Gel: A Generic Extensible Language

Jose Falcon, William R. Cook

Department of Computer Science
University of Texas at Austin
jofalcon@mail.utexas.edu, wcook@cs.utexas.edu

Abstract. Both XML and Lisp have demonstrated the utility of generic syntax for expressing tree-structured data. But generic languages do not provide the syntactic richness of custom languages. Generic Expression Language (Gel) is a rich generic syntax that embodies many of the common syntactic conventions for operators, grouping and lists in widely-used languages. Prefix/infix operators are disambiguated by white-space, so that documents which violate common white-space conventions will not necessarily parse correctly with Gel. With some character replacements and adjusting for mismatch in operator precedence, Gel can extract meaningful structure from typical files in many languages, including Java, CSS, Smalltalk, and ANTLR grammars. This evaluation shows the expressive power of Gel, not that Gel can be used as a parser for existing languages. Gel is intended to serve as a generic language for creating composable domain-specific languages.

1 Introduction

The traditional approach to implementing concrete syntax for a language is to define a custom grammar and a parser to read the language, and possibly a pretty-printer to output or reformat programs. Examples include programming languages, grammars for parser generators, configuration files, CSS styles, and makefiles.

C/Java:	`int m(int[] a) { return o.m(2 * a[x++], !done); }`
CSS:	`a:link { font-family: courier; color: #FF0000 }`
Smalltalk:	`^ o m: 2 * (a at: x inc) n: done not.`
ANTLR:	`call : ID '(' (a=e (',' b=e { a.add(b); })* )? ')';`

A custom-designed language is generally easy for humans to read and write, although they must learn specialized syntax and lexical conventions for each language. There are many tools for creating custom languages [19, 25, 17] and also for creating extensions to custom languages [7].

A second approach is to use a generic language that provides a standard concrete syntax representing generic abstract trees. Examples of this approach include XML [9] and Lisp S-Expressions [22]. A custom language can be defined within a generic language as a *subset*: the Lisp programming language is a subset of S-Expressions, XHTML is a subset of all possible XML documents. Krishnamurthi has called this technique *bicameral parsing* [21]. A language designer can choose how to encode high-level concepts using the generic syntax. For example, if x<3 then print(x) could be represented this way:

Lisp: (if (< x 3) (print x))

XML: <if><test op=”lt”><var name=”x”/><const>3</const></test>
<then><call fun=”print”><arg>x</arg></call></then></if>

It is easy to embed or compose different languages in one document. Humans only have to learn one set of syntactic conventions. Parsing, pretty-printing, and other tools can be reused.

A major negative of generic languages is that humans generally find them less appealing to read and write than custom languages. Compared to languages with a custom grammar, both Lisp and XML are impoverished syntactically: a few delimiters and simple syntactic forms are repeated many times.

In this paper we present Generic Extensible Language (Gel), a language that embodies many of the common syntactic conventions popular languages, including C/Java, Smalltalk, CSS and EBNF. The goal of this research is to define a generic language based on the syntactic conventions that have evolved over the last 40 years.

Gel has a uniform fixed syntax supporting arbitrary prefix, suffix, and infix operators, lists, grouping, keywords, sequences of adjacent expressions. and string interpolation. It has a novel quoting construct to support meta-languages.

We evaluate Gel by analyzing how well the Gel AST corresponds to the AST created by a traditional parser. Ignoring some conflicts in operator precedence, Gel extracts a good representation of the structure of Java programs, CSS styles, Smalltalk programs, and Corba IDL definitions. These examples demonstrate the expressive power of Gel. The goal is not to create actual parsers for these languages, but to use Gel as a standard input format for future domain-specific languages. The benefit is that Gel can easily parse embedded languages; for example, allowing a Java-like language to include CSS-like fragments directly in an expression, without switching to a different parser.

2 Introduction to Gel

This section introduces Gel by example. For reference, an informal summary of Gel syntax in EBNF is given in Figure 1. A formal grammar is given in Section 3. Gel expressions include familiar identifiers, numbers, strings which can be combined with binary operators and grouped in the familiar way.

`s1 = x * 3 && c == "str"` (&& (= s1 (* x 3)) (== c ”str”))

`(s1 = (x * 3)) && (c == "str")` (&& (= s1 (* x 3)$^{()}$)$^{()}$ (== c ”str”)$^{()}$)

`{s1 = [x * 3]} && [c == "str"]` (&& (= s1 (* x 3)$^{[]}$)$^{\{\}}$ (== c ”str”)$^{[]}$)

The expression on the left is input text. The expression on the right is a Lisp-like presentation of the generic abstract syntax (GAST) that results from parsing the text on the left using Gel. The grouping symbols are indicated in GAST by a superscript. This convention emphasizes that parenthesis are often ignored in the semantic processing of

```
                                 Precedence of expressions, highest first
e := s | t | (e) | [e] | ␣•␣     9. symbol, string, group, op
   | ‘e                          8. quoted expression
   | ee                          7. sequence/function application
   | e•e                         6. binary without spaces
   | ␣•e | e•␣ | ␣•e•␣           5. unary prefix and/or suffix
   | e␣e                         4. sequence with space
   | e␣•␣e                       4. binary with spaces
   | e␣? , ␣?e                   3. comma list
   | e: | :e | :e: | {e}         2. keyword forms and braces
   | e␣? ; ␣?e                   1. semicolon list
• := ‘*[., ^~, */%, +-, @#, <>, !=, &, |, :?$]+  arbitrary operators
s := [a-zA-Z0-9_]+                               symbols
t := 'r*' | "p*"                                 strings
r := \xXX | \uXXXX | \[tnr'"\$] | char           text encoding
p := $s•??g* | r                                 string interpolation
g := [e] | (e) | {p*}                            interpolation group
␣ := white-space or begin/end of group/file
```

Fig. 1. Informal summary of Gel syntax with expression precedence

expressions. Remember that Gel only specifies syntax; the semantics of these notations are defined by the particular language encoded using Gel.

Gel interprets any contiguous digits, letters, and underscores as *symbols*. As a result, Gel accepts `3F5BA2`, `10pt`, `3_D` and `10e23` as symbols. The validity of these symbols is determined by the client program using Gel. Handling of more complex floating point formats is discussed in Section 2.4.

The set of operators is not fixed. Instead, operators are constructed like identifiers: any combination of operator symbols is an operator.

```
{1..9} :-> [c =*= "str" ]          (:-> (.. 1 9){} (=*= c ”str”)[])
```

Several other languages, including Haskell[18], Scala[23] and Smalltalk[15], allow arbitrary infix operators.

The precedence of most operators is defined by their first character as defined in Figure 2. There is a special case for assignment operators [23] which end in [=] and do not start with [!=<>]. Throughout this paper, [abc] represents the *set* of characters {a,b,c}.

The comma, semicolon, and grouping characters are called *punctuation* in Gel. Punctuation symbols do not combine with other operators, and are always taken as single characters. Also, white space is always ignored around punctuation, while it is significant around other operators, as described below.

Multiple uses of the same operator are collected together into an n-ary application, so they have no associativity. Different operators with the same precedence level use right-associativity. While the operators resemble the precedence of many languages, they do not match any perfectly. Although Gel can parse Java code, some operators are

precedence	*first character*	*middle*	*last*	*description*
13	[.]	any	not [=]	dots
12	[^~]	any	not [=]	high
11	[*/%]	any	not [=]	multiplicative
10	[+-]	any	not [=]	additive
9	[@#]	any	not [=]	middle
8	[<>]	any	any	relational
7	[!=]	any	any	equality
6	[&]	any	not [=]	and
5	[|]	any	not [=]	or
4	[:?$]	any	not [=]	low
3	not [!=<>] if len>1	any	[=]	assignment
2	[,]	—	—	comma list
1	[;]	—	—	semicolon list

where any = [^~*/%+-@#!=<>&|:?$‘]

Fig. 2. Gel precedence table of operator patterns and precedence levels.

given the wrong precedence; the goal of Gel is not to create a better Java parser, but to be able to parse Java-like languages generically. Gel does not support ternary operators, but Java's `c ? a : b` operator can be parsed as a combination of binary operators.

`c ? a : b + 2`	(? c (: a (+ b 2)))

Many programming languages use comma and semicolon to represent lists of identifiers and lists of statements.

`{ 2, 3, 5, 7, 13 }`	(, 2 3 5 7 13)$^{\{\}}$
`one; two; three`	(; one two three)
`(a, 1); (a + 1, b + 2)`	(; (, a 1)$^{()}$ (, (+ a 1) (+ b 2))$^{()}$)
`a, 1; a + 1, b + 2`	(; (, a 1) (, (+ a 1) (+ b 2))

Comma has higher precedence than semicolon, and they both have lower precedence than other operators, so the last two examples above are equivalent. An empty object ϵ is inserted when list items are missing, even at the end of a list:

`a,,b`	(, a ϵ b)
`{ a *= b + 1; }`	(; (*= a (+ b 1)) ϵ)$^{\{\}}$

Operators are treated as symbols in situations where they do not make sense as binary (or unary) operations. Comma and semicolon are always treated as operators, unless they are directly enclosed in a group.

`ops = (*, +, -, /)`	(= ops (, * + - /)$^{()}$))
`others = [(,) + (;) + ($)]`	(= others (+ ,$^{()}$;$^{()}$ \$$^{()}$)$^{[]}$)

2.1 Unary Operators

Any operator (other than comma and semicolon) can be used as a prefix or suffix unary operator, on any expression:

`x?, *p++, !done, pat*`	(, [x]? *[p]++ ![done] [pat]*)

In the abstract notation on the right, unary operators have a special notation. For any operator $\circ$, the prefix form is $\circ[x]$, the suffix form is $[x]\star$ and a combined prefix/suffix form is $\circ[x]\star$.

The combination of binary and unary operators allows Gel to represent the typical notation for regular expressions which are also used in modern versions of Extended BNF and other notations for patterns.

`(a+ | b+)? | x*`	(| [(| [a]+ [b]+)]? [x]*)

Although a period is often used to represent a wildcard pattern, in Gel it works better to use _ since it is a symbol, not an operator.

Gel does not support compound grouping symbols, although they can be represented by a prefix and/or suffix operator on a standard group.

`@[ "a", "b" ]`	@[(, “a” “b”)$^{[]}$]
`=[x, y, z]=`	=[(, x y z)$^{[]}$]=
`<{ #2342; @:option** }>`	<[(; #[2342] @:[option]**)$^{\{\}}$]>

2.2 Sequences

Gel allows *sequences* of expressions that are not separated by an operator. In Haskell sequences of expressions denote function application. In Smalltalk a sequence of identifiers following an expression represent postfix unary operators. In both cases sequences have higher precedence than binary operators.

Haskell: `f a 3 + g 10`	(+ (_ f a 3) (_ g 10))
Smalltalk: `obj size + item max`	(+ (_ obj size) (_ item max))

In the abstract syntax on the right, a sequence a b is represented by an underscore operator: (_ a b). It does not matter to Gel that the interpretation of these syntactic forms is completely different in Haskell and in Smalltalk. What matters is that they follow common syntactic conventions.

Java and C do not have explicit sequence operators, but sequences arise in declarations, statements and in some expressions.

`static int f (int x, bool y)`	(_ static int f (, (_ int x) (_ bool y))$^{()}$)
`if (x > y) { return x; }`	(_ (> x y)$^{()}$ (; (_ return x) ϵ)$^{\{\}}$)
`(String) x == a [i]`	(== (_ String$^{()}$ x) (_ a i$^{[]}$)

Sequences are also used in grammars and regular expressions.

`p ::= id | '(' p ')'`	(::= p (| id (_ ’(’ p ’)’)))
`('+' | '-')? ('0' .. '9')+`	(_ [(| ’+’ ’-’)$^{()}$]? [(.. ’0’ ’9’)$^{()}$]+)

Sequences enable Gel to parse compound expressions without any specific information about what the sequence should contain.

2.3 Spaces

The combination of arbitrary infix, prefix and suffix operators with sequences of expressions is highly ambiguous. There would not be any reasonable way to parse the following generic grammar without additional syntactic clues:

```
e ::= e op e | op e | e op | e e
```

The simple expression `a + * b` can be parsed five different ways (assuming + and * are separate operators). Gel is based on common conventions for formatting expressions, using white spaces, that distinguish these cases. Parsers are traditionally written to ignore white-space, but humans do not ignore it. Gel uses white-space to distinguish three of the interpretations of this expression (here, an equivalent parenthesized version is provided):

```
a + *b ≡ a + (*b)          (+ a *[b] )
a+ * b ≡ (a+) * b          (* [a]+ b)
 a+ *b ≡ (a+) (*b)         (␣ [a]+ *[b] )
```

Two other interpretations require parentheses in Gel:

```
(a+)* b ≡ ((a+)*) b        (␣ [ [a]+ ]* b)
a +(*b) ≡ a (+(*(b))       (␣ a +[ *[b] ] )
```

There is one remaining way to include white-space in the expression. It is not clear how this expression should be parsed.

```
a + * b
```

One option is to make it an error. However, it is similar to a more common situation with a freestanding operator before or after an expression. In this case, Gel interprets the operator as if it were in parentheses:

```
   * x + 3 ≡ (*) x + 3                (␣ * (+ x 3))
 a [@ 1] $ ≡ a [(@) 1] ($)            (␣ a (␣ @ 1)▯ $)
[+ -, * /] ≡ [(+) (-), (*) (/)]       (, (␣ + -) (␣ * /))▯
   a + * b ≡ a (+) * b                (* (␣ a +) b)
```

The final example illustrates how the ambiguous expression above is covered by this rule. The last operator is taken as a binary operator, while previous operators are parsed as symbols. The motivation for this choice is to allow Gel to act as a flexible tokenizer. Gel does not reject expressions that might have a meaningful interpretation.

Spaces *are* significant in most languages. For example, in Java `int x` does not mean the same thing as `intx`. Fortran is the only language we know for which spaces are truly optional [1].

The precedence rules for sequences and operators do not allow Java to be parsed perfectly, even using the pretty-printing conventions. In some cases sequences should have higher precedence than comma, and in other cases comma should have higher precedence:

```
fun(int x, int y)                  (_ fun (, (_ int x) (_ int y))⁰)
int x, y;                          (, (_ int x) y)
```

There is no way to parse both `int x, int y` and `int x, y` correctly with generic precedence for sequences and comma. Gel assigns sequence higher precedence than comma so that function headers, as in the first example, could be parsed correctly. This precedence ordering, however, does not correctly parse the second example.

2.4 Spaced and Non-Spaced Operators

Spacing also affects the interpretation of binary operators in Gel. The examples in Section 2.2 depend on sequences of expressions having higher precedence than binary operators. However, there are other situations in which binary operators should have higher precedence. One example comes from ANTLR grammars, which use an equal sign as a high-precedence binary operator.

```
exp : a=term ( ’+’ b=term )*
```

Does “`a=b c`” parse as (= a (_ b c)) or as (_ (= a b) c)? In this case the desired parse is the latter, but this violates the rule that sequences have higher precedence than binary operators. Note that the convention in ANTLR is to have *no white-space* around the = operator, in contrast to the convention when formatting assignment operators in Java.

The solution in Gel is to make operators without white space have higher precedence than operators surrounded by white-space, including sequences separated by white-space. This is clearly a controversial decision. It matches conventions in languages as diverse as Haskell and ANTLR, as examples illustrate below. Using this rule, the ANTLR expression parses correctly in Gel:

```
exp : a=term (’+’ b=term)*
                         (: exp (_ (= a term)⟨⟩  [(_ ’+’ (= b term)⟨⟩)⁰]* ))
```

The general rule is that a chunk of text with no spaces or punctuation is always parsed as a unit, as if it were parenthesized. These chunks are then combined by any operators with spaces. The same precedence rules are applied to non-spaced and spaced operators. Thus white-space acts as an implicit grouping operator, in effect a kind of parentheses. This idea is represented explicitly in the abstract representation using a ⟨⟩ as a grouping operator.

Unary prefix and suffix operators can only occur at the beginning or end of a chunk, and they always apply to the result of the entire chunk.

```
2 * -3.14^20                       (* 2 –[(^ (. 3 14) 20)] )
x=A* | y=B?                        (| [(= x A)]*  [(= y B)]? )
&a+b+c*                             &[(+ a b c)]*
```

The first example illustrates how the decimal point in floating point numbers is interpreted as a binary operator. Java breaks sequences of operator characters into tokens, but Gel does not. For example, the Gel expression `x--*++y` has the binary operator `--*++` but in Java it parses as `(x--)*(++y)`. In Gel it must be written `x-- * ++y`. Java is not completely consistent in this respect, because it fails to parse `x+++++y`.

A sequence without spaces, which has high precedence than all other operators, can be used for casting, function application and array access in Java. Note that sequence has higher precedence than dot in Gel, but lower precedence in Java.

`f(x, y)[n]`	(_ f (, x y)$^{()}$ n$^{[]}$)$^{\langle\rangle}$
`(Integer)a.b`	(. (_ Integer$^{()}$ a) b)$^{\langle\rangle}$
`(Integer) a.b`	(_ Integer$^{()}$ (. a b)$^{\langle\rangle}$)
`o.m(a)`	(. o (_ m a$^{()}$))$^{\langle\rangle}$
`o.m (a)`	(_ (. o m)$^{\langle\rangle}$ a$^{()}$)

These examples illustrate how spaces affect the grouping of operators. The punctuation characters (parentheses, brackets, braces, comma and semicolon) are always interpreted the same whether or not they have white-space around them.

Gel can also parse typical email addresses and URLs, although it does not conform to the full specification of either.

`wcook@cs.utexas.edu`	(@ wcook (. cs utexas edu))$^{\langle\rangle}$
`http://google.com/search?query=Gel&n=1#m`	(:// http (? (/ (. google com) search) (# (& (= query Gel) (= n 1)) m)))$^{\langle\rangle}$

This example is only meant to be suggestive of the kinds of notations that Gel could parse, in more restricted contexts. The actual email and URL standards [11, 2] allow many other characters that would be interpreted as operators in Gel and ruin the parse.

The Haskell period symbol uses a special case of the general rule for spaces and operators. Without spaces, the period between identifiers represents module paths, but with spaces it is a binary operator, as seen in this one-line implementation of the Unix sort command:

```
(sequence . map putStrLn . List.sort . lines) =<< getContents
```

Gel parses this Haskell expression correctly:

(=<< (. sequence (_ map putStrLn) (. List sort)$^{\langle\rangle}$ lines)$^{()}$ getContents))

2.5 Keywords and Curly Braces

A keyword is a special identifier often used to indicate a particular syntactic structure. In most languages keywords are reserved words that cannot be used for any other purpose. One common use is to identify control flow structures, for example `for`, `while`, `if`/`else`, `switch`/`case`, `try`/`catch` and `return`. Some keywords act as operators, for example `new` and `instanceof` in Java. The set of keywords differs from language to language. Some languages, including Smalltalk, do not have any keywords.

Many uses of keywords in Java can be parsed in Gel without any specific information about keywords.

`while (!b) { b = next(); }`	$(_{\_}\ \text{while}\ ![\text{b}]^{()}\ (;\ (=\text{b}\ (_{\_}\ \text{next}\ \epsilon^{()})^{\langle\rangle})\ \epsilon)^{\{\}})$
`p = new Point(3, 4)`	$(=\text{p}\ (_{\_}\ \text{new}\ (_{\_}\ \text{Point}\ (,\ 3\ 4)^{()})^{\langle\rangle}))$
`if (a>b) f(i); else return;`	$(;\ (_{\_}\ \text{if}\ (>\text{a b})^{()}\ (_{\_}\ \text{f x}^{()})^{\langle\rangle})\ (_{\_}\ \text{else return}))$
`if (e instanceof Point) m(e)`	$(_{\_}\ \text{if}\ (_{\_}\ \text{e instanceof Point})^{()}\ (_{\_}\ \text{m e}^{()})^{\langle\rangle})$
`for (i = 9; i > 1; i--) f(i)`	$(_{\_}\ \text{for}\ (;\ (=\text{i 9})\ (>\text{i 1})\ [\text{i}]--)^{()}\ (_{\_}\ \text{f i}^{()})^{\langle\rangle})$

This is not a general solution, however. The statement `return x + y` parses incorrectly as `(return x) + y` because sequence has higher precedence than +. A similar situation happens in ML or Haskell, which do not require parentheses as in Java and C, so control flow statements do not parse correctly in Gel.

`if a = b then 1 else 2`	$(=\ (_{\_}\ \text{if a})\ (_{\_}\ \text{b then 1 else 2}))$

These examples illustrate a common purpose for keywords — to label or combine expressions to form statements. When viewed from this perspective, keywords can be understood as a kind of low-precedence operator. In Gel, keywords are identified by a prefix or suffix unary colon operator. Keywords enable more of Java to be parsed correctly with Gel:

`return: x + y;`	$(;\ (_{=}\ [\text{return}]\text{:}\ (+\ \text{x y}))\ \epsilon)$
`if: a = b then: 1 else: 2`	$(_{=}\ [\text{if}]\text{:}\ (=\text{a b})\ [\text{then}]\text{: 1}\ [\text{else}]\text{: 2})$

Keywords have precedence greater than semicolon but less than comma. In the abstract syntax (on the right) keywords are combined by a double-barred sequence operator, $_{=}$. Gel generalizes the notion of a keyword to allow any expression with a prefix or suffix colon operator to be a keyword.

`n-val: 23; (test): 5`	$(;\ (_{=}\ [(\text{- n val})]\text{: 23})\ (_{=}\ [\text{test}^{()}]\text{: 5}))$

In addition, groups in curly braces are also treated as keywords. This convention mirrors usage in C/Java and CSS, where such groups are not included in sequences. Compare these examples:

`a + b [ more ]` $\equiv$ `a+(b[more])`	$(+\ \text{a}\ (_{\_}\ \text{b more}^{[]}))$
`a + b { more }` $\equiv$ `(a+b){more}`	$(_{=}\ (+\ \text{a b})\ \text{more}^{\{\}})$

As a result, Gel parses these forms correctly:

`class: C implements: A, B { ... }`

$(_{=}\ [\text{class}]\text{: C}\ [\text{implements}]\text{:}\ (,\ \text{A}\ (_{\_}\ \text{B}\ \ldots^{\{\}})))$

`.info,h1 { color: #6CADDF }`

$(_{=}\ (,\ .[\text{info}]\ \text{h1})\ (_{=}\ [\text{color}]\text{:}\ \#[\text{6CADDF}])^{\{\}})$

`if: (b) { ... } a = 3;`	$(;\ (_{=}\ [\text{if}]\text{:}\ (_{\_}\ \text{b}^{()}\ \ldots^{\{\}}))\ (=\text{a 3})\ \epsilon)$

If these groups were not treated the same as keywords, they would parse as

```
   implements: A, (B { ... })
```
and
```
   if: (((b) { ... } a) = 3);
```
The last example above illustrates a final special case: when a curly group is inside a semicolon operator, the group has an implicit semicolon added after it. In C++ the semicolon is required after a class declaration, but not after a method body. This special case for curly groups affects some other languages badly. For example, many parser generators use curly groups to enclose parser actions, so they do not parse correctly in Gel. The solution is to add a unary operator to the group, as in `*{ ...}`, or to use a different grouping operator.

Gel also cannot meaningfully parse languages that use keywords for grouping, e.g. `begin`/`end` or `if...end if`. Parsing these examples correctly would require specific knowledge of the structure of statements.

There is a special case for keywords or curl braces that are the direct argument of a binary operator. In this case they keyword is nested inside the binary operator.

`p = new: Point(3, 4)`	(= p ($\_{=}$ [new]: ($\_$ Point (, 3 4)$^{()}$)$^{\langle\rangle}$))
`x = {a} + b * test: x`	($\_{=}$ (= x (+ a$^{\{\}}$ (* b ($\_{=}$ [test]: x)))))
`b * k1: k2: 99`	(* b ($\_{=}$ [k1]: [k2]: 99))

The design of keywords is the most difficult part of Gel. We explored the option of user-defined keywords in a document or block header, but this complicated the language and interrupted the flow of content in a document. The colon marker is lightweight and explicit.

2.6 Quoting

Quoting is useful to indicate that a syntactic form has a special meaning. In Lisp, any expression can be quoted. Syntactically, this wraps the expression in a list beginning with the symbol quote, which tells the Lisp interpreter to use the expression as a literal data value. Quotes are also useful in defining grammars. They can be used to distinguish the syntax being defined from the meta-syntax of the grammar definition language. To illustrate, first consider a conventional presentation of the syntax of EBNF in EBNF:

```
grammar EBNF {
  grammar ::= "grammar" id "{" rule (";" rule)* "}" ;
  rule    ::= id "::=" pat ;
  pat     ::= id | str | pat pat | pat "|" pat | pat "*" ;
  id      ::= letter+ ;
  str     ::= quote any* quote
}
```

This is a typical grammar for parsing text streams, in which the tokens of the language being defined are enclosed in quotes. It assumes that the patterns `letter` and `quote` are predefined. This grammar is highly ambiguous, requiring significant work to resolve these ambiguities. More work would be needed to deal with white-space.

Gel suggests another possibility where the operators `"|"` and `"*"` are parsed as actual operators rather than strings. The operators that are part of the language being defined are marked with a backquote character:

```
id | ‘id | pat pat | pat‘|pat | pat‘*
```

This is a tree grammar [14] that recognizes Gel trees that represent EBNF patterns. The expression `pat‘|pat` is written as a chunk (without spaces) so that it will have higher precedence than the other `|` operators. In the example below it is parenthesized instead. The full grammar is below:

```
grammar EBNF {
 grammar ::= (‘grammar ID { rule* });
 rule    ::= (ID ‘::= pat);
 pat     ::= ID | ‘ID | pat pat | (pat ‘| pat) | pat‘*;
}
```

In Gel any expression or operator can be quoted. A quoted operator has exactly the same precedence as its unquoted version. That is, Gel will create the same structure for a quoted expression and an unquoted version.

```
(_ grammar EBNF
  (; (::= grammar (_ ‘grammar ID [rule]* ^{})^())
     (::= rule (‘::= ID pat)^())
     (::= pat (| ID ‘ID (_ pat pat) (‘| pat pat) [pat]‘* )))^{})
```

Gel quoting can also be combined with a prefix operator to implement back-quote substitution as in Lisp. This kind of structural substitution has a counterpart in strings as defined in the next section.

2.7 Strings and Interpolation

Many languages allow variables or expressions to be embedded inside a string, a technique called *string interpolation*. For example, `"the $nth word"` is equivalent to `"the " + nth + " word"`. String interpolation is a short-hand for string concatenation. In Gel the `$` character can be followed by an optional symbol, then an optional operator, and then any number of groups. The parenthesis and square bracket groups contain Gel, while the curly bracket groups enclose *strings*. That is, the text inside `${...}` is implicitly quoted and can contain additional interpolations.

```
"$heading[2+n]{Section $n} equation: $={2+n}"
(+ (_ heading * (+ 2 n)^[] (“Section ” n^$)^{})^$ “ equation ” (_ = “2+n”^{})^$)
```

Note that Gel’s interpolations generalized both Perl notation and also TEX [20]. After substituting `$` for \, Gel can extract meaningful structure from many (but not all) TEX documents. Gel could be used for a Latex-like formatting language, but the Gel operator syntax could be used for math instead of text encoding as in TEX.

3 Gel Specification

Gel is defined by a concrete grammar, an abstract syntax, and a set of rewrite rules to handle keywords. The grammar of Gel is given in Figure 3. As is standard, x* means zero or more repetitions of x, x+ is one or more, and x? means zero or one copy of x. A set of characters in brackets `[abc]` represents exactly one character from the set. Character sets preceded by a ¬ symbol represents exactly one character that is not in the character set, and sets superscripted by a number n represent n repetitions. Ranges may also appear in superscripts as $n-m$. White-space tokens are not ignored, but are represented explicitly in the grammar as `[ ]`. Comments can only occur in conjunction with white-space. The reference parser for Gel is defined using Rats! [17], a Parsing Expression Grammar system [13]. Syntactic predicates are needed in the rule for B_{n+1} to identify extra operators as defined at the end of Section 2.3. More details and the Gel implementation are available for download at `http://www.utexas.edu/users/wcook/Gel`.

```
 expression ::= list quote? [;] expression | list
       list ::= optional quote? [,] list | optional
   optional ::= [ ]? B_3? [ ]?
        B_i ::= B_{i+1} [ ] op_i [ ] B_i | B_{i+1}        for i ∈ {3..n}
    B_{n+1} ::= op [ ]?  | (op [ ])* sequence ([ ] op )*
   sequence ::= chunk ([ ] chunk)*
      chunk ::= op? C_3 op?
        C_i ::= C_{i+1} op_i C_i | C_{i+1}        for i ∈ {3..n}
    C_{n+1} ::= primary+
       op_i ::= quote? [‘:$@?|&!=<>+-*/\%˜ˆ#.]+
                   where i is the precedence as defined in Figure 2
         op ::= op_1 | . . . | op_n
      quote ::= [‘]+
    primary ::= quote? (symbol | group | string1 | string2)
     symbol ::= [a-zA-Z0-9_]+
      group ::= [{] expression [}] | exprGroup
    string1 ::= [′] (escape | ¬[′] _)* [′]
    string2 ::= ["] (escape | interpolate | ¬["] _)* ["]
     escape ::= [\][u][0-9A-F]^4 | [\][0-7]^{1−3} | [\] _
interpolate ::= [$] symbol? op? (string3 | exprGroup)*
    string3 ::= [{] (escape | interpolate | ¬[}] _)* [}]
  exprGroup ::= [(] expression [)] | [[] expression []]
     ignore ::= [/][/] (¬newline)* newline | [/][*] any* [*][/]
```

Fig. 3. Gel grammar, where n is the number of operator precedence levels

The first three productions represent lists, separated by semicolon (op_1) and comma (op_2), of optional items. The nonterminals B_3 through B_n represent binary expressions with op_i surrounded by spaces. The `[ ]` terminal represents any number of white-space

$$\begin{array}{r|l}
(\star\ (\star\ \bar{x})\ x) \Rightarrow (\star\ \bar{x}\ x) & (\star\ x\ (\star\ \bar{x})) \Rightarrow (\star\ x\ \bar{x}) \\
(\_\ k\ x) \Rightarrow (\underline{\_}\ k\ x) & (\_\ x\ k) \Rightarrow (\underline{\_}\ x\ k) \\
(\circ\ (\underline{\_}\ \bar{x}\ v_1)\ v_2) \Rightarrow (\underline{\_}\ \bar{x}\ (\circ\ v_1\ v_2)) & (\circ\ v_1\ (\underline{\_}\ v_2\ \bar{x})) \Rightarrow (\underline{\_}\ (\circ\ v_1\ v_2)\ \bar{x}) \\
(\_\ (\underline{\_}\ \bar{x}_1)\ (\underline{\_}\ \bar{x}_2)) \Rightarrow (\underline{\_}\ \bar{x}_1\ \bar{x}_2) & \\
(\_\ (\underline{\_}\ \bar{x})\ x) \Rightarrow (\underline{\_}\ \bar{x}\ x) & (\_\ x\ (\underline{\_}\ \bar{x})) \Rightarrow (\underline{\_}\ x\ \bar{x})
\end{array}$$

$$(;\ \bar{x}_1\ (\underline{\_}\ \bar{x}_2\ x^{\{\}}\ \bar{x}_3)\ \bar{x}_4) \Rightarrow (;\ \bar{x}_1\ (\underline{\_}\ \bar{x}_2\ x^{\{\}})\ (\underline{\_}\ \bar{x}_3)\ \bar{x}_4)$$

$\star, \circ \in$ op, $\circ \notin [\![$`;`$]\!]$, x is any Gel, $k \in \{\ [x]:,\ :[x],\ x^{\{\}}\ \}$, $v \notin \{\ x:, :x, x^{\{\}}, (\underline{\_}\ \bar{x})\ \}$

Fig. 4. Keyword rewrite rules

characters, including single spaces, tabs, return feeds and new lines. These nonterminals have lower precedence than *sequence*, which is a list of chunks that do not contain spaces. The B_{n+1} rule allows operators to be part of a sequence, when they cannot be interpreted as binary operators. The nonterminals C_i are analogous to B_i except the operators do not have spaces. The C_{n+1} rule allows sequences of primaries that are not separated by spaces.

Compound operators are composed of any sequence of operator characters. The precedence order of operators is given by the table in Figure 2. For most operators the precedence is given by the precedence of the first character. There is a special case for *assignment operators*, which end with equal [`=`] and do not begin with [`!=<>`].

A primary is a symbol, string, or group. The *string1* and *string2* rules define strings with single and double quotes, respectively. Both strings allow Java-style escaping with backslash. The [`$`] character is an interpolation character in double-quoted strings. It allows interpolation of Gel expressions into a string. The single back-quote character (`‘`) is used for quoting. Any operator or primary may be quoted.

$$\begin{array}{rl}
x \in \textit{Gel} = & \textit{symbol} \mid \text{“}str\text{”} \mid (\star\ x_1 \ldots x_n) \mid \star[x] \mid [x]\star \mid \star[x]\star \mid \text{‘}x \mid x^G \mid \epsilon \\
\textit{symbol} \in & [\![\texttt{a-zA-Z0-9\_}]\!]+ \\
\star \in & ; \mid , \mid [\![\texttt{‘:\$@?|\&!=<>+-*/\textbackslash\%\~{}\~{}\^{}\#.}]\!]+ \mid \_ \mid \underline{\_} \\
G \in \textit{Group} = & () \mid \{\} \mid [] \mid \langle\rangle
\end{array}$$

where $\_$ means sequence, $\underline{\_}$ means keyword sequence, $\langle\rangle$ means chunk

Fig. 5. Gel abstract syntax.

The behavior of keywords in Gel is not implemented by the parser, but is handled by the rewrite rules in Figure 4 during construction of the abstract syntax tree. The first rule combines operators to eliminate associativity. Keywords in a sequence are moved outside of other operators. The last rule adds an implicit semicolon after a group. The abstract syntax of Gel is defined in Figure 5.

4 Evaluation

We evaluate Gel by testing how well it can extract the structure from existing languages that are defined by a custom grammar. It is not enough to determine whether Gel accepts a given input, because Gel accepts almost any input with balanced grouping operators. The key question is whether Gel can extract meaningful structure from typical documents that follow standard formatting conventions. These tests were instrumental in designing Gel.

Let S be a source file of a language L, and let $L(S)$ be the AST of S created by the L parser. The same source file S can be parsed with Gel to produce a GAST, $Gel(S)$. The goal is to determine if $Gel(S)$ has the same structure as $L(S)$.

However, the $L(S)$ AST cannot be compared to the GAST because each uses a different abstract syntax. To overcome this problem, we apply the idea that the structure of an abstract tree can be made explicit in concrete syntax by adding parentheses at every level of the tree. The implementation of this idea starts with a printer P for language L having the property that $L(P(T)) = T$ for any abstract tree T in L. We then convert P into a *parenthesizing printer* P' that prints a tree T while adding parentheses around every abstract node as it is printed. The output $P'(T)$ may not be a valid instance of language L, but it can be parsed with Gel. The extra parentheses force Gel to create a parse three that mirrors the structure of $L(S)$. Gel has captured the structure of L if $Gel(P'(L(S))) = Gel(S)$ ignoring parentheses. As an example, consider this Smalltalk fragment and its parenthesized versions:

```
x min to: args size * 2 do: aBlock
            (= (_ x min) [to]: (* (_ args size) 2) [do]: aBlock)
((x) min) to: (((args) size) * (2)) do: (aBlock)
            (= (_ x^() min)^() [to]: (* (_ args^() size)^() 2^())^() [do]: aBlock^())
```

Not all languages follow Gel's syntactic standards. Although there can be significant differences, sometimes the differences are small, for example comment markers and operator choices may conflict. Smalltalk separates statements with a period, which is a high-precedence binary operator in Gel. If Smalltalk used a semicolon, as in Java, Gel would parse it more accurately. We handle minor syntactic issues by converting symbols before parsing with Gel. This change preserves the key characteristics of Smalltalk; it just uses a different symbol. A fixup transformation T for language L is applied to the files before they are parsed by Gel. These transformations are simple character or reserved word substitutions. With transformation, the comparison is $Gel(T(P'(L(S)))) = Gel(T(S))$. We have successfully applied this technique to Smalltalk, Java, CSS and CORBA IDL. For a small set of representative sample documents, Gel extracts the exact same structure as the custom parser, in all but a few cases as mentioned below. There may be other syntactic mismatches that did not show up in our test documents.

Java Gel operators and precedence are based on Java, but Gel does not have exactly the same operator precedences, so it will not parse Java precisely. We tested Gel against Java documents whose operators align with Gel precedence. Other issues in java are related to sequences, where two syntactic structures are placed next to each other with just a space between them.

- Declarations of multiple variables do not parse correctly, as described at the end of Section 2.4.
- Java keywords do not parse correctly unless they are marked as Gel keywords as mentioned in Section 2.5.
- The grammar we used for Java parses `o.m()` as (_ (. `o` `m`) ϵ^0), while Gel parses it as (. `o` (_ `m` ϵ^0)). It is debatable which of these is correct.
- Generics in Java are declared using the < and > characters, as in `Stack<String>`. We translated these to [...] before parsing.
- Typical white-space conventions must be followed: using white-space after a colon, and around binary operators. Typical white-space means that `int[] x` must not be written `int []x` although this is legal in Java, it violates coding conventions. Similarly, `p = *p2` must not be written `p =* p2`. We found that the reformat command in Eclipse corrects most spacing issues in Java documents so that they parse correctly with Gel.

Smalltalk The Gel syntax closely resembles and generalizes Smalltalk grammar. Keywords in Smalltalk are identified by a colon suffix. Arbitrary binary operators use infix notation, and have higher precedence than keywords. Unary messages are represented by a sequence of symbols separated by spaces, with higher precedence than binary operators. Parentheses, braces, and brackets are used for grouping.

There are problems with parsing using Gel:

- Statements are terminated or separated by periods. We translated these semicolons before parsing with Gel.
- Cascaded message sends are separated by semicolons. These become ambiguous if period is replaced by semicolon. We insert a special "previous" token after the semicolon to make reuse of the previous message target explicit. These message sends must also be enclosed in parentheses if the target object is returned.
- Binary operators in Smalltalk all have the same precedence.
- The conventional storage format for Smalltalk programs (the method change list) does not have grouping constructs that can be parsed by Gel.
- Typical white-space conventions must be followed: using white-space after a colon, and around binary operators.

CSS Most of CSS follows a typical structure with semi-colons and braces. CSS also uses keywords tagged with colon. It uses a variety of prefix and infix operators. However, there are problems with parsing CSS with Gel:

- Identifiers that include hyphens, e.g. `background-color`, parse as chunks in Gel. This works reasonably well, although Gel is breaking up more tokens than are necessary.
- Typical white-space conventions must be followed: using white-space after a colon, and not separating prefix operators from their.

– Pseudo-classes look like binary colon operators, of the form `link:visible`. According to one CSS grammar they should be parsed as `link (:visible)` but Gel parses them as `(: link visible)`. This does not seem like a major issue.
– The use of numbers with a dimension, as in `16pt`, is handled in Gel as an identifier, not as a sequence of number `16` and `pt`. It is simple to process these tokens to extract the dimension.

Python Although Python does not adhere to many of the conventions discussed, Gel is able to parse Python programs. The following problems must be addressed to parse Python correctly:

– In Gel, logical blocks of code can only be created using the three types of grouping operators. However, Python uses indentation to specify logical blocks of code. This is currently handled by a pre-processor, which inserts {`...`} groups according to indentation rules of Python. This preprocess is a lexical transformation.
– Many statement constructs in Python use the colon character, as in `if x is True:`. These can be discarded once grouping operators are created around the logical block.
– Python uses newline to separate statements. However, these would parse as white-space tokens in Gel, so semicolons must be inserted.

ANTLR and other parser generators We have used Gel to parse grammar specification languages, including ANTLR [24] and Rats! [17]. These languages use {`...`} as parser actions within a rule. A prefix or suffix must be added to prevent actions from terminating the expression (according to the keyword rule in Section 2.5). In addition, Rats! uses `[A-Z]` as a character class, in effect quoting an arbitrary set of characters, as in `[({]`. These must quoted as strings, or converted to the form used by ANTLR: `'A'..'Z'`.

5 Related Work

Gel is related to other generic and extensible languages, include Lisp, XML and JSON. Gel can parse Lisp-like data [22], if single-quote is converted to backqoute, comma to `$`, comments to `//`. Common lisp atoms `*val-list*` are converted to Gel chunks *[(-val list)]* with prefix/suffix * operators, which means that they have been over-analyzed but are still recognizable. Any sequence of non-punctuation characters without spaces can be parsed as Gel. Operators are the main problem, since Lisp always treats them as ordinary symbols, but Gel may parse them as binary operators. Thus `(a + b)` is incorrectly parsed as (+ a b) in Gel. To express something like the correct Lisp structure (_ a + b) the operator must be changed, for example enclosed in a group `(a {+} b)`.

XML [9] cannot be parsed by Gel at all. It uses < and > as grouping characters, and tags as grouping for large-scale units. To parse XML-like structures, a more C-like notation is needed.

<tag attr="value" ...>...</tag> ⇒ tag: attr="value" ... { ... }
text ⇒ "*text*"

Alternatively, Gel could simulate the text-oriented nature of XML and its history in HTML by using an interpolation-based translation:

$<$tag attr=”value” ...$>$...$<$/tag$>$ $\Rightarrow$ $tag(attr=”value” ...)$\{$ ... $\}$

The JavaScript Object Notation (JSON) is a subset of JavaScript that is frequently used as a generic data encoding language [12]. Correct parsing of JSON depends on consistent white-space conventions. It works well if colon is treated as a binary operator.

```
"val" : 3, "name" : "Test"   (, (: “val” 3) (: “name” “Test”))
"val": 3, "name": "Test"     (_= [“val”]: (, 3 (_= [“name”]: “Test”)))
"val": 3; "name": "Test"     (; (_= [“val”]: 3) (_= [“name”]: “Test”))
```

The keyword notation in the second example groups the values awkwardly: the second keyword is within the body of the first keyword because of the comma. If JSON used semi-colons then Gel could parse the keyword form more naturally, as in the third example.

Another approach to extensible languages involves languages whose syntax can be extended with additional rules. This approach has the advantage that specific syntax is recognized and checked during parsing. Brabrand and Schwartzbach [5] provide a detailed summary and comparison of different systems for syntax extension [10, 27, 6, 8, 16]. It is difficult to perform a direct comparison between the extensible syntax and the idea of generic extensible languages, because the two approaches are so different in their fundamental assumptions. Each clearly has drawbacks and advantages that can only be evaluated in the context of a larger system in which domain specific languages are defined and manipulated. Examples of such systems include `<bigwig>` [4, 5] and Stratego [28]. Lisp and Scheme macros provide a similar benefit in the context of the generic syntax of Lisp S-Expressions. Gel does not yet part of a complete system for language definition and syntactic extension, so it is difficult to compare its effectiveness at this level. Given that Gel is essentially a syntactic variant of Lisp S-Expressions, the techniques developed for Lisp/Scheme should work for Gel as well. This kind of validation will not be possible until different researchers experiment with using Gel in their own systems.

6 Conclusions

Gel is designed to be used as a front-end for domain-specific languages. To define a language within Gel, appropriate operators and syntactic forms are chosen, and a structure grammar is defined. The output tree from Gel must then be parsed to verify that it matches the DSL structure. This process is very much like validating against an XML Schema [26, 3] but is beyond the scope of this paper. Gel allows easy syntactic composition or embedding of different languages within each other. It may also be possible to define a generic pretty-printer for Gel.

One argument against Gel may be that its use of white spaces makes it too fragile for casual use. However, most programming languages are sensitive to adding new arbitrary spaces, or completely removing spaces. Gel accepts nearly every input document without error, as long as grouping symbols are balanced. When used for a specific DSL, error messages will come from later phases, when the output of Gel is validated against the DSL structure.

During the design of Gel numerous alternatives were tried. We have worked hard to eliminate special cases. Currently the only special cases are for assignment operators and curly braces. These special cases are relatively simple for users and provide useful options to language designers when designing a new notation. We have resisted allowing the grammar to be customized, for example by allowing external definition of a set of keywords. We plan to gather feedback on Gel for a short period of time before fixing the language specification.

References

1. J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, R. A. Hughes, L. B. Mitchell, R. A. Nelson, R. Nutt, D. Sayre, B. P. Sheridan, H. Stern, and I. Ziller. *Fortran Automated Coding System For the IBM 704*. International Business Machines Corporation, New York, 1956.
2. T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738, Internet Engineering Task Force, Dec. 1994. `http://ds.internic.net/rfc/rfc1738.txt`; accessed August 23, 1997.
3. P. V. Biron and A. Malhotra. XML Schema part 2: Datatypes. The World Wide Web Consortium `http://www.w3.org/TR/xmlschema-2/`, May 2001.
4. C. Brabrand, A. Møller, and M. I. Schwartzbach. The <bigwig> project. *ACM Trans. Interet Technol.*, 2(2):79–114, 2002.
5. C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *In Proceedings of Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2002. ACM*, pages 31–40. ACM Press, 2002.
6. C. Brabrand, M. I. Schwartzbach, and M. Vanggaard. The metafront system: Extensible parsing and transformation. In *in Proc. 3rd ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications, LDTA '03*, 2003.
7. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.
8. M. Bravenboer and E. Visser. Designing syntax embeddings and assimilations for language libraries. In *Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, pages 34–46, Berlin, Heidelberg, 2008. Springer-Verlag.
9. T. Bray, J. Paoli, C. M. Sperberg-McQueen, Eve, and F. Yergeau, editors. *Extensible Markup Language (XML) 1.0*. W3C Recommendation. W3C, fourth edition, August 2003.
10. L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical report, Research Report 121, Digital SRC, 1994.
11. D. H. Crocker. *Standard for the Format of ARPA Internet Text Messages*. University of Delaware, Department of Electrical Engineering, Newark, DE 19711, August 1982. `www.faqs.org/rtcs/rfc822.html`.
12. D. Crockford. Rfc 4627. the application/json media type for javascript object notation (json). online, http://www.json.org/, 2006.
13. B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Symposium on Principles of Programming Languages*, pages 111–122, 2004.
14. A. V. Gladky and I. A. Melčuk. Tree grammars (= Δ-grammars). In *Proceedings of the 1969 Conference on Computational linguistics*, pages 1–7, Morristown, NJ, USA, 1969. Association for Computational Linguistics.

15. A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.
16. R. Grimm. Practical packrat parsing. *New York University Technical Report, Dept. of Computer Science, TR2004-854*, 2004.
17. R. Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM.
18. P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, 1992.
19. S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.
20. D. E. Knuth. *The TEXbook*. Addison-Wesley, 1984.
21. S. Krishnamurthi. *Programming Languages: Application and Interpretation*. 2006. `http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/`.
22. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.
23. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A comprehensive step-by-step guide*. Artima Inc, August 2008.
24. T. Parr and R. Quong. ANTLR: A Predicated-LL (k) Parser Generator. *Software - Practice and Experience*, 25(7):789–810, 1995.
25. T. Parr and R. Quong. ANTLR: A Predicated-LL(k) parser generator. *Journal of Software Practice and Experience,*, 25(7):789–810, July 1995.
26. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema part 1: Structures. The World Wide Web Consortium `http://www.w3.org/TR/xmlschema-2/`, May 2001.
27. E. Visser. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE02*, pages 299–315. Springer-Verlag, 2002.
28. E. Visser. Program transformation with stratego/xt: Rules, strategies, tools, and systems in stratego/xt 0.9. In C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2003.