

# The **metafront** System: Extensible Parsing and Transformation

Claus Brabrand<sup>a,1</sup>, Michael I. Schwartzbach<sup>b,2</sup>,  
and Mads Vanggaard<sup>b,3</sup>

<sup>a</sup> *INRIA/LaBRI, ENSEIRB, 1, Avenue du Docteur Albert Schweitzer, Domaine  
Universitaire - BP 99, F-33402 Talence Cedex, France*

<sup>b</sup> *BRICS, Department of Computer Science, University of Aarhus  
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark*

---

## Abstract

We present the **metafront** tool for specifying flexible, safe, and efficient syntactic transformations between languages defined by context-free grammars. The transformations are guaranteed to terminate and to map grammatically legal input to grammatically legal output.

We rely on a novel parser algorithm that is designed to support gradual extensions of a grammar by allowing productions to remain in a natural style and by statically reporting ambiguities and errors in terms of individual productions as they are being added.

Our tool may be used as a parser generator in which the resulting parser automatically supports a flexible, safe, and efficient macro processor, or as an extensible lightweight compiler generator for domain-specific languages. We show substantial examples of both kinds.

---

## 1 Introduction

We present the **metafront** tool for specifying safe, flexible, efficient, and extensible syntactic transformations between languages defined by context-free grammars. Safety means that **metafront** statically guarantees that the transformation of grammatically legal input will always terminate and produce grammatically legal output. Flexibility means that the expressive power is sufficient for realistic tasks and that both the source and target languages

---

<sup>1</sup> Email: [Claus.Brabrand@inria.fr](mailto:Claus.Brabrand@inria.fr)

<sup>2</sup> Email: [mis@brics.dk](mailto:mis@brics.dk)

<sup>3</sup> Email: [mvj@brics.dk](mailto:mvj@brics.dk)

may be extended with little overhead. Efficiency means that, given a grammar and transformation, the parsing and transformation is linear in the size of input and generated output. Extensibility means that post-hoc extensions of the source language are easily reflected in similar extensions of the transformation.

We have two main usage scenarios in mind for this versatile tool. First, `metafront` can be used for lightweight domain-specific compiler prototypes, e.g. for translating Java programs into HTML documentation in the style of JavaDoc. Second, if the source language is a small extension of the target language, then the syntactic transformation is equivalent to a powerful macro mechanism.

In all cases, the programmer may greatly benefit from the advantages that `metafront` offers. Thus, our tool captures the niche where full-scale compiler generators are too general and where simpler techniques for syntactic transformation are not expressive enough or do not offer sufficient safety guarantees.

### 1.1 Language Design

The `metafront` tool works with two kinds of files: definitions of *languages* and definitions of *transformations*.

Languages are defined using fairly standard context-free grammars with nonterminals, terminals, and productions. A simple module system allows languages to be defined through a DAG of sublanguages that refer to each other. Classes of terminals are defined by full regular expressions, including intersection and complement.

A central part of such a tool is of course the parsing algorithm that is employed. We have been led to develop a novel algorithm, called *specificity parsing*, which is a scannerless top-down parser where ambiguities are resolved through notions of specificity. At any stage, the remainder of the input string is confronted with a *set* of *candidates*, which are sentential forms stemming from different right-hand sides of productions. First each candidate suggests what the next token should be, and the most specific one wins. The candidates that can accept this token will then each suggest which action to take, and the most specific action wins. This action is then performed and those candidates that agreed on this choice survive to the next challenge round.

This method of parsing is tailored to our intended applications, where languages are extended by different programmers. This requires that the syntax is written in a natural style and that errors and ambiguities can be explained sensibly in terms of the individual productions that are being added. Since we employ a top-down approach, we of course cannot handle left-recursive nonterminals but, apart from this restriction, productions may be written in a quite intuitive manner. Also, when a new production is added we can statically decide if it may cause ambiguities during subsequent parsing. Furthermore, error messages are phrased locally in terms of the added production.

Transformations are specified relative to a source and a target language, which are imported from other files. Each production in the source language is instrumented with a transformation rule. The parse trees that correspond to the nonterminals of the right-hand side are inductively subjected to transformations before the results are inserted into a template that constructs a parse tree of the target language. Transformation rules may accept parse trees as arguments and produce parse trees as results. Since users often specify transformations from an extended language to a core language, each source production has the identity as its default transformation rule.

There are three important characteristics of our notion of transformations. First, they are designed to allow only elaborate well-founded induction, so termination is ensured. Second, we can statically decide if a transformation is guaranteed to map grammatically legal input to grammatically legal output. Third, the rules are expressive enough to allow sophisticated transformations that rearrange trees in a non-local manner.

The `metafront` tool accepts as arguments a transformation and a term of the input language. It will then analyze the source and target grammars and the transformation rules and provide error diagnostics or construct the corresponding term of the target language.

## 1.2 Related Work

There are three main bodies of work that we must relate to: *parser generators*, *macro processors*, and *compiler generators*.

We differ from all other parser generators, such as JavaCC or the Lex/Yacc family in automatically instrumenting the generated parser with a powerful syntactic macro processor similar to the one we have earlier hardwired for the `<bigwig>` language [3]. Formally, our parsing algorithm is incomparable to both LL(1) and LR(1) parsers; however, we claim that it has unique benefits as outlined above. Traditional parser generators allow productions to be instrumented with action code, whereas we only allow inductive transformations. This is the basis for the safety guarantees that we are able to provide. In any case, action code can be emulated by performing a transformation into e.g. Java code which must then be executed afterwards (without any guarantees besides syntactic correctness, of course).

Regarding macro processors, we refer to the comprehensive survey that we provide in [3]. Our present tool is unique in simultaneously being parameterized with the grammar of the host language and providing strong safety guarantees.

Compiler generators, such as [11,1,8,13], have wider ambitions than our work, supporting specifications of full-scale compilers including static and dynamic semantics. Invariably, this involves Turing-complete computations on parse trees which of course precludes our level of safety guarantees.

The extensible grammars of [5] share our aims in many ways. The resulting

tool is a parser generator that allows subsequent extensions of the language which must then be desugared into the original language. It offers safety guarantees similar to ours, but does not handle arbitrary source and target languages and provides less expressive transformations.

The system that most closely compares to `metafront` as a compiler generator is ASF+SDF [13]. It uses a scannerless generalized LR-parser to produce a forest of parse trees which is continually filtered and transformed with respect to a set of rewrite rules. The end result is hoped to be a single, normalized parse tree. By imaginative use of rewriting of syntactic encodings, it is possible to construct complete compilers including symbol tables, type-checking, and code generation. It is of course also possible to *encode* the kinds of transformations that `metafront` supports. If  $S$  and  $T$  are the source and target languages, then possible encodings are to define the transformations as rewritings on a combined language such as  $S \cup T$  or  $S \times T$ . While ASF+SDF statically guarantees that each rewrite step will respect the given grammar, there are two kinds of termination problems. First, the transformation may loop, which cannot be statically determined since rewritings are Turing-complete. Second, the transformation may terminate too soon, leaving unprocessed pieces of the input language. Again, it is undecidable to determine if this problem may occur.

The two parser algorithms also have different characteristics. Generally, the ASF+SDF parser generates a forest of parse trees, and it is undecidable if filtering results in a single tree. It is scannerless by processing each character as a separate token, which results in some overhead. In comparison, we tokenize the input string using ordinary DFAs that are selected dynamically based on the parser context. Finally, the ASF+SDF parser has a running time that depends on the number of parse trees being produced. Our parser is guaranteed to run in linear time, but is of course restricted to a certain class of grammars.

In summary, `metafront` is a domain-specific language focusing on syntactic transformations as a subset of compiler generator applications and offering advantages in terms of flexibility, safety and efficiency.

## 2 Parsing

As mentioned in the introduction, we place key emphasis on *extensibility*: we want different kinds of users to be able to incrementally add new productions and even new user-defined terminals and nonterminals.

To achieve this, language designers must read the grammar and find hooks where new extensions and transformations may be attached. This requires the grammar to be phrased in a natural style. Additionally, error messages should only involve the part of the grammar written by the user. These requirements, however, are not satisfied by other common parsing strategies.

The LR(k) family of bottom-up parsing algorithms is unable to provide lo-

calized error messages. Consider for example the Yacc version of the LALR(1) grammar for Java, which contains the production:

```
GuardingStatement : SYNCHRONIZED '(' Expression ')' Statement;
```

If we clumsily tried to allow synchronization on multiple objects by adding the production:

```
GuardingStatement : SYNCHRONIZED '(' Expression Expression ')' Statement;
```

then the Yacc tool reacts by producing 29 shift/reduce and 26 reduce/reduce errors. None of those errors occur in the parser states corresponding to the inserted production, but in seemingly arbitrary places involving nonterminals such as e.g. `NotJustName` and `ShiftExpression`. The reason for this avalanche of non-local errors is that LALR(1) parser errors arise in terms of a table *derived* from the grammar, and not in terms of the grammar itself.

It is often also necessary to rewrite a grammar into a less natural style. This is evident for the LL(k) family of top-down parsing algorithms, where for example the productions:

```
Class : "class" Identifier '{' ... '}'
      | "class" Identifier "extends" Identifier '{' ... '}' ;
```

must be rewritten into the less intuitive form:

```
Class : "class" Identifier ExtendsOpt '{' ... '}' ;
ExtendsOpt : "extends" Identifier
           | ;
```

which is also less susceptible to extension; there is no nonterminal where, for instance, a concept of *anonymous classes* with a “`class { ... }`” syntax may subsequently be added.

In contrast, Earley’s algorithm [7] and generalized LR(k) parsing [12] allow any grammar, but sacrifice linear-time processing. Also, they ignore ambiguities by constructing all parse trees, and choosing the right one at the end requires non-local reasoning.

Our goal is to obtain an efficient parsing algorithm that allows productions to be added incrementally and that locally and statically detects and reports ambiguity errors.

## 2.1 Specificity Parsing

A *specificity grammar* is a five-tuple:

$$G = \langle \Sigma, T, N, s, \pi \rangle$$

where  $\Sigma$  is a finite set of symbols known as the alphabet;  $T \subseteq \text{Reg}(\Sigma)$  is a finite set of *regular languages* over  $\Sigma$  known as the set of *terminal languages*;  $N$  is a finite set of nonterminals;  $s \in N$  is the start nonterminal; and  $\pi : N \rightarrow \mathcal{P}(E^*)$ , where  $E = T \cup N$ , is the production function identifying the grammar rules for each nonterminal.

This definition resembles the definition of context-free grammars, but with an explicit notion of lexical structure described separately as the set of *terminal languages*,  $T$ .

This separation plays a crucial role in specificity parsing that works on two integrated levels, a lexical and a syntactic, each with its own notion of specificity for deterministic ambiguity resolution that is independent of definition order.

The terminal languages induce a *lexical specificity* relation which is used to deterministically select a terminal language and input tokenization in case there are multiple choices. Specifically, the lexical specificity relation,  $\sqsubseteq_{lex} \subseteq \langle T \times \Sigma^* \times \Sigma^* \rangle \times \langle T \times \Sigma^* \times \Sigma^* \rangle$  (see Appendix C.1 in [4]), is defined on *terminalizations* which are triples comprised of terminal languages, concrete input tokens, and remainder input. The lexical specificity ordering is defined as the lexicographical composition of longest consumable token and terminal regular language inclusion. We have hardwired the preference of longer tokens due to a general consensus in programming languages as evident in scanner generators such as Lex. For simplicity, terminals are in the following assumed not to include the empty string,  $\varepsilon$ ; this can be amended with minor modifications to the algorithm.

Given a lexical layer, the rest of the parser handles syntactic aspects using a notion of *head-sets* (see Appendix B.2 in [4]) which resemble first-sets, but are sets of terminal languages and nonterminals; in contrast, a first-set is a subset of  $\Sigma^*$ . A special terminal,  $\square$ , represents the end of a sentential form and is included in a head-set if there is a token-less path to it.

Head-sets induce a *syntactic specificity* relation on sentential forms which is used to deterministically pick a production among multiple choices. Specifically, two syntactic specificity relations,  $\sqsubseteq_E \subseteq E \times E$  and  $\sqsubseteq_{syn} \subseteq E^* \times E^*$  (see Appendix C.2 in [4]), are defined in terms of head-set inclusion. The nonterminals are present in the head-sets in order to define this relation and are used later to guide parsing towards nonterminal gaps (introduced in Section 3). As a consequence, two nonterminals are only ordered if they are related by a token-less path from one to the other (i.e. not if they happen to have related head terminals).

### 2.1.1 The Specificity Parsing Algorithm

We now present our *Specificity Parsing* algorithm and evaluate it in Section 2.2. Algorithm 1 depicts the nine steps of a so-called specificity parsing *challenge round* in which a *set of sentential forms*,  $A$ , is confronted with an input string,  $\omega$ . The algorithm parses as much input as possible and returns the remainder input.

**Steps 1-4** work on a lexical level in defining a context-sensitive scanner that determines the *terminalization* which is comprised of the terminal language,

---

**Algorithm 1** *The specificity parsing algorithm:*

---

<u>parse</u> ( $A, \omega$ ) :	$\mathcal{P}(E^*) \times \Sigma^* \rightarrow \Sigma^*$
1. $H = \bigcup \{\text{head-set}(\alpha) \mid \alpha \in A\}$	<i>find head-sets: <math>H \subseteq E \cup \{\square\}</math>;</i>
2. $X = \{\langle x, \omega_1, \omega_2 \rangle \mid x \in H, \omega = \omega_1 \omega_2, \omega_1 \in x\}$	<i>find possible terminalizations;</i>
3. <b>if</b> $X = \emptyset$ <b>then</b>	<i>if no terminalizations:</i>
3a. <b>if</b> $\varepsilon \in A$ <b>then return</b> $\omega$	<i>return, if an option;</i>
3b. <b>if</b> $\square \notin H$ <b>then</b> $\text{error}(H)$	<i>issue error message;</i>
3c. <b>else</b> $\langle x, \omega_1, \omega_2 \rangle = \langle \square, \varepsilon, \omega \rangle$	<i>select sent. form end, <math>\square</math>;</i>
4. <b>else</b> $\langle x, \omega_1, \omega_2 \rangle = \text{most-specific-terminal}(X)$	<i>select best terminalization;</i>
5. $Y = \{\alpha \in A \mid x \in \text{head-set}(\alpha)\}$	<i>find applicable sent. forms;</i>
6. $e = \text{most-specific-entity}(Y)$	<i>select best head entity: <math>e \in E</math>;</i>
7. <b>case</b> $e$ <b>of</b>	<i>parse winner entity, <math>e</math>:</i>
7a. $t \in T$ : $\omega' = \omega_2$	<i>consume input token, <math>\omega_1</math>;</i>
7b. $n \in N$ : $\omega' = \text{parse}(\pi(n), \omega)$	<i>parse <math>n</math> recursively;</i>
8. $A' = \bigcup \{\text{advance}(\alpha, e) \mid \alpha \in A\}$	<i>tails of those with head <math>e</math>;</i>
9. <b>return</b> <u>parse</u> ( $A', \omega'$ )	<i>parse next challenge round.</i>

$x$ , the actual input token,  $\omega_1$ , and the remainder input,  $\omega_2$ , when this token is consumed:

**1.** We calculate the union of the *head-sets* of the sentential forms. This set,  $H$ , corresponds to looking for terminals in all directions from the current parsing context. This is context-sensitive in that only terminals “visible” from the current parsing context are considered.

**2.** Based on the set of visible terminals,  $H$ , we now determine the set,  $X$ , of terminals applicable to the current input string (along with what they are capable of consuming and what input remains).

**3.** In case there were no applicable terminals ( $X = \emptyset$ ) there are three cases:

**3a.** if the empty sentential form is an option ( $\varepsilon \in A$ ), we stop parsing and *return* to the previous challenge round parse;

**3b.** otherwise, if we cannot get to the end of a sentential form,  $\square$ , the parser is stuck and we generate an error message containing precisely the set of terminals,  $H$ , expected at the current position in the parser.

**3c.** otherwise we select  $\langle \square, \varepsilon, \omega \rangle$  as the terminalization, so that we are guided towards the end of a sentential form,  $\square$ .

**4.** If the set of possible terminalizations,  $X$ , is non-empty, we select the *most specific* terminalization according to lexical specificity; uniqueness of a *most specific* choice is statically ensured in Section 2.2.2.

**Steps 5-7** work on a syntactic level and are responsible for parsing *this* challenge round by determining the entity to parse in this round and by reacting appropriately to it. The rest of input to parse is left in  $\omega'$ .

**5.** We determine the non-empty set,  $Y$ , of sentential forms capable of consuming the winner terminal,  $x$ .

**6.** From the set of applicable sentential forms,  $Y$ , we select the most specific ones according to *syntactic specificity* which is defined in terms of inclusion of sentential form head-sets. There may be many different sentential forms with the same head-set, but their head element is unique and is extracted into  $e$ ; uniqueness is statically ensured in Section 2.2.2.

**7.** We are now ready to *parse* the winner head entity which is either a terminal or a nonterminal:

**7a.** if the winner entity is a terminal, we consume the input token by assigning the remainder input to the rest of input to parse,  $\omega'$ . Since  $x \in \text{head-set}(t\beta)$  (cf. step 5), we know that  $x = t$ .

**7b.** if the winner entity is a nonterminal,  $n$ , we recursively parse its production “right-hand sides”,  $\pi(n)$ .

**Steps 8-9** prepare for and parse the *next* challenge round:

**8.** We advance the parser by extracting the tails of all sentential forms that have the winner entity as head; the resulting set,  $A'$ , may have one or more elements.

**9.** We recursively parse the next challenge round with the surviving set of sentential form tails.

## 2.2 Evaluation

We now evaluate the specificity parsing algorithm with respect to flexibility, safety, and efficiency.

### 2.2.1 Flexibility

The most important advantage of specificity parsing is *extensibility*. Both lexical and syntactic specificity are deterministic disambiguation mechanisms that provide local conflict resolution; extensions have local effect and any errors are guaranteed to involve only locally the extended parts of the grammar. Also, the specificity selection is independent of definition-order so that language composition is symmetric and language modules may be loaded in any order. These properties permit incremental and modular grammar design with grammar-level parsing, reasoning, and disambiguation.

Since the specificity parser operates relative to a *set* of sentential forms, syntax can be conveniently overloaded by the addition of syntactic variants (as with the anonymous classes in Section 2).

Specificity parsing is *scannerless* in that the scanner is implicitly synthe-

sized from the grammar. This alleviates many tedious and error-prone tasks of manually keeping a state correspondence between the scanner and parser; in Lex, this correspondence is often emulated via a notion of *start-conditions*. Having a truly context-sensitive scanner avoids *keywordification* meaning that keywords are not necessarily global; different parts of a program may have different keywords. This is good for languages with many different constituent DSLs, such as <bigwig> [2]. As previously mentioned, our scanner may be extended to cope with terminals containing the empty string.

### 2.2.2 Safety

We perform three static analyses on grammars:  $W_{\text{NLR}}$ , intercepts left-recursion;  $W_{\text{DER}}$ , checks that all nonterminals have (finite) derivations; and  $W_{\text{USW}}$ , ensures that lexical and syntactic specificity always have unique final winners. These three specificity grammar wellformedness safety checks are formalized in Appendix D in [4].

The *no-left-recursion* wellformedness check,  $W_{\text{NLR}}$ , ensures termination of the parsing strategy by essentially making sure that the parser is unable to oscillate between nonterminals without consuming input. Termination is then obtained from the fact that the grammar and input is finite. The *derivability* check,  $W_{\text{DER}}$ , has no implications on safety; it is just included as a convenient sanity check. The *unique-specificity-winner* check,  $W_{\text{USW}}$ , guarantees that the parser is deterministic in its choice of terminals and productions.

### 2.2.3 Efficiency

Given a grammar, the algorithm as presented above parses the input in linear time without any backtracking. In Section 2.4, we show how to add controlled backtracking without compromising this time bound. Also, since every challenge round has a unique winner (in step 6) the parser commits to one entity without any state explosion.

All head-set unions (in step 1) can be statically precomputed for all the finitely many challenge round parser positions. This information can then be used to statically factor out all dynamic syntactic specificity checks by topologically sorting all productions according to this partial ordering. At parse-time, the algorithm may then test them in this sequence and dispatch on the first applicable entity.

Although scannerless, the parser retains all efficiency benefits of a scanner-full approach in employing minimized deterministic finite automata, DFAs, for deciding regular language membership.

Also, since recursive parser calls in step 7b do not consume input nor increase head-sets, the terminalization steps 1-4 may be cached as the same terminal wins again.

Finally, global analysis of the grammar may lead to further optimization like inlining of nonterminals.

### 2.3 Comments and Whitespace

Comments and whitespace are handled through a special terminal, **omit**, that may be assigned a regular expression of tokens to omit. Since different parts of a program may have different omit structure, omits do not have global effect, but are instead bound to all subsequently defined nonterminals and implicitly added between all entities in their sentential forms.

```
terminal {
  WhiteSpace      = { [ \t\n\r]+ }
  MultiLineComment = { "/*" .. "*/" }
  EndOfLineComment = { "//" .. \n }
  omit = { ( <WhiteSpace> | <MultiLineComment> | <EndOfLineComment> )+ }
}
```

This fragment defines a Java-like omit structure, discarding standard whitespace, multi-line comments, and end-of-line comments. The binary infix regular expression operator “ $R..S$ ” is a convenient *from-to* construction, defined as  $R(\Sigma^*S\Sigma^*)^cS$ ; it can be added to metafront through self-application. The **omit** construction defaults to the regular expression `WhiteSpace` defined above.

#### 2.3.1 Example: The Lambda Calculus

We first use simple extensions of the untyped lambda-calculus to show the complete contents of metafront files. The basic syntax is defined as:

```
language Lambda {
  terminal Id = { [a-z]+ }
  nonterminal Exp;

  Exp[id]      --> <Id>          ;
  [lambda]    --> \\ <Id> . <Exp> ;
  [apply]     --> ( <Exp> <Exp> ) ;
}
```

Next we *extend* this base language with numerals by simply adding the required productions:

```
language LambdaNum extends Lambda {
  Exp[zero]  --> 0              ;
  [succ]     --> succ <Exp>     ;
  [pred]     --> pred <Exp>     ;
}
```

The *extend* operator is similar to the ones proposed in [9,14].

### 2.4 Attractors

Consider the following subset of the Java grammar:

```
language JavaSubset {
  Statement[decl] --> <Declaration>          ;
  [exp]          --> <Expression> ";"        ;
}
```

```

Declaration[var] --> <Identifier> <Identifier> ";" ;
Expression[id]   --> <Identifier> ;
}

```

This language is statically intercepted by  $W_{\text{USW}}$  which produces the following error:

```
*** specificity clash: Statement[decl vs. exp] round #1 on <Identifier>
```

The reason is that we cannot discern `Statement[decl]` from `Statement[exp]` by looking at the terminal `<Identifier>` only. To solve this without rearranging the grammar and introducing phony nonterminals, we introduce a limited form of *lookahead* through a concept of *attractors*. Their syntax is either `<?t?>` where  $t$  is a terminal language or `<?n:k?>` where  $n$  is a nonterminal and  $k$  is an integer constant. Attractors are placed on the right-hand sides of productions, as in the example:

```
Statement[decl] --> <?Declaration:2?> <Declaration> ;
```

which solves our problem. If the parser can successfully consume the specified prefix of the input string, in this case the two first tokens of a `Declaration`, then it backtracks and continues with the rest of this sentential form, disregarding all other candidates. If the attractor fails, then the candidate is removed. We statically check that all attractors correspond to disjoint prefixes, so no ambiguity can be introduced (see Appendix E in [4]).

The notation `<?Declaration:2?>` is preferable to the more explicit alternative `<? <Identifier> <Identifier> ?>` because updates of the grammar are automatically reflected. Attractors can be evaluated efficiently by running the ordinary parsing algorithm while maintaining a counter of tokens consumed. Note that the complete parsing algorithm remains constant-time, since the lookahead is bounded by the constant  $k$ .

#### 2.4.1 Traps

When computing head-sets of sentential forms including attractors, we use the rule that  $\text{head-set}(\langle ?n:k? \rangle \beta) = \text{head-set}_E(n)$ . This allows us to use attractors as *traps*. To illustrate this, consider the standard Java grammar which causes a problem for our parsing algorithm. The operator `&` is a prefix of the operator `&&` but has a higher precedence. This means that an expression such as `x && y` will never be parsed correctly, since `exp[and]` will “steal” one ampersand and we will get a parse error. The solution is to add a conjunction attractor:

```

terminal AndAndTrap = { && }
AndExpressionRest[andandtrap] --> <?AndAndTrap?> ;

```

which forces `AndExpressionRest` to consume only the empty string. A different situation arises with `switch` statements, where the parsing of a branch should terminate at the following `case` construct:

```

Statement[switch] --> switch ( <Expression> ) { <SwitchBody> } ;
Statements[none]  --> ;

```

```

    [more]    --> <Statement> <Statements>           ;
SwitchBody[one] --> <Case>                           ;
    [more]    --> <Case> <SwitchBody>               ;
Case[case]     --> case <Expression> : <Statements>   ;

```

This will not happen, however, since `case` is recognized as an identifier which belongs to the head-set of statements. The solution is to apply another trap:

```

terminal CaseTrap = { case }
Statements[casetrap] --> <?CaseTrap?> ;

```

This mechanism can also be used to exclude keywords from identifiers in specific parts of the grammar.

### 3 Transformation

Transformations are typed with input and output languages and transform syntactically legal input terms to syntactically legal output terms. For each input term, three steps are performed: first, it is parsed to produce a syntax tree of the input language; secondly, this input tree is subjected to a syntax tree transformer, producing a syntax tree of the output language; and finally, this output tree is unparsed (see Section 3.4) according to the output syntax to produce the output term.

The actual transformer is thus run on parse-trees of the input language; each production kind dispatches a corresponding rule of the syntax tree transformer which names immediate constituent parse-trees, inductively applies transformers on them, and reassembles the transformed results into a resulting output syntax tree.

In order to specify result syntax trees as output terms augmented with place-holder gaps for inductively transformed terms, we extend Algorithm 1 to parse relative to a *gap environment* (as formalized in Appendix F in [4]). A gap environment,  $\tau : \overline{G} \rightarrow E$ , maps a finite set of *gap names*,  $\overline{G}$ , to *gap types* which are either terminals or nonterminals.

Example 3 illustrates the basic concepts of a transformation; it desugars the numeral extensions of the calculus, `LambdaNum`, by transforming it into the basic lambda calculus, `Lambda`. The first line names the transformation `LambdaNum2Lambda` and specifies its type by designating the *source* and *target languages*; this instructs `metafront` to load these two language definitions. The second line declares a transformer action, `Xexp`, and specifies its *source* and *target nonterminals*, respectively belonging to the source and target languages; both called `Exp` in the example.

Hereafter comes the actual rules for how to transform a `LambdaNum.Exp` syntax tree into one from `Lambda.Exp`; there must be a rule for each production of the source nonterminal. Each individual rule has four parts; a production name, a binding part naming all constituent terminal and nonterminal variables, a number of inductive transformer applications, and a result construction part.

**Example 3.1** *A Transformation, LambdaNum2Lambda*


---

```

transformation LambdaNum2Lambda: LambdaNum ==> Lambda {
  transform Xexp: Exp ==> Exp;

  Xexp[id]      (I)                ==>  << <I>      >>
    [lambda]    (I,E)  E.Xexp()=>X  ==>  << \ <I> . <X> >>
    [apply]     (E,F)  E.Xexp()=>X, F.Xexp()=>Y ==> << ( <X> <Y> ) >>
    [zero]      ( )                ==>  << \z.z      >>
    [succ]      (E)      E.Xexp()=>X  ==>  << \ n . <X>   >>
    [pred]      (E)      E.Xexp()=>X  ==>  << ( <X> \z.z ) >>

  ┌──────────┐ ┌────────┐ ┌──────────────────────────┐ ┌──────────────────┐
  │ productions │ bindings │ inductive transformers │ result construction │
  └──────────┘ └────────┘ └──────────────────────────┘ └──────────────────┘
}

```

---

Consider the second last **Xexp** rule, **[succ]**. The rule name, **succ**, refers to the production with the same name in the source nonterminal, **LambdaNum.Exp**. Since this production has one terminal-or-nonterminal variable, namely the nonterminal **<Exp>**, it must be bound in the binder part; this rule names it **E**. The inductive transformer part, **E.Xexp()=>X**, means that the syntax tree contained in **E** is inductively subjected the transformer, **Xexp**, to produce an output syntax tree which is named, **X**. Finally, the result of this transformer rule is obtained by inserting the syntax tree held in **X**, into the result construction for the place-holder gap, **<X>**.

*3.1 Lexical Transformation*

The terminals of the source and target languages need not be related in any way. Thus, a syntactic transformation must also perform lexical transformations. An example of such a specification is:

```

transform Esc: String ==> PCDATA;
transform Escape: StringContents ==> PCDATA;

Esc      -->  \" <Escape E> \"          ==>  << <E> >>

Escape   -->  "&" <Escape E>           ==>  << &amp; <E> >>
          -->  ">" <Escape E>           ==>  << &gt; <E> >>
          -->  "<" <Escape E>           ==>  << &lt; <E> >>
          -->  <StringContent S> <Escape E> ==>  << <S> <E> >>

```

Here, transformers are typed with terminal languages. In case of ambiguities during the processing of a token, the most specific input terminal is chosen.

Some structural restrictions apply to the transformation rules. The input productions are required to constitute a regular grammar and the output productions to constitute a regular or left-linear grammar. This ensures that we can compute regular languages describing both the possible input and

output strings. To provide the desired static safety guarantee we check that these languages are in the appropriate contravariant relationship with the declared source and target terminal languages.

### 3.2 Evaluation

Again, we have divided our evaluation into flexibility, safety, and efficiency.

#### 3.2.1 Flexibility

Our transformations are sufficiently expressive to handle many useful cases. They will of course always be limited compared to Turing-complete alternatives. Note though that we can perform more than linear transformations, since the output term may be exponentially larger than the input term. The liberal grammar structure that our parsing algorithm allows is essential to specify transformations. If we were forced to write:

```
ClassDeclaration[both] --> class <Identifier> <ExtendsOpt> <ClassBody> ;
ExtendsOpt[extends]   --> extends <Identifier> ;
                        [simple]      --> ;
```

instead of the more straightforward:

```
ClassDeclaration
  [simple] --> class <Identifier> <ClassBody> ;
  [extends] --> class <Identifier> extends <Identifier> <ClassBody> ;
```

then we could not specify independent transformations for the two kinds of classes. Finally, the inherent extensibility of `metafront` helps in structuring transformations.

#### 3.2.2 Safety

Transformations are statically checked to be type-safe with respect to the input and output languages. This is done by parsing the right-hand sides of transformations relative to an environment mapping place-holder gaps into terminals and nonterminals. Termination is also guaranteed, since inductive transformations can only be invoked on subtrees.

#### 3.2.3 Efficiency

Given a transformation,  $x$ , and input  $\omega$ , the transformation runs in optimal time  $\mathcal{O}(|\omega| + |x(\omega)|)$ . The current implementation is a prototype, but as for parsing there are ample opportunities for optimizations.

### 3.3 Default Transformations

When using `metafront` as a macro mechanism where the source is an extension of the target language, we want to write only transformations for the extended syntax. To this end we take two measures. First, the tool defines identity transformers with the same name as the nonterminals on all

overlapping productions. Second, we provide  $E()=>X$  as short-hand notation for  $E.N()=>X$ , where  $N$  is the name of the nonterminal type of  $E$  (naming a transformer which is possibly generated by default). The lambda calculus transformation above can now be written as:

```
transform LambdaNum2Lambda: LambdaNum ==> Lambda {
  Exp[zero] ()          ==> << \z.z >>
  [succ] (E) E()=>X    ==> << \n.<X> >>
  [pred] (E) E()=>X    ==> << ( <X> \z.z ) >>
}
```

Both measures can also be achieved with self-application of the `metafront` tool.

### 3.4 Unparsing

To control unparsing, we have augmented production right-hand sides with four *pretty print directives* that are ignored by the parser: `<+>`, for increasing indentation after newlines; `<->`, for decreasing indentation after newlines; `</>`, for inserting newlines followed by indentation whitespaces; and `<>`, to suppress whitespace printing between sentential form entities which is default. For convenience, these directives may be grouped sequentially, e.g. as `<+/>`.

## 4 Examples

We illustrate the use of `metafront` by sketching a number of small and larger examples. The full details of all examples are available from our project Web site at <http://www.brics.dk/metafront/>.

### 4.1 More Lambda Extensions

Continuing with the lambda calculus we add syntax for booleans:

```
language LambdaBool extends LambdaNum {
  Exp[true]  --> true          ;
  [false]   --> false         ;
  [if]      --> ( if <Exp> <Exp> <Exp> ) ;
}
```

and extend the desugaring accordingly:

```
transformation LambdaBool2LambdaNum: LambdaBool ==> LambdaNum {
  Exp
  [true] ()          ==> << \x.\y.x >>
  [false] ()        ==> << \x.\y.y >>
  [if] (E1,E2,E3) E1()=>X1, E2()=>X2, E3()=>X3 ==> << ( ( <X1> <X2> ) <X3> ) >>
}
```

This example could of course be extended to a full language with operators and functions.

## 4.2 Java Extensions

Some more substantial examples involve the full Java syntax. We use the grammar copied directly from the language definition [10], with all EBNF constructions desugared away. The full grammar contains 144 nonterminals and 335 productions. Our specificity parser reported specificity clashes in six places where we have added attractors and traps to disambiguate the grammar. Our first extension is to add a **C# foreach** construction:

```
language ForEach extends Java {
  Statement[foreach] -->
    foreach ( <Type> <Identifier> in <Expression> ) <Statement> ;
}
```

which is desugared by a corresponding small transformation:

```
transformation ForEach2Java: ForEach ==> Java {
  Statement[foreach](T,I,E,S) T()->xT, E()->xE, S()->xS ==> << {
    Iterator iterator = (<xE>).iterator();
    while (iterator.hasNext()) {
      <xT> <I> = (<xT>) iterator.next();
      <xS>
    }
  } >>
}
```

A larger Java extension originates from the Jwig project [6], which uses domain-specific syntax to manipulate XML fragments in Web services. The extension involves numerous modifications at various levels of the Java grammar. A small part introduces a new operator, “x<[g=y]”, for plugging together XML fragment values:

```
language Jwig extends Java {
  nonterminal Plugs;

  PostfixExpression[plug] --> <PrimaryExpression> <Plugs> ;

  Plugs[one] --> "<[" <Identifier> = <Expression> "]" ;
  [more] --> "<[" <Identifier> = <Expression> "]" <Plugs> ;
}
```

which uses the following transformation to produce nested invocations of a plug method:

```
transformation Jwig2Java: Jwig ==> Java {
  transform Plugs: Plugs ==> PrimaryExpressionRest;

  PostfixExpression[plug](E,P) E()->xE, P()->xP ==> << ( ( <xE> ) <xP> ) >>

  Plugs[one] (I,E) E()->xE ==> << .plug( <I> , <xE> ) >>
  [more](I,E,P) E()->xE, P()->xP ==> << .plug( <I> , <xE> ) <xP> >>
}
```

Another example shows the need for multiple transformations. We extend Java with a hypothetical mechanism for reserving named resources. To avoid

deadlock, a sequence of resources must be released in the opposite order in which they were acquired. The Java extension looks as follows:

```
language Reserve extends Java {
  nonterminal Identifiers;

  Statement[reserve] --> reserve ( <Identifiers> ) <Statement> ;
  Identifiers[one]   --> <Identifier> ;
                    [more] --> <Identifier> , <Identifiers> ;
}
```

and the transformation is defined as follows:

```
transformation Reserve2Java: Reserve ==> Java {
  transform Acqs: Identifiers ==> Statement;
  transform Rels: Identifiers ==> Statement;

  Statement[reserve](Is,S) Is.Acqs()=>xA, S()=>xS, Is.Rels()=>xR
                        ==> << { <xA> <xS> <xR> } >>

  Acqs[one] (I) ==> << acquire( <I> ); >>
  [more] (I,Is) Is.Acqs()=>xA ==> << { acquire( <I> ); <xA> } >>

  Rels[one] (I) ==> << release( <I> ); >>
  [more] (I,Is) Is.Rels()=>xR ==> << { <xR> release( <I> ); } >>
}
```

### 4.3 Java Enumerations

An example involving arguments to transformers is the definition of enumeration types in Java:

```
language Enum extends Java
  Declaration[enum] --> enum <Identifiers> ";" ;
```

The transformation uses an argument K to build an appropriate enumeration constant expressed as simple sums:

```
transformation Enum2Java: Enum ==> Java {
  transform Enums(Expression K): Identifiers ==> Declarations;

  Declaration[enum] (E) E.Enums( << 1 >> )=>X ==> <<
    static final int <I> = 0;
    <X>
  >>

  Enums[empty] ( ) ==> << >>
  [more] ( ) E.Enums( << <K> + 1 >> )=>X ==> <<
    static final int <I> = <K>;
    <X>
  >>
}
```

#### 4.4 Questionnaires

A complete example of a domain-specific language involves online questionnaires. We have defined a syntax for asking a series of questions, each with a fixed number of options as possible answers. A dependency relation ensures that each question may only be asked when other questions have been given certain answers.

We have then defined a transformation into the above mentioned Jwig extension of Java, which in turn generates a customized interactive Web service that permits users to answer questionnaires and administrators to view statistics.

#### 4.5 Self-Applications

We have observed many occasions for applying `metafront` to itself. We have defined a transformation to HTML syntax, which provides online documentation of `metafront` files in the style of JavaDoc.

Transformations from `metafront` to itself are also interesting. For the common cases of language extensions being desugared, our tool provides useful default transformations for all productions. As mentioned in Section 3.2, those can be defined through an explicit preprocessing defined as a `metafront` transformation. Similarly, we can extend the syntax for grammars to include EBNF right-hand sides.

The most ambitious self-application introduces explicit directives for precedence and associativity of operators. This can be given a semantics through a transformation into the basic `metafront` language.

## 5 Future Work

We plan to implement all optimizations mentioned in Section 2.2.3 and instead of interpretation, generate code for a parser that uses tables and control-flow embedded DFAs. The resulting parser should obtain performance comparable to that of Lex/Yacc combinations.

We want to create a typed algebra of languages and transformations, allowing operators like products and compositions. The `extends` mechanism will induce a subtype relation.

It is possible to allow transformations to use symbol tables and derived def-use links, while still retaining the safety guarantees. This would extend the expressive power considerably and enable transformations to read and write typed trees across those links.

Finally, we would like to provide alternative characterizations of the class of languages that specificity parsing can recognize.

## 6 Conclusion

The `metafront` tool provides a flexible, safe, and efficient means for parsing and performing syntactic transformations. Parsing is conducted by a novel parser algorithm, known as *specificity parsing*, which is designed to support gradual extension of a grammar by statically reporting ambiguities and errors in terms of individual productions as they are being added. Transformations are guaranteed to terminate and to map grammatically legal input to grammatically legal output.

The `metafront` tool may be used as a parser generator in which the resulting parser automatically supports a powerful syntactic macro mechanism, or as an extensible lightweight compiler generator for domain-specific languages. The implementation is available in 6300 lines of Java code under an open-source license.

## References

- [1] Lex Augusteijn. The Elegant compiler generator system. In *Attribute Grammars and their Applications*, volume 461 of *LNCS*. Springer-Verlag, 1990.
- [2] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The <bigwig> project. *ACM Transactions on Internet Technology*, 2(2), 2002.
- [3] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '02*, January 2002.
- [4] Claus Brabrand, Michael I. Schwartzbach, and Mads Vanggaard. The metafront system: Extensible parsing and transformation. Technical Report RS-03-7, BRICS, 2003.
- [5] Luca Cardelli, Florian Matthes, and Martin Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, 1994.
- [6] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. Technical Report RS-02-11, BRICS, March 2002.
- [7] J. Earley. An efficient context-free parsing algorithm. *CACM*, 13(2):94–102, 1970.
- [8] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete flexible compiler construction system. *CACM*, 35(2):121–130, 1992.
- [9] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Multiple attribute grammar inheritance. *Informatica*, 24(3):319–328, September 2000.
- [10] Sriram Sankar. Javacc java grammar, 2002. <http://www.cobase.cs.ucla.edu/pub/javacc/java/>.

- [11] Friedrich Wilhelm Schroer. *The GENTLE Compiler Construction System*. Oldenbourg Verlag, 1997.
- [12] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized lr parsers. In *Proc. Compiler Construction 2002*. Springer-Verlag, 2002.
- [13] M. G. J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: a component-based language development environment. In *Proc. Compiler Construction 2001*. Springer-Verlag, 2001.
- [14] Eric van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In Nigel Horspool, editor, *11th International Conference on Compiler Construction*, volume 2304, pages 128–142. Lecture Notes in Computer Science, Springer-Verlag, 2002.