\_];
5  include lexical restriction D+ -/- [0-9];
6  include context-free restriction LAYOUT? -/- [\ \t\011\n\r\012];
7  ...
8 );
```

2.4.6 "fixed an ambiguity in the lexical syntax, 'IS*?' was produced, changed to 'IS*'."

This ambiguity isn't generated by the 'yacc2sdf' script we used. Nor by the manual addition of the lexical syntax from the LEX file. It seems in the original specification the definition of 'IS' was changed from '[uUlL]* -> IS' to '[uUlL] -> IS', changing all symbols 'IS' in other productions to 'IS*' to compensate. There are a few symbols 'IS?' that would change into 'IS*?' if 'IS' were to be replaced by 'IS*' based on syntax. Correcting this is easy, just replace all occurrences of 'IS*?' with 'IS*'.

If we wanted to correct this using FST we would have to exclude the ambiguous definitions and include their correct version. Of course, FST must be able to work on ambiguous grammars for this to work. The 'preserve' operator allows us to replace the symbol with the restriction that the old and the new definition must be equivalent. For example:

FST
```
1 focus on module Generated
2 do
3 (
4  preserve [0] [xX] H+ IS*? in HexadecimalConstant as [0] [xX] H+ IS* ;
5  preserve D+ IS*? in IntegerConstant as D+ IS* ;
6  ...
7 );
```

In FST one can use *patterns* to more generically select from a grammar. The pattern '...' represents 'zero or more' symbols, and the pattern '.' represents 'exactly one' symbol. For example:

FST
```
1  exclude ... IS*? ... from HexadecimalConstant;
```

will remove all productions that define 'IS*?' as part of the definition for the sort 'HexadecimalConstant'. However, using the patterns with the 'preserve' operator is not possible, unless the symbols matched by the patterns ('...') can be stored and retrieved so that they can be reintroduced in the 'as' part of the 'preserve' operator.

The following *h*FST function (see 2.4.2) will remove all occurrences of 'IS*?' from the grammar. It starts with three variable declarations, specifying a type as well as a pattern. '$S' represents the sort defined by '$P1 IS*? $P2', where '$P1' and '$P2' each represent a symbol with the pattern '...'. A new operator 'for each' is introduced that will execute a statement for each production matching its argument 'sort $S defining $P1 IS*? $P2'.

*h*FST
```
1 function &disambiguate() (
2  var Sort $S:.;
3  var Symbol $P1:...;
4  var Symbol $P2:...;
5
6  for each sort $S defining $P1 IS*? $P2
7  do preserve $P1 IS*? $P2 in $S as $P1 IS* $P2 ;
8 );
```

The empty variables '$S', '$P1' and '$P2' are matched by the 'for each' operator using their given pattern, and they store the result. They are subsequently used with the 'preserve' operator to remove the ambiguous definition and introduce the unambiguous one for the sort '$S'. Once the operation is concluded, the next match is made and processed until there are no more matches left.

2.4.7 "re-factored all right-associative lists using SDF2 list syntax"

Traditionally, right associative lists are defined recursively as follows; using the definition for the comma-separated list 'ArgumentExprList' as an example:

SDF
```
90  context-free syntax
91    AssignmentExpr -> ArgumentExprList
92    ArgumentExprList "," AssignmentExpr -> ArgumentExprList
```

In SDF2 an extended notation was introduced that has the same semantics, but is easier to read as well as maintain. The previous example in SDF2 notation looks like this:

SDF
```
1  context-free syntax
2    { AssignmentExpr "," }+ -> ArgumentExprList
```

We found the following sorts that define right-associative comma-separated lists:

ArgumentExprList	StructDeclaratorList	InitDeclaratorList
ParameterIdentifierList	IdentifierList	ParameterTypeList
ParameterList	InitializerList	EnumeratorList
StatementList		

Space-separated right-associative lists, which are also defined recursively, can be re-factored to use the Kleene Plus ('+'), which signifies 'one or more'. For example the definition for 'StructDeclarationList' after the trivial transformation looks like this:

SDF
```
239  context-free syntax
240    StructDeclaration -> StructDeclarationList
241    StructDeclarationList StructDeclaration -> StructDeclarationList
```

It can be written as follows in SDF2 syntax:

SDF
```
1  context-free syntax
2    StructDeclaration+ -> StructDeclarationList
```

There are three space-separated list definitions in the grammar: StructDeclarationList, TypeSpecifierList and DeclarationList.

In the original specification all the lists in the grammar were manually re-factored. In FST this can be established by excluding the two productions defining the list and including a single production which defines the same list using SDF2 notation. For the given examples this would amount to the following FST scripts:
for the comma-separated 'ArgumentExprList' example:

FST
```
include { AssignmentExpr "," }+ in ArgumentExprList;
exclude AssignmentExpr from ArgumentExprList;
exclude ArgumentExprList "," AssignmentExpr from ArgumentExprList;
```

for the space-separated '`StructDeclarationList`' example:

FST
```
include StructDeclaration+ in StructDeclarationList;
exclude StructDeclaration from StructDeclarationList;
exclude StructDeclarationList StructDeclaration from StructDeclarationList;
```

To create a more generic script for this type of transformation we would need to match two productions per sort. In the next *h*FST script a boolean '`and`' operator is introduced for this purpose. We also need to check that the two productions we're matching are the only productions defining the list. We do this by first matching and removing them, after which we check that there are no more productions defining the sort, i.e. the two matched productions were the only ones. If that check succeeds we add the new list notation, if it fails we fail the entire '`for each`' cycle, restoring the state of the grammar to what it was before the '`for each`' loop started.

The following code transforms all comma-separated right-assiciative lists in a grammar.

*h*FST
```
focus on module Generated
do
(
 var Sort $S:.;
 var Symbol $S1:.;

 for each ( sort $S defining $S1 and sort $S defining $S "," $S1 )
 do
 (
  exclude $S1 from $S;
  exclude $S "," $S1 from $S;
  if not defined sort $S then
   include { $S1 "," }+ in $S;
  else fail;
 );
);
```

2.4.8 "renamed all non-terminals from '`LogicalAndExpression`' up-to and not including '`UnaryExpression`' to '`LogicalOrExpression`'"

The item goes on to state that:
"This was a single injection chain with left recursive binary operators. Each non-terminal up the chain introduced a higher priority. The priorities are now expressed using context-free priorities to cope with the disappearance of the non-terminals. All trivial productions

from 'LogicalOrExpression' to 'LogicalOrExpression' that were introduced by the previous actions have been removed to fix the obvious circularity."

The following sorts were renamed to 'LogicalOrExpression':

CastExpr	RelationalExpr	InclusiveOrExpr
MultiplicativeExpr	EqualityExpr	LogicalAndExpr
AdditiveExpr	AndExpr	
ShiftExpr	ExclusiveOrExpr	

The last sort in the list, 'LogicalAndExpr', is used in the definition of 'LogicalOrExpr'. The sort 'InclusiveOrExpr' is used in the definition of 'LogicalAndExpr' and so on, with 'CastExpr', which is used in the definition of 'MultiplicativeExpr', being at the bottom.

The definition of each of the sorts has the following layout, using the sorts 'LogicalAndExpr' and 'LogicalOrExpr' as an example:

SDF
```
153  context-free syntax
154    InclusiveOrExpr -> LogicalAndExpr
155    LogicalAndExpr "&&" InclusiveOrExpr -> LogicalAndExpr
156
157  context-free syntax
158    LogicalAndExpr -> LogicalOrExpr
159    LogicalOrExpr "||" LogicalAndExpr -> LogicalOrExpr
```

Renaming the sort 'LogicalAndExpr' to 'LogicalOrExpr' will result in the following set of productions:

SDF
```
153  context-free syntax
154    InclusiveOrExpr -> LogicalOrExpr
155    LogicalOrExpr "&&" InclusiveOrExpr -> LogicalOrExpr
156
157  context-free syntax
158    LogicalOrExpr -> LogicalOrExpr
159    LogicalOrExpr "||" LogicalOrExpr -> LogicalOrExpr {left}
```

Note that the definition of 'LogicalOrExpr' has changed with the renaming. An associativity attribute is also added. The trivial injection:

SDF
```
158    LogicalOrExpr -> LogicalOrExpr
```

has to be removed.

To preserve the priorities between the productions we must now add SDF2 type context-free priorities. In the example 'LogicalAndExpr' has a higher priority than 'LogicalOrExpr'. This is represented in SDF2 by adding the following context-free priority:

SDF

```
context-free priorities
  LogicalOrExpr "&&" InclusiveOrExpr -> LogicalOrExpr {left}
  >
  LogicalOrExpr "||" LogicalOrExpr -> LogicalOrExpr {left}
```

Since this is a clear set of steps that have to be taken for each sort in the list, we can create a generic *h*FST script that takes such a sort as an argument. The script will then rename the sort to '`LogicalOrExpr`', remove the trivial injection and add the relevant context-free priority. If we call this script for each of the sorts in the list we will have transformed the injection-chain.

*h*FST

```
function &Rename_With_Priority(Sort $S:.) (
 var Symbol $Y1:.;
 var Symbol $Y2:.;
 var Literal $L1:.;
 var Literal $L2:.;
 var Group $G1:.;
 var Group $G2:.;

 %% create the first group for the priorities ...
 for each sort $Y1 defining $Y1 $L1 $S do (
  include $S $L2 $Y2 -> $S in $G1;
  include attribute {left} in $Y1;
 );

 %% create the second group for the priorities ...
 for each sort $S defining $S $L2 $Y2 do (
  include $Y1 $L1 $S -> $Y1 in $G2;
  include attribute {left} in $S;
 );

 %% include the context-free priority ...
 include context-free priorities $G1 > $G2;

 %% rename the sort ...
 rename $S to LogicalOrExpr;

 %% remove the trivial injection ...
 exclude LogicalOrExpr from LogicalOrExpr;
);
```

The separate creation of the groups to be used in the priority grammar is nessesary because there might be more than one production defining the given sort, or the sort using the given sort in its definition.

The first '`include`' statement used in the for each loops adds a production to a *group*, not a grammar. The SDF2 syntax of groups is implied. The second '`include`' statement

in the loops include an attribute defining left-associativity in the definition of the sort in question.

The last '`include`' statement in the function adds a priority grammar containing the two groups created to the focus. After this, the sort is renamed and the trivial injection is removed.

2.4.9 "added lexical restriction on '`D+`'"

Adding follow restrictions to a sort was discussed in 2.4.5. This lexical restriction was already listed there as part of the additions made to the original specification. It was also part of the example *h*FST script.

2.4.10 "removed '`UnaryOperator`' non-terminal by inlining the alternative operators"

This resembles the transformation made in 2.4.2, where terminal identifiers were unfolded and eliminated. The difference is that '`UnaryOperator`' represents more than one literal. These are the productions defining '`UnaryOperator`':

SDF
```
101 context-free syntax
102    "&" -> UnaryOperator
103    "*" -> UnaryOperator
104    "+" -> UnaryOperator
105    "-" -> UnaryOperator
106    "~" -> UnaryOperator
107    "!" -> UnaryOperator
```

We need to add a copy of each production encorporating the sort '`UnaryOperator`', per literal defining it. Take for instance the production:

SDF
```
97    UnaryOperator CastExpr -> UnaryExpr
```

The FST '`unfold`' operator expects only a single literal meaning we cannot use it. To unfold '`UnaryOperator`', we have to include six new productions defining '`UnaryExpr`', namely the following:

SDF
```
1     "&" CastExpr -> UnaryExpr
2     "*" CastExpr -> UnaryExpr
3     "+" CastExpr -> UnaryExpr
4     "-" CastExpr -> UnaryExpr
5     "~" CastExpr -> UnaryExpr
6     "!" CastExpr -> UnaryExpr
```

Then we can remove the original production.

The FST script would look like this:

FST
```
focus on module Generated
do
(
 include "&" CastExpr in UnaryExpr;
 include "*" CastExpr in UnaryExpr;
 include "+" CastExpr in UnaryExpr;
 include "-" CastExpr in UnaryExpr;
 include "~" CastExpr in UnaryExpr;
 include "!" CastExpr in UnaryExpr;

 exclude UnaryOperator CastExpr from UnaryExpr;
);
```

2.4.11 "removed 'AssignmentOperator' non-terminal by inlining the alternatives"

This item needs the same functionality as the previous one where '`UnaryOperator`', which was defined by multiple literals, was unfolded. The sort '`AssignmentOperator`' is defined by the following productions:

SDF
```
  context-free syntax
    "=" -> AssignmentOperator
    "*=" -> AssignmentOperator
    "/=" -> AssignmentOperator
    "%=" -> AssignmentOperator
    "+=" -> AssignmentOperator
    "-=" -> AssignmentOperator
    "<<=" -> AssignmentOperator
    ">>=" -> AssignmentOperator
    "&=" -> AssignmentOperator
    "^=" -> AssignmentOperator
    "|=" -> AssignmentOperator
```

The FST code required to do this transformation is the same as in 2.4.10, but using a diferent sort.

A more generic function unfolding any sort defined by multiple literals would use the *h*FST '`for each`' operator introduced in 2.4.6. It takes the terminal sort as an argument:

*h*FST
```
function &unfold_many(Sort $S:.) (
 var Symbol $Y1:...;
 var Symbol $Y2:...;

 for each sort $S1 defining $Y1 $S $Y2
 do
 (
  var Literal $L:.;

```

```
10   for each sort $S defining $L
11   do include $Y1 $L $Y2 in $S1;
12
13   exclude $Y1 $S $Y2 from $S1;
14  );
15 );
```

The first '`for each`' loop matches all the productions encorporating the given terminal sort '`$S`' one by one. The second '`for each`' loop includes a new production for each literal defining the given terminal sort '`$S`'. In the case of '`AssignmentOperator`' these are '`=`', '`*=`', '`/=`', '`%=`', '`+=`', ... After that the original production is excluded from the grammar.

Using this function we can now unfold the non-terminals '`UnaryOperator`' and '`AssignmentOperator`'. The *h*FST code, not including the definition of the function, looks like this:

*h*FST

```
1 focus on module Generated
2 do
3 (
4  &unfold_many(UnaryOperator);
5  &unfold_many(AssignmentOperator);
6 );
```

2.4.12 "renamed '`LogicalOrExpression`' to '`BasicExpression`'"

This comes down to renaming a sort. All productions defining '`LogicalOrExpr`' must be changed to define '`BasicExpr`' instead. Also, all productions referencing '`LogicalOrExpr`' must be made to reference '`BasicExpr`' instead.

FST includes the '`rename`' refactoring operator for this purpose:

FST

```
1 focus on module Generated
2 do rename LogicalOrExpr to BasicExpr;
```

Note that in the original specification all sorts that ended in '`Expr`' were changed to end in '`Expression`', hence the difference between the original explanation and this one. This '`Expr`' transformation is not listed in the summary of the original specification and it will be addressed in 2.5.

2.4.13 "renamed '`PostfixExpression`' to '`PrimaryExpression`' and removed the trivial injection"

The first part of this transformation is the same as in 2.4.12. However, one of the productions prior to renaming reads as follows:

SDF

```
79   context-free syntax
80     PrimaryExpr -> PostfixExpr
```

If we rename ‘PostfixExpr’ to ‘PrimaryExpr’ we would end up with:

SDF
```
79  context-free syntax
80    PrimaryExpr -> PrimaryExpr
```

which introduces a loop. This production needs to be removed. The resulting FST script looks like this:

FST
```
1 focus on module Generated
2 do
3 (
4  rename PostfixExpr to PrimaryExpr;
5  exclude PrimaryExpr from PrimaryExpr;
6 );
```

2.4.14 “folded empty and non-empty use of ‘ArgumentList’ in ‘PrimaryExpression’”

In 2.4.7 the right-associative list defining ‘ArgumentExprList’ was refactored to SDF2 syntax. The resulting definition is as follows:

SDF
```
1   context-free syntax
2     { AssignmentExpr "," }+ -> ArgumentExprList
```

Since ‘ArgumentExprList’ is only ever used in the definition of ‘PrimaryExpr’ (formerly ‘PostfixExpr’, see 2.4.13) we can change this definition to

SDF
```
1   context-free syntax
2     { AssignmentExpr "," }* -> ArgumentExprList
```

and remove the empty use of ‘ArgumentExprList’ from the definition of ‘PrimaryExpr’.

The *h*FST script looks like this:

*h*FST
```
1 focus on module Generated
2 do
3 (
4  preserve { AssignmentExpr "," }+ in ArgumentExprList
5   as { AssignmentExpr "," }*;
6  exclude PrimaryExpr "(" ")" from PrimaryExpr;
7 );
```

2.4.15 “Introduced separate lexical non-terminals for each type of constant: ‘IntegerConstant’, ‘HexadecimalConstant’, ‘CharacterConstant’ and ‘FloatingPointConstant’”

From the LEX specification we manually added among others the following productions defining the sort ‘Constant’:

SDF
```
lexical syntax
  [0] [xX] H+ IS* -> Constant                  %% HexadecimalConstant
  [0] D+ IS* -> Constant                       %% IntegerConstant
  D+ IS* -> Constant                           %% IntegerConstant
  [\'] ([\\]~[] | ~[\\\'] )+ [\'] -> Constant %% CharacterConstant

  D+ E FS? -> Constant                         %% FloatingPointConstant
  D* [\.] D+ E? FS? -> Constant                %% FloatingPointConstant
  D+ [\.] D* E? FS? -> Constant                %% FloatingPointConstant
```

They are however different types of constants. We appended these as a comment to each production.

We can express this transformation by folding each definition to their respective constant type, which also maintains an injection of that type into the sort '`Constant`'. The FST script for this transformation; note the specific focus on the sort '`Constant`':

FST
```
focus on sort Constant
do
(
 fold [0] [xX] H+ IS* to HexadecimalConstant;
 fold [0] D+ IS* to IntegerConstant;
 fold D+ IS* to IntegerConstant;
 fold [\'] ([\\]~[] | ~[\\\'] )+ [\'] to CharacterConstant;

 fold D+ E FS? to FloatingPointConstant;
 fold D* [\.] D+ E? FS? to FloatingPointConstant;
 fold D+ [\.] D* E? FS? to FloatingPointConstant;
);
```

This script results in the following productions:

SDF
```
lexical syntax
  [0] [xX] H+ IS* -> HexadecimalConstant
  [0] D+ IS* -> IntegerConstant
  D+ IS* -> IntegerConstant
  [\'] ([\\]~[] | ~[\\\'] )+ [\'] -> CharacterConstant

  D+ E FS? -> FloatingPointConstant
  D* [\.] D+ E? FS? -> FloatingPointConstant
  D+ [\.] D* E? FS? -> FloatingPointConstant
```

SDF
```
  HexadecimalConstant -> Constant
  IntegerConstant -> Constant
  CharacterConstant -> Constant
  FloatingPointConstant -> Constant
```

In the original specification the sort '`Constant`' was subsequently unfolded, eliminating it from the grammar. This transformation is unlisted in its summary and will be adressed in 2.5.

2.4.16 "Removed production '`[0] D+ IS* -> IntegerConstant`' because '`D`' includes '`[0]`' and the production '`D+ IS* -> IntegerConstant`' exists too"

The sort '`D`' is defined as

SDF
```
1 [0-9]        -> D
```

and as such includes the character '`0`'. The production:

SDF
```
1 [0] D+ IS* -> IntegerConstant
```

is no longer nessessary since it is covered by the production:

SDF
```
1 D+ IS* -> IntegerConstant
```

In FST its removal is expressed by a simple exclusion:

FST
```
1 focus on module Generated
2 do exclude [0] D+ IS* from IntegerConstant;
```

2.5 Unlisted Transformations Encountered

There are a number of transformations that were done for the original specification but that were not listed in its summary.

2.5.1 Problems encountered during step 2.4.1

The trivial transformation did not introduce any sort declarations, these had to be added by hand, since FST cannot handle sort declarations directly. Indirectly, sort declarations are added every time a new sort definition is introduced, and removed when all definitions for a sort are removed.

2.5.2 Problems encountered during step 2.4.2

The following changes were made in accordance with the transformations done for the original specification.

- The '`RANGE`' lexical definition was missing but the sort is never used so we removed the (empty) definition.

- The '`ELIPSIS`' lexical definition was missing, the sort is used in two productions defining '`ParameterIdentifierList`' and '`ParameterTypeList`'. An elipsis is depicted by three dots ('...') signifying a continuance of a pattern. We added the lexical syntax to the (empty) definition.
- The '`TYPENAME`' lexical definition was missing, the sort is used in '`TypeSpecifier`'. We removed the (empty) lexical definition and the '`TypeSpecifier`' injection.
- The '`IDENTIFIER`' lexical definition is missing but can be extracted. One of the LEX definitions returns a function call to '`check_type()`', which is defined in the same LEX file. That function always returns '`IDENTIFIER`'. We have taken the lexical definition returning the function call to be the lexical definition of '`IDENTIFIER`'.
- The sort '`IDENTIFIER`' was neither unfolded nor eliminated in the original specification, instead it was renamed to '`Identifier`'.

2.5.3 All sorts that ended in "`Expr`" were changed to end in "`Expression`"

This is a renaming operation, however, as with item 2.4.3 where all dash-separated sorts were renamed to CamelCase, we need to be able to match parts of a sort-name itself to be able to create a new name based on the old. In this case, using '`for each`', we need to match all sorts ending in '`Expr`', storing the part before it to rename the sort using a new ending '`Expression`'.

2.5.4 The sort "`Constant`" was unfolded, eliminating it from the grammar

This is an unfolding operation for multiple definitions of a sort, and was described in 2.4.10. However, the *h*FST function given only works with literals being the terminal part. If we change that to type '`Symbol`' we can unfold sorts as well as literals, giving us the ability to unfold the sort '`Constant`' using the function. The *h*FST function for unfolding multiple definitions of a sort:

*h*FST

```
function &unfold_many(Sort $S:.) (
 var Symbol $Y1:...;
 var Symbol $Y2:...;

 for each sort $S1 defining $Y1 $S $Y2
 do
 (
  var Symbol $L:.;

  for each sort $S defining $L
  do include $Y1 $L $Y2 in $S1;
  exclude $Y1 $S $Y2 from $S1;
 );
);
```

Using the created function on the sort 'Constant':

*h*FST
```
1 focus on module Generated
2 do &unfold_many(Constant);
```

An explanation for the script can be found at 2.4.10.

2.5.5 The sort "ConditionalExpression" was renamed to 'ConstantExpression'

This is a straightforward renaming:

FST
```
1 rename ConditionalExpression to ConstantExpression;
```

2.6 Conclusions

The implementations of the various steps in this case study provide a number of issues with their possible solutions that need to be addressed in the new transformation language. Most of the constructs part of FST should be part of the new language, possibly extended to provide a semantics that can transform any and all parts of an SDF2 specification. These include the functions for including/excluding SDF constructs, the condition checking etcetera.

Most of the steps in the case study can be implemented literally in FST. But implementing transformations that have to be repeated many times using the same operations but different input comes down to a lot of copy-paste work, which is time consuming.

One way of making this easier is to introduce *functions*, which can be seen in step 2.4.2. The *h*FST script created to handle this step creates a function that holds the statements that need to be evaluated per item. To feed the function with the relevant information we need *variables*, which are *typed* to prevent inproper use of a function. The repetition still exists but has become less expansive since only the function-call has to be repeated.

Another way of dealing with repetition is by introducing repetitive *control-structures*, i.e. *loops*. The *h*FST script in step 2.4.6 introduces a *for each* statement which takes a list of constructs and evaluates a group of statements for each item in the list. Again, variables are nessesary to hold the current item for evaluation. This statement is also used in steps 2.4.7, 2.4.8 and 2.4.11. It could also have been used in step 2.4.2 by creating a list of tuples to use as input for the function.

In FST, any transformation that fails will also fail the compound statement containing the transformation and so forth. The *if* control-structure which is part of FST can be used to ensure continued evaluation of the script even though a test failed. It is incorporated in the *h*FST script for step 2.4.7.

What a number of steps show is that the ability to include or exclude *SDF2 constructs* other than productions or sorts is needed. Steps 2.4.4 and 2.4.5 deal with import- and restriction- grammars respectively, and step 2.4.8 adds a specific attribute to a production. The SDF2 constructs that are added in the aforementioned steps cannot be added using

FST. In the *h*FST scripts for those steps this is solved by extending the semantics of include and exclude. The new transformation language must address this issue.

Step 2.4.6 shows the use of FST *patterns* to transform all productions for '`HexadecimalConstant`' containing the symbol '`IS*?`'. These patterns are also used with the variables in the *h*FST script for the step. Since a variable as used in step 2.4.2 does not have a pattern specification, it is not known if a variable is allowed to contain "exactly one" or "zero or more" of a certain type. To allow for this the variable declarations in this *h*FST script include a pattern-, as well as type- specification. Using variables with patterns also allows the programmer to specify a kind of sub-type, i.e. "all productions defining a specific sort" as a *type-constraint* instead of using a test each time the content of the variable changes.

Chapter 3

A Grammar Transformation Language

3.1 Introduction

Using the conclusions of the case study, we have created an SDF grammar transformation language incorporating the concepts introduced in the hypothetical FST programs.

GTL encompasses strong typing, variables, user patterns and functions to give the user full control over the transformation process. Through modularization the user is capable of setting up libraries of functions to be included in any GTL program.

These features combined provide a means for users to define just about any transformation. For instance, the operators of FST can largely be implemented as functions in a GTL library (see appendix B, p.115). Programs can be created for menial tasks such as listing all productions in a specification adhering to a certain pattern, or restructuring them so that productions defining identical sorts are grouped together. More complicated tasks can also be implemented, such as extracting all productions and declarations required for the definition of a given sort, and then generating a new specification for that definition.

In this chapter we will start by explaining the type- and pattern- systems at the root of the language. Then we will discuss variables, which are a special kind of pattern. Lastly we will elaborate on the composition of a GTL program and its sections.

3.2 Types

All GTL constructs have an associated type. GTL *Types* are representations of the constructs that are part of the SDF2 specification, like symbols or aliases. They have the same name as the sort of the SDF construct, meaning that a symbol is represented by the type `Symbol` in GTL.

A GTL type represents the most basic pattern of a construct and can only match that pattern. Any correct specification in SDF2 can be matched with a GTL type.

3.3 Patterns

A GTL *pattern* is an instantiation of a type which can be used to match an SDF construct of the same type. This means that a pattern of type `Symbol` can be used to match an SDF symbol. Since each type is a direct representation of an SDF construct, type-patterns can only match such a construct *literally*.

GTL incorporates patterns and pattern operators for matching for instance 'exactly one' or a sequence of SDF constructs. These facilitate matching multiply occurring- or unidentified- constructs and allow the programmer to create a pattern for matching a subset of a type. For instance 'all `Productions` defining a certain `Sort`', where the exact definition is not specified. See section 3.3.3, p.39 for a GTL notation of this pattern.

Pattern matching is *recursive*. This means that if any subpattern of a pattern fails, the entire pattern fails. For instance, the type `Symbols` is part of the definition of a `Production`, this means that if a symbol does not match, the pattern match of the encompassing `Production` will also not match. For a sequence pattern this means that if a pattern in the sequence does not match, the sequence does not match.

First, the pattern operators are explained. Then two patterns are introduced that are taken directly from the FST notation, namely the '*exactly one*' pattern and the '*zero or more*' pattern.

3.3.1 Generic Pattern Operators

Most of the GTL *Pattern Operators* have the same semantics and notation as some of the operators for *symbols* in SDF2. There are the *Sequence*-, the *Alternative*-, the *Option*- and the *Repetition*- operators [6, section 2.5].

The operators described here are *generic* in the sense that they operate on GTL patterns of any type using the same semantics.

The Sequence Operator

The *Sequence Operator* in SDF [6, sec. 2.5.2] is defined as a grouping of symbols, delimited by '`(`' and '`)`'. This is treated as syntax by GTL patterns of type `Symbol`, to allow literal matching.

To be able to match groupings of SDF constructs of any type, GTL implements the *Sequence* operator expressed by parenthesis enclosed in angle brackets containing a semicolon delimited list of patterns:

`<(`⟨*pattern*⟩`;` ⟨*pattern*⟩`; ... )>`

All patterns contained in the GTL sequence must have the same type. This is also the type of the sequence itself. Each successive pattern in a pattern sequence must be matched in the same order to a list of constructs for the sequence to match.

The following pattern can match a list of exactly two SDF *productions*, using patterns '`"GTL" -> GTLName`' and '`"SDF" -> SDFName`':

GTL pattern:

```
1 <( "GTL" -> GTLName ;
2    "SDF" -> SDFName )>
```

matches

SDF sequence:

```
1 "GTL" -> GTLName
2 "SDF" -> SDFName
```

but *rejects*

```
1 "GTL" -> GTLName
2 "too" -> Bad
3 "SDF" -> SDFName
```

The Set Operator

In SDF, a choice between two symbols at a single position can be expressed either by defining two productions containing either choice, or by using the *alternative* operator ('`|`') [6, sec. 2.5.4]. For instance '`true|false`' represents that either a '`true`' symbol or a '`false`' symbol may occur at a single position.

To be able to match a single SDF construct agains *any of* a list of patterns, GTL incorporates the *Set* operator expressed by braces enclosed in angle brackets containing a comma-delimited list of patterns:

`<{`⟨*pattern*⟩`,` ⟨*pattern*⟩`, ... }>`

A given SDF construct is tried against every pattern in the list. This implies that all the patterns in the set are required to be of the same GTL type, namely the same type as the SDF construct the pattern set is matched against. If it matches at least one of the patterns in the set, the match is successful, if it matches none of the patterns, the match fails.

The following pattern can match a single production that matches either pattern '`"GTL" -> GTLName`' or '`"SDF" -> SDFName`':

GTL pattern:

```
1 <{ "GTL" -> GTLName ,
2    "SDF" -> SDFName }>
```

matches

SDF construct (either of):

- `"GTL" -> GTLName`
- `"SDF" -> SDFName`

The Option- and Repetition-Operators

In SDF, the *Option* operator '`?`' [6, sec. 2.5.1] describes an optional part in a syntax rule. For instance, '`GTL?`' defines *no-* or *exactly one-* occurrence of the sort '`GTL`'. The SDF *Repetition* operators '`*`' (zero or more) and '`+`' (one or more) [6, sec. 2.5.3] express that a symbol should occur several times. For example the SDF symbol '`GTL*`' expresses that either symbols '', '`GTL`' or '`GTL GTL ...`' can occur at that position. Repetition operators allow for the creation of flat lists. They are sometimes referred to as *lists* because of this.

These operators and their semantics are copied in GTL as GTL pattern operators, with the restriction that they can only be used with GTL sequence- or set-operators. This is mainly to distinguish them from the syntactical representation of their SDF counterparts for *symbols*, otherwise it would not be possible to know whether the pattern '`GTL*`' refers to an SDF symbol '`GTL*`' literally, or a list of zero or more SDF symbols '`GTL`'.

An example of matching the same pattern one or more times:

GTL pattern:

```
1 <( "GTL"* )>+
```

matches

SDF constructs (either of):

- `"GTL"*`
- `"GTL"* "GTL"*`
- `"GTL"* "GTL"* "GTL"* ...`

...

3.3.2 The 'exactly one' Pattern

The GTL pattern '`.`' can be used to match *exactly one* undefined SDF construct of a given type. GTL maintains the semantics as defined for this pattern in the language FST [5].

Whenever an SDF construct of any type is to be matched using a GTL pattern, we can also use '`.`' to signify that it doesn't matter what the construct defines, but that *exactly one* construct of that type *must* be matched.

The SDF production:

SDF

```
1 "This" "is" An -> Example
```

can either be matched *directly*, using the pattern:

GTL

```
1 "This" "is" An -> Example
```

or *indirectly* using the pattern:

GTL

```
1 .
```

Both patterns are of type `Production`.

The '`.`' pattern can also be used as a pattern of type `Symbol`, part of the `Production` pattern in the example, to match *exactly one* `Symbol`:

GTL

```
1 "This" . An -> Example
```

This pattern will match all SDF productions defining the sort '`Example`' to represent a sequence of three symbols starting with '`This`' and ending with '`An`', with *exactly one* unspecified symbol in between. This pattern will match the original production by matching '`"is"`' with '`.`'. It will also match in the following cases, where the emphasized parts of the SDF productions are the constructs that are matched with the pattern '`.`':

GTL pattern:

```
1 "This" . An -> Example
```

matches

SDF constructs (either of):

- `"This" "was" An -> Example`
- `"This" "will be" An -> Example`
- `"This" MightBe An -> Example`

...

3.3.3 The 'zero or more' Pattern

The pattern '`...`' is copied from the language FST and is maintained for legibility. It works much the same way as the '`.`' pattern, except that it can match *zero or more* SDF constructs of a given GTL type. It is semantically identical to the GTL pattern '`<(.)>*`', which represents "any construct of a certain type ('`.`'), repeated zero or more times".

An example using the '`...`' pattern as part of a production pattern:

GTL pattern:

```
1 ... -> Prod
```

matches

SDF construct (either of):

```
• -> Prod
• "This" Is A "Match" -> Prod
• So "is this" -> Prod
 ...
```

3.3.4 Type-Specific Pattern Operators

Type-Specific Pattern Operators are pattern operators that provide matching capabilities specific to a certain type. For instance the *'any repetition'* operator taken from the FST language is a `Symbol` type-specific pattern operator for matching the SDF option- and repetition-operators part of a symbol. It cannot be used with any other type.

We introduce the *Lexical* operator which provides lexical construction for the types `Sort` and `Literal`.

The 'any repetition' Operator

To be able to match an SDF symbol with an unknown repetition operator, GTL provides the *any repetition* operator '`_`'. It is used solely with `Symbol` patterns in the same way as the option- or repetition- operators it can match. It also matches if neither option- nor repetition- operator is part of the symbol. GTL maintains the semantics as defined for this operator in the language FST [5].

It is provided so that the programmer can match symbols regardless of option- or repetition- operators. The following example shows a literal match, which rejects the presence of a repetition operator:

GTL pattern:

```
1 Try "this" -> Match
```

matches

SDF production:

```
Try "this" -> Match
```

but *rejects*

```
Try "this"+ -> Match
```

Using the 'any repetition' operator, we can match the following productions:

GTL pattern:

```
1 Try "this"_ -> Match
```

matches

SDF production (any of):

```
• Try "this" -> Match
• Try "this"? -> Match
• Try "this"* -> Match
• Try "this"+ -> Match
```

The Lexical Operator

The *Lexical* operator provides lexical -matching and -construction for the lexical types `Sort`, `Literal` and `ModuleId`. This allows the programmer to automate the transformations that were implied by the case study. In step 2.4.3, p.19 non-terminal sorts had to be renamed from dash separated to CamelCase, and in step 2.5.3, p.32, all sorts ending in '`Expr`' had to be renamed to end in '`Expression`'.

The lexical operator is expressed by square brackets enclosed in angle brackets containing a sequence of lexical patterns:

`<[`⟨*lexical pattern*⟩ ⟨*lexical pattern*⟩ ... `]>`

A lexical pattern can be a quoted literal, a character class or a construct of the same type that the lexical operator represents in its context. As stated this can be either a `Sort`, a `Literal` or a `ModuleId`.

A lexical pattern can also be any GTL construct that has a lexical operator as a pattern, or one of the lexical constructs as a type, meaning it can be used recursively. This allows for the use of GTL -patterns, -variables and -functions within a lexical operator.

To allow for lexical matching or construction, the same constructs that are used in lexical syntax productions in an SDF2 specification are also allowed within the GTL lexical operator. These are *character classes* and *quoted literals*. Character classes can only be used for matching, not for construction. Quoted literals can be used both for matching and construction.

The following patterns of type `Sort` can *match* the sort '`MyExample`':

GTL
```
1 • <[ "My" "Example" ]>
2 • <[ "My" Example ]>  %% where 'Example' is an SDF sortname
3 • <[ "My" . ]>
4 • <[ "My" "E" "x" "a" "m" "p" "l" "e" ]>
5 • <[ "My" "E" [a-z]+ "e" ]>
6 • <[ [A-Za-z]+ ]>
```

When used in assignment, the lexical operator can not contain any GTL patterns other than initialized variables, functions or literals, and of course SDF lexical constructs depending on the type of the assignment.

Of the previous example, only the following can be used to *construct* a new Sort '`MyExample`':

GTL
```
1 • <[ "My" "Example" ]>
2 • <[ "My" Example ]>  %% where 'Example' is an SDF sortname
3 • <[ "My" "E" "x" "a" "m" "p" "l" "e" ]>
```

3.3.5 Typed Patterns

The GTL patterns introduced in section 3.3.1, p.36 are type-less. They can be used with an SDF construct of *any* type. For most patterns, the type of the pattern can be ascertained from the pattern itself. For instance, the pattern:

GTL
```
1 sorts ...
```

is obviously a pattern of type `Grammar`, in this case a sorts grammar. In other cases the type of the pattern can be discovered by looking at previous declarations in the program (see section 3.4.1, p.43, section 3.7.1, p.51, ...).

Sometimes however, the type of a pattern needs to be made explicit. For instance for the GTL functions that can take type-less arguments, like '`select`' (see section 3.9.3, p.64), or '`extract`' (see section 3.9.3, p.64). These functions can select or extract any SDF construct type from an SDF specification, so for some patterns, like the 'exactly one' pattern, it needs to be made explicit which type is intended.

Assigning a type to a pattern can be done by adding a type specification to the pattern as follows:

⟨*type*⟩ : ⟨*pattern*⟩

The following is an example of using a typed pattern to match a single '`Literal`':

GTL pattern:

```
1 Literal: .
```

matches

SDF construct (either of):

- `"one"`
- `"two"`
- `"three"`

3.3.6 Complex Pattern Examples

The GTL pattern operators offer a versatile way of matching SDF constructs. A number of relatively complex patterns and their posssible matches are presented here.

The following example is a pattern to match a sequence of symbols, in this case literals. The pattern states that the SDF sequence has to begin with '`"one"`', followed by zero or more times either '`"two"`' or '`"three"`'.

GTL pattern:

```
1 <( "one"; <{ "two", "three" }>* )>
```

matches

SDF construct (either of):

- `"one"`
- `"one" "two"`
- `"one" "three"`
- `"one" "three" "two"`

...

The following example is a slightly more confusing pattern stating that the SDF sequence of symbols can start with '`"one"`' followed by '`"two"`', or start directly with '`"two"`'. This then has to be followed by zero or more '`Three`'.

GTL pattern:

```
1 <( <("one")>?; "two"?; <(Three)>* )>
```

matches

SDF construct (either of):

```
• "two"?
• "one" "two"?
• "two"? Three Three
• "one" "two"? Three Three
 ...
```

Using the '`.`' pattern we can match any symbol. Notice the recurrence of '`"two"`' between matching sequences in the following example:

GTL pattern:

```
1 <( <( . )>?; "two"; <( . . )>? )>
```

matches

SDF construct (either of):

```
• "two"
• Geit "two"
• "two" "Kees" "Poes"
• Teun "two" "Geit" Three
 ...
```

The following is a pattern for an SDF lexical syntax grammar. Note that this pattern will *only* match lexical syntax grammars that only contain productions defining the sort '`Prod`'. If the grammar contains other productions the match will fail.

GTL pattern:

```
1 lexical syntax
2   <(... -> Prod)>+
```

matches

SDF construct (either of):

```
• lexical syntax
    -> Prod
• lexical syntax
    "Zomaar" "iets" -> Prod
    Whatever "you" Want -> Prod
 ...
```

but *rejects*

```
1 • lexical syntax
2     "This one" Is OK -> Prod
3     "This" One "is" NOT -> TooBad
4  ...
```

3.4 Variables

Variables are special GTL patterns for storing and retrieving SDF constructs. As patterns, they provide a way to extract information from a given SDF specification or construct by storing the construct(s) that are matched. The stored constructs can then be used in another match, or as input to subsequent GTL functions or operators, or as output of the GTL program or function being written.

Variables are declared in the '`variables`' section of a GTL program.

3.4.1 Variable Declarations

Variable Declarations are part of a GTL '`variables`' section (see section 3.12.2, p.72). They must provide a *name*, a *type* and a *pattern* for the type. They can only be declared once, and have the following layout:

⟨*type*⟩ ⟨*name*⟩ [`:`⟨*pattern*⟩] [`:=` ⟨*SDF construct*⟩]`;`

Note that the pattern specification is also optional. If the pattern specification is not part of the declaration then the 'exactly one' pattern ('`.`') is assumed to be the pattern for the variable.

The following is an example of a simple declaration of a variable:

GTL
```
1 Grammar MyGrammar: . ;
```

'`MyGrammar`' is the variable name, its type is `Grammar`, and its pattern is '`.`', conforming to the pattern of its type.

If we leave out the pattern specification, the 'exactly one' pattern ('`.`') is assumed to be the pattern for the variable. This means that the following declaration declares the same variable as the previous:

GTL
```
1 Grammar MyGrammar;
```

A variable can also be initialized in its declaration, using the matching operator '`:=`' (see section 3.9.3, p.63). A variable initialization part of a declaration may not contain references to itself.

Initializing the variable using a declaration with an assignment match:

GTL
```
1 Grammar MyGrammar:. := context-free syntax
2                          "MyLiteral" -> MySort;
```

The variable declaration information is the same as in the previous example. An SDF context-free syntax `Grammar`, conforming to the pattern '`.`' of the variable, is assigned as content.

3.4.2 The Variable Pattern

A *Variable* pattern allows the programmer to either add content to a variable storage in the environment through matching, or use the existing content of a variable as a pattern to match against.

A variable pattern consists of a *name* enclosed in *angle brackets* or *chevrons* as follows:

`<`⟨*name*⟩`>`

Non-empty variables are said to be *initialized*. If a variable that is used as a pattern is uninitialized, it is assigned content by matching its pattern to the SDF construct at the position of the reference. If the variable is already initialized, then its content is used as if it were the pattern at the current position in the parent pattern, and matching is resumed.

An SDF construct mixed with initialized variables can be fully reduced to a "pure" SDF construct by replacing the initialized variables with their respective content. Such a mixed SDF construct is called an *extended SDF construct*.

Example

The following example shows how variables can be used to match two SDF constructs that have to be identical. Let's match the following SDF production:

SDF
```
1 Three Three -> Example
```

against the GTL pattern:

GTL
```
1 <MyVar> <MyVar> -> Example
```

where '`<MyVar>`' represents a variable named '`MyVar`' of type `Symbol` and pattern '`.`'. How to declare such a variable will be explained in section 3.12.2, p.72.

The match is made from left to right. Since the variable is uninitialized, its pattern '`.`' is used to match the first SDF construct, '`Three`'. The match is successful and the SDF construct '`Three`' is stored with the variable information in the GTL environment.

The second pattern is the same as the first, however, the variable was just matched and initialized. Since it already has content, namely the '`Three`' construct from the first match, this content is used as a pattern, meaning the second SDF construct is now required to match the GTL pattern '`Three`'. This is also successful.

The remainder of the GTL pattern and SDF production are identical ('`-> Example`'), and so they match. The entire pattern has now been successfully matched with the SDF production.

Some more matches using this pattern:

GTL pattern:
```
1 <MyVar> <MyVar> -> Example
```

The variable '`MyVar`' is of type `Symbol` and has pattern '`.`'.

	SDF construct (either of):	*post-match variable content:* *MyVar*
matches	• "any" "any" -> Example • Geit Geit -> Example ...	• "any" • Geit ...

3.4.3 The Reset Variable Pattern

Since initialized variables cannot be assigned new content, appending content to a variable presents a problem. To this end the *Reset Variable* pattern is introduced. This pattern clears the content of the variable after matching but before assignment. This implies that a variable can be assigned its own content, which is nessesary if the programmer wants to append to it. This will be explained further in section 3.9.3, p.63.

A reset variable pattern consists of a *name* enclosed in *angle brackets* or *chevrons* as follows:

```
>⟨name⟩<
```

When the reset variable pattern is used for matching, the *pattern* of the *variable* is always used, never its content. After the pattern has matched the given SDF construct, the variable is cleared and subsequently assigned the matched part of the construct. This order is nessesary to allow the already initialized variable to be used in the extended SDF construct which is matched against. Note that a variable reset pattern can not be part of an extended SDF construct.

In the following example, assume the variable '`MyVar`' has been declared and initialized with type `Symbol`, pattern '`...`', and content '`"one"`':

GTL pattern:

1 `>MyVar<`

The variable '`MyVar`' is of type `Symbol` and has pattern '`...`', content '`"one"`'

SDF construct (either of):

matches

- `"two"`
- `<MyVar> "two"`

post-match variable content:

MyVar

- `"two"`
- `"one" "two"`

In the first example the reset variable pattern is matched against the symbol '`"two"`'. Since a variable reset pattern only uses its *pattern*, which is '`...`', for matching, the variable is successfully matched and then cleared, after which it is reinitialized with the matched symbol '`"two"`'.

The second example is more complex. Again, the pattern of the variable ('`...`') is used for matching. In matching, the variable reference '`MyVar`' in the extended SDF construct was expanded to its content '`"one"`'. The pattern '`...`' of the variable matches both symbols '`"one"`' and '`"two"`' of the expanded SDF construct. The variable content is then cleared and will be assigned the matched constructs, namely '`"one" "two"`'.

3.4.4 Variables within a Set-Pattern

As stated in section 3.3.1, p.37, a GTL set-pattern will match a given SDF construct if one of its child patterns does. These child patterns can be overlapping making other matches inconsequential. This is not a problem, since only one of the child patterns has to match. If the set contains variables however, the situation changes, since a matching SDF construct is no longer discarded, but stored.

If a variable in the set is the only pattern to match a given SDF construct, it will be handled correctly. In the following example the variable '`MyVar`' is of type `Symbol` and has pattern '`"one"`'.

GTL pattern:

```
1 <{ <MyVar>, "two" }>
```

The variable '`MyVar`' is of type `Symbol` and has pattern '`"one"`'.

	SDF construct (either of):	*post-match variable content:* `MyVar`
matches	• `"one"` • `"two"`	• `"one"` • ⟨*empty*⟩

The construct '`"one"`' can only be matched by the variable, and will be stored accordingly. The construct '`"two"`' can only be matched directly by the pattern '`"two"`', implying that the variable is never matched and therefore will not be given content; it will remain empty.

If however the variable '`MyVar`' in the set were to be asigned the pattern '`"two"`', and the set is subsequently matched against the SDF construct '`"two"`', it is unclear which of the two child patterns in the set will be used to match the construct, since both can match it.

The following pattern is such a set. This example shows behaviour not part of the GTL pattern semantics, it is provided merely to explain the issue.

GTL pattern:

```
1 <{ <MyVar>, "two" }>
```

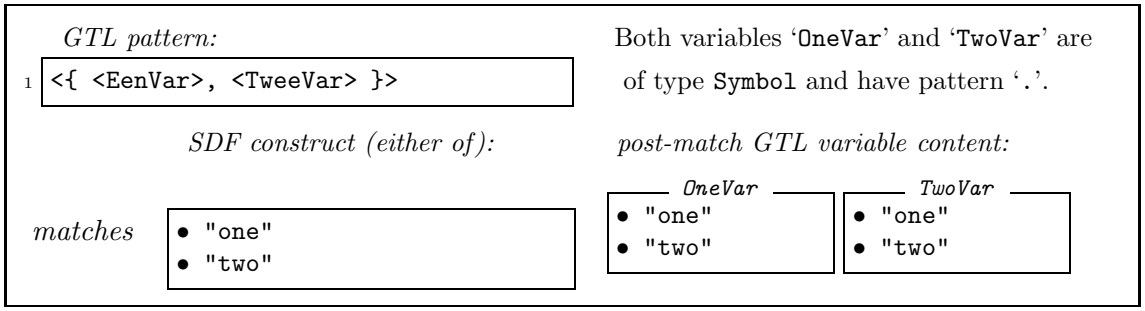
The variable '`MyVar`' is of type `Symbol` and has pattern '`"two"`'.

	SDF construct (either of):	*post-match variable content:* `MyVar`
matches	• `"two"`	• `"two"` or ⟨*empty*⟩

The variable has pattern '`"two"`', which is the same pattern as the second child in the set. When this set is matched against the SDF construct '`"two"`', both child patterns can match it. Which one of the two patterns in the set is used decides whether the variable is initialized or not.

To correct this non-deterministic behaviour, all variable references within a set-pattern will always be matched against the given SDF constructs by default. This means that regardless of which child pattern was used to match the entire set-pattern, all the variables referenced in the set and matching the SDF construct will be initialized.

For example, the following set-pattern contains two variables with different names but identical type and pattern:



Any SDF construct of type `Symbol` will be matched by both variables. Both will be initialized as per the semantics just described.

3.4.5 Complex Variable Declarations

The SDF constructs used in an assignment match are extended SDF constructs, meaning that they can contain initialized variable references. For instance:

GTL
```
1 Grammar MyGrammar:. := context-free syntax
2                          "MyLiteral" -> MySort;
3 Module MyModule:. := module MyModule
4                      exports
5                      sorts MySort
6                      <MyGrammar>;
```

defines two variables, '`MyGrammar`' and '`MyModule`'. '`MyModule`' is initialized using an SDF construct including a reference to the '`MyGrammar`' variable. After matching, '`MyModule`' is initialized as follows:

SDF
```
1 module MyModule
2 exports
3 sorts MySort
4 context-free syntax
5   "MyLiteral" -> MySort
```

The order of the declarations is not important, so long as any referenced variables are declared and initialized in the same scope (see section 3.14, p.76).

Creating a small SDF definition by adding a variable declaration:

GTL
```
1 Grammar MyGrammar:. := context-free syntax "MyLiteral" -> MySort;
2 Module MyModule:. := module MyModule exports sorts MySort <MyGrammar>;
3 Definition MySpec:. := definition <MyModule>;
```

After assignment the '`MySpec`' variable will contain the fully expanded definition:

SDF
```
definition
module MyModule
exports
sorts MySort
context-free syntax
  "MyLiteral" -> MySort
```

3.5 Arguments

User-defined patterns, functions or programs can all take *Arguments* which can be used to provide additional imput. This input is subsequently available as an initialised variable. An *Argument Declaration* is nothing more than an uninitialized variable declaration. When the user-defined pattern, function or program is invoked, the initialization of the argument variables must be provided with the invocation.

Argument declarations are part of pattern-, function- or program- declarations, which will be described in section 3.6.2, p.49, section 3.7.1, p.51 and section 3.13.1, p.74 respectively. They do not appear anywhere else.

An argument declaration has the following layout:

⟨*type*⟩ ⟨*name*⟩ [:⟨*pattern*⟩]

Note that the pattern specification is optional. If the pattern specification is not part of the declaration then the 'exactly one' pattern ('`.`') is assumed to be the pattern for the argument.

An example declaration of an argument '`MyArg`' of type `Production` and pattern '`... -> Boek`':

GTL
```
Production MyArg:... -> Boek
```

This means that the argument defined by this declaration must be a production defining the sort '`Boek`'. In the subsequent user-defined pattern, function or program, this argument will be available as an initialized variable with the name '`MyArg`'.

3.6 User-Defined Patterns

In GTL, the programmer can define new patterns that can then be used for matching. This allows the programmer to keep the code readable by using descriptive names for the defined patterns. It also makes the code more maintainable, since changing a user defined pattern is easier than searching and replacing all occurences of a pattern or sequence of patterns.

3.6.1 Pattern Declarations

Pattern Declarations are made in a GTL '`patterns`' section (see section 3.12.1, p.72). A pattern declaration has to provide a *name*, *type* and *pattern*. The name can be interpreted as an alias for the provided pattern, and *must* start with a '`#`' to identify it as a *user-defined* pattern. The type reference is used to force the subsequent pattern to match only SDF constructs of the given type. This means the pattern given in the declaration has to adhere to the type-pattern for the declaration to be type-correct.

A pattern declaration has the following layout:

⟨*type*⟩ ⟨*name*⟩ = ⟨*pattern*⟩

A simple pattern declaration looks like this:

GTL
```
1 Grammar #all_aliases = aliases ...
```

where '`#all_aliases`' is the pattern name, its type is `Grammar` and the pattern it represents is '`aliases ...`', which is of type `Grammar` as required.

This pattern will match at least in the following cases:

GTL pattern:

```
1 #all_aliases
```

matches

SDF construct (either of):

```
• aliases
• aliases
    "Any" -> OldAlias
    "My" -> Favourite
• aliases
    ...
```

User defined patterns are not allowed to reference themselves, but they can be used in the declaration of other patterns. An example of this is the definition of the '`#all_grammars` pattern in figure 3.1, p.50.

GTL
```
1 Grammar #all_grammars = <{ #all_aliases, #all_restrictions, #all_priorities,
2                            #all_sorts, #all_imports, #all_syntax }>
```

Note that the requirement that all child patterns in the set are of type `Grammar` (see section 3.3.1, p.37) applies.

Also note that, as with variable declarations, the order of the declarations is not important. However, any user-defined pattern used in the declaration of a new pattern must be declared in the same scope (see section 3.14, p.76).

3.6.2 Pattern Declarations with Arguments

A GTL pattern declaration can define arguments. This allows for the creation of more dynamic user patterns, and is especially useful for gathering specific information.

Such a pattern declaration looks like this:

GTL
```
Grammar #all_aliases = aliases ... ;
Grammar #all_restrictions = <{ restrictions ..., lexical restrictions ...,
                               context-free restrictions ... }> ;
Grammar #all_priorities = <{ priorities ..., lexical priorities ...,
                             context-free priorities ... }> ;
Grammar #all_sorts = sorts ... ;
Grammar #all_imports = imports ... ;
Grammar #all_syntax = <{ syntax ..., lexical syntax ...,
                         context-free syntax ... }> ;
Grammar #all_grammars = <{ #all_aliases, #all_restrictions, #all_priorities,
                           #all_sorts, #all_imports, #all_syntax }> ;
Section #all_sections = <{ exports ..., hiddens ... }> ;
Module #all_modules = module ... ;
```

Figure 3.1: An example '`patterns`' section

⟨*type*⟩ ⟨*name*⟩`(` ⟨*argumentdeclarationlist*⟩ `)` `=` ⟨*pattern*⟩

The variables declared as arguments can then be used as part of the pattern.

In FST , the pattern '`definition of` ...' was defined to specifically focus on a production defining a given sort. This pattern can easily be created in GTL using the patterns that have been defined so far.

The following declaration defines the '`#definition_of`' user pattern:

GTL
```
Production #definition_of(Sort S: . ) = ... -> <S> ;
```

The variable declaration '`Sort S:.`', defined as an argument of the user pattern, declares a variable named '`S`', with type `Sort` and pattern '`.`'. The pattern defined by this declaration is '`... -> <S>`', which matches any production defining a sort stored in the variable '`S`'.

The following pattern will match any SDF context-free syntax grammar containing at least one production defining the sort '`Three`':

GTL pattern:

```
context-free syntax
  <( ...;
     #definition_of(Three);
     ...; )>
```

matches

SDF construct (either of):

- `context-free syntax`
  `"one" "two" -> Three`
- `context-free syntax`
  `"Any" Old -> Prod`
  `"one" "two" -> Three`
  `"My" Favourite -> Prod`

3.7 Functions

GTL implements user defined *Functions*, providing an easy way for the programmer to implement a series of transformations that can be used more than once. In the case study

the need for this functionality was evident in step 2.4.2, p.18 and subsequent transformations in step 2.4.6, p.21 and step 2.4.11, p.27. The hypothetical declarations of a function in step 2.4.2, p.18 and step 2.4.11, p.27 also showed that we need to be able to pass arguments to functions.

Functions in GTL can return an SDF construct as the result value of a function-call. This will enable them to be used as patterns with the same semantics as an initialized variable reference. That means the output of a function can be used for matching or assignment.

3.7.1 Function Declarations

Functions are declared in the '`functions`' section (see section 3.12.4, p.73). A *Function Declaration* must provide a name, a return -type and -pattern, followed by a comma-delimited list of zero or more argument declarations, and finally the function body. *Function names* must begin with an ampersand ('`&`'). The *function body* consists of zero or more GTL sections declaring the variables, patterns, imports and (sub-) functions that are part of the body. The GTL sections are followed by a begin ... end compound statement enclosing a list of semi-colon delimited statements to be evaluated when the function is called.

A function declaration has the following layout:

```
⟨type⟩ ⟨name⟩( ⟨argumentdeclarationlist⟩ ) [:⟨pattern⟩] {
  ⟨sections⟩
  begin
    ⟨statement⟩;
    ⟨statement⟩;
     ...
  end;
}
```

Note that the pattern specification is optional. If the pattern specification is not part of the declaration then the 'exactly one' pattern ('`.`') is assumed to be the pattern for the function.

In the following example we use a *matching operator* as a statement. The operator was already introduced as part of variable declarations with an assignment (see section 3.4.1, p.43).

To keep it simple, no sections are included, they will we discussed in section 3.12, p.72.

GTL

```
1 Production &MyFunction(Symbol A1:.):... -> Boek {
2   begin
3     <MyFunction> := <A1> -> Boek;
4   end;
5 }
```

This declaration defines a function called '`MyFunction`' with a return type `Production` and pattern '`... -> Boek`'. This means that any production returned by this function must match that pattern.

The declaration defines one argument to the function, namely '`A1`' which is of type `Symbol` and has pattern '`.`', meaning that any single symbol will do. Before the statements are evaluated, the argument '`A1`' is declared as a variable and initialized with the value passed as an argument at invocation.

GTL has no '`return`' statement as in C or Java. In GTL the function name, in this case '`MyFunction`', is declared as a variable whose content after evaluation of the statements will represent the return value of the function. This implies that the return value of a function can only be an SDF construct.

The begin ... end compound statement defines a single assignment match, where the return variable '`MyFunction`' is assigned a `production` '`<A1> -> Boek`'. The variable '`<A1>`' was initialized by the argument passed to the function, it will represent the definition of the sort '`Boek`'.

If we were to call the function with the `Symbol` '`"Dit"`' as an argument, the argument variable '`<A1>`' would be initialized to that value. The return variable '`<MyFunction>`' would be assigned a production '`"Dit" -> Boek`', which represents the result of the function. That production could then be used in another statement.

3.7.2 The Function Pattern

A *Function Pattern* consists of the name of the function followed by a comma-delimited list of arguments which will be passed to it.

⟨*name*⟩`(` ⟨*argumentdeclarationlist*⟩ `)`

In the previous section an example function declaration for the function '`MyFunction`' was given. We will now create a pattern for that function.

The variable declaration:

GTL
```
1 Production P:. := &MyFunction( "Dit" )
```

will initialize the variable '`P`' of type `Production` and pattern '`.`' to the production '`"Dit" -> Boek`', which is the return value of the invocation of the function '`MyFunction`', as explained in the previous section (section 3.7.1, p.51).

Since the result of a function is guaranteed to be an SDF construct, it can be used as a pattern to be matched against as well as a value to be assigned.

As such, the following variable declaration is also correct:

GTL
```
1 Production P:&MyFunction( "Dit" )
```

The variable '`P`' of type `Production` is declared with the result of the function call '`&MyFunction( "Dit" )`', namely the production '`"Dit" -> Boek`', as a pattern. The variable is not initialized. Note that the return type of the function must be the same as the type of the variable being declared.

The same variable is declared as if the declaration were as follows:

GTL
```
1 Production P:"Dit" -> Boek
```

3.8 Selection

In the previous sections we explained various GTL patterns and how they can be used to match specific SDF constructs.

To be able to work on portions of an SDF grammar construct, whilst maintaining the whole grammar, GTL encorporates the concept of a focus as introduced in FST, extending it to include *all* SDF types instead of only sorts and modules. In GTL, we speak of a *Selection*, which holds both the part of the SDF construct that is operated on, which is called the *Focus*, as well as the remaining part of the SDF construct, which is called the *Remainder*.

In GTL, the concept of a selection is largely hidden from the programmer, primarily to make a program more legible. In effect, all SDF constructs *produced* by operations in GTL are part of a selection. The content of variables, the SDF construct produced by a function-call and even the output of a GTL program is part of a selection.

A selection has no syntactic representation in GTL but can be manipulated using vairous operators. It will be represented as follows in the various examples and explanations in this thesis:

$\langle focus | remainder \rangle$

Most operators and expressions operate on the focus, but some imply changes to the remainder as well. The semantics of using a selection within an expression will be discussed per operator introduced in section 3.9, p.59 about expressions.

In this section, we will discuss the impact of the produced results being a selection. The semantics of pattern matching as has already been discussed in section 3.3, p.36 is revisited with selections in mind. We will also introduce a number of operators that specifically target a selection.

3.8.1 Producing a Selection

The semantics of the patterns in pattern matching as discussed so far have to be fine-tuned to account for their operating on- and producing- a selection. Their primary semantics are the same since they operate only on the focus, but it needs to be clear what actually happens to the remainder.

An SDF construct is said to be held "in focus" if the selection holds that construct in the focus. An SDF construct is said to be held "out of focus" if the selection holds that construct in the remainder.

Primitive Patterns

The examples given in section 3.3, p.36 on patterns deal only with success or failure of a match. However, when used as part of an operation the result of a match, namely the matched SDF construct, will likely be used. This result is a selection.

Matching with simple patterns was introduced in section 3.3, p.36, and deals with patterns without variable- or function- patterns. A selection that is produced after a

successful match using a simple pattern holds the matched SDF construct in focus. The remainder is empty because the pattern is either a full match or no match at all.In the latter case a result will not be produced.

Extending one of the examples given on page 38 to show the produced selection we get:

GTL pattern:

```
1 "This" . An -> Example
```

matches

SDF construct:

- `"This" "was" An -> Example`
- `"This" "will be" An -> Example`
- `"This" MightBe An -> Example`

...

resulting selection

- ⟨`"This" "was" An -> Example` | –⟩
- ⟨`"This" "will be" An -> Example` | –⟩
- ⟨`"This" MightBe An -> Example` | –⟩

...

Variable Patterns

Using an *initialized* variable reference in a pattern implies that the stored construct is used as a pattern. Since a variable stores a selection, only the construct in focus is used as a pattern, and the remainder is ignored. The same semantics apply to the initialized variable pattern being used as part of an extended SDF construct; it will be expanded to the construct held in focus. Apart from this, pattern matching using the initialized variable pattern is the same as explained in section 3.4.2, p.43.

Using an *uninitialized* variable reference in a pattern implies that the pattern of the variable is used for matching (see section 3.4.2, p.43), and that the matched construct will be stored. The matched construct will be stored in focus in the resulting selection.

The following is an example using an uninitialized variable pattern, note that the variable content is now represented by a selection:

GTL pattern:

```
1 . <MyVar> ... -> Example
```

The variable '`MyVar`' is of type `Symbol` and has pattern '`.`'.

matches

SDF construct:

```
"one" "two" "three" -> Example
```

post-match variable content:

MyVar

⟨`"two"` | –⟩

Function Patterns

A function pattern used within a pattern will be expanded to its result if the function-call is successful. This result is a selection. The semantics are the same as if an initialized variable had been used; only the focus will be used and the remainder will be discarded. Other than this, the semantics of the function pattern as explained in section 3.7.2, p.52 are the same.

3.8.2 Operations on Selections

Selections provide a way to focus on a specific part of an SDF grammar construct. Apart from creating a selection as discussed in the previous section, we also need to be able to "undo" the selection to retrieve the whole grammar. This is called *imploding* a selection. To do this we need to merge two SDF grammar contructs, namely the focus and the remainder of a selection. The specific semantics of merging two SDF grammar constructs is explained in section 3.16, p.77.

In the following sections a "plus" infix operator ('+') is used to signify a merge between two SDF grammar constructs. The semantics of merging two SDF constructs is explained in section 3.16, p.77.

Creating a Selection

As explained at the beginning of section 3.8, p.53, every SDF construct produced by an operation in GTL is in fact a selection. There is one operator however that is specifically intended to create a selection using a given pattern and SDF construct: the '`select`' operator. Its use is explained in section 3.9.3, p.64.

Imploding a Selection

As stated, a selection can be "undone" by imploding it. The result will be a new selection with the merged focus and remainder in focus.

It has the following semantics:

$$\langle\, focus_1 \,|\, remainder_1 \,\rangle \;\rightarrow\; \langle\, focus_1 + remainder_1 \,|\, - \,\rangle$$

Narrowing a Selection

Narrowing a selection means selecting a part of the construct in focus, and making that the new focus of the selection. The remainder, those constructs which were part of the original focus but are now part of the new remainder, will need to be merged to the original remainder that was already present in the selection. Narrowing a selection is basically moving some of the SDF constructs from the focus to the remainder, creating a selection with a smaller focus and a bigger remainder.

The semantics are as follows; we start with a selection:

$$\langle\, focus_1 \,|\, remainder_1 \,\rangle$$

We want to narrow the selection by selecting from the original as follows; note that '`select`' works only on the focus:

$$\texttt{select}\ \langle pattern \rangle\ \texttt{from}\ \langle\, focus_1 \,|\, remainder_1 \,\rangle \;\rightarrow\; \langle\, focus_2 \,|\, remainder_2 \,\rangle$$

If we implode new selection, the focus of the original will be the result. This means that the remainder of the original has disappeared, which is not what we want. We have to merge the original remainder with the remainder of the new selection to get to the following result:

$\langle\, focus_2 \,|\, remainder_1 + remainder_2 \,\rangle$

This represents the new narrowed selection.

Widening a Selection

Widening a selection works much in the same way as narrowing, only now we are selecting from the remainder, after which we merge the produced focus with the original focus. Narrowing a selection is basically moving some of the SDF constructs from the remainder to the focus, creating a selection with a bigger focus and a smaller remainder.

The semantics are as follows; we start with a selection:

$\langle\, focus_1 \,|\, remainder_1 \,\rangle$

We want to widen the selection by selecting from the original. The '`select`' operator works only on the focus, so to select from the remainder we use the '`remainder`' operator which will return the remainder of its argument in focus. The operation looks like this:

`select` $\langle pattern \rangle$ `from remainder(` $\langle\, focus_1 \,|\, remainder_1 \,\rangle$ `)` $\rightarrow$ $\langle\, focus_2 \,|\, remainder_2 \,\rangle$

If we implode new selection, the remainder of the original will be in focus of the resulting selection. This means that the focus of the original has disappeared, which is not what we want. We have to merge the original focus with the focus of the new selection to get to the following result:

$\langle\, focus_1 + focus_2 \,|\, remainder_2 \,\rangle$

This represents the new widened selection.

3.8.3 Selection Operators

Sometimes it can be useful to operate on the remainder of a selection instead of the focus, or to drop either the focus or remainder part of a selection. To facilitate this we introduce a number of selection operators.

A "plus" infix operator ('+') is used to signify a merge between two SDF grammar constructs. The semantics of merging two SDF constructs is explained in section 3.16, p.77.

The Focus Operator

The *focus* operator takes a selection as an argument. It returns a new selection containing the focus of that argument, without the remainder. It is depicted by the keyword '`focus`', and has the following layout:

`focus(` ⟨*selection*⟩ `)`

Its semantics explained:

`focus(` ⟨*focus*|*remainder*⟩ `)` → ⟨*focus*|– ⟩

The Remainder Operator

The *remainder* operator takes a selection as an argument. It returns a new selection containing the remainder of the argument in focus. It is depicted by the keyword '`remainder`', and has the following layout:

`remainder(` ⟨*selection*⟩ `)`

Its semantics explained:

`remainder(` ⟨*focus*|*remainder*⟩ `)` → ⟨*remainder*|`-` ⟩

The Invert Operator

The *invert* operator takes a selection as an argument. It returns a new selection containing the inverted selection of that argument. The focus of the argument is now the remainder and the remainder of the argument is now the focus. It is depicted by the keyword '`invert`', and has the following layout:

`invert(` ⟨*selection*⟩ `)`

Its semantics explained:

`invert(` ⟨*focus*|*remainder*⟩ `)` → ⟨*remainder*|*focus*⟩

The Implode Operator

The *implode* operator takes a minimum of one selection as an argument. It returns a new selection containing the merged focus and remainder of its arguments as a focus, with the remainder empty. It is depicted by the keyword '`implode`', and has the following layout:

`implode(` ⟨*selection*⟩`,` ⟨*selection*⟩`, ... )`

Its semantics explained for a single argument:

`implode(` ⟨*focus*|*remainder*⟩ `)` → ⟨*focus*+*remainder*|`-`⟩

If there is more than one argument all the foci and remainder of it arguments are merged together and produced as the new focus in the result.

Its semantics explained for a single argument:

`implode(` ⟨$focus_1$|$remainder_1$⟩`,` ⟨$focus_2$|$remainder_2$⟩`, ... )` →
⟨$focus_1$+$remainder_1$+$focus_2$+$remainder_2$+ ...|`-`⟩

The Merge Operator

The *merge* operator takes a minimum of two selections as arguments. It returns a new selection containing the merged foci of its arguments as the new focus and the merged remainders of its arguments as the new remainder. It is depicted by the keyword '`merge`', and has the following layout:

`merge(` ⟨*selection*⟩ `,` ⟨*selection*⟩`, ... )`

The selections offered as arguments do not have to be of the same type, however, not all combinations of types can be merged, so care should be taken in using this operator. The semantics of merging two SDF constructs is covered in section 3.16, p.77.

The merge of two selections explained:

`merge(` ⟨$focus_1$|$remainder_1$⟩ `,` ⟨$focus_2$|$remainder_2$⟩ `)` →
⟨$focus_1$+$focus_2$|$remainder_1$+$remainder_2$⟩

If there are more than two arguments, the resulting selection of the merge of the first two selections is merged with the third argument, after which the result is merged with the fourth argument and soforth until there are no more arguments left.

The Narrow Operator

The *narrow* operator takes a selection and a pattern as arguments. It returns a new selection containing the narrowed focus as the new focus and the merged remainders of the match and the original as the new remainder. It is depicted by the keyword '`narrow`', and has the following layout:

`narrow` ⟨*selection*⟩ `using` ⟨*pattern*⟩

The selection and the pattern do not have to be of the same type. This means that in some cases the pattern must be *typed* (see section 3.3.5, p.40) to avoid ambiguities.

It is implemented as:

```
merge(select ⟨pattern⟩ from focus(⟨selection⟩),
      invert( remainder(⟨selection⟩) ) );
```

The Widen Operator

The *widen* operator takes two selections as arguments. It returns a new selection containing the widend foci of its arguments as the new focus and the widend remainders of its arguments as the new remainder. It is depicted by the keyword '`widen`', and has the following layout:

```
widen ⟨selection⟩ using ⟨pattern⟩
```

The selection and the pattern do not have to be of the same type. This means that in some cases the pattern must be *typed* (see section 3.3.5, p.40) to avoid ambiguities.

it is implemented as:

```
merge(select ⟨pattern⟩ from remainder(⟨selection⟩), focus(⟨selection⟩));
```

3.9 Expressions

GTL *Expressions* are used for condition testing. An expression can either succeed or fail, this is called the *result condition*. The result condition can be influenced using the logical operators *And*, *Or* and *Not*.

Certain operators used in expressions can also return a selection, called the *result value*. For instance equivalence operators, which can be used to test for existance or equivalence of portions of a grammar, generate a result value if they are successful.

Expressions can be used in conditional statements (see section 3.10.3, p.68), as statements directly (see section 3.10.2, p.67), or in an assignment match statement (see section 3.9.3, p.63).

3.9.1 Logical Operators

GTL defines three *logical operators* to aid in testing for specific conditions. They are the *and*-, *or*- and *not*- operators. These operators result only in success or failure.

Any operator that produces a return value will have this value discarded if it is used in conjunction with a logical operator.

The and- and or- operators combine the failure- or success- result conditions of two expressions. The not operator changes the result condition of a single expression.

The And Operator

The *And* operator has the following layout:

```
⟨expression⟩ && ⟨expression⟩
```

Its evaluation succeeds if both expressions are successfully evaluated, and fails otherwise.

The Or Operator

The *Or* operator has the following layout:

⟨*expression*⟩ `||` ⟨*expression*⟩

Its evaluation succeeds if at least one of the expressions is successfully evaluated, and fails otherwise.

The Not Operator

The *Not* operator '`not`' has the following layout:

`not(`⟨*expression*⟩`)`

Its evaluation succeeds if the evaluation of its argument expression fails, and fails otherwise.

3.9.2 Equivalence Operators

The *Equivalence Operators* are a class of operators used for comparing SDF grammars.

The Equals Operator

The *Equals* operator is used to compare two SDF constructs. Its evaluation succeeds if the constructs are structurally equal, producing the SDF grammar construct in focus as a result value. The equals operator is *right-associative*, and is depicted by the symbol '`==`'. It has the following layout:

⟨*extended SDF construct*⟩ `==` ⟨*extended SDF construct*⟩

The constructs on either side of the operator are *extended SDF constructs*, meaning they are allowed to reference initialized variables, which will be expanded before comparison. They must be of the same type.

In the following uses of the equals operator, the '`MyVar`' variable is predeclared with type `Symbol` and pattern '`.`'. It is either initialized or uninitialized as depicted in the column on the right:

statement: *post-evaluation variable content:*

successful:

```
1 "one" == "one";
```

„ :

```
1  ...
2 "one" "two" "drie" -> Test ==
3   <MyVar> "two" "drie" -> Test;
```

MyVar

```
1 "one"
2 "one"
3 "one"
```

The Subgrammar Operators

The *Subgrammar* operators test whether an SDF grammar construct is structurally equivalent to- or a subgrammar of- the other. If its evaluation is successful, the *matched* part of the grammar encompassing the other is produced in focus and the remainder of that grammar is produced out of focus as a result value. The subgrammar operators are *left-associative*, and they are depicted by the symbols '`(=`' and '`=)`'.

The subgrammar operators test for structural equivalence *modulo renaming* (see section 3.15.2, p.77) of one grammar and a (sub)grammar. The SDF constructs do not have to be of the same type. They have the following layout:

⟨*SDF (sub)grammar construct*⟩ `(=` ⟨*SDF grammar construct*⟩

If the grammar on the left-hand side is structurally equal to-, or a subgrammar of-, the grammar on the right-hand side, the evaluation of this operator is successful. In that case, the subgrammar of the right-hand side that the left-hand side matches with is produced in focus of the resulting selection. The remainder of that grammar is produced out of focus of the resulting selection.

The other operator works in the same way, but with the left-hand side and right-hand side reversed.

⟨*SDF grammar construct*⟩ `=)` ⟨*SDF (sub)grammar construct*⟩

The constructs on either side of the operator are *extended SDF constructs*, meaning that they are allowed to reference initialized variables, which will be expanded before comparison.

The following example shows uses of the subgrammar operators; the '`MyVar`' variable is predeclared with type `Symbol` and pattern '`.`'. It is either initialized or uninitialized as depicted in the column on the right:

statement: *post-evaluation variable content:*

successful:

```
1 "one" (= "one";
2 "one" =) "one";
```

MyVar

```
1 "one"
2 "one"
3 "one"
4 "one"
```

" *:*

```
1  ...
2 "one" "two" "drie" -> Modulo
3 (=
4 <MyVar> "two" "drie" -> Renaming;
```

The following is a more complex example where two SDF context-free grammars are compared:

GTL statement *resulting selection*

```
1 context-free syntax
2   "one" Two -> Many
3   "two" -> Two
4 (=
5 context-free syntax
6   "three" -> Three
7   "one" Deux -> Beaucoup
8   "two" -> Deux
```

⟨ `context-free syntax` / `"one" Deux -> Beaucoup` / `"two" -> Deux` | `context-free syntax` / `"three" -> Three` ⟩

Using structural equivalence modulo renaming, the SDF grammar on the left-hand side of the operator is a subgrammar of the one on the right, when renaming the sorts '`Two`' to '`Deux`' and '`Many`' to '`Beaucoup`'.

The Covers Operator

The *Covers* operator is depicted by the symbol '`(=)`', and provides a way of testing whether two SDF grammars are *structurally equivalent modulo renaming* (see section 3.15.2, p.77).

It is not obvious which of the two arguments should be used as a result value, the one on the left, or the one on the right. They can differ in the sort names they employ, so it is not an arbitrary choice. Since both constructs are provided by the programmer as arguments to the operator, they are assumed to be reproducible by the programmer as well. Since this is the case, this operator does not produce a result value.

The operator has the following layout:

⟨*SDF grammar construct*⟩ `(=)` ⟨*SDF grammar construct*⟩

The constructs on either side of the operator are *extended SDF constructs*, meaning they are allowed to reference initialized variables, which will be expanded before comparison. The SDF constructs on either side must be of the same type.

The covers operator has identical semantics to the following GTL expression:

GTL

```
1 <C1> (= <C2>  &&  <C1> =) <C2>
```

where '`C1`' and '`C2`' are variables of the same type containing the SDF constructs to be compared.

An example of a successful use of the covers operator:

GTL

```
1 context-free syntax
2   AB -> S
3   "a" -> A
4   "b" -> B
5 (=)
6 context-free syntax
7   CD -> T
8   "a" -> C
9   "b" -> D
```

3.9.3 Matching Operators

These are the operators that require a pattern to be given as an argument. They use the pattern to match all or parts of an SDF construct to perform a function.

The Matching Operator

The *Matching* operator is depicted by the symbol '`:=`' and allows GTL patterns to be matched against SDF constructs. It is *right-associative*, and has the following layout:

⟨*pattern*⟩ `:=` ⟨*extended SDF construct*⟩

The extended SDF construct and the pattern must be of the same type. The left-hand side of the operator is a GTL pattern. It is used to match the extended SDF construct on the right-hand side. If the match is successful, any uninitialized variables that are part of the pattern are initialized with their matched SDF constructs (see section 3.4.2, p.43). The entire extended SDF construct which was matched against is produced as a result value. If the match fails, any uninitialized variables part of the pattern remain uninitialized, and no result value is produced.

The initialization of variables referenced in the pattern is a *side effect* which is not present in the other expression operators. In this, the use of an assignment operator resembles a statement, except that it can succeed or fail.

The most simple use of a matching operator is the initialization of a single variable. For example, assuming that a variable '`MyVar`' has been declared with type `Symbol` and pattern '`.`':

GTL statement:

```
1 ...
2 <MyVar> := "one";
3 ...
```

post-evaluation variable content:

MyVar

```
1 ⟨empty⟩
2 "one"
3 "one"
```

After evaluation the variable '`MyVar`' is initialized with value '`"one"`'.

The left-hand side of a matching operator can be any GTL pattern. Assume the same uninitialized variable '`MyVar`' as before:

GTL statement:

```
1 ...
2 <MyVar> . -> . := "one" "two" -> Many;
3 ...
```

post-evaluation variable content:

MyVar

```
1 ⟨empty⟩
2 "one"
3 "one"
```

Matching the `Production` pattern on the left-hand side to the SDF production on the right-hand side results in the variable '`MyVar`' having been initialized to hold '`"one"`'.

A pattern without variables can also be used to match against, implying that the match has no side-effects and simply acts as an expression to be used for testing:

statement:

successful:
```
1 ... -> . := "one" "two" -> Many;
```

unsuccessful:
```
1 "two" . -> . := "one" "two" -> Many;
```

The Select Operator

The *Select* operator allows a GTL pattern to be used to gather all matching SDF constructs that are part of a given grammar construct into the focus of a selection. All constructs not matching the pattern will be put in the remainder. It is depicted by the keyword '`select`' and a GTL pattern, followed by the keyword '`from`' and an extended SDF construct.

It has the following layout:

`select` ⟨*pattern*⟩ `from` ⟨*extended SDF construct*⟩

The GTL pattern is used to match each of the sub-constructs of the extended SDF construct. The selection and the pattern do not have to be of the same type. This means that in some cases the pattern must be *typed* (see section 3.3.5, p.40) to avoid ambiguities.

A select operator fails if no matches were found, producing no result value. If it is successful, it will produce a selection.

The following is an example of a selection from a context-free grammar:

GTL statement

```
1 select . -> .
2 from context-free syntax
3        "one" "two" -> Many
4        "one" -> Many
```

resulting selection

⟨ `context-free syntax` `"one" -> Many` | `context-free syntax` `"one" "two" -> Many` ⟩

The Extract Operator

The extract operator works just like the select operator, but differs in it's result value. Where the select operator returns a selection of the same type as the selection given as an argument, the extract operator returns a list of constructs of the same type as the pattern.

It has the following layout:

`extract` ⟨*pattern*⟩ `from` ⟨*extended SDF construct*⟩

The selection and the pattern do not have to be of the same type. This means that in some cases the pattern must be *typed* (see section 3.3.5, p.40) to avoid ambiguities.

The following is an example of an extraction from a context-free grammar, the resulting selection is of the same type as the given pattern:

GTL statement

```
1 extract . -> .
2 from context-free syntax
3        "one" "two" -> Many
4        "one" -> Many
```

resulting selection

⟨ `"one" -> Many` | `"one" "two" -> Many` ⟩

The Remove Operator

This operator will remove all SDF constructs matching the given pattern from the focus of the selection.

It has the following layout:

```
remove ⟨pattern⟩ from ⟨selection⟩
```

The semantics is the same as the following GTL statement:

```
merge(remainder(select ⟨pattern⟩ from ⟨selection⟩),
      invert(remainder(⟨selection⟩)));
```

A select is used to single out the intended constructs. This means that the selection and the pattern do not have to be of the same type. In some cases the pattern must be *typed* (see section 3.3.5, p.40) to avoid ambiguities.

The following is an example of the removal of a production from a context-free grammar:

GTL statement

```
1 remove . -> .
2 from context-free syntax
3        "one" "two" -> Many
4        "one" -> Many
```

resulting selection

```
⟨ context-free syntax
    "one" "two" -> Many | _ ⟩
```

The Insert Operator

This operator will insert the given SDF construct into the focus.

It has the following layout:

```
insert ⟨extended SDF construct⟩ into ⟨selection⟩
```

The semantics is the same as the following GTL statement:

```
merge(⟨extended SDF construct⟩, ⟨selection⟩);
```

A merge is used to insert the construct into the selection. This means that the selection and the extended SDF construct do not have to be of the same type (see section 3.16, p.77).

The following is an example of the insertion of a production into a context-free grammar:

GTL statement

```
1 insert "three" -> More
2 into context-free syntax
3        "one" "two" -> Many
4        "one" -> Many
```

resulting selection

```
⟨ context-free syntax
    "one" "two" -> Many
    "one" -> Many
    "three" -> More | _ ⟩
```

The Replace Operator

This operator will replace all SDF constructs in focus that match the given pattern. The given SDF construct will serve as a replacement. The given SDF construct can be a function call using the match of the pattern as an argument, so one can interpret this as "apply function ... to all ... in ⟨*selection*⟩".

`replace` ⟨*pattern*⟩ `with` ⟨*extended SDF construct*⟩ `in` ⟨*selection*⟩

The matched construct will be replaced at the position it was encountered. The pattern and the SDF construct must be of the same type. Note that an uninitialized variable used in the pattern is reset for each match made. This variable is initialized in a match and can be used as part of the extended SDF construct.

The following is an example of the replacement of a production in a context-free grammar:

GTL statement

```
1 replace "one" -> Many
2 with "three" -> More
3 in context-free syntax
4      "one" "two" -> Many
5      "one" -> Many
```

resulting selection

⟨ `context-free syntax` `"one" "two" -> Many` `"three" -> More` | `-` ⟩

3.9.4 Operator Precedence

The precedence between the operators is as follows:

'`(=)`', '`=)`', '`(=`' before '`==`', '`:=`' before '`&&`', '`||`'

A deviation from this can be forced by using a simple compound statement (see section 3.10.2, p.67).

3.10 Statements

Traditionally, statements do not return results and are executed solely for their side effects, like for example variable initialization. Since GTL is a language based mainly on pattern matching, which can fail if there is no match, traditional statements like the assignment statement can influence the program execution. This is why it was introduced as the matching operator to be used in expressions.

GTL allows expressions to be used as statements, so that an assignment using the matching operator does not always have to be enveloped by a conditional statement. This however implies that some statement types can fail, meaning that they are conditional by themselves.

The consequences of this will be explained per statement type in this section.

3.10.1 Expressions as Statements

GTL allows expressions to be used as statements directly. Any result value produced by such an expression is discarded, however, the result condition is not. This means that certain statements can fail and as such influence the program execution path.

The primary reason for allowing expressions to be used as statements are assignment statements. If an assignment were not an expression but a statement which can not fail, its correct function could not be guaranteed.

An assignment is simply a pattern match, since the assigned SDF construct has to adhere to the pattern of the variable it is assigned to. Such a match can fail, implying that none of the uninitialized variables referenced in the pattern, if any, will have been initialized. If the assignment cannot fail, then that would mean that any subsequent statement using these variables will not have the correct input. To be able to guarantee the correct initialization of variables, statements are allowed to fail, so that subsequent statements are never reached.

To catch a possible failure of an expression or statement, the programmer can use a conditional statement like the '`if`' statement. This way the programmer can provide statements to handle such a failure.

The following assignment fails, since the pattern of the variable does not match the SDF construct assigned to it. Assume the variable '`MyVar`' has been declared using type `Production` and pattern '`. -> Many`':

GTL

```
1 <MyVar> := "one" "two" -> Many
```

The pattern of the variable will only match productions defined by a single symbol, yet the SDF construct assigned to the variable is defined by two symbols.

3.10.2 Sequences

Multiple statements can be grouped together using a *Compound* statement. If any of the statements contained in such a group fails, the compound statement also fails, discarding any side effects achieved by the evaluation of the statements in the group.

There are two types of compound statement. One can only be used as a statement, and the other can only be used in program and function declarations.

The Simple Compound Statement

The simple compound statement is expressed by a semi-colon delimited list of statements enclosed in braces:

```
{
  ⟨statement⟩;
  ⟨statement⟩;
   ...
}
```

No new declarations can be made in this statement, it serves only as a grouping of statements. This statement is intended to be used in conditional statements, but can be used anywhere where a statement is allowed.

The simple compound statement can have both a result condition and a result value. The result condition is the same as if all the enclosed statements were evaluated as a series of expressions concatenated using the *and* operator (see section 3.9.1, p.59). If any one enclosed statement fails the entire compound statement fails.

If the result condition is "success", the result value as produced by the last statement in the group will be produced as the result value for the compound statement.

The following example shows two compound statements containing conditional statements which will succeed and fail respectively. Assume the variable '`MyVar`' has been declared using type `Symbol` and pattern '`.`':

GTL statement: *post-evaluation results:*

```
1 {
2   <MyVar> := "one";
3   <MyVar> == "one";
4 }
```

```
<MyVar>:
1 ⟨empty⟩
2 "one"
3 "one"
4 "one"
```

condition: success
value: "one"

```
1 {
2   <MyVar> := "one";
3   not( <MyVar> == "one" );
4 }
```

```
<MyVar>:
1 ⟨empty⟩
2 "one"
3 "one"
4 "one"
```

condition: failure
value: ⟨*none*⟩

The `begin ... end` Compound Statement

This compound statement is expressed by a semi-colon delimited list of statements enclosed in the keywords '`begin`' and '`end`':

```
begin
  ⟨statement⟩;
  ⟨statement⟩;
   ...
end
```

It may be used in program (see section 3.13.1, p.74) or function (see section 3.7.1, p.51) declarations, but nowhere else.

The begin ... end statement implies a new scope, including a possible result value. It is part of a program- or function- declaration, where the various GTL sections preceeding a begin ... end statement are used to populate the new scope with variables, patterns, etc.

3.10.3 Conditional Statements

A *Conditional* statement is used to influence program evaluation. It has no side effects in that it does not change the execution environment, only the execution path.

The If Statement

The GTL *if* statement is used for the conditional evaluation of statements. As a statement it cannot fail, and as such the evaluation of statements following the '`if`' statement will continue regardless of the results.

It has the following layout:

`if` ⟨*expression*⟩ `then` ⟨*statement*⟩ [`else` ⟨*statement*⟩]

When an '`if`' statement is encountered, the expression directly following the keyword '`if`' is evaluated as a condition. Only the result condition is taken into account, a possible result value is discarded. If the result condition after evaluation is "success", the statement following the keyword '`then`' is evaluated, otherwise the statement following the keyword '`else`' (if it is present) is evaluated.

Including an '`else`' part with the '`if`' statement is optional. If it is not present and the result condition after evaluation is "failure", nothing is done.

The following example shows an '`if`' statement whose result condition is "success":

GTL statement:

```
1 if "one" == "one" then
2   <MyVar> := "one";
3 else
4   <MyVar> := "two";
5 ...
```

post-evaluation variable content:

<MyVar>

```
1 ⟨empty⟩
2 "one"
3 [not evaluated]
4 [not evaluated]
5 "one"
```

The following example shows an '`if`' statement whose result condition is "failure":

GTL statement:

```
1 if "one" == "two" then
2   <MyVar> := "one";
3 else
4   <MyVar> := "two";
5 ...
```

post-evaluation variable content:

<MyVar>

```
1 ⟨empty⟩
2 [not evaluated]
3 ⟨empty⟩
4 "two"
5 "two"
```

The Case Statement

The *Case* statement can be used to match a given SDF construct to a set of patterns connected to statements. If one of the patterns is a match the statement connected to the matching pattern is evaluated. Its result -condition and -value (if any) will be produced as the result -condition and -value of the case statement.

A '`case`' statement starts with the keyword '`case`' followed by an extended SDF construct which will be used to match against. This is foloowed by the keyword '`of`' after which a comma-delimited list of patterns with statements must be given. The patterns must be of the same type as that of the extended SDF construct.

The '`case`' statement has the following layout:

```
case ⟨extended SDF construct⟩ of
  ⟨pattern⟩:⟨statement⟩,
  ⟨pattern⟩:⟨statement⟩,
   ...
  default:⟨statement⟩;
```

There is one additional match called the *default match*, it is depicted by the keyword '`default`', and it *must* be the last item in the set of patterns. If none of the supplied patterns match and a default match is present, it is evaluated.

An example of a '`case`' statement is part of the GTL function '`&toCamelCase`' which is presented in appendix **??**, p.**??**. The '`case`' statement in question is used to convert a lower case letter to upper case. The variable named '`S`' represents a character. It must be converted to uppercase, and assigned to another variable named '`ToUpper`'.

GTL

```
28        <ToUpper> := case <S>: { a:A; b:B; c:C; d:D; e:E; f:F;
29                                 g:G; h:H; i:I; j:J; k:K; l:L;
30                                 m:M; n:N; o:O; p:P; q:Q; r:R;
31                                 s:S; t:T; u:U; v:V; w:W; x:X;
32                                 y:Y; z:Z; default:<S> };
```

The '`case`' statement tries to match the content of the variable '`S`' to each of the patterns supplied. These patterns represent the entire lower case alphabet. If any of the patterns is a match, the connected statement, which represents the upper case version of the matched letter, is produced as a result value to the case statement. This result value is assigned to the variable '`ToUpper`' using a matching operator.

The default match is supplied to catch any other than lower case letters. The content of the variable '`S`' is then produced as a result value, meaning that any characters other than lower case letters are produced unchanged. This is useful if the character is already an upper case letter.

The Foreach Statement

The *Foreach* statement is used to evaluate a statement for every match made within a given SDF grammar construct. The statements starts with the keyword '`foreach`', followed by an uninitalized variable declaration. Next, the keyword '`in`' and an extended SDF construct must be supplied. The keyword '`do`' preceeds the statement to be evaluated if a match is made. The foreach statement has the following layout:

```
foreach ⟨variable declaration⟩ in ⟨SDF grammar construct⟩ do ⟨statement⟩
```

In the evaluation of the foreach statement the uninitialized variable will be used as a pattern. It is matched against *all* SDF constructs part of the supplied grammar construct. A successful match is used to initialize the variable, after which the statement is evaluated. When the statement has been evaluated, the next match is made and used to initialize

the variable. The statement is evaluated once more and so on, until the evaluation of the statement fails, or there are no more matches to be made.

If the evaluation of the statement fails, the foreach statement fails. It will produce neither result -condition nor -value.

In the following example of a foreach statement, assume that the variable '`Output`' has been declared uninitialized with type `QLiteral` and pattern '`...`'.

GTL
```
1 foreach QLiteral MyVar:<( . )> in <( "one" "two" -> Many )> do
2   >Output< := <Output> <MyVar>;
```

The foreach statement will evaluate its statement two times, once with the variable '`MyVar`' initialized to hold '`"one"`', and once initialized to hold '`"two"`' in that order. The matching statement that is evaluated per match appends the content of the variable '`MyVar`' to the content of the variable '`Output`'. When no more matches are found the foreach statement is finished, and the variable '`Output`' holds '`"one" "two"`' as content.

Altering the contents of a variable referenced in the '`in`' part of the statement during evaluation is allowed, but does not change the SDF construct initially passed to the statement. This means that all matches will be made using the original content of the variable, not the altered one. The foreach statements part of the scripts in section 5.3.7, p.98, section 5.3.8, p.99 and section 5.3.11, p.100 make use of these semantics.

3.11 Imports

GTL supports modularization through *imports*. Any GTL library or program is considered a *module* (see section 3.13, p.74). It can be imported by another module, or import other modules itself. Through imports the various sections defined in the imported module are added to the importing one.

Importing a GTL program is allowed; it is treated as if it were a library. This will be described in section 3.13.3, p.75.

3.11.1 Import Declarations

Imports are declared in the '`imports`' section (see section 3.12.3, p.73). An import declaration must provide the *filename* of the module to be imported, without extension.

A filename can be preceeded by a series of directory references, the path, to identify the location of the file in the filesystem. This path is relative to the location of the program mocule being evaluated.

An import declaration has the following layout:

`imports` ⟨*relative path*⟩/⟨*modulename*⟩

The following example shows an import of the module '`ANSI-c.gtl`' from the directory '`language/syntax`':

GTL
```
1 imports language/syntax/ANSI-c
```

Importing '`strings.gtl`' from the directory '`libraries`':

GTL
```
1 imports libraries/strings
```

3.12 Sections

There are four different types of GTL *sections*: '`patterns`', '`variables`', '`imports`' and '`functions`'. They can be used in programs, libraries an functions. The different sections will be discussed here.

3.12.1 The '`patterns`' Section

The '`patterns`' section allows the programmer to create named patterns that can be used as aliases within any other pattern specification. The '`patterns`' section has the following layout:

```
patterns
  ⟨pattern declaration⟩;
  ⟨pattern declaration⟩;
  ...
```

It starts with the keyword '`patterns`' followed by a semi-colon delimited list of pattern declarations.

Pattern declarations were discussed in section 3.6.1, p.49. In the following, part of an example which was also used in that section is implemented in a '`patterns`' section:

GTL
```
1 patterns
2   Grammar #all_aliases = aliases ... ;
3   Grammar #all_restrictions = <{ restrictions ..., lexical restrictions ...,
4                                  context-free restrictions ... }> ;
5   ...
6   Section #all_sections = <{ exports ..., hiddens ... }> ;
7   Module #all_modules = module ... ;
```

3.12.2 The '`variables`' Section

Variables as declared in the '`variables`' section are used in matching and assignment statements. The '`variables`' section has the following layout:

```
variables
  ⟨variable declaration⟩;
  ⟨variable declaration⟩;
  ...
```

It starts with the keyword '`variables`' followed by a semi-colon delimited list of variable declarations. Variable declarations were discussed in section 3.4.1, p.43.

An example '`variables`' section with variable declarations taken from section 3.4.5, p.47 looks like this:

GTL
```
variables
  Grammar MyGrammar:. := context-free syntax
                           "MyLiteral" -> MySort;
  Module MyModule:. := module MyModule
                       exports
                       sorts MySort
                       <MyGrammar>;
```

3.12.3 The '`imports`' Section

Imports are declared in the '`imports`' section and are used for modularization of the program. The section has the following layout:

```
imports
  ⟨import declaration⟩,
  ⟨import declaration⟩,
  ... ;
```

It starts with the keyword '`imports`' followed by a comma-delimited list of import declarations. Import declarations were discussed in section 3.11.1, p.71.

3.12.4 The '`functions`' Section

Functions are declared in the '`functions`' section (see section 3.12.4, p.73). It has the following layout:

```
functions
  ⟨function declaration⟩;
  ⟨function declaration⟩;
  ...
```

It starts with the keyword '`functions`', and is followed by a semi-colon delimited list of function declarations. Function declarations were discussed in section 3.7.1, p.51.

Using the example function declaration from that section, we create an example functions section:

GTL
```
functions
  Production &MyFunction(Symbol A1:.):... -> Boek {
    begin
      <MyFunction> := <A1> -> Boek;
    end;
  };
```

3.13 Modules

A *module* is a text file containing a GTL program- or library- declaration. A module with the extension '`.gtp`' contains a *program declaration*, and a module with the extension '`.gtl`' contains a *library declaration*.

The distinction between programs and libraries lies only in the fact that programs can be directly executed and libraries cannot. This implies that the top module that is executed must contain a program declaration, since it declares a primary function. Program declarations are discussed in the next section.

3.13.1 Programs

A program module contains a *Program Declaration*. A program declaration is almost the same as a function declaration, preceded by the keyword '`program`'. This function is called a *Primary Function*. A primary function declaration has the same syntax as a normal function declaration (see section 3.7.1, p.51). In effect, a GTL program is almost the same as a function (see section 3.7, p.50). The difference is that the statements of a primary function are executed immediately upon program execution, and that the arguments it requires are passed to it through command-line options or through a unix *pipe*, specifically `<STDIN>`. The return value of a program function is passed to the pipe `<STDOUT>`. See section 4.8, p.92 for a more thorough explanation.

As with functions, the name of the primary function, or program, is declared as a variable before execution of the statements. After execution, its contents constitute the result value of the program. The result value must adhere to the declared pattern of the program.

A GTL program has the following layout, differing from the layout of a function declaration only by the addition of the keyword '`program`':

```
program
⟨type⟩ ⟨name⟩( ⟨argumentdeclarationlist⟩ ) [:⟨pattern⟩];
⟨sections⟩
begin
  ⟨statementlist⟩;
end;
```

Note that the pattern specification is optional. If the pattern specification is not part of the declaration then the 'exactly one' pattern ('`.`') is assumed to be the pattern for the program.

An example of a GTL program:

GTL

```
program
Module &MakeModule():.;

patterns
  Module #all_modules = module ... ;

variables
  Grammar MyGrammar:. :=
    context-free syntax
      "MyLiteral" -> MySort;
  Module MyModule:#all_modules :=
    module MyModule
    exports
    sorts MySort
    <MyGrammar>;

begin
  <MakeModule> := <MyModule>;
end;
```

3.13.2 Libraries

A GTL *library* can contain all GTL sections except the '`variables`' section, meaning variables cannot be exported.

A *library declaration* is a list of sections, preceded by the keyword '`library`'. It has the following layout:

`library` ⟨*name*⟩
⟨*sections*⟩

Note that a library declaration does not define type, pattern or arguments, since it can neither receive nor return them.

3.13.3 Importing Modules

GTL implements modularization through import declarations in the '`imports`' section (see section 3.12.3, p.73). Both programs and libraries can be imported from any module.

Importing a library means that all the sections defined in that library will be added to the section declarations in the importing module, making them available for use from within that module.

Importing a program works largely in the same way. The sections defined as part of the primary function will be added to the section declarations in the importing module. Also, the primary function as declared in the imported program is treated as if it were a 'regular' function. It is also made available in the importing module. This was done to allow programs to take advantage of the primary functionality provided by other programs.

3.14 Declaration Scope

A scope is the frame of execution where a declaration is valid. All variables, patterns, imports and functions are part of a scope.

The Program Scope

A program has the primary scope. All the declarations made in the various sections part of a program declaration are added to that scope. The declared patterns and variables, either declared in the program declaration or imported, are valid to be used in the begin ... end statement part of the program declaration, as are the functions.

The Function Scope

Each function declaration gets its own scope, just like a program, so that all declarations made within are valid for that function.

3.14.1 Inheritence

Certain declarations are inherited by a function from its parent scope, which is either the scope of a program or of another function.

These declarations are patterns and functions. Patterns as well as functions declared in the parent scope or higher can also be used in the current scope. It has the same effect as if the patterns and functions were declared in a library and imported in each scope.

Variables cannot be inherited, but they can be passed to a function using its arguments.

3.15 Grammar Equivalance

Grammars are said to be equivalent if they can produce the same strings as output. Proving two grammars equivalent is undecidable [8], but a different type of equivalence, namely *structural equivalence* can be used. This method for grammar comparisin was also used in the FST [5].

3.15.1 Structural Equivalence

With structural equivalence, two grammars must provide the same *definition*; the same sorts with the same names and each sort must be defined using the same terms. A more formal description can be found in [7].

Structural equivalence does not cover all the grammars that can produce the same output. The following two SDF grammars can parse the same string for the sort '`S`', namely '`ab`'. They are equivalent, but they are not *structurally* equivalent, since a definition for the sort '`B`' does not exist in the grammar on the right, and the definition for the sort '`S`' is different:

```
1 context-free syntax
2   AB -> S
3   "a" -> A
4   "b" -> B
```

```
1 context-free syntax
2   A "b" -> S
3   "a" -> A
4
```

3.15.2 Structural Equivalence Modulo Renaming

With structural equivalence modulo renaming the sort names do not have to be identical, as long as they are defined using the same terms (modulo renaming), and they are used in the same terms defining other sorts.

The following two SDF grammars are structurally equivalent modulo renaming:

```
1 context-free syntax
2   AB -> S
3   "a" -> A
4   "b" -> B
```

```
1 context-free syntax
2   CD -> T
3   "a" -> C
4   "b" -> D
```

They can both parse the same string '`ab`', the grammar on the left using the sort '`S`' and the grammar on the right using the sort '`T`'.

3.16 Grammar Merging

The basic functionality behind imploding, narrowing and widening a selection as discussed in section 3.8.2, p.55 is the merging of grammars. In the examples this was expressed by a "plus" infix operator ('+'). The implementation of SDF construct merging is covered in section 4.5.2, p.82.

3.16.1 Merging Identical Types

In a selection both the focus and the remainder are of the same type. A merge of two identical types is rather straigthforward, though the semantics differ for each type. The relevant semantics include:

allow/disallow doubles
: There may be no modules with the same name, but a production may contain infinite references to the same sort/literal/etc.

ordered/unordered
: The order in which grammars appear in a section does not matter, but the order in which symbols appear in a production does.

3.16.2 Merging Different Types

Merging two different types is possible only when one of the types is a sub-type of the other, that is, when one of the types is used in the definition of the other. A production can be merged with a context-free syntax grammar by merging the production with the

productions present in the grammar. However, production can not be merged with a sorts grammar, since neither is defined using the other.

Allowing the '`merge`' operator to take arguments of differing types prevents the programmer from having to program a series of matches to get to the intended types, then do the merge and then recreate the original structure using the constructs matched before the merge.

Checking whether a merge between differing types is possible, only has to be done once. The arguments to a merge, whether used directly with the '`merge`' operator, or implicitly as part of one of the other selection operators, are typed. This can be used to warn the programmer at compile time about 'incompatible' types, like a production and a sorts grammar.

Chapter 4

Implementation

4.1 Introduction

In implementing GTL a few goals were kept in mind. The maintainability and flexibility of the implementation being the most important ones. The case study shows that for GTL to remain usable with a changing specification of SDF the implementation needs to be easily adaptable and extensible. This means we must use a programming language with the features and functions that will aid the reaching of these goals.

We will start by explaining the choice of the programming language for the implementation of GTL. Then an overview of the implementation is given, after which the implementation is discussed.

4.2 Choosing A Programming Language

The choice of a programming language for implementing an SDF transformation language was an obvious one. The ASF+SDF meta-environment was developed for creating languages and implementing transformations on those languages, including interpreters or compilers. This, and more, makes it a good choice for implementation of GTL.

The specification of SDF is already implemented in SDF and is provided with the ASF+SDF distrubution, so creating a parser for SDF is not nessesary. Also, any new version of the SDF specification will be part of a new distribution as well, meaning that the maintainer of GTL only has to adapt the GTL implementation, not implement the changes in the new SDF specification.

The distrubution also includes various libraries whose functions can be used in the implementation of GTL. These include functions for matching and testing character classes, which are part of the SDF as well as the GTL language.

The modularization capabilities present in SDF also help to create a maintainable interpreter. Modules in SDF can be parameterized, allowing the functionality expressed in one module to be used as part of multiple sorts. Sorts, like modules, can also be parameterized, allowing for the creation of a more legible and maintainable implementation. These SDF features are heavily relied upon in the implementation of GTL.

There are also downsides to using the ASF+SDF meta-environment. The GTL pattern system is intended to be very closely modeled after the SDF constructs it is intended to match. This means that any 'literal' patterns, that is those patterns that have the exact same syntax as their SDF counterparts, will introduce ambiguities between the SDF constructs and the GTL patterns in the parser. Creating a well defined specification of GTL employing the various disambiguation techniques that SDF provides should overcome most if not all of these problems.

Another downside has to do with the maintainability of the implementation. If the SDF specification changes, the implementation will have to be adapted. However, since the implementation is done in SDF itself, it must first be adapted to use the new SDF syntax and possibly semantics. This is not a problem as long as the new SDF specification is backward compatible with the old. If this is not the case however, it could mean that the implementation of GTL will need a lot of work to update it to the new SDF specification.

These downsides can be overcome, at least to a high degree. The benefits of using the ASF+SDF meta-environment for the implementation of GTL in our opinion still outweigh the downsides mentioned.

4.3 Overview

The implementation of GTL is split into four groups of modules. The *Environment* group contains the definition of the GTL environment. The *Pattern Matching* group is the matching kernel and it contains the untyped GTL pattern definitions. The *Types* group contains a number of modules each defining the pattern -syntax and -semantics specific to a GTL type. Lastly the *Evaluator* group contains the specification of the GTL syntax and semantics.

4.3.1 SDF Specific Features Used

GTL is defined in SDF2, and makes use of various features such as like module- and sort-parameterization. Module parameterization allows for the definition of generic modules, so that we can create one set of modules for pattern matching which will be imported once for each type.

The operations defined in these modules are independent of a specific type, which is important since we want the generic GTL patterns like the 'exactly one' pattern ('`.`') to be able to represent any SDF construct. We need to avoid ambiguities however, so, to distinguish between identical patterns for different GTL types we will use parameterized sorts, which increases readability of the source and avoid name clashes between different instances of the same module defining the patterns.

4.4 The Environment group

This group defines the environment and the functionality needed to store and retrieve data from it. The environment structure is devided into four sections, the '`variables`'-,

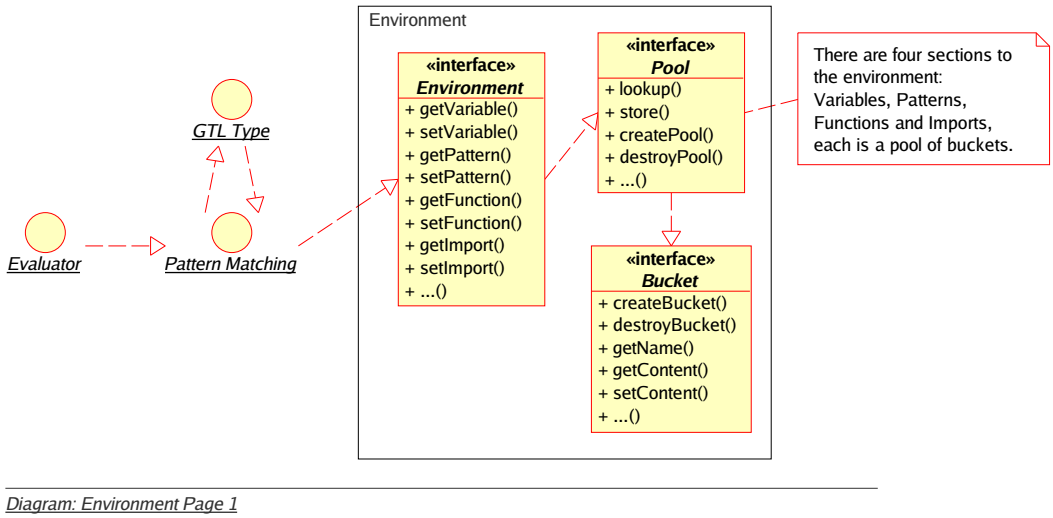


Figure 4.1: The Environment Interfaces

'`patterns`'-, '`functions`'- and '`imports`'- sections, to hold the stored items.

The individual items are stored in buckets, which in turn are stored in a pool of buckets. Each section is a simple list of pools. Each time a scope is added a pool is created and appended for each section. When a scope is destroyed the last pool in the list will be destroyed.

The bucket and pool structures are defined in two parameterized modules, '`Bucket`' and '`Pool`'. The parameter provides an ID for the structures which is used as a parameter to the sorts defining the bucket and the pool. This is done to avoid ambiguities and to allow for multiple distinguishable instances.

4.4.1 The Bucket Interface

The '`Bucket`' module defines the storage of a single item. A bucket is a parameterized sort '`Bucket[[TypeID]]`' and contains three constructs, the given '`TypeID`', a sort '`Name[[TypeID]]`' and a sort '`Content[[TypeID]]`'. The last two sorts should be defined by the importing module. A number of functions are also provided to set and retrieve the name or content of a bucket, as well as create a new bucket and test whether it has content. By leaving the name- and content- sorts to be defined by the importing module, the specification of a bucket is truly independent of its name and content, and can be reused for all the sections in the environment structure.

4.4.2 The Pool Interface

The '`Pool`' module imports the '`Bucket`' module passing along its parameter '`TypeID`'. It can store a list of buckets. A pool contains two constructs, the given ID and a list of buckets of the type. A number of functions are provided to lookup in-, remove from- or

add a bucket to- the list, as well as create a new pool. Some functions from the '`Bucket`' module are overloaded to work on a pool as well by implicitly performing a lookup in the pool. As with the bucket structure, a pool is independent of its content and can be reused for all the sections in the environment structure.

4.4.3 The Environment Interface

The environment structure is defined in the '`Environment`' module. It imports the '`Pool`' module to represent a single scope. Each section is a list of scopes.

The '`Environment`' module also defines the ID's and name syntax needed for the definition of the pools and buckets that make up the sections in the environment. It is nessesary to provide the name syntax to be able to provide the lookup- and store- functions for the environment structure. Since we import the '`Pool`' module four times, once for each section, we need a way to keep them apart. The semantics for these lookup- and store- functions are specific to each section, since some of the stored items can be inherited between scopes, and others cannot. For instance, variables and imports are looked up only in the top scope, while functions and patterns are looked for in the entire list of scopes because they are inherited.

A number of functions are provided to take care of appending-, creating- or destroying- a scope.

4.5 The Pattern Matching group

The GTL patterns that are type-independent are defined in the pattern matching group. For example the sequence- and set- patterns and the GTL repetition operators. This group also provides functions and sort declarations to allow for the addition of new types.

There are two separate interfaces, one is used solely by modules that define new types, and the other provides a type-independent interface to be used in the evaluator. The former is parameterized with the information needed to add a type, like the type identifier and the construct it represents. The latter is not parameterized and encompasses all the types and type-less GTL patterns.

4.5.1 The Matching Kernel Interface

This interface defines and implements three basic functions '`match`', '`select`' and '`merge`'. They handle the semantics of all GTL patterns, including the types created by implementing the functions in the Pattern Matching interface. They are the core functions that are used by the evaluator and the matching functions.

4.5.2 The Pattern Matching Interface

This interface provides three functions '`matchSingle`', '`selectSingle`' and '`mergeSingle`' to handle single type-specific patterns. They are implemented only by type defining modules. Default rules are provided that will fail them, so an importing module that defines

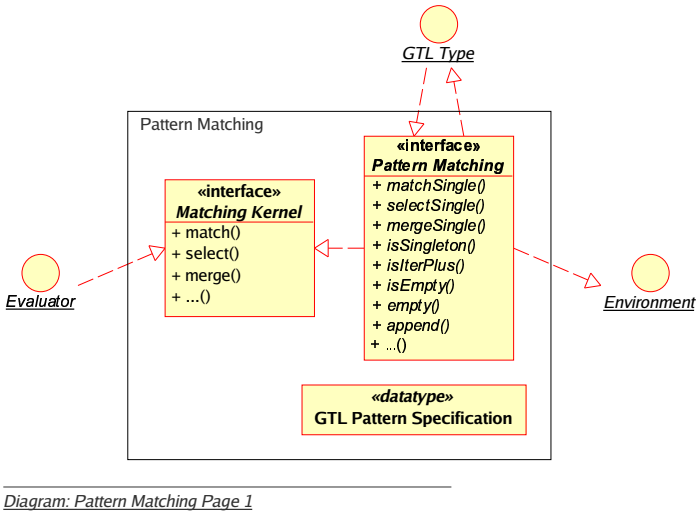


Figure 4.2: The Pattern Matching Interfaces

a new type will need to redefine them to supply the correct semantics. This way, the semantics of a new type can be added by simply providing rewrite rules to handle these functions for the type.

The three '`-Single`' functions are recusive in the sense that they call their namesake functions for the types used in their definition. The functions for type `Production` call the same function for `Symbol` and `Attributes`. These types are part of the definition of a `Production`. If these calls are successful, the calling function will handle the relevant semantics and return its results.

For '`selectSingle`', the results of the recusive calls to the sub-types will be used to built two new constructs of the same type as the current function, one is the focus and the other is the remainder. These are used to create a selection wich is returned as a result. For '`matchSingle`', the result is either a match or not, if it is a match, the entire construct is returned as the result. Both functions use the same matching engine, which returns a selection as a result. In the case of matchSingle, the remainder of the resulting selection is empty, and the focus contains the entire matched construct.

For '`mergeSingle`', the handling is different. The function for each type can only merge two constructs of the same type, again using recursion. Merging two different types is handled by the pattern matching module. Merging two different types is done by taking one construct and trying to merge it with each of the children of the other, top-down left to right. The first type-match made will induce the merge, after which the merged result is used to create the new construct. If no merge is induced, the next child of the original construct will be handled in the same manner. If the one construct in the function call does not appear as a child in the other, the two constructs are swapped and the merge is retried. If this still does not produce a merge, for instance when merging a production with a sorts grammar, the merge fails.

Apart from single constructs, we also need to be able to match lists. There are two types of lists in SDF, flat lists which are defined using SDF -repetition operators and -list operator, and recursive lists which represent the definition of a list by referencing themselves. An example of the latter list type is the SDF type grammar. To keep as much as possible of the kernel code unique, five functions independent of the list type were created that represent the list operations needed. The pattern matching group only defines the default rules which will fail the match. Each type defining module importing the group must define these functions to correctly handle lists of its type.

Some of the types in GTL are made up of characters, for instance the types `Sort` and `ModuleId`. A type-specific GTL pattern, the lexical pattern, was introduced to provide the ability to match and alter parts of these types. Since this pattern can be used for more than one type, the basic syntax and semantics are part of the pattern matching group. The modules providing this pattern are imported only by the types that need it. These modules are made part of the pattern matching group even though they are not fully type-independent.

4.5.3 Pattern Ambiguities

For the definitions of the types, simply injecting all the SDF constructs into a single sort would create a lot of ambiguities, because some types are injected into others. This makes it impossible to decide which type is actually intended. A literal for instance is injected into the symbol type, so when parsing a literal, the parser does not know which type is intended, symbol or literal.

To circumvent this in cases where the intended type is known, each type is declared using a parameterized sort. One parameterized sort for the type-pattern and one for the construct it represents. This avoids most of the ambiguity problems, since we can now specify which pattern type can match which SDF construct directly.

The pattern matching operators of GTL can take any type to operate on, so in parsing them, the injection problem persists. This can be solved by preferring some types over others in parsing, using the '`prefer`' and '`avoid`' attributes.

4.5.4 The Pattern Structure

To separate the types, all sorts that define a pattern are parameterized, having the textual representation and the represented SDF construct of the type as a parameter.

Some of the patterns can be used in more than one GTL construct. A variable for instance can be part of a pattern but also part of an extended SDF construct. Thus, the variable pattern needs to be defined separate so that it can be included in the definition of both the GTL pattern structure and the extended SDF structure.

We also need to be able to restrict specific GTL patterns in their use. For instance an extended SDF construct may not include the GTL option- or repetition- operators, since these can not be reduced to an SDF construct.

A structure of sort injections is set up here to allow specific patterns to be restricted to certain GTL constructs, and to avoid the ambiguities discussed.

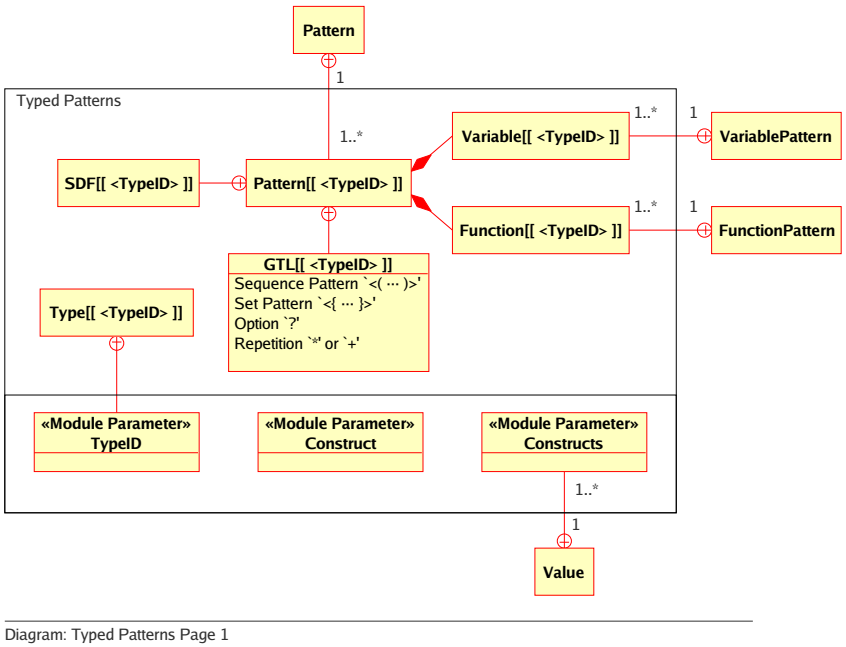


Figure 4.3: The Typed Patterns Structure

The Typed Patterns Structure is defined in a separate module named '`Constructs`', see appendix A.3, p.113.

The Typed Pattern Structure

The following list describes the sorts that hold the definitions for a single type. Their structure is depicted in the top section of figure 4.3.

'`Pattern[[`⟨*TypeID*⟩`]]`'
: At the core of the pattern structure is the sort '`Pattern[[`⟨*TypeID*⟩`]]`'. It will encompass all the patterns for a single type.

'`GTL[[`⟨*TypeID*⟩`]]`'
: The sort '`GTL[[`⟨*TypeID*⟩`]]`' holds the definitions for the GTL sequence-, set-, option- and repetition- operators. Notice that the lexical- and 'any repetition'- operators are missing, this is because they are type-dependent.

'`SDF[[`⟨*TypeID*⟩`]]`'
: The sort '`SDF[[`⟨*TypeID*⟩`]]`' will hold the pattern which is used to match its represented SDF construct. It is declared here but not defined, since it is type-specific. Its definition must be provided by the importing modules defining the type.

'`Variable[[`⟨*TypeID*⟩`]]`' and '`Function[[`⟨*TypeID*⟩`]]`'
: The variable- and function- patterns are held in the sorts '`Variable[[`⟨*TypeID*⟩`]]`' and '`Function[[`⟨*TypeID*⟩`]]`' respectively. These are defined exactly the same for each type, so no external definition is needed. They are defined separate from the

other sorts because they are allowed to be used in patterns as well as extended SDF constructs.

'`Type[[`⟨*TypeID*⟩`]]`'

The sort '`Type[[`⟨*TypeID*⟩`]]`' will hold the textual representation of the type for reference in declarations in a GTL program. It is defined by the '`TypeID`' parameter given to the kernel modules.

The Encompassing Pattern Sorts

There are four sorts without parameter. They hold the definitions of for *all* the types through injection. They are shown outside of the box in figure 4.3.

'`Pattern`'

This sort will hold the sorts '`Pattern[[`⟨*TypeID*⟩`]]`' for all types, and is used in the specification in places where a pattern of *any* type is allowed.

'`VariablePattern`'

This sort will hold the sorts '`Variable[[`⟨*TypeID*⟩`]]`' for all types, and is used in the specification of the Unknown type, where it is not known which type is intended. This will be discussed later.

'`FunctionPattern`'

This sort will hold the sorts '`Function[[`⟨*TypeID*⟩`]]`' for all types, and is used in the specification of the Unknown type, where it is not known which type is intended. This will be discussed later.

'`Value`'

This sort holds all the different SDF constructs introduced by the modules defining the GTL types. It is never used for matching since that would introduce ambiguities. It is however used for the declaration of type-less functions. The function to reduce an extended SDF construct is one such function.

The Module Parameters

At the bottom of figure 4.3 the three module parameters that are required to create a new type are given. They are the following:

'`TypeID`'

a type identifier, which is a quoted literal representing the textual name of a type. It is used as a parameter to the sorts that hold the definitions of the patterns for the type. It also represents the type in a GTL program as part of declarations.

'`Construct`'

the SDF construct the type represents. It is used primarily as the type of a variable in the rewrite rules describing the semantics of the patterns.

'`Constructs`'

a parameter representing a list of the same construct to handle the type-specific lists. This parameter is included to allow the handling and representation of different types of lists, namely flat- and recursive- lists. It is used in the same way as the '`Construct`' parameter.

4.5.5 More Pattern Ambiguities

The GTL specific patterns do not infer a type from their syntax. For instance the '`.`' and '`...`' patterns, but also variables and functions. This means that a large pattern that does not contain a recognisable pattern of a type does not infer a type from its syntax either. The pattern '`<( ...; . )>`' is such a pattern which can be matched againts *any* type. If the type of the SDF construct being matched is known from syntax, this does not present a problem. But using extended SDF constructs allows us to use initialized variables. The variable pattern syntax does not infer type, so when we match an initialized variable against a type-less pattern, it can not be discerned from syntax which type semantics should be used for matching.

The solution is to parse these patterns using a single, but hidden type that has *preference* over *all* other types in parsing. It is called the '`Unknown`' type. Its semantics are that its rewrite rules try to extract the intended type from the pattern and SDF constructs. This is done by retrieving the types of the variables and functions present in both. When the type is extracted the appropriate rules can be used to evaluate the match.

In matching, we always need a pattern as well as an SDF construct. As stated the pattern can be type-less, meaning a type for the pattern cannot always be extracted from it. For the extended SDF construct this is different, since the only GTL patterns it may contain are variables and functions. The types for both of these can be retrieved from the environment, since they must be declared with a type. If an extended SDF construct does not contain any GTL variables or functions, it respresents an SDF construct directly and its type can be inferred from syntax, taking into account the preference between the various types. So, the type of an extended SDF construct can always be ascertained, implying that for a match the intended type can also always be discovered, albeit sometimes after parsing it using the '`Unknown`' type by retrieving the type of a referenced variable or function.

4.5.6 The '`Unknown`' type

Sometimes the type of a pattern can not be extracted from the sub-patterns because it is made up solely of type-less GTL patterns. The pattern '`<( ...; . )>`' is such a pattern. To allow matching of these patterns they are typed to a hidden type, called the `Unknown` type. Like all public types, this type imports the type-independent GTL patterns. However, the 'exactly one' and 'zero or more' patterns are also defined for this type. The semantics for these are that they can be matched to *any* type. So, a pattern of type `Unknown` can be matched to any other type. It does not represent an SDF construct, and as such can not be used in an extended SDF construct because it cannot be reduced.

The `Unknown` type was added to avoid ambiguities when parsing the type-less GTL patterns, see 4.5.5, p.87.

Since it does not represent an SDF construct, it is defined in the Pattern Matching group. It must be declared and defined separately because it does not contain an type-specific pattern.

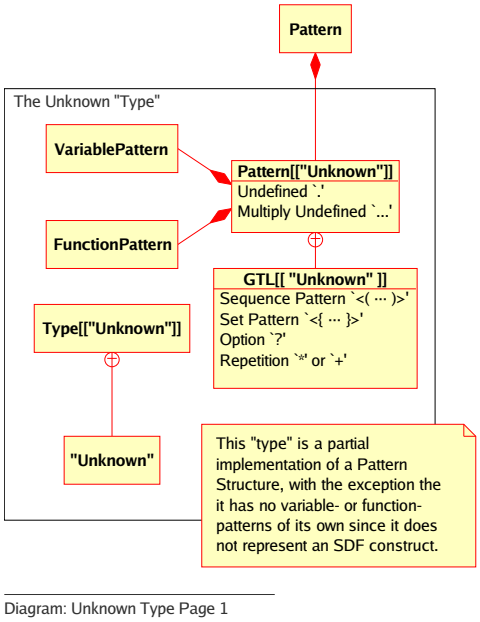


Figure 4.4: The Unknown Type Structure

The 'Unknown' Type Structure

The definitions of the type are parameterized in the same way as typed patterns, using the type identifier '`"Unkown"`'. The structure is shown in figure 4.4

The sort '`Pattern[["Unknown"]]`' contains definitions for the 'exactly one' and 'zero or more' patterns. The sort '`SDF[["Unknown"]]`' does not exist and the type has no variable or function patterns of its own because the type does not represent an SDF construct. It does contain the encompassing pattern sorts for variables and functions, since it needs to be able to match them all to ascertain their type for subsequent matching.

Other than these changes, the structure is the same as the one for typed patterns.

To see how the '`Unknown`' Type Structure interacts with the Typed Pattern Structure a diagram of the two together is provided in appendix A.1, p.111.

The '`Unknown`' Type Structure is defined in a separate module named '`Unknown`', see appendix A.3, p.113.

4.5.7 The Lexical Pattern

This pattern can be used to match parts of the SDF types that are made up of characters, like literals and sorts. Since the pattern is implemented by more than one type, it is defined separately to be included by those types that require it. The syntax and semantics of this pattern and its supporting functions are defined in two modules, '`CharMatching`' and '`CharSupport`', see appendix A.3, p.113.

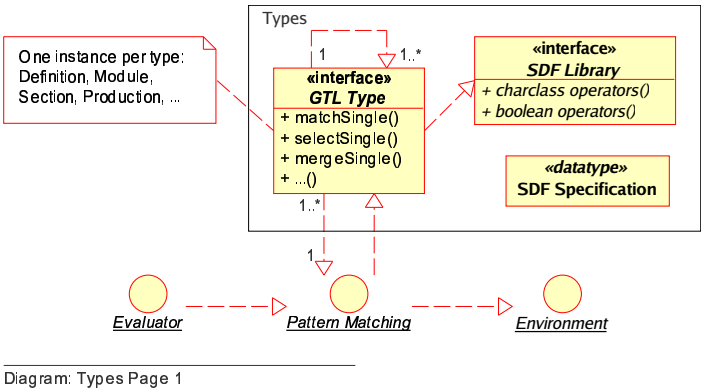


Figure 4.5: The Types Interfaces

4.6 The Types group

The types group contains the type-defining modules. For each type, an number of sort definitions have to be provided, as well as rewrite rules to handle the matching of a single instance of the type to the SDF construct it represents.

Each module imports the pattern matching modules from the pattern matching group at least once, supplying an SDF construct and type identifier string as parameters. The module must provide a parameterized sort '`SDF[[`⟨*typeid*⟩`]]`' that defines the pattern equivalent of the SDF construct it represents.

To facilitate the handling of lists of each type, the rewrite rules for five additional functions need to be defined. Since there are only two types of lists involved, namely flat- and recursive- lists, these are defined in separate modules where they can be imported into the module defining the type. Since there is only one type that employs a recursive specification, the grammar type, only the module defining the functions to handle flat lists is actually implemented. This module is called '`Matching`', see the figure in appendix A.3, p.113. It is this module that eventually imports the modules from the Pattern Matching group.

In figure 4.5, p.89, the interfaces part of- and included in- the Types group are shown. The interface to the SDF library is a simplification representing a group of modules imported and used by various types. These include the syntax and semantics for booleans and various functions that operate on SDF character classes.

4.6.1 A Type-Definition Example

In this example, the type `Production` is defined and implemented.

The SDF sort '`Production`' is represented in GTL by the parameterized sort '`SDF[["Production"]]`'. The SDF definition of a production is as follows:

SDF

```
1 Symbol* "->" Symbol Attributes -> Production
```

The corresponding GTL type-pattern definition '`SDF[["Production"]]`' is defined similarly, using only GTL type-patterns.

SDF
```
1 Pattern[["Symbol"]]* "->" Pattern[["Symbol"]] Pattern[["Attributes"]] ->
2                                                              SDF[["Production"]]
```

Using the sort '`Pattern[[`⟨*TypeID*⟩`]]`' in the definition of the pattern allows the programmer to use GTL patterns as part of the pattern for a `Production`.

Rewrite rules specific to this type have to be defined for the functions '`matchSingle`', '`selectSingle`' and '`mergeSingle`'. In this case, each of these functions calls the pattern matching functions of its sub-patterns for the types `Symbol` and `Attributes` to handle their matching. This means that the module defining this type-pattern will have to import the modules defining the `Symbol` and `Attributes` types.

To facilitate the matching of (flat) lists of this type the '`Matching`' module is imported using the parameters '`Production`' and '`"Production"`' as the SDF construct and type-id representation. The '`Matching`' module defines the nessesary list functions and imports the required modules from the Pattern Matching group.

4.6.2 More Pattern Ambiguities

As stated before, injections pose a problem in parsing. This is no different with pattern definitions. Take for instance one of the definitions for the SDF type symbol:

SDF
```
1 Literal -> Symbol
```

The definition of the sort '`SDF[["Symbol"]]`' for this injection would look like this:

SDF
```
1 Pattern[["Literal"]] -> SDF[["Symbol"]]
```

However, this will create problems when matching a literal represented by a symbol against the pattern '`.`', which is a part of the sort '`Pattern[["Literal"]]`' as well as the sort '`Pattern[["Symbol"]]`'. It cannot be ascertained which of the two types is intended.

To avoid this problem, the definition of the sort '`SDF[["Symbol"]]`' for this injection looks like this:

SDF
```
1 SDF[["Literal"]] -> SDF[["Symbol"]]
```

where only the single representation for the literal type is injected, which does not contain the pattern '`.`'. So, when matching a literal represented by a symbol against the pattern '`.`', it is now clear that the type `Symbol` is intended, since the '`.`' pattern is not part of the definition of the sort '`SDF[["Literal"]]`'.

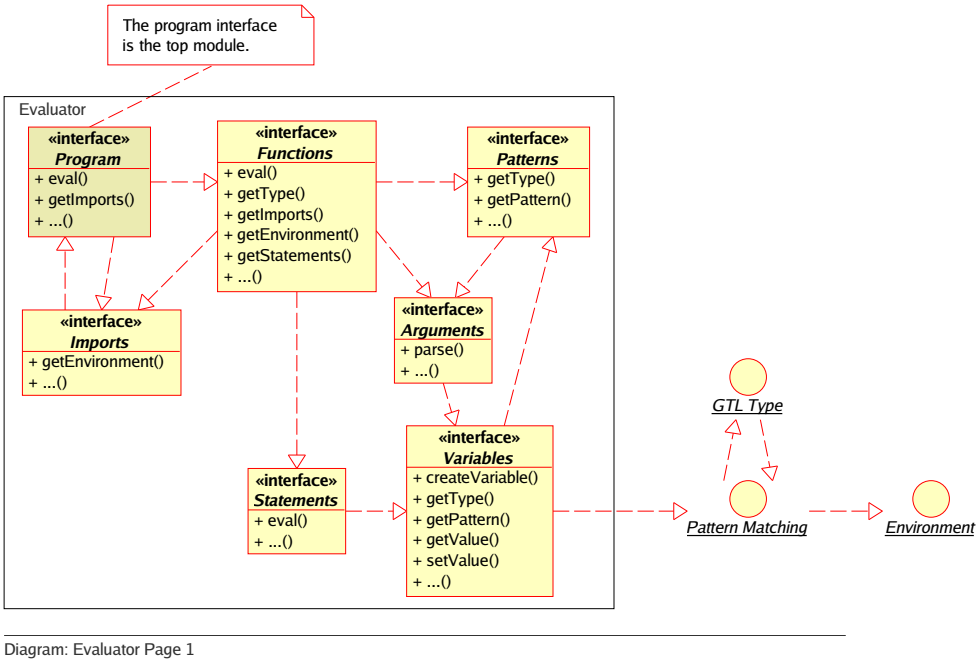


Figure 4.6: The Evaluator Interfaces

4.7 The Evaluator group

This group contains the modules that declare and define the syntax and semantics of the GTL language. There are seven interfaces that introduce the various constructs. These are shown in figure 4.6, p.91.

Four of the interfaces, the functions-, patterns-, imports-, and variables- interfaces, introduce the pattern-, declaration- and section- semantics of their name-sake GTL constructs.

The interface for arguments was created because arguments are used in both patterns and functions, it is prudent to implement them only once.

The interface for GTL statements is used only in functions. It provides the semantics of the various GTL operators and statements.

Lastly, the program interface provides the functions for evaluating a GTL program. It does not use the arguments interface directly even though it can have them. This is because a GTL program is implemented as a function. A GTL program is also called the primary function.

4.7.1 Four Sections

There are four sections to a GTL -program or -library, the functions-, patterns-, imports-, and variables- sections. Each is implemented in its own module that contains the defini-

tions and declarations for the GTL construct. See the Evaluator cluster in the dependency graph in appendix A.3, p.113.

The `Sections` and `Declarations` Modules

Since the four GTL constructs all have their declarations done in sections, and these sections are handled in the same way, the functionality for evaluating them has been put in two separate modules, `Sections` and `Declarations`. Both modules are parameterized and take a GTL construct identifier as an argument. This argument is used as a parameter to the sorts defining the section- and declaration- syntax for the section. These sorts are declared in these modules, but must be defined by the importing module. The semantics for the evaluation of a declaration must also be defined in the importing module. The `Declarations` module only contains a function to evaluate a list of declarations, as does the `Sections` module for a list of GTL sections.

The Four Section Modules

The four modules that define a section also define the syntax and semantics for the pattern and the declaration of the GTL construct. Only the `Imports` module does not define a pattern, since there is none for the construct. The other three provide rewrite rules that extend the three matching functions '`match`', '`select`' and '`merge`' for the pattern they introduce. They also provide functionality to simplify the interaction with the GTL environment.

4.7.2 Statements

The GTL statements and operators are defined in the `Statements` module. A function '`Eval(`⟨*statement*⟩`)`' is declared and defined to handle their evaluation.

Expressing their semantics is a simple matter of projecting them onto the basic matching functions. In most cases, only one rewrite rule is needed to handle a GTL statement or operator.

4.7.3 Programs and Libraries

The `Program` module defines the GTL program and library. Two functions '`Eval(`⟨*program*⟩`)`' and '`Eval(`⟨*library*⟩`)`' are declared and defined to handle their evaluation.

In both cases the environment is filled with the declarations made in the program or library. The difference is that a program is treated as a special function which is always evaluated. A library merely adds declarations to the environment.

4.8 The GTL Interpreter

The GTL interpreter is intended to be used as any unix scripting language. This means that the first line of any GTL program can start with:

GTL
```
#!\usr\local\bin\gtli ...
```

where 'gtli' is the filename of the GTL interpreter.

There are a number of options that can be given to the interpreter. Apart from the name of the GTL program to be interpreted, its arguments and its input, these include providing root-directories pointing to where the GTL libraries reside. Other options, like controlling the format of the output, can be envisioned but are not implemented as yet.

The interpreter has been implemented and tested with a rudimetary GTL program, but because of problems encountered in the implementation (see section 6.3, p.109), it was not finished.

4.9 Deviations from SDF standards

There are a number of issues in the implementation where the semantics deviate from the official semantics of SDF.

• Currently, there is no checking whether two modules of the same name exist in the grammar being transformed. The merge function will correctly merge two modules of the same name, but it has to be called explicitly. Two modules of the same name can be put into a specification by manually assigning them to a `Definition`. For instance:

GTL
```
variables
  Module M1:module A
              sorts S
            exports
            lexical syntax
              "S" -> S;
  Module M2:module A
              sorts S
            exports
            lexical syntax
              "K" -> S;
  Definition D := definition
                    <M1>
                    <M2>;
```

This behaviour is not according to the SDF specification, which states that all module names must be unique.

• Sortnames in SDF must start with a capital letter. In GTL, a sortname is allowed to start with a lower case letter, which does not conform to the SDF specification. However, there is no active checking for this in the ASF implementation either. This was left in because the only way to correct the behaviour would be to disallow the use of uninitialized variables of type Sort. The other types do not have this problem. Checking for this would break the modularization of the implementation, since it would have to be implemented in the generic functions for the lexical pattern. These generic functions are used by all the types that can be used in a lexical pattern.

However, there is a way around this. When the programmer wishes to match the beginning and end of a capitalized sortname, the end part might start with a lowercase letter. Currently, this can be assigned to a variable of type `Sort`. The way around this is to match the lower case parts of a sortname using '`Literals`' instead of sorts. These '`Literals`' can be used to create a new sort-name by employing a lexical pattern.

Chapter 5

The Case Study in GTL

5.1 Introduction

In chapter 2, p.15 a case study was performed to find out what kind of functionality is needed in an SDF transformation language. The study was previously performed by Jurgen Vinju and Hayco de Jong, and the recreation tried to implement the various transformations in FST as well as a hypothetical version of FST, referred to as *h*FST.

In this chapter these implementations in *h*FST will be used as a template for implementing the transformations from the case study in GTL. The aim is to show that GTL covers all the added functionality as introduced in the *h*FST scripts, as well as some added functionality.

5.2 Approach

Each step in the case study is recreated in GTL. The headers for the steps bear the same name as in chapter 2, p.15, to facilitate a comparisin. All transformations are implemented as functions named '`Step`$\langle n \rangle$' where n is the number of the section describing the step. These functions are intended to be part of the same GTL program and as such can reference each other at will.

The transformations that are very similar to their *h*FST counterparts will not be explained, apart from inline comments.

Some of the functions used in the *h*FST programs are not present in GTL and have to be expressed using other statements. As such some of the GTL functions can be somewhat more complex.

5.3 Retracing the Previous Case Study

Each of the transformation steps described in the previous case study is listed here. A recount is given on the implementation of each of the FST or *h*FST scripts that were devised as part of those steps.

5.3.1 “first a one-to-one translation from YACC to SDF”

Since this transformation is automated using the `yacc2sdf` transformer provided with XT there is no difference with the step taken in section 2.4.1, p.17.

5.3.2 “replaced terminal identifiers (‘RETURN’) by literal symbols (‘"return"’)”

A function was created in *h*FST that takes a sortname and literal as arguments. The empty definition for the sort is removed, after which all references to the sort in definitions are replaced by the literal. The function can then be called for all the terminal identifiers that need to be replaced.

GTL

```
functions
  Definition &Replace_ID(Definition D, Sort S, Literal L) {
    variables
      Symbol S1;
      Symbol S2, S3:...;
      %% remove the empty production defining <S>
      Definition D' := remove Production: -> <S> from <D>;
    begin
      %% check that no more defining productions exist
      not(<(.)>+ := extract Production:... -> <S> from <D'>);
      %% for each production containing the sort <S> unfold it.
      foreach Production P:<S1> <#S> <S2> -> <S3> in <D'> do {
        <D'> := replace Production:<P>
                with Production:<S1> <#L> <S2> -> <S3> in <D'>;
      };
      <Replace_ID> := <D'>;
    end;
  };

  Definition &Step2(Definition D) {
    begin
      <D> := &Replace_ID(<D>, RETURN, "return");
      <D> := &Replace_ID(<D>, CONTINUE, "continue");
      <D> := &Replace_ID(<D>, BREAK, "break");
      %% the remaining terminal identifiers are not listed
      %% ...
      <Step2> := <D>;
    end;
  };
```

5.3.3 “renamed non-terminals from dash separated to CamelCase”

Since this step was already performed by the ‘`yacc2sdf`’ transformer, no implementation in *h*FST was given. However, it was concluded that the functionality is desirable in GTL,

and as such the lexical pattern for matching/creating a sort was added. An implementation for this transformation can be found in appendix **??**, p.**??**.

5.3.4 "imported lexical definition of C comments from the CPP grammar"

No implementation in *h*FST was given for this step since it involved a simple inclusion of existing SDF modules.

5.3.5 "then added follow restrictions on identifiers and some unary operators"

The transformation implemented in GTL:

GTL
```
functions
  Definition &Step5(Definition D) {
    begin
      <D> := insert lexical restriction
                       Identifier -/- [0-9a-zA-Z\_]
                       D+ -/- [0-9]
             into <D>;
      <D> := insert context-free restriction
                       LAYOUT? -/- [\ \t\011\n\r\012]
             into <D>;
      <D> := insert context-free restriction
                       "&" -/- [\&]
                       "-" -/- [\-]
                       "+" -/- [\+]
             into <D>;
      <Step5> := <D>;
    end;
  };
```

5.3.6 "fixed an ambiguity in the lexical syntax, 'IS*?' was produced, changed to 'IS*'"

Since the function '`&Preserve`' does not exist in GTL '`replace`' is used. Note that a '`foreach`'-loop is not nessesary since the pattern used in the replace statement will match all the productions containing '`IS*?`' one at a time. Replacement continues until no more matches exist.

The transformation implemented in GTL:

GTL
```
functions
  Definition &Step6(Definition D) {
    variables
      Symbol S;
      Symbol S', S'':...; %% used for matching any symbols surrounding IS*?
```

```
6     begin
7       %% for each Production containing IS*? replace IS*? with IS* ...
8       <D> := replace Production:<S'> IS*? <S''> -> <S>
9              with Production:<S'> IS* <S''> -> <S> in <D>;
10      <Step6> := <D>;
11    end;
12  };
```

5.3.7 "re-factored all right-associative lists using SDF2 list syntax"

The function for transforming all comma-separated right-associative lists in a definition:

GTL

```
1  functions
2    Definition &Step7(Definition D) {
3      variables
4        Symbol S, S';
5      begin
6        %% for each recursive definition of a comma-separated
7        %% right-associative list ...
8        foreach Production P:<S> "," <S'> -> <S> in <D> do {
9          <D> := remove Production:<P> from <D>;
10         <D> := remove Production:<S'> -> <S> from <D>;
11
12         if not(extract Production:... -> <S> from <D>) then {
13           %% there are no more productions defining <S>
14           <D> := insert { <S'> "," }+ -> <S> into <D>;
15         } else
16           %% there were more productions defining <S>
17           %% we cannot be sure about the transformation so fail
18           fail;
19       };
20       <Step7> := <D>;
21     end;
22   };
```

The '`foreach`' loop will iterate over all the productions recursively defining a comma-separated list. If one such production is found, it is removed together with the production defining a single item in the list.

The '`if`' statement test whether the list defining productions were the only productions for the sort. If so, a new production defining the list in SDF2 syntax is inserted into the definition. If there are other defining productions for the sort, we can not be sure that the changes we make keep the original semantics. In this case fail the '`if`'-statement through the use of the '`fail`' statement on line 18. This will in turn fail the current iteration of the '`foreach`' loop, discarding any changes made in this iteration so far. The '`foreach`' loop will continue with the next iteration, if any.

5.3.8 "renamed all non-terminals from 'LogicalAndExpression' up-to and not including 'UnaryExpression' to 'LogicalOrExpression'"

This is almost a literal implementation of the *h*FST script in GTL. The inclusion of the attribute in the production was done here by replacing the original production with a new one based on the old.

GTL

```
functions
  Definition &Step8(Definition D, Sort S) {
    variables
      Symbol Y1, Y2;
      Literal L1, L2;
      Group G1, G2;
    begin
      %% create the first group of priorities...
      foreach Production P:<Y1> <L1> <S> -> <Y1> in <D> do {
        <G1> := include Production:<S> <L2> <Y2> -> <S> in <G1>;
        <D> := replace Production:<P>
                 with Production:<Y1> <L1> <S> -> <Y1> {left} in <G1>;
      };

      %% create the second group of priorities...
      foreach Production P:<S> <L2> <Y2> -> <S> in <D> do {
        <G2> := include Production:<Y1> <L1> <S> -> <Y1> in <G2>;
        <D> := replace Production:<P>
                 with Production:<S> <L2> <Y2> -> <S> {left} in <G1>;
      };

      %% include the context-free priority ...
      <D> := include Grammar:
               context-free priorities
                 <G1> > <G2>
             in <D>;

      %% rename the sort ...
      <D> := replace Sort:<S> with Sort:LogicalOrExpr in <D>;

      %% remove the trivial injection ...
      <D> := exclude Production:LogicalOrExpr -> LogicalOrExpr from <D>;

      <Step8> := <D>;
    end;
  };
```

5.3.9 "added lexical restriction on 'D+'"

This step was implemented as part of step 5. It is the first insert in the function.

5.3.10 "removed 'UnaryOperator' non-terminal by inlining the alternative operators"

The implementation of the *h*FST script:

GTL
```
functions
  Definition &Step10(Definition D) {
    <D> := insert Production:<( "&" CastExpr -> UnaryExpr;
                                "*" CastExpr -> UnaryExpr;
                                "+" CastExpr -> UnaryExpr;
                                "-" CastExpr -> UnaryExpr;
                                "~" CastExpr -> UnaryExpr;
                                "!" CastExpr -> UnaryExpr )> into <D>;
    <D> := remove Production:UnaryOperator CastExpr -> UnaryExpr from <D>;
    <Step10> := <D>;
  };
```

Note that the productions are inserted as a list of productions, this implies that they will be grouped together in the resulting definition.

5.3.11 "removed 'AssignmentOperator' non-terminal by inlining the alternatives"

Since this type of transformation was done "manually" as part of step 10, a generic '&Unfold_Many' function was created in *h*FST to take care of both transformations.

Here is its implementation in GTL:

GTL
```
functions
  Definition &Unfold_Many(Definition D, Sort S) {
    variables
      Sort S';
      Symbol S1,S2,S3:...;

    begin
      %% for each definition using the sort <S> ...
      foreach Production P1:>S1< <S> >S2< -> >S'< in <D> do {
        %% remove the production ...
        <D> := remove Production:<P1> from <D>;
        %% add a production for each definition of the sort <S>
        %% for the sort <S'>
        foreach Production P2:>S3< -> <S> in <D> do {
          <D> := insert Production:<S1> <S3> <S2> -> <S'> into <D>;
        };
      };
      %% after unfold, remove all productions defining <S>
      <D> := remove Production:... -> <S> from <D>;
      <Unfold_Many> := <D>;
    end;
```

```
  };

  Definition &Step11(Definition D) {
    begin
      <D> := &Unfold_Many(<D>, UnaryOperator);
      <D> := &Unfold_Many(<D>, AssignmentOperator);
      <Step11> := <D>;
    end;
  };
```

5.3.12 "renamed 'LogicalOrExpression' to 'BasicExpression'"

GTL

```
functions
  Definition &Step12(Definition D) {
    begin
      <D> := replace Sort:LogicalOrExpr with Sort:BasicExpr in <D>;
      <Step12> := <D>;
    end;
  };
```

5.3.13 "renamed 'PostfixExpression' to 'PrimaryExpression' and removed the trivial injection"

GTL

```
functions
  Definition &Step13(Definition D) {
    begin
      <D> := replace Sort:PostfixExpr with Sort:PrimaryExpr in <D>;
      <D> := remove Production:PrimaryExpr -> PrimaryExpr from <D>;
      <Step13> := <D>;
    end;
  };
```

5.3.14 "folded empty and non-empty use of 'ArgumentList' in 'PrimaryExpression'"

GTL

```
functions
  Definition &Step14(Definition D) {
    begin
      <D> := replace Production:{ AssignmentExpr "," }+ -> ArgumentExprList
             with Production:{ AssignmentExpr "," }* -> ArgumentExprList
             in <D>;
      <D> := remove Production:PrimaryExpr "(" ")" -> PrimaryExpr from <D>;
      <Step14> := <D>;
    end;
  };
```

5.3.15 "Introduced separate lexical non-terminals for each type of constant: 'IntegerConstant', 'HexadecimalConstant', 'CharacterConstant' and 'FloatingPointConstant'"

GTL

```
functions
 Definition &Step15(Definition D) {
    begin
      <D> := remove Production:<{
                [0] [xX] H+ IS* -> Constant,                  %% Hexadecimal
                [0] D+ IS* -> Constant,                       %% Integer
                D+ IS* -> Constant,                           %% Integer
                [\'] ([\\]~[] | ~[\\\'] )+ [\'] -> Constant,  %% Character
                D+ E FS? -> Constant,                         %% FloatingPoint
                D* [\.] D+ E? FS? -> Constant,                %% FloatingPoint
                D+ [\.] D* E? FS? -> Constant                 %% FloatingPoint
              }> from <D>;

      <D> := insert Grammar:
                lexical syntax
                  [0] [xX] H+ IS* -> HexadecimalConstant
                  [0] D+ IS* -> IntegerConstant
                  D+ IS* -> IntegerConstant
                  [\'] ([\\]~[] | ~[\\\'] )+ [\'] -> CharacterConstant
                  D+ E FS? -> FloatingPointConstant
                  D* [\.] D+ E? FS? -> FloatingPointConstant
                  D+ [\.] D* E? FS? -> FloatingPointConstant
              into <D>;

      <D> := insert Grammar:
                context-free syntax
                  HexadecimalConstant -> Constant
                  IntegerConstant -> Constant
                  CharacterConstant -> Constant
                  FloatingPointConstant -> Constant
              into <D>;

      <Step15> := <D>;
    end;
  };
```

The '`remove`' uses a set-pattern so that wherever the productions may reside in the original definition they are each removed. Both '`insert`' statements insert a grammar into the definition. This is to keep the productions together as well as introduce the first insertion as a `lexical syntax` grammar instead of the default `context-free syntax` grammar.

5.3.16 "Removed production '[0] D+ IS* -> IntegerConstant' because 'D' includes '[0]' and the production 'D+ IS* -> IntegerConstant' exists too"

This transformation is a simple removal of a production.

GTL
```
functions
  Definition &Step16(Definition D) {
    begin
      <Step16> := remove Production:[0] D+ IS* -> IntegerConstant from <D>;
    end;
  };
```

5.4 Unlisted Transformations Encountered

This section provides scripts in GTL for the transformations not listed in the original adaptation of the C grammar.

5.4.1 Problems encountered during step 2.4.1

The following is a generic script to add sort declarations for all the defined sorts in a grammar. It assumes that no sort declarations are present.

GTL
```
functions
  Definition &Unlisted1(Definition D) {
    variables
      Symbol S;
      Grammar SG;

    begin
      %% process each defined sort
      foreach Production:... -> <S> in <D> do
        if not(select <S> from <SG>) then
          insert <S> into <SG>;
        else
          ; %% otherwise do nothing since the sort is already declared

      %% now add the sort declarations to the definition
      <Unlisted1> := insert <SG> into <D>;
    end;
  };
```

5.4.2 Problems encountered during step 2.4.2

The following script deals with each of the listed transformations:

GTL
```
functions
  Definition &Unlisted2(Definition D) {

    begin
      <D> := remove -> RANGE from <D>;

      <D> := replace -> ELIPSIS with "..." -> ELIPSIS in <D>;

      <D> := remove -> TYPENAME from <D>;
      <D> := remove TYPENAME -> TypeSpecifier from <D>;

      <D> := replace -> IDENTIFIER with L ( L | D )* -> IDENTIFIER in <D>:

      <D> := replace Sort:IDENTIFIER with Sort:Identifier in <D>;
    end;
  };
```

5.4.3 All sorts that ended in ‘‘Expr’’ were changed to end in ‘‘Expression’’

Since the part of the sort-names that needs to be replaced is known, this transformation can be handled by a single ‘replace’-statement.

First a sort is matched with ‘<[<S1> "Expr"]>’. Since the variable ‘S1’ has type Sort and pattern ‘...’ the whole pattern will match all sorts ending in ‘Expr’. The part of the sort-name before ‘Expr’ is stored in the variable ‘S1’.

Any match is replaced by a new sort-name formed by the contents of the variable ‘S1’ followed by a new ending, namely ‘Expression’.

GTL
```
functions
  Definition &Unlisted3(Definition D) {
    variables
      Sort S1:...;
    begin
      <Unlisted3> := replace Sort:<[ <S1> "Expr" ]>
                     with Sort:<[ <S1> "Expression" ]> in <D>;
    end;
  };
```

5.4.4 The sort ‘‘Constant’’ was unfolded, eliminating it from the grammar

For this unfolding transformation the ‘&Unfold_Many’ function part of step 11 is used.

GTL
```
functions
  Definition &Unlisted4(Definition D) {
    begin
      <Unlisted4> := &Unfold_Many(<D>, Constant);
```

```
5     end;
6   };
```

5.4.5 The sort "`ConditionalExpression`" was renamed to 'ConstantExpression'

GTL

```
1  functions
2    Definition &Unlisted5(Definition D) {
3      begin
4        <Unlisted5> := replace Sort:ConditionalExpression
5                       with Sort:ConstantExpression
6                       in <D>;
7      end;
8    };
```

5.5 Conclusions

The conclusions drawn from the implementation of the case study in *h*FST were used to create GTL.

The implementation of those transformations in GTL show that the extended functionality of GTL opens up possibilities for more generic scripting solutions through the use of functions, variables and patterns. The '`&Unfold_Many`' function which can unfold a sort that has more than one definition is an example of that.

However, there is a down side. Since GTL is less restrictive in the sense that is performs almost no implicit grammar consistancy checking. Tests like grammar equivalence and others are available in GTL, but they have to be envoked manually by the programmer as part of the script. FST functions like '`preserve`' check whether a definition introduced as a replacement for another is structurally equal to that other definition. In GTL that check would have to be done explicitely as part of the script. However, most, if not all, of the FST functions can be implemented in GTL *with* the checks in place. Some of these implementations can be viewed in appendix B, p.115. The implementation for '`&Preserve`' is at section B.3.1, p.120.

GTL syntax tries to adhere as closely as possible to the syntax of SDF in that both the SDF constructs and GTL patterns represent or match SDF constructs directly. As such, the language GTL focusses more on the syntax of SDF specifications, whereas FST focusses more on the semantics side by providing functions like '`definition of ...`', and allowing the programmer to focus on a sort, which implicitly includes its declaration and all its definitions.

It is up to the programmer which approach of the two is preferred, however, appendix B, p.115 shows that it is for the most part possible to implement FST in GTL, after which the programmer could decide to use those functions in building transformations.

Chapter 6

Conclusion

When work was started on GTL many months ago, a number of goals were already stated even before FST or the GDK were studied or the case study was performed. The primary goals were to create a language that would be capable of transforming any and all parts of an SDF specification through a clear interface, while maximizing the maintainability of both the scripts in GTL as well as the language GTL itself. The former should of course be a basic premise for any language, and the latter is nessesary to allow GTL to be adapted to future versions of SDF. The fact that the implementation of FST, which was specifically targeted to SDF, is no longer valid for the current version of SDF amplifies this.

The GDK is an implementation of the same principles as used in the implementation of FST. The transformation language part of the kit is also called FST. However, it employs its own specification language called LLL. Although an SDF specification can be transformed into it for use with the GDK, a lot of the SDF specific constructs and semantics is lost in that process. It is because of this that we focussed primarily on the FST tailored to SDF for the case study. In the following, any reference to FST implies this SDF specific implementation.

6.1 The Core

It was decided that the best language to use to implement GTL would be ASF+SDF itself. This is primarily because the specification of SDF is maintained in SDF itself, but also because creating languages is, in effect, what ASF+SDF is intended for.

For the setup of the core of GTL we looked at FST. FST employs a separate traversal of an SDF specification for most of its functions. This has the advantage that new functions to be added to the language have little dependancy upon the rest of the implementation, apart from the primitive functions. The disadvantage is that a change in the specification of SDF implies a possible change in the traversal of an SDF grammar, meaning that all functions employing a traversal would need to be adapted.

Looking at the implementation of FST it was apparent that maintainability of the GTL sources means creating a clear devide between the semantics and syntax of SDF, and the

syntax and semantics of GTL. To this end, each SDF construct is implemented in it's own module which encompasses its definition as well as the semantics needed to traverse it. This implies that a moderate change to the specification of SDF would nessesitate an adaptation only to the module defining the construct in question, as well as the modules defining its parent constructs. The resulting specification tree closely resembles the tree structure of the current version of SDF, version 2. See appendix A.3, p.113.

With the minimization of traversals in the implementation of GTL in mind, a set of primitive functions was envisioned that can be used as a basis for others. Since most if not all of the functionality needed in GTL revolves around selecting, matching or merging grammars, these are the functions that are created to do all the traversing of a specification. They should be the only ones to have to be defined in each of the modules declaring and defining an SDF construct. All of the other functions can be expressed using some or all of these primitive functions.

Having these three as the only functions also implied that we needed a generic way of selecting a specific SDF construct, whether for matching or to focus on it for further action. Using patterns was the obvious solution.

Each module defining an SDF construct creates a pattern for it to be used in GTL pattern matching. This pattern should mimic the SDF definition to the letter. We now have a tree of patterns which can match an SDF specification in the same way the SDF specification itself can. The difference is that we can now introduce patterns specific to GTL to specify things like "exactly one" or "one of these", which can be used as any of the SDF constructs. Variables typed to a specific SDF construct could now be part of a pattern, as was shown to be required in the case study.

On the basis of this core, the GTL language was built.

6.2 The Language

The core is flexible enough to have allowed us to implement an extended version of FST, but then the resulting scripts would lose out in maintainability. What the case study showed was that we needed a more modular language, including functions and libraries of functions. The language also needs to be more removed from the semantics of SDF as was already concluded.

To make GTL easy to program its syntax was kept as close as possible to the syntax of SDF. The pattern matching engine already implies this since it closely mimics the specification of SDF, but the layout of a GTL program or library was envisioned to do the same. A GTL program contains a number of sections where the declarations of the variables, patterns and functions reside. The section containing the actual transformations was modeled after the language pascal, using a begin ... end statement. To ensure that the GTL constructs cannot be mistaken for an SDF construct each of them was given a syntax that would not be recognized by the SDF specification. Functions must start with an ampersand, variables are placed within angle brackets, and user defined patterns start with a hash-sign ('#').

The semantics of the various GTL constructs are not much different from most other

languages. Functions can be called directly of be used as a pattern or an SDF construct depending on where it is referenced, just like variables and user defined patterns. There is a difference however in that a function in GTL can fail. This is because for instance a select statement does not actually select anything simply because its pattern is not part of the specification being matched against. It was decided that a failure in one of the statements would fail the encompassing function, which would in turn fail the function-call andsoforth, just like in FST. This implied that any function can also be used as a condition check. An if-statement can be used as a try-catch block to ensure continued evaluation of a program.

6.3 The Implementation

Due to unforseen and unintended problems with the ASF+SDF meta-environment, we were not able to achieve a fully working version of a GTL interpreter. It was too late to redesign the implementation to circumvent these problems, although that would probably have solved the issue in exchange for losing some of the modularity and maintainability of the implementation. A few very specific and low level programs in GTL were evalutated successfully however, and we are confident that a full implementation can be achieved. This does imply however that all of the examples of GTL programs and functions are untested. They should be looked on as reference implementations.

6.4 Using GTL

The re-implementation of the case study in GTL shows that the functionality deemed nessesary in the previous case study is part of GTL. It also shows that the case study itself is not a very good example of the capabilities of GTL, since most of the transformations are very specific to the case study. A few however, like the '`&Unfold_Many`' function, can be used with any SDF specification. They show more of the generic transformations that GTL is capable of.

Most of the functionality present in GTL was rarely used in the case study. Things like the case-statement or the lexical pattern were not even nessesary, but this is mostly because they are intended to be useful for more generic transformations, i.e. transformations that can be performed on any SDF specification. The '`&toCamelCase`' function given in appendix C, p.125 is an example of such a transformation on sorts.

6.5 Future Work

Allowing patterns as arguments

This would expand the functionality that can be expressed by user defined functions. Functions can be created that employ selection using a given pattern.

This would mean that for instance internal functions '`narrow`' and '`widen`', which take a pattern as an argument, could be implemented as a used defined function.

Function Overloading

Allow functions to be redefined using different arguments. This will allow the creation of a function that can for instance operate on either a `Production` or a `Module`.

Error Reporting

GTL improves a lot on the functionality of FST, but there remain a few things that would be desirable to implement. First of all is good error reporting, not only from the language itself but also reporting specific to the script, which would have to be supplied by the programmer.

User Interface Generation

Since with GTL the input and output of a program is specified with pattern and type, it is easy to create a user interface for the script. The standard output would be an SDF construct of the same type and pattern as the primary function, and the input is based on the arguments given to the same function.

Transforming both SDF and ASF

In the long term, extending GTL to also be able to transform ASF rewrite rules would open up a whole new set of possibilities. Changes made in a specification could be used to determine which rewrite rules are affected and need to be adapted. Also, when selecting a subtree to copy and merge into another specification, relevant rewrite rules could also be incoporated in the merge. A lot of work on the semantics of this has yet to be done however, but using the same syntactic approach as with GTL at this stage, the actual functionality could be left to be implemented in GTL itself as with the implementation of FST.

Appendix A

Diagrams

A.1 Pattern Structure

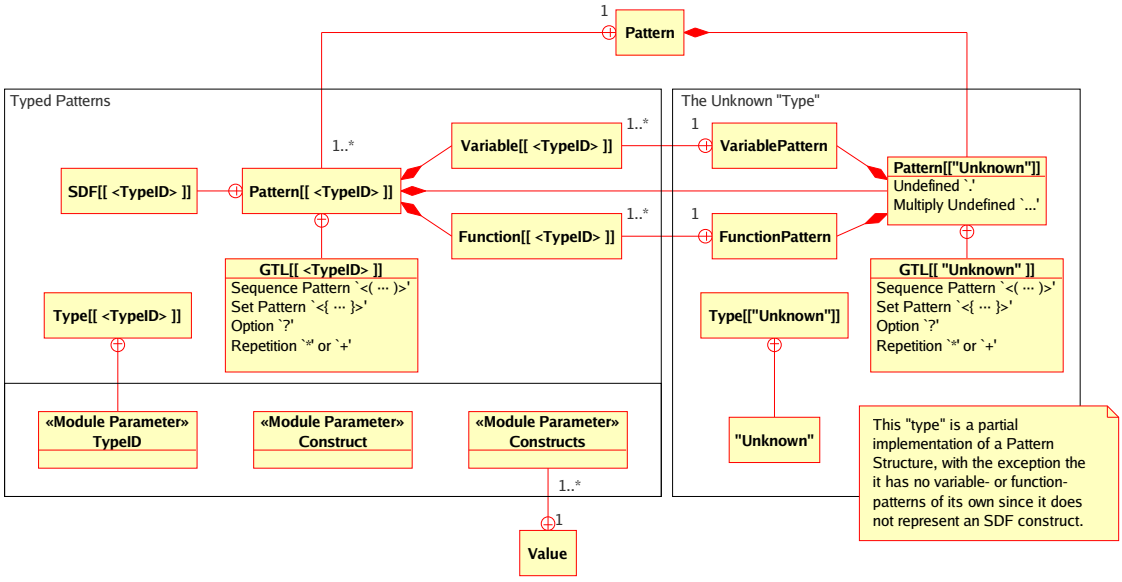
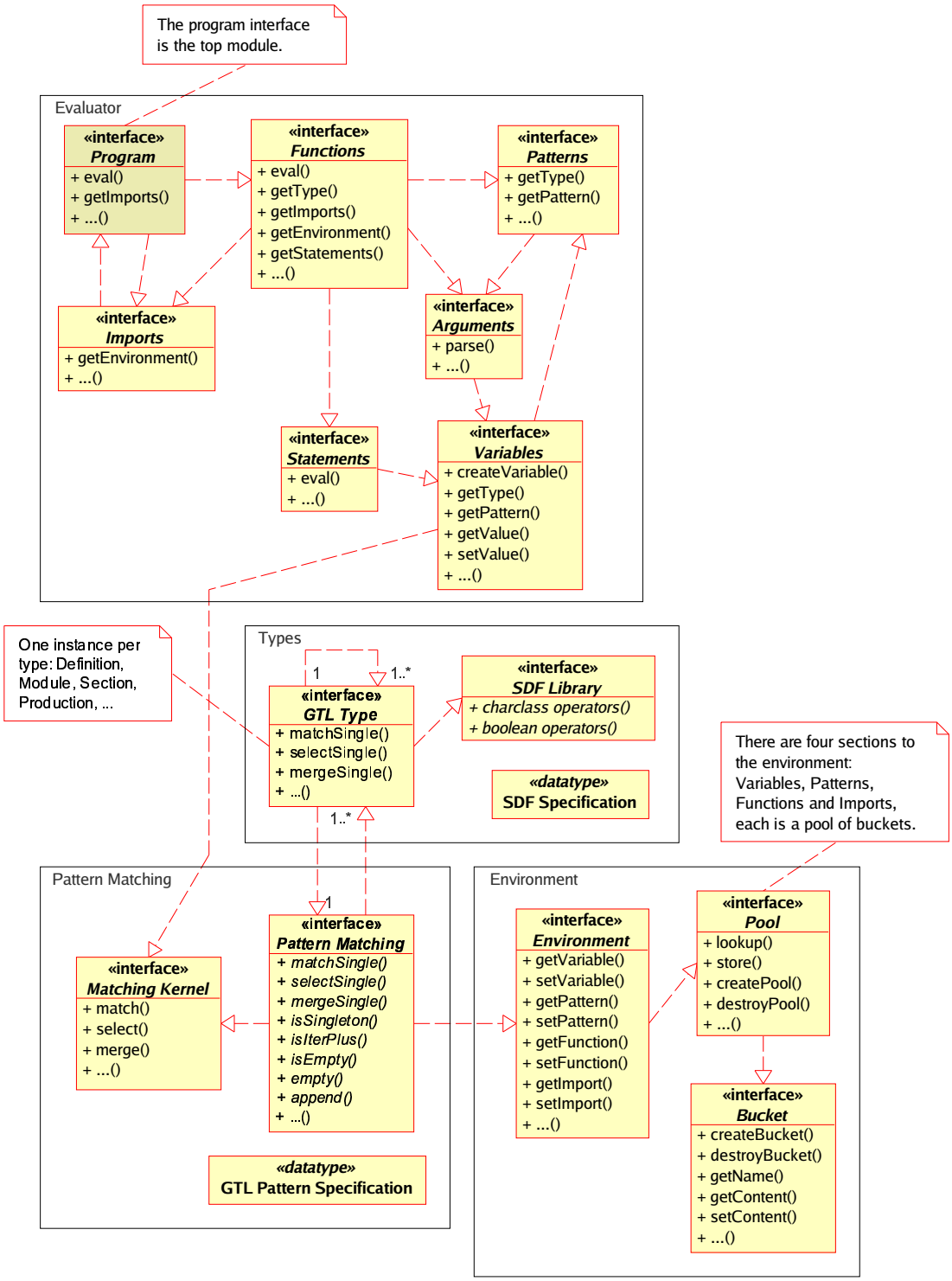
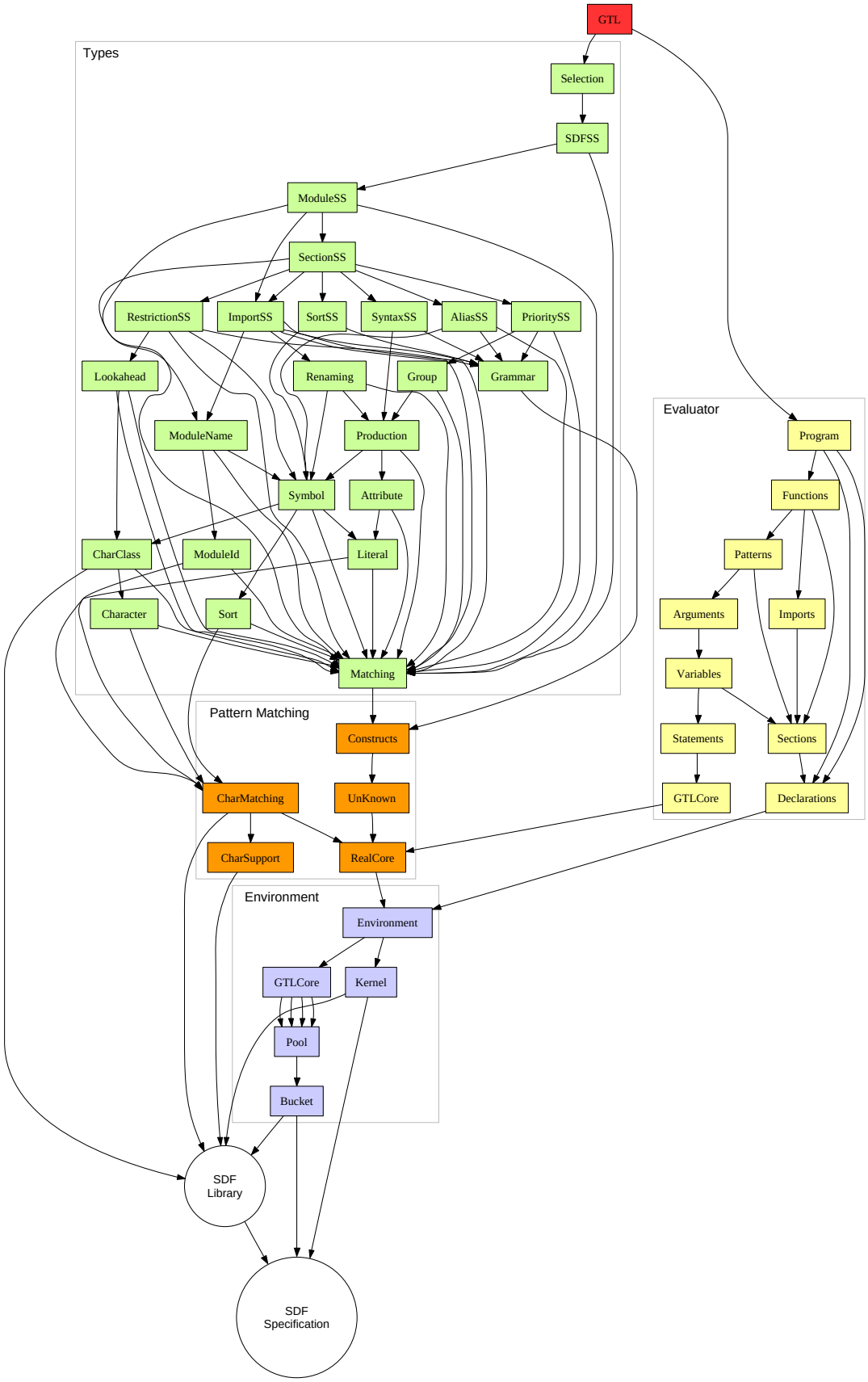


Diagram: Pattern Structure Page 1

A.2 GTL Implementation Interfaces



A.3 GTL Implementation Dependency Graph



Appendix B

Implementing FST in GTL

B.1 Introduction

Implementing FST in GTL is not that hard in most cases. However, we need to take care to mimic the semantics of FST as closely as possible, given the difference in the implementation of the focus and other changes.

The implementation given here is complete in that it represents the framework as implemented in GTL. This framework however does not provide functions to deal with SDF specific constructs like priorities, restrictions or modularization. These are not provided here either, since that would amount to extending the FST, which is exactly what GTL was created to do. This implementation shows that the functionality as provided by FST can be expressed in GTL as well.

B.2 The Transformation Framework

The functions described in this chapter mimic the original framework, rather than the implementation of FST. This framework is described in [7]. Since FST is also based on this framework, most of the relevant functions will be incorporated.

There are a few points where the FST implementation in GTL differs from the described framework.

Focus
: The focus as described in the framework is a global one. All functions operate within that focus. There can only be one focus at a time. In GTL, each (extended) SDF construct has a focus, which is part of the selection it represents. There is no global focus, so it has to be presented as an argument to *all* functions. This means that all the functions must take care to preserve the focus, unless they were intended to change it.

Function Overloading
: FST has overloaded functions, which can work on either sorts or modules, like the '`&Add`' and '`&Sub`' functions. GTL does not provide this functionality yet (see sec-

tion 6.5, p.110). In this implementation we will adhere to the original framework and ignore modules, although separate functions could be created to rectify this.

Pattern Arguments

GTL functions can only accept SDF constructs as arguments. In the case of the '`&Focus`' operator, this means that it can only focus on literal constructs, not use patterns, since these cannot be passed to the function (see section 6.5, p.109).

B.2.1 Primitives

These are the primitive functions, all other functions are based on these.

GTL

```
functions
  Definition &Id(Definition F) {
    begin
      %% id merely returns the given selection
      <Id> := <F>;
    end;
  };

  Definition &Fail(Definition F) {
    begin
      %% this constraint will fail
      "true" == "false";
      %% no result value is needed since this position will never be reached,
      %% in FST, the result value is undefined
    end;
  };

  Definition &Reset(Definition F) {
    begin
      %% reset undoes the selection, the focus will be empty, the remainder will
      %% contain the entire definition
      <Reset> := invert(implode(<F>));
    end;
  };

  Definition &Add(Definition F, Production P) {
    begin
      %% insert will add the given production to the focus of <F>
      <Add> := insert <P> into <F>;
    end;
  };

  Definition &Sub(Definition F, Production P) {
    begin
      %% this will remove the given production from the selection <F>
      %% <P> must be in the focus of <F>
      <Sub> := remove <P> from <F>;
    end;
```

```
39  };
40
41  Definition &Substitute(Definition F, Sort N, Sort N') {
42    begin
43      %% the GTL function replace does the same, for all possible
44      %% instances of <N>
45      <Substitute> := replace <N> with <N'> in <F>;
46    end;
47  };
48
49  Definition &Replace(Definition F, Symbol P, Symbol P') {
50    begin
51      %% in FST, this function replaces one phrase with another.
52      %% A phrase is a sequence of symbols part of a production.
53      %% The GTL replace function can be used in the same way.
54      <Replace> := replace <P> with <P'> in <F>;
55    end;
56  };
```

B.2.2 Combinators

The FST combinators like the conditional ('`if`') and the sequential composition are already part of GTL, albeit in a different form.

The FST '`focus`' operator wil be expressed using the GTL widen operator. Note that, since GTL patterns cannot be given as arguments, this operator does not adhere to the FST implementation. In cases where a pattern is required, a GTL operator must be used.

The '`!`' operator will be given the name '`&Effectively`', since the exclamation mark cannot be part of a function name.

GTL

```
1  functions
2    Definition &Focus(Definition F, Sort N:<( . )>+ ) {
3      begin
4        %% first reset the current focus
5        <F> := &Reset(<F>);
6        %% for each sort in the given list, widen the focus with its defining
7        %% productions
8        foreach Sort S in <N> do {
9          <F> := widen <F> using Production:... -> <S>;
10       };
11       <Focus> := <F>;
12     end;
13   };
14
15   Definition &Effectively(Definition F, Definition F') {
16     begin
17       %% <F'> holds the result of a function-call. It is this result that must
18       %% be different from <F>.
19       not(<F> == <F'>);
```

```
20      %% <F> and <F'> differ. Reproduce <F'> as the result
21      <Effectively> := <F'>;
22    end;
23  };
```

B.2.3 Constraints

Since in GTL, functions must have a result value, the FST constraint functions are implemented to take and return a definition. The definition given as an argument is used in some cases, like with the functions '`&Defined`' and '`&Used`'. The return value of each of the constraint functions is never used by the programmer. Only the success or failure of the function is of importance.

The FST function '`not`' is already a part of GTL, and is not included here.

GTL

```
1  functions
2    Definition &Defined(Definition F, Sort N) {
3      begin
4        %% 'extract' will produce a list of productions defining <N>.
5        %% If the list has one or more members, the sort is "defined"
6        <(.)>+ == extract Production:... -> <N> from <F>;
7        <Defined> := <F>;
8      end;
9    };
10
11   Definition &Used(Definition F, Sort N) {
12     begin
13       %% 'extract' will produce a list of productions using <N> in their
14       %% definition. If the list has one or more members, the sort is "used"
15       <(.)>+ == extract Production:... <N> ... -> . from <F>;
16       <Used> := <F>;
17     end;
18   };
19
20   Definition &Bottom(Definition F, Sort N) {
21     begin
22       %% the 'bottom' constraint expressed using other constraints
23       not(&Defined(<N>)) && &Used(<N>);
24       <Bottom> := <F>;
25     end;
26   };
27
28   Definition &Top(Definition F, Sort N) {
29     begin
30       %% the 'top' constraint expressed using other constraints
31       &Defined(<N>) && not(&Used(<N>));
32       <Top> := <F>;
33     end;
34   };
```

```
35
36  Definition &Fresh(Definition F, Sort N) {
37    begin
38      %% the 'fresh' constraint expressed using other constraints
39      not(&Defined(<N>)) && not(&Used(<N>));
40      <Fresh> := <F>;
41    end;
42  };
43
44  Definition &Focused(Definition F, Sort N) {
45    begin
46      %% the 'focused' constraint expressed using other constraints
47      not(&Defined(<N>)) && &Used(<N>);
48      <Focused> := <F>;
49    end;
50  };
51
52  Definition &Covers(Definition F, Symbol P:... , Symbol P':...) {
53    begin
54      %% the FST 'covers' function has a different semantics from the covers
55      %% operator '(=)', part of GTL, so it is not used.
56      <P> =) <P'>;
57      <Covers> := <F>;
58    end;
59  };
60
61  Definition &Equiv(Definition F, Symbol P:... , Symbol P':...) {
62    begin
63      %% the FST 'equiv' function has the same semantics as the GTL
64      %% operator '(=)'
65      <P> (=) <P'>;
66      <Covers> := <F>;
67    end;
68  };
```

B.2.4 Symbolic Operands

Symbolic operands in FST can be used as patterns with the FST '`focus`' operator. These are covered by using GTL patterns. A few are listed here using user-defined patterns. Note that patterns cannot be used with the '`&Focus`' function presented in this chapter.

In FST, only the operands '`all sorts`' and '`all modules`' were implemented. These have been extended here with a few other operands.

GTL

```
1 patterns
2   Grammar #all_aliases = aliases ... ;
3   Grammar #all_restrictions = <{ restrictions ..., lexical restrictions ...,
4                                  context-free restrictions ... }> ;
5   Grammar #all_priorities = <{ priorities ..., lexical priorities ...,
```

```
                                context-free priorities ... }> ;
  Grammar #all_sorts = sorts ... ;
  Grammar #all_imports = imports ... ;
  Grammar #all_syntax = <{ syntax ..., lexical syntax ...,
                           context-free syntax ... }> ;
  Grammar #all_grammars = <{ #all_aliases, #all_restrictions, #all_priorities,
                             #all_sorts, #all_imports, #all_syntax }> ;
  Section #all_sections = <{ exports ..., hiddens ... }> ;
  Module #all_modules = module ... ;

  Production #definition_of(Sort S) = ... -> <S> ;
```

The symbolic operand '`definition of`' must also be implemented as a function returning the phrase used in the definition of its argument sort. It is used as such in the function '`&Unfold`' in section B.3.1, p.120.

GTL

```
functions
  Symbol &Definition_Of(Definition F, Sort N) {
    begin
      %% The equality test with the dot pattern ensures that only one
      %% definition of <N> can exist.
      . == extract >Definition_Of< -> <N> from <F>;
    end;
  };
```

B.3 The Operator Suite

Here, the main functions of the operator suite are presented. They are expressed using the primary functions already implemented. The implementation for each function mimics the semantics as given in [7]. The functions are divided in three groups, namely refactoring, construction and destruction.

B.3.1 Refactoring

GTL

```
functions
  Definition &Preserve(Definition F, Symbol P:..., Symbol P':...) {
    %% original: preserve P in F as P'
    begin
      &Equiv(<F>, <P>, <P'>);
      <Preserve> := &Effectively(<F>, &Replace(<F>, <P>, <P'>));
    end;
  };

  Definition &Fold(Definition F, Symbol P:..., Sort N) {
    %% original: fold P in F to N
    begin
```

```
      <Fold> := &Preserve(&Introduce(<F>, <N>, <P>), <P>, <N>);
    end;
  };

  Definition &Unfold(Definition F, Sort N) {
    %% original: unfold N in F
    begin
      <Unfold> := &Preserve(<F>, <N>, &Definition_Of(<F>, <N>));
    end;
  };

  Definition &Introduce(Definition F, Sort N, Symbol P:...) {
    %% original: introduce N as P
    begin
      &Fresh(<F>, <N>);
      <Introduce> := &Add(<F>, P -> N);
    end;
  };

  Definition &Eliminate(Definition F, Sort N) {
    %% original: eliminate N
    begin
      <Eliminate> := &Reject(<F>, <N>);
      &Fresh(<Eliminate>, <N>);
    end;
  };

  Definition &Rename(Definition F, Sort N, Sort N') {
    %% original: rename N to N'
    begin
      &Fresh(<F>, <N'>);
      <Rename> := &Effectively(<F>, &Substitute(<F>, <N>, <N'>));
    end;
  };
```

B.3.2 Construction

GTL

```
functions
  Definition &Generalise(Definition F, Symbol P:..., Symbol P':...) {
    %% original: generalise P in F to P'
    begin
      &Covers(<F>, <P'>, <P>);
      <Generalise> := &Effectively(<F>, &Replace(<F>, <P>, <P'>));
    end;
  };

  Definition &Include(Definition F, Symbol P:..., Sort N) {
    %% original: include P for N
    begin
```

```
      &Defined(<F>, <N>);
      <Include> := &Effectively(<F>, &Add(<F>, <P> -> <N>));
    end;
  };

  Definition &Resolve(Definition F, Sort N, Symbol P:...) {
    %% original: resolve N as P
    begin
      &Bottom(<F>, <N>);
      <Resolve> := &Add(<F>, <P> -> <N>);
    end;
  };

  Definition &Unify(Definition F, Sort N, Sort N') {
    %% original: unify N to N'
    begin
      &Bottom(<F>, <N>);
      not(&Fresh(<F>, <N'>));
      <Unify> := &Replace(<F>, <N>, <N'>);
    end;
  };
```

B.3.3 Destruction

GTL

```
functions
  Definition &Restrict(Definition F, Symbol P:..., Symbol P':...) {
    %% original: restrict P in F to P'
    begin
      &Covers(<F>, <P>, <P'>);
      <Restrict> := &Effectively(<F>, &Replace(<F>, <P>, <P'>));
    end;
  };

  Definition &Exclude(Definition F, Symbol P:..., Sort N) {
    %% original: exclude P from N
    begin
      <Exclude> := &Effectively(&Sub(<F>, <P> -> <N>));
      &Defined(<F>, <N>);
    end;
  };

  Definition &Reject(Definition F) {
    %% original: reject F
    begin
      <Reject> := &Reset(<F>);
    end;
  };

  Definition &Separate(Definition F, Sort N, Sort N') {
```

```
26      %% original: separate N in F as N'
27      begin
28        &Fresh(<F>, <N'>);
29        <Separate> := &Effectively(<F>, &Replace(<F>, <N>, <N'>));
30        not(&Fresh(<F>, <N>));
31      end;
32    };
33
34    Definition &Delete(Definition F, Symbol P:...) {
35      %% original: delete P in F
36      begin
37        not(&Covers(<P>, ));
38        <Delete> := &Effectively(<F>, &Replace(<F>, <P>, ));
39      end;
40    };
41
```

Appendix C

Example of a GTL Program

C.1 '&toCamelCase'

Here is a program that performs the transliteration from dash-separated to CamelCase for a given sort.

GTL

```
program
Sort &ToCamelCase(Sort S) {
  functions
    Sort &Capitalize(Sort S) {
      variables
        Sort Head:<[ [A-Za-z] ]>;
        Sort Tail:<[ [A-Za-z]+ ]>;

      begin
        if <[ <Head> <Tail> ]> := <S>; then
          <Capitalize> := <[ &ToUpper(<Head>) &ToLower(<Tail>) ]>;
        else
          <Capitalize> := &ToUpper(<S>);
      end;
    };

    Sort &ToUpper(Sort S) {
      variables
        Sort Head:<[ [A-Za-z] ]>;
        Sort Tail:<[ [A-Za-z\-\_]+ ]>;

      begin
        if <[ <Head> <Tail> ]> := <S>; then
          %% the sort is longer than one character...
          <ToUpper> := <[ &ToUpper(<Head>) &ToUpper(<Tail>) ]>;
        else
          %% the sort is one character long
          <ToUpper> := case <S>: { a:A; b:B; c:C; d:D; e:E; f:F;
                                   g:G; h:H; i:I; j:J; k:K; l:L;
```

```
30                                  m:M; n:N; o:O; p:P; q:Q; r:R;
31                                  s:S; t:T; u:U; v:V; w:W; x:X;
32                                  y:Y; z:Z; default:<S> };
33      end;
34    };
35
36    Sort &ToLower(Sort S) {
37      variables
38        Sort Head:<[ [A-Za-z] ]>;
39        Sort Tail:<[ [A-Za-z\-\_]+ ]>;
40
41      begin
42        if <[ <Head> <Tail> ]> := <S>; then
43          %% the sort is longer than one character...
44          <ToLower> := <[ &ToLower(<Head>) &ToLower(<Tail>) ]>;
45        else
46          %% the sort is one character long
47          <ToLower> := case <S>: { A:a; B:b; C:c; D:d; E:e; F:f;
48                                  G:g; H:h; I:i; J:j; K:k; L:l;
49                                  M:m; N:n; O:o; P:p; Q:q; R:r;
50                                  S:s; T:t; U:u; V:v; W:w; X:x;
51                                  Y:y; Z:z; default:<S> };
52      end;
53    };
54
55  variables
56    Sort Head:<[ [A-Za-z]+ ]>;
57    Sort Tail:<[ [A-Za-z\-\_]+ ]>;
58
59  begin
60    %% match S to a \- or \_ delimited Head and Tail
61    if <[ <Head> [\-\_]+ <Tail> ]> := <S>; then
62      <ToCamelCase> := <[ &Capitalize(<Head>) &ToCamelCase(<Tail>) ]>;
63    else
64      <ToCamelCase> := &Capitalize(<S>);
65  end;
66 };
```

Appendix D

Indices

GTL Constructs

Index

Appendix E

Nomenclature

ASF - *Algebraic Specification Formalism*; a language for rewriting terms parsed using an SDF grammar.

SDF - *Syntax Definition Formalism*; a language for constructing grammars, based on EBNF but differing in syntax and semantics.

ASF+SDF - *Meta-Environment*; an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language. It can be found at: `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment`.

BNF/EBNF - *Backus-Naur Form / Extended Backus-Naur Form*; BNF is a formal notation for specifying the production rules of a syntax devised by John Backus and Peter Naur. EBNF is BNF augmented by the use of regular expressions on the right hand side of productions.

LEX - *A LEXical analyser generator*; helps write programs whose control flow is directed by instances of regular expressions in the input stream. LEX is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

YACC - *Yet Another Compiler Compiler*; provides a general tool for describing the input to a computer program. The YACC user specifies the structures of his input, together with code to be invoked as each such structure is recognized. YACC turns such a specification into a subroutine that handles the input process.

LEX+YACC - *LEX+YACC*; Since the parser generated by Yacc requires a lexical analyzer, it is often used in combination with a lexical analyzer generator, in most cases the Lex program. The IEEE POSIX P1003.2 standard defines the functionality and requirements to both Lex and Yacc.

FST - *Framework for SDF Transformation*; a prototype implementation for SDF of the grammar adaptation framework described in [7]. It cannot be downloaded, but it's homepage is: `http://www.cs.vu.nl/grammarware/fst/`

*h*FST - *hypothetical FST*; a hypothetical language based on FST but employing additional functionality lacking in FST to better handle the various transformations done in the case study. The resulting scripts are not executable, since the language does not actually exist. The language is used to find out what functionality is needed in GTL.

GDK - *Grammar Deployment Kit*; provides tool support for grammar deployment, the process of turning a given grammar specification into a working parser. It is based on firm grammar engineering methods.
The kit can be found at: `http://gdk.sourceforge.net/`

LLL - *Lightweight LL parsing*; should be read as "L-Cube". A simple EBNF-based grammar format that is used in the GDK as the basic specification language. An SDF specification can be transformed into an LLL specification so that the tools that are part of the GDK can be used on it. Afterwards the LLL specification can be transformed back into an SDF specification.

XT - *XT, transformation tools*; a bundle of tools for building program transformation systems. The tools include parser generation, pretty-printing, abstract syntax tree representation, tree transformation, and building and bundling of systems. The toolkit can be found at:
`http://www.program-transformation.org/Tools/WebHome`

GTL - *Grammar Transformation Language*; The language described in this thesis. It provides basic functionality for transforming SDF specifications. It extends on FST in that it provides modularization and functionality specifically geared towards SDF2.

Bibliography

[1] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers, *The Syntax Definition Formalism SDF — Reference Manual. SIGPLAN Notices*, 24(11):43-75, 1989.

[2] E. Visser. *Syntax Definition for Language Prototyping.* PhD Thesis, University of Amsterdam, September 1997.

[3] J. Rekers, *Parser Generation for Interactive Environments.*, PhD thesis, University of Amsterdam, 1992.

[4] M. Tomita, *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems.*, Kluwer Academic Publishers, 1985.

[5] R. Lämmel, G. Wachsmuth, *Transformation of SDF syntax definitions in the ASF+SDF environment*, Electronic Notes in Theoretical Computer Science 44 No.2, 2001.

[6] M.G.J. van den Brand, P. Klint, *ASF+SDF Meta-Environment User Manual, preliminary revision 1.146*, September 20, 2004.
`http://www.cwi.nl/projects/MetaEnv/meta/doc/asfsdfmanual/user-manual.html`

[7] R. Lämmel, *Grammar Adaptation*, In *Proc. Formal Methods Europe (FME'01)*, volume 2021 of *LNCS*, pages 550-570, Springer-Verlag, 2001.

[8] A. Aho & J. Ullman. *Theory of parsing, Translation and Compiling, vol. 1 (Parsing)* in Automatic Computation, Prentice Hall, 1972, chap. 2.6.3, p199.

[9] P. Klint, R. Lämmel, C. Verhoef. *Toward an Engineering Discipline for Grammarware*, ACM Transactions on Software Engineering and Methodology, Vol.14, No. 3, July 2005, Pages 331-380.