Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

ASF+SDF Parsing Tools Applied to ELAN

M.G.J. van den Brand, C. Ringeissen

Software Engineering (SEN)

SEN-R0029 November 30, 2000

Report SEN-R0029
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

ASF+SDF Parsing Tools Applied to ELAN

M.G.J. van den Brand
email: Mark.van.den.Brand@cwi.nl

C. Ringeissen
email: Christophe.Ringeissen@loria.fr

LORIA-INRIA
615 rue du Jardin Botanique, BP 101,
F-54602 Villers-lès-Nancy Cedex, France

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

This paper describes the development of a new ELAN parser using ASF+SDF parsing technology. ASF+SDF and ELAN are two modern rule-based systems. Both systems have their own features and application domains, however, both formalism have user-defined syntax for defining rewrite rules. The ASF+SDF Meta-Environment uses powerful and efficient generic parsing tools, whereas the ELAN parser is based on an Earley parser. Furthermore, the ELAN syntax is "hard-wired" in the parser, which makes adaptations of the syntax cumbersome. The use of ASF+SDF parsing technology makes the ELAN syntax more open and adaptable, however, some features of the ELAN syntax makes the development of a parser a challenging problem.

1998 ACM Computing Classification System: D.2.6, D.3.1, D.3.4, F.4.2
Keywords and Phrases: parsing, rewriting systems, language definition, intermediate format
Note: To appear in *Proceedings of Third International Workshop on Rewriting Logic and its Applications (WRLA'2000)*, 2000
Note: Work carried out under project SEN1.4, ASF+SDF

1. Introduction

This document contains an overview of the work performed in order to improve the parsing technology used in the ELAN environment. ELAN [KKV95, BKK+98a] is a specification language based on rewriting logic [KK98]. Some of the characteristic features of ELAN are rewriting, AC-matching, and strategies to control the non-determinism induced by non-confluent rewrite systems. Hence, AC-matching and strategies are two sources of non-determinism. The specificity of ELAN consists of integrating the two forms of non-determinism plus deterministic rule-based computations in the same environment. The development of ELAN specifications is supported by an environment which contains, among others, a parser, interpreter [KKV95], and compiler [Vit96, MK98].

The ELAN environment can be considered as a monolithic piece of software which is hard to maintain and not really open. The ELAN syntax, for instance, is "hard-wired" in the current implementation of the parser. The first steps to open the system is performed by introducing the REF [BJMR98] and developing a new ELAN compiler [MK97, MK98] which is quite independent of the rest of the system. The compiler interacts with the rest of the system through REF.

A few years ago it was decided to use more generic language technologies when designing and implementing the new ELAN environment. The technology applied in the old ASF+SDF Meta-Environment [Kli93] as well as the new ASF+SDF Meta-Environment [vdBKMO97] was considered as a possible solution to improve the structure and maintainability of the ELAN environment and to make adaptations of the syntax easier. ASF+SDF [vDHK96] is an algebraic specification formalism designed for the definition of the syntax and semantics of (programming) languages, its main application area is in the domain of language prototyping and program transformations. The ASF+SDF Meta-Environment is an integrated programming environment to develop these language definitions and to generate a programming environment given a language definition. Two technical developments of ASF+SDF proved to be very useful for the development of an ELAN environment, namely ATERMS [vdBdJKO00] and the generic parsing technology. The ATERMS format is a generic formalism for the representation of structured information, like (abstract) syntax tree, parse tables, environments, etc. The generic parsing technology consists of a parse table generator and a parser. The parser is a scannerless generalized LR parser (SGLR) [Vis97].

A number of experiments were performed in order to see how the various problems and requirements set by the ELAN language could be solved using ASF+SDF technology. Two aspects of the ELAN were identified for which "ASF+SDF" technology could be useful. First, a new intermediate format for ELAN was designed, based on the ATERMS format, in order to replace the REF in the future. Second, given the parser generator and parsing technology used in the ASF+SDF Meta-Environment a new parser for ELAN was developed. Appendix 3 show some preliminary results on the efficiency of the new ELAN parser.

The paper is organized as follows. In Section 2 an overview of the basic ASF+SDF technology is presented. In Section 3, we discuss the new intermediate format EFIX for ELAN, which is an instance of ATERMS used in the ASF+SDF environment. Then, we outline the new parser developed using ASF+SDF parsing tools (Section 4). The effects on the new syntax are summarized in Section 5, and a complete example is detailed in the old syntax and in the new one (Section 6). In Section 7, we describe the current implementation of the new parser. Eventually, we conclude in Section 8 with future works that will lead to its integration in the ELAN environment.

2. Basic ASF+SDF technology

We will discuss the basic ASF+SDF technology that has been used to develop the new ELAN parser. First, we will discuss the intermediate format, ATERMS, used within the ASF+SDF Meta-Environment. Next, we will give a short overview of the parser generation technology. Finally, we will discuss the parser itself.

2.1 Intermediate format

ATERMS [vdBdJKO00] is a generic formalism to represent structured information like (abstract) syntax trees. It is readable for humans and easy to process for computers. A number of libraries that implement the functionality of creating and manipulating terms provide an API for the ATERMS formalism. These libraries provide functionality to read, write, and manipulate terms. Furthermore, both libraries ensure maximal subterm sharing and automatic garbage collecting when processing terms.

The primary application area of ATERMS is the exchange of information between components of a programming environment, such as a parser, a (structure) editor, a compiler, and so on. The following data are typically represented as ATERMS: programs, specifications, parse tables, parse trees, abstract syntax trees, proofs, and the like. A generic storage mechanism, called *annotation*, accommodates associating extra information that may be of relevance somehow to specific ATERMS under consideration.

Examples of objects that are typically represented as ATERMS are:

- *constants*: `abc`.
- *numerals*: `123`.
- *literals*: `"abc"` or `"123"`.

- *lists*: `[]`, `[1, "abc", 3]`, or `[1, 2, [3, 2], 1]`.
- *functions*: `f("a")`, `g(1,[])`, or `h("1", f("2"), ["a","b"])`.
- *annotations*: `f("a"){[g,g(2,["a"]])}` or `"1"{[l,[1,2]],[s,"ab"]}`.

ATERMS can be qualified as an *open*, *simple*, *efficient*, *concise*, and *language independent* solution for the exchange of data structures between distributed applications.

The concrete syntax of ATERMS is presented in ELAN.

```
module aterm

import global int string;
end

sort ATerms ATermList AFun ATerm Ann;
end

operators global
  @           : ( ATerm ) ATerms;
  @ , @       : ( ATerm ATerms ) ATerms;
  []          : ATermList;
  [ @ ]       : ( ATerms ) ATermList;
  @           : ( int ) AFun;
  @           : ( string ) AFun;
  @           : ( ATermList ) ATerm;
  @           : ( AFun ) ATerm;
  @ ( @ )     : ( AFun ATerms ) ATerm;
  '{' @ '}'   : ( ATerms ) Ann;
  @ @         : ( ATermList Ann ) ATerm;
  @ @         : ( AFun Ann ) ATerm;
  @ ( @ ) @   : ( AFun ATerms Ann ) ATerm;

end

end
```

The ATERMS library is documented extensively in its user manual [dJO99].

2.2 Parser Generator

The parser generator, part of the current ASF+SDF Meta-Environment [vdBKMO97], is one of the components that can be (re-)used to generate parse tables for user-defined syntax in ELAN.

It generates parse tables, suitable for later use by the SGLR parse table interpreter (see Section 2.3) from SDF syntax definitions. The process of generating parse tables consists of two distinct phases. In the first one the SDF definition is normalized to an intermediate, rudimentary, formalism: *Kernel-SDF*. In the second phase this Kernel-SDF is transformed to a parse table.

Grammar Normalization The grammar normalization phase, which derives a Kernel-SDF definition, consists of the following steps:

- A modular SDF specification is transformed into a flat specification.
- Lexical grammar rules are transformed to context-free grammar rules.
- Priority and associativity definitions are transformed to lists of pairs, where each pair consists of two production rules for which a priority or associativity relation holds. The transitive closure of the priority relations between grammar rules is made explicit in these pairs.

Parse Table Generation The actual parse table is derived from the Kernel-SDF definition. To do so, a straightforward SLR(1) approach is taken. However, shift/reduce or reduce/reduce conflicts are not considered problematic, and are simply stored in the table. Some extra calculations are consequently performed to reduce the number of conflicts in the parse table. Based on the list of priority relation pairs the table is filtered; see [KV94] for more details. The resulting table contains a list of all Kernel-SDF production rules, a list of states with the actions and gotos, and a list of all priority relation pairs. The parse table is represented as an ordinary ATERM.

2.3 Scannerless Generalized LR Parsing

Even though parsing is often considered a solved problem in computer science, every now and then new ideas and combinations of existing techniques pop up. SGLR (Scannerless Generalized LR) parsing is a striking example of a combination of existing techniques that results in a remarkably powerful parser.

Generalized LR Parsing for Context-Free Grammars LR parsing [ASU86] is a well-known parsing technique used in many well-known implementations, e.g. LEX/YACC [LS86, Joh86]. LR parsers are based on the shift/reduce principle; a (conflict-free) LR(k) ($k \geq 0$) parse table, containing actions and gotos, is used. A conventional LR parser consist of a scanner, that splits the input stream into tokens, and a parser that processes the tokens and either generates error messages or builds a parse tree.

The ability to cope with arbitrary context-free grammars is important if one wishes to allow a modular syntax definition formalism. Due to the fact that LR(k)-grammars are not closed under union, a more powerful parsing technique is required. Generalized LR-parsing [Tom85, Rek92] (GLR-parsing) is a natural extension to LR-parsing, from this perspective. GLR-parsing does not require the parse table to be conflict-free. Allowing conflicts to occur in parse tables, GLR is equipped to deal with arbitrary context-free grammars. The parse result, then, might not be a single parse tree; in principle, a forest consisting of an arbitrary number of parse trees is yielded. Ambiguity produces multiple parse trees, each of which embodies a parse alternative. In case of an LR(1) grammar, the GLR algorithm collapses into LR(1), and exhibits similar performance characteristics. As a rule of thumb, the simpler the grammar, the closer GLR performance will be to LR(1) performance.

Eliminating the Scanner The use of a scanner in combination with GLR parsing leads to a certain tension between scanning and parsing. The scanner may sometimes have several ways of splitting up the input: a so-called lexical ambiguity occurs. In case of lexical ambiguities, a scanner must take some decision; at a later point, when parsing the tokens as offered by the scanner, the selected tokenization might turn out to be not quite what the parser expected, causing the parse to fail.

Scannerless GLR parsing [Vis97] solves this problem by unifying scanner and parser. In other words, the scanner is eliminated by simply considering all elementary input symbols as input tokens for the parser. Each character becomes a separate token, and ambiguities on the lexical level are dealt with by the GLR algorithm. This way, in a scannerless parser lexical and context-free syntax are integrated into a single grammar, describing the defined language entirely and exhaustively. Neither knowledge of the (usually complex) interface between scanner and parser nor knowledge of operational details of either is required for an understanding of the defined grammar.

3. THE NEW INTERMEDIATE FORMAT: EFIX

The abstract syntax trees representing ELAN specifications will be represented as ATERMS [vdBdJKO00].

By instantiating the nonterminal `AFun`, in the definition of the ATERMS syntax presented in Section 2.1, a language specific version of ATERMS can be created. For each abstract syntax construction a new `AFun`-symbol has to be defined. The ATERMS for ELAN will be called from now on EFIX.

3.1 Abstract syntax for ELAN *specifications*

For each language construct in ELAN an abstract syntax rule is defined which can be represented as an EFIX term. For example, the abstract syntax rule for module is:

```
<Module> ::= module ( <FormalModuleName>,
                      <Imports>,
                      <SortDefinition>,
                      <OperatorDefinition>,
                      [{<FamilyOfRule> ","}*])
```

The "keywords" like **module** corresponds to the `AFun` instantiations of ATERMS for ELAN. The sort names like <*Imports*> represent the abstract syntax *sub*trees. `[{`<*FamilyOfRule*>` ","}*]` represents a possible empty list of <*FamilyOfRule*> subtrees. For each abstract syntax rule in ELAN an equivalent "ATERMS" rule is defined. All redundant information, like layout, comments, keywords, etc., is lost in this EFIX representation.

The parsing of ELAN specifications is a two-phase process, see Section 4, in the first phase the specification is parsed modulo the rule bodies, whereas the second phase takes care of parsing these rule bodies. The EFIX format should allow the representation of the abstract syntax trees for both phases.

3.2 Abstract syntax for rules and terms

Section 4 discusses the parsing of rule bodies. In order to represent these rules in EFIX a number of new `AFun`s are introduced, namely `rule_body`, `if_cond`, and `where_cond`. A rule is now represented as:

```
rule_body(<Lhs>, <Rhs>,[{<Cond> ","}*])
```

Where `[{`<*Cond*>` ","}*]` is a list of conditions containing both `if_cond` and `where_cond`:

```
if_cond(<BoolTerm>)
where_cond(<Lhs>, <Rhs>)
```

<*Lhs*>, <*Rhs*> and <*BoolTerm*> are terms represented as:

```
appl(
  operator_decl(
    simple_formal_module_name(...),
    e_name(...),
    sorts_to_sort(...),
    options(...)),
  [{<Arg> ","}*])
```

Where `[{`<*Arg*>` ","}*]` represents a list of arguments of the form above or

```
variable(simple_variable(...),simple_sort_name(...))
```

4. THE NEW PARSER

The ELAN language has a number of features which makes the development of a new parser quite a challenge.

- The language allows the definition of mixfix operators and the use of it when defining rewrite rules.
- The preprocessing syntax, a kind of macro mechanism, allows a very concise way of writing down specifications.

- The concrete syntax of operators can be modified by means of the "alias" mechanism.

Furthermore, the current syntax of ELAN is to a large extent influenced, even polluted, by the parsing technology currently used (the Earley parser [BKK+98b]). A number of syntactic "Earley" adaptations will be given later on.

Given the ELAN User Manual [BKK+98b] the concrete syntax of ELAN has been defined in SDF, this exercise revealed some syntactical mismatches between the manual and the actual implementation. Furthermore, a mapping from the concrete syntax to the abstract syntax in EFIX was defined, see Section 3. Given the new parser generator of the ASF+SDF Meta-Environment an alternative parser was available, although it missed quite some functionality. It did not support the parsing of mixfix terms, preprocessing syntax, and aliasing. So, this parser could be seen as a skeleton parser which could be used to perform the first phase of parsing. The architecture of the skeleton parser is depicted in Figure 1. Appendix 1 gives the SDF definition of the concrete syntax of the operator definitions, whereas Appendix 2 gives the syntax definition of the "Family of Rules". Note that the bodies of the rewrite rules are parsed as flat strings.

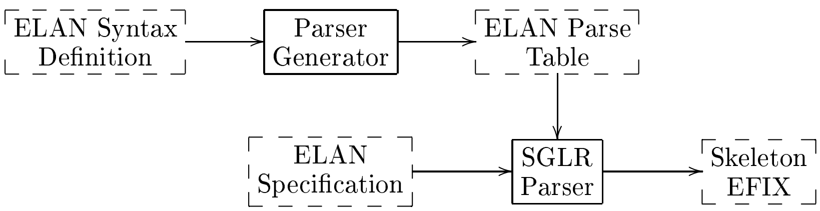

Figure 1: ELAN Skeleton Parser

Given this parser and the mapping to EFIX, ELAN specifications could be parsed and translated into an abstract format. The mixfix terms were stored as strings in this abstract format.

```
module assoc

sort Int;
end

operators global
   0 : Int;
   s (@) : (Int) Int;
   @ + @ : (Int Int) Int assocLeft pri 20;
   @ * @ : (Int Int) Int assocLeft pri 40;
   @ ^ @ : (Int Int) Int assocRight pri 60;
end

rules for Int
  x,y,z: Int;
global
  [] 0 + x => x end
  [] s(x) + y => s(x+y) end
  [] 0 * x => 0 end
  [] s(x) * y => x + z where z := ()x * y end
  [] x ^ 0 => s(0) end
  [] x ^ s(y) => x * z where z := ()x ^ y end
end

end
```

A part of the EFIX representation for this simple ELAN specification looks like

```
module(
...
rules_family(simple_sort_name("Int"),
[variable_declare([simple_variable("x"),
simple_variable("y"),
simple_variable("z")],
simple_sort_name("Int"))],
global_rules([non_labelled_rule(elan_string("0 + x => x end")),
non_labelled_rule(elan_string("s(x) + y => s(x+y) end")),
non_labelled_rule(elan_string("0 * x => 0 end")),
non_labelled_rule(
  elan_string("s(x) * y => x + z where z := ()x * y end")),
non_labelled_rule(elan_string("x ^ 0 => s(0) end")),
non_labelled_rule(
  elan_string("x ^ s(y) => x * z where z := ()x ^ y end"))])
...
```

Given the abstract syntax tree in EFIX it is possible to extract the relevant information, like defined sorts, operator definitions, and variables, and generate a parse table which can then be used to parse the unparsed mixfix terms, occurring in rules.

The following issues had to be solved:

- ELAN is a modular specification formalism with a powerful import mechanism, it allows parameterization of modules and renaming.
- Operator definitions may be global or local and the import of modules may also be both global and local.
- Per "Family of Rules" a new set of local variables is defined.

The import mechanism means that in order to parse a module all imported modules have to be inspected (thus being parsed) and all global definitions have to be retrieved. For now, we restrict to the case where imported modules have no parameters, and rules are declared as global.

The fact the per "Family of Rules" a fresh set of variables is defined lead to the observation that for each "Family of Rules" a new parse table had to be generated. This latter requirement means that the speed of parse table generation is quite important in order to make the system workable.

Given the set of visible sorts, operator definitions, variables, and the sort of the "Family of Rules" a parse table is generated per "Family of Rules". The parse table generation is performed in two steps. First, the EFIX representation of the sorts, operator definitions, and variables is translated into an intermediate formalism: Kernel-SDF. During this translation context-free grammars rules defining the structure of a "rule body", "if condition", "where condition" as well as the primitive strategy operators, such as `repeat*`, `first`, etc., are added. The Kernel-SDF representation also contains rules for recognizing comments and layout. This Kernel-SDF representation is then used to generate the parse table which will be used to parse the text of the rule bodies in the "Family of Rules". The architecture of this "mixfix" parser is shown in Figure 2.

Given this parse table, the unparsed terms have to be located and parsed and the derived EFIX representation has to be inserted in the original tree.

For example, the EFIX subterm

```
elan_string("s(x) + y => s(x+y) end")
```

is replaced by:

```
rule_body(
 appl(
```

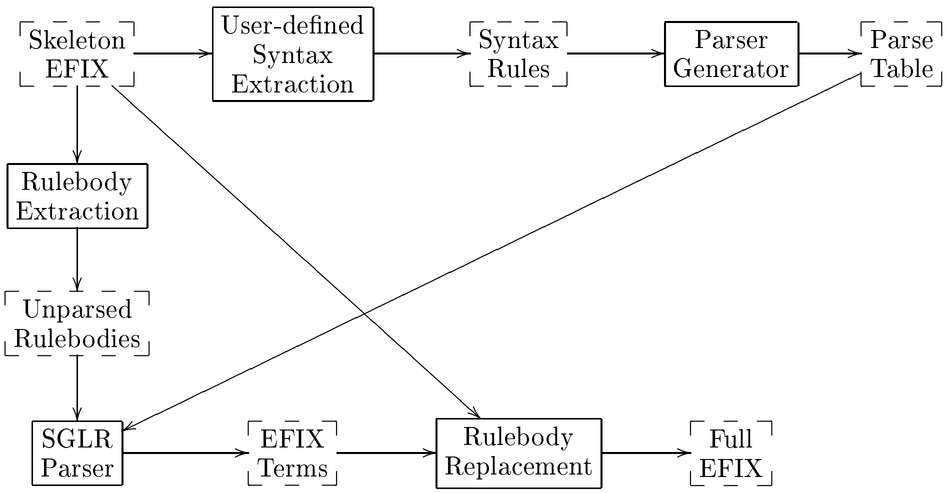

Figure 2: ELAN Mixfix Parser

```
  operator_decl(...,e_name([placeholder,"+",placeholder]),...),
  [appl(
    operator_decl(...,e_name(["s","(",placeholder,")"]),...),
    [variable(simple_variable("x"),simple_sort_name("Int"))]
   ),
   variable(simple_variable("y"),simple_sort_name("Int"))
  ]
 ),
 appl(
  operator_decl(..., e_name(["s","(",placeholder,")"]),...),
  [appl(
    operator_decl(...,e_name([placeholder,"+",placeholder]),...),
    [variable(simple_variable("x"),simple_sort_name("Int")),
     variable(simple_variable("y"),simple_sort_name("Int"))
    ]
   )
  ]
 ),
 []
)
```

In the current prototype we left out two things:

1. We restrict ourselves to programs without preprocessing constructs.

2. We do not consider parameterized modules and local rules

The second point is left out because of time constraints, not because of some technical difficulty. Indeed, a parameterized module can be already fully parsed, but we still have to add the functionality that enables us to instantiate the formal parameter (string) by the effective string in the EFIX program.

The first point is left out, because it is unclear whether preprocessing constructs should be or not considered as ELAN syntax. The status of preprocessing constructs should be further be clearified before serious work in this direction can be done.

5. Effects on the syntax

The use of the Earley parser has strongly influenced the concrete syntax of ELAN. The current ELAN syntax can be characterized as an "end" syntax. All main syntactic constructs are ended by an semi-colon or the "end" keyword, this does not improve the readability of a specification and caused even some serious problems in the skeleton parser for ELAN. Since ELAN (r)evolution is out of scope of this paper, we decided to keep all occurrences of "end".

5.1 Operators declaration

Alias option The alias option allows the programmer to declare different syntaxes for the same operator. In the ELAN parser, the last declared syntax is used as the representative of the operator in terms occurring in rules. With the new parser, we choose more naturally the first declared syntax as representative. For example, `@ + @ : (int int) int alias plus(@,@);` the binary plus is an alias for the prefix plus. The parser can recognize both the binary plus operator as well as the prefix plus, but in both cases the prefix plus operator will be inserted in the abstract syntax tree.

Bracket option A "bracket" option has been added in order to be able to use a "bracket" operator like in ASF+SDF. A "bracket" operator does not occur in EFIX terms, but it guides the parsing of terms. The alias option was often use to mimic the brackets, for example, `(@ + @) : (int int) int alias @ + @;`

User-defined strategy operators A user-defined strategy operator is declared just like other term operators. There is a specific sort for a strategy that applies on terms of sort s_1 and returns results of sort s_2. Now, the sort of such a strategy is denoted by $(s_1 \rightarrow s_2)$. Usually, we have $s_2 = s_1$, and in that case the sort $(s_1 \rightarrow s_1)$ is abbreviated by $< s_1 >$.

Built-in strategy operators In addition to user-defined strategy operators, we also add automatically some declarations for each built-in strategy operators (`dk`, `dc`, `dc one`, `first`, `first one`, `repeat*`, `repeat+ iterate*`, `iterate+`, `id`, `fail`, `;`), in order to be able to parse strategy expressions. These built-in strategy operators are sort-preserving, so that we consider a declaration of such an operator for each strategy sort occurring in PSS, the set of "Potential Strategy Sorts" defined as the set of sorts $(s_1 \rightarrow s_2)$ occurring as codomains of visible user-defined strategy operators. For example, the `repeat` operator is declared as follows:

`repeat(@) : ((`s_1` -> `s_2`)) (`s_1` -> `s_2`) // for each` $(s_1 \rightarrow s_2) \in PSS$

Rules of sort s are declared as strategy constant operators of sort $< s >$. Eventually, the application of strategies are performed via two kinds of *application* operators. The first one consists in applying a strategy of sort $(s_1 \rightarrow s_2)$ to a term of sort s_1. It returns a set of results of sort s_2, denoted by a built-in sort, named $set[s_2]$. The second operator applies the leftmost-innermost strategy (also called "the empty strategy") to a term t of sort s_2, and it yields a singleton of sort $set[s_2]$ containing the unique normal form of t. The application operators are automatically declared as follows:

`(@) @ : ((`s_1` -> `s_2`) `s_1`) set[`s_2`] // for each` $(s_1 \rightarrow s_2) \in PSS$

`() @ : (`s_2`) set[`s_2`] // for each` $(s_1 \rightarrow s_2) \in PSS$

5.2 Rules definition

The fact that a sort name has to be given in the left-hand side of a `where` part is another example of where the parser influenced the language design. This construction is no more needed with the new parser, and so it has been removed. Now, a `where` assignment is parsed according to the following declarations:

`where @ := @ : (`s_2` set[`s_2`]) WherePart // for each` $(s_1 \rightarrow s_2) \in PSS$

The reader may note that the two members of `where` are not of the same sort. Indeed, the right-hand side denotes a set of results of sort s_2, whereas the left-hand side is a term of sort s_2. The assignment is performed successively for each result thanks to the backtracking mechanism.

5.3 Strategies definition

There are two ways for defining a strategy. First, a strategy can be defined implicitly as a rule for the sort $(s_1 \rightarrow s_2)$. The right-hand side of this rule is a term involving built-in strategy operators as well as user-defined operators, provided that the codomain of the top-most operator is of sort $(s_1 \rightarrow s_2)$. Second, it is also possible to define a strategy explicitely as a rule for the built-in sort $set[s_2]$, by using one of the two application operators. The sort $set[s_2]$ can only occur in the `rules for` construct, since it cannot be used to declare user-defined operators. Indeed, the $set[]$ sorts are inhabited only by the built-in application operators.

6. An example: the old syntax vs. the new syntax

In this section, we present a very simple example, a specification of Booleans, first in the old (executable) syntax, and then in the new syntax (not yet executable). In this specification, we use implicit and explicit strategy definitions. We also introduce an unnecessarily complex rule in order to have a `where` assignment with a non-variable pattern.

6.1 Booleans in the old syntax

```
module Bool

import local strat[Bool] ;
end

sort Bool ;
end

operators global
True: Bool ;
False: Bool ;
@ & @: (Bool Bool) Bool assocLeft pri 200 ;
@ | @: (Bool Bool) Bool assocLeft pri 100 ;
!(@) : (Bool) Bool ;
end

stratop global
  outermostStrat : <Bool> bs ; // basic strategy
  oneStep : <Bool> ; // user-defined strategy
end

rules for Bool
B: Bool ;
global
[R1] True | B => True   end
[R2] False | B => B     end
[R3] True & B => B      end
[R4] False & B => False end
[R5] !(False) => True end
[R6] !(True) => False end
end

rules for Bool
B1,B2,S1,S2: Bool ;
```

```
global
[C1] B1 & B2 => S1 & S2
                where S1:=(outermostStrat) B1
                          // basic strategy applications
                where S2:=(outermostStrat) B2 end
[C2] B1 | B2 => S1 | S2
                where S1:=(outermostStrat) B1
                where S2:=(outermostStrat) B2 end
[C3] !(B1) => !(S1)
              where S1:=(outermostStrat) B1 end
[C4] !(B1) => !(S1)
              where (Bool) !(S1) := [first(C3)] !(B1) end
end

strategies for Bool
implicit
[] outermostStrat => repeat*(first(R1,R2,R3,R4,R5,R6,C1,C2,C4)) end
// basic strategy definition
end

strategies for Bool
X: Bool ;
explicit
[] [oneStep] X => [first(R1,R2,R3,R4,R5,R6)] X end
// user-defined strategy definition
end

end
```

6.2 Booleans in the new syntax
The different changes are given below.

Operators declaration Strategy operators and term operators are declared in the same declaration part.

```
operators global

True: Bool ;
False: Bool ;

@ & @: (Bool Bool) Bool assocLeft pri 200 ;
@ | @: (Bool Bool) Bool assocLeft pri 100 ;
!(@) : (Bool) Bool ;

outermostStrat : <Bool> ;
oneStep : <Bool> ;
end
```

Rules definition The syntax of a `where` assignment becomes much more simple since it is no more necessary to know the sort of the non-variable pattern to be parsed.

```
rules for Bool
B1,B2,S1,S2: Bool ;
global
...
```

```
[C4] !(B1) => !(S1)
              where !(S1) := (first(C3)) !(B1) end
...
end
```

Implicit strategy definition An implicit strategy is defined as a rule for a *strategy* sort.

```
rules for <Bool>
global
[] outermostStrat => repeat*(first(R1,R2,R3,R4,R5,R6,C1,C2,C4)) end
end
```

Explicit strategy definition An explicit strategy is defined as a rule for an *application* sort.

```
rules for set[Bool]
X: Bool ;
global
[] (oneStep) X => (first(R1,R2,R3,R4,R5,R6)) X end
end
```

It is important to note that a unique notation is now used for the application of strategies. We do not reuse anymore square brackets as in the old syntax, where we had `()` brackets for basic strategies and `[]` brackets for user-defined strategies. Of course, after parsing, we still have to make a distinction between basic strategies and user-defined strategies, but it is no longer a parsing matter.

7. Current implementation

The current implementation is a script `parselan` that consists of two tools. These two tools correspond to the two main parsing phases. The first one, called `elan2efix`, produces an EFIX term, where the rule bodies are still unparsed and just occur as strings. This is the so-called skeleton parser. The second tool performs the actual mixfix parsing, it parses the unparsed rule bodies, constructs EFIX subterms for them and inserts these EFIX subterms in the original EFIX term.

In a module, a rule is identified by a pair (i, j) of integers, where i is the index of its family of rules in the whole list of family of rules, and j is the index of the rule in the list of rules defining the family of rules. A new parse table must be constructed for each family of rules. The set of operators visible in the module of interest is computed by `visible-sig`. Besides the globally visible operators the local variables defined within a family of rules are also needed for generating a parse table. `extract-sig` takes the signature obtained via `visible-sig` and adds the local variables to it. Given the results of `visible-sig` and `extract-sig`, `efix2table` builds a parse table for each family of rules. This parse table is used for parsing all rules in the family of rules. For each rule to be parsed, the string is extracted by `extract-rule` and parsed by `sglr`. The result is eventually plugged in the EFIX term using `replace-rule`. All these tools `visible-sig`, `extract-sig`, `efix2table`, `sglr`, `extract-rule`, `replace-rule` handle EFIX ATERMS or plain ATERMS. They are integrated in one main `C` program.

8. Conclusion and future works

In this document, we report our first experiments in the development of an ELAN parser using the available ASF+SDF parsing tools and the ATERM representation. For now, we have developed a first prototype. The first results are quite promising (see Appendix 3). In the next future, we plan to tackle the remaining issues like, aliases, parameterized modules, and may be even preprocessing syntax. Hence, we believe to develop a complete parser for ELAN, which will be both efficient and easy to maintain.

The use of ASF+SDF parsing technology to develop a ELAN parser is quite a logical choice. Both formalism support user-defined syntax, although in a slightly different manner. Alternative parsers could have been CIGALE [Voi86] used within ASSPEGIQUE [BC85], Cocke-Younger-Kasami parser

[HU79], or Earley parser [Ear70]. The latter one was already used within the ELAN system. Of course, LEX/YACC based parsers could also be used, but this would restrict the user-defined syntax in order to prevent conflicts. The ASF+SDF parsing technology has been used to develop parser for similar language, such as CASL [CL98] and Stratego [VBT98]. The architecture of this CASL parser [vdBS00] is quite similar to the architecture of the ELAN parser discussed here.

Finally, we still have to adapt existing interpreter and compiler for executing EFIX programs. Even if it is obviously not the best solution, we currently develop in this direction a translation tool from EFIX to REF, which is an executable format in the ELAN environment. The actual prototype of this translation tool only deals with REF programs without strategies, and so it needs to be further investigated.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BC85] M. Bidoit and C. Choppy. ASSPEGIQUE: an integrated environment for algebraic specifications. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Formal Methods and Software Development - Proceedings of the International Joint Conference on Theory and Practice of Software Development 2*, volume 186 of *LNCS*, pages 246–260. Springer-Verlag, 1985.

[BJMR98] P. Borovanský, S. Jamoussi, P.-E. Moreau, and Ch. Ringeissen. Handling ELAN Rewrite Programs via an Exchange Format. In *Proc. of [KK98]*, 1998.

[BKK+98a] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and Ch. Ringeissen. An Overview of ELAN. In *Proc. of [KK98]*, 1998.

[BKK+98b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN *V 3.0 User Manual*. Inria Lorraine & Loria, Nancy (France), second edition, January 1998.

[CL98] CoFI-LD. CASL – The CoFI Algebraic Specification Language – Summary, version 1.0. Documents/CASL/Summary-v1.0, in [CoF98], 1998.

[CoF98] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by HTTP[1] and FTP[2], 1998.

[dJO99] H.A. de Jong and P. Olivier. ATerm Library User Manual, 1999. Available via HTTP[3].

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[HU79] J. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Language, and Computation.* Addison-Wesley, 1979.

[Joh86] S.C. Johnson. *YACC: yet another compiler-compiler.* Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[KK98] C. Kirchner and H. Kirchner, editors. *Second Intl. Workshop on Rewriting Logic and*

[1] `http://www.brics.dk/Projects/CoFI/`
[2] `ftp://ftp.brics.dk/Projects/CoFI/`
[3] `http://www.wins.uva.nl/pub/programming-research/software/aterm/`

its Applications, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France), September 1998. Elsevier.

[KKV95] C. Kirchner, H. Kirchner, and M. Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. MIT press, 1995.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[KV94] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proceedings ASMICS Workshop on Parsing Theory*, pages 1–20, 1994. Published as Technical Report 126–1994, Computer Science Department, University of Milan.

[LS86] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[MK97] P.-E. Moreau and H. Kirchner. Compilation Techniques for Associative-Commutative Normalisation. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97*, Workshops in Computing, Amsterdam, September 1997. Springer-Verlag.

[MK98] P.E. Moreau and H. Kirchner. A compiler for rewrite programs in associative-commutative theories. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[Tom85] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.

[VBT98] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizes with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, pages 13–26, 1998.

[vdBdJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[vdBKMO97] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Design and implementation of a new asf+sdf meta-environment. In A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Workshops in Computing, Amsterdam, 1997. Springer/British Computer Society.

[vdBS00] M.G.J. van den Brand and J. Scheerder. Development of Parsing Tools for CASL using Generic Language Technology. In D. Bert and C. Choppy, editors, *Workshop on Algebraic Development Techniques (WADT'99)*, volume 1827 of *LNCS*. Springer-Verlag, 2000.

[vDHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[Vis97] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[Vit96] M. Vittek. A Compiler for Nondeterministic Term Rewriting Systems. In Harald

Ganzinger, editor, *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168. Springer-Verlag, July 1996.

[Voi86] F. Voisin. CIGALE: a tool for interactive grammar construction and expression parsing. *Science of Computer Programming*, 7(1):61–86, 1986.

1. SDF definition of Operator Definition

```
module OperatorDefinition

imports Name Sorts

exports
  sorts OperatorDefinitionOpt GlobalOrLocalOperatorDefinition
        SymbolDeclaration NewSymbolDeclaration
        SymbolAlias Rank Option NamedSortName
  context-free syntax
                                                     -> OperatorDefinitionOpt
  "operators" GlobalOrLocalOperatorDefinition* "end" -> OperatorDefinitionOpt

  "global" SymbolDeclaration+ -> GlobalOrLocalOperatorDefinition
  "local" SymbolDeclaration+  -> GlobalOrLocalOperatorDefinition

  NewSymbolDeclaration ";" -> SymbolDeclaration
  SymbolAlias ";"          -> SymbolDeclaration

  Name ":" Rank Option* -> NewSymbolDeclaration

  Sort                        -> Rank
  "(" Sort+ ")" Sort          -> Rank
  "(" NamedSortName+ ")" Sort -> Rank

  "assocLeft"   -> Option
  "assocRight"  -> Option
  "pri" Number  -> Option
  "(" "AC" ")"  -> Option
  "bracket"     -> Option
  "code" Number -> Option

  Id ":" SortName -> NamedSortName

  NewSymbolDeclaration "alias" Name -> SymbolAlias
```

2. SDF definition of Family of Rules

```
module FamilyOfRules

imports Name ElanStrings Sorts

exports
  sorts FamilyOfRules GlobalRulesOpt LocalRulesOpt
        LabelledOrNonLabelledRule LabelledRule
        NonLabelledRule RuleLabel VariableDeclare

  context-free syntax
    "rules" "for" Sort VariableDeclare*
                  GlobalRulesOpt LocalRulesOpt "end" -> FamilyOfRules

                                       -> GlobalRulesOpt
    "global" LabelledOrNonLabelledRule+ -> GlobalRulesOpt

                          -> LocalRulesOpt
    "local" LabelledRule+ -> LocalRulesOpt

    LabelledRule    -> LabelledOrNonLabelledRule
    NonLabelledRule -> LabelledOrNonLabelledRule

    "[" RuleLabel "]" BodyString -> LabelledRule

    Id -> RuleLabel

    "[" "]" BodyString -> NonLabelledRule

    {VariableName ","}* ":" SortName ";" -> VariableDeclare
```

3. Some measurements

We have made experiments on a number of examples, including the specification of Booleans seen in Section 6. In the following, we also give results obtained with a class of examples generated automatically. These examples are parsed on a PC Linux 500Mhz equipped with 128 Mb.

Example	ELAN *parser*	*New Skeleton*	*New Mixfix*
`Bool`	0.70 s	0.47 s	1.59 s
`enum10`	0.04 s	0.58 s	0.58 s
`enum20`	0.10 s	1.00 s	1.01 s
`enum30`	0.29 s	1.42 s	1.52 s
`enum40`	0.56 s	2.24 s	2.19 s
`enum50`	Fail	3.34 s	3.17 s
`enum60`	Fail	4.48 s	4.18 s
`enum70`	Fail	5.81 s	5.44 s
`enum80`	Fail	7.43 s	7.10 s
`enum90`	Fail	9.14 s	8.84 s
`enum100`	Fail	11.31 s	10.66 s
`enumNW50`	0.23 s	1.14 s	1.48 s
`enumNW100`	1.86 s	2.80 s	4.32 s
`enumNW200`	22.39 s	10.04 s	18.93 s
`enumNW250`	50.61 s	15.44 s	31.36 s

The ELAN program called $\texttt{enum}_n$ consists of a rule body with n rules, one for each $i = 1, \ldots, n$:

```
[] enum(i) => X_1 U ... U X_i U emptySet
      where X_1 := () 1
      ...
      where X_i := () i
      end
```

Similarly, the ELAN program $\texttt{enumNW}_n$ consists of the following rules, for $i = 1, \ldots, n$:

```
[] enum(i) => 1 U ... U i U emptySet end
```

With these examples, there is no difference between the old syntax and the new one. Therefore, it is possible to parse them using ELAN with the option `--export`, which is the only way to call the parser without the interpreter. One may note that ELAN fails in most of examples because there are too many local variables defined in rule bodies. When it does not fail, this option of ELAN produces a REF program which has nothing to do with an abstract syntax of an ELAN program. Therefore, it is quite difficult to fairly compare the two parsers. We recall that the new parser first execute the skeleton parser (third column), and then the mixfix parser (fourth column). Therefore, we must add the execution times in the last two columns in order to obtain the total parsing time. Note that the new parser is already faster than the ELAN parser on a large example (the last one) and it has no problems with huge numbers of local variables. Moreover, in all examples, the new parser run on a PC Linux 500 Mhz is faster than the ELAN parser run on a DEC Alpha 300 Mhz.