Commodity Transformation for DSLs
LPT With lex, yacc, and XSLT

Crutcher Dunnavant crutcher@samedi-studios.com, Trevor Jay aka1@samedi-studios.com,
Joel Jones jones@cs.ua.edu, Katrina Roan roan001@bama.ua.edu,
Charles Ward omnisoft@unix.eng.ua.edu, Nathan Wiegand atreus@unix.eng.ua.edu, and
Derek Woodham derek@unix.eng.ua.edu

University of Alabama, Computer Science
Tuscaloosa, AL USA 35487-0290

Abstract. We believe that the appropriate characterization of when a language is a DSL, and when it is not, is best made through examination of the relative assignment of resources to a project. We present a new approach to the construction of text transform tools for the prodution of DSLs, and some constraints which shape this approach; not an examination of a specific implementation or architecture. We emphasize the use of ubiquitous tools to do transforms. A reader with concrete experience in some lexer and some parser generation packages, and some XML library with XSLT features, should be able to produce a tool using this approach in a day or two. The presentation is at a high enough level that the reader without this concrete and immediately applicable knowledge can still follow the approach. The goal is to make the production of small domain-specific languages common by promoting simplicity of implementation and use of the tools at hand. We use as an example the development of a parser for a simple parenthesis language. We also present a novel way of dealing with delimited lists in a parser specification.

1 The DSL Cost Domain

Though DSLs specify a semantic application of language development, the literature of DSL discussion has always carried with it an understanding that the languages are, in some greater or lesser way, under tighter development budgets than other languages. We formalize this idea as the DSL Cost Domain.

A Cost Domain is a relative distribution of resources. As an example, in the realm of compilation, consider the Fast Code and the Small Code Cost Domains. In the Fast Code Cost Domain, the resources are distributed in such a way that target code speed is more important than space consumption; whereas the reverse is true in the Small Code Cost Domain.

But Cost Domains are not limited to the assignment of "computing" resources. The assignment of monetary, temporal, and knowledge resources may also be described by a Cost Domain. Consider the Time Critical Cost Domain, wherein time is given the highest priority. In order to meet these time constraints,

monetary resources will be assigned in a manner which would be considered too costly outside of this domain.

We are interested in the DSL Cost Domain. We believe that the appropriate characterization of when a language is a DSL, and when it is not, is best made through examination of the relative assignment of resources to a project. While it is difficult to precisely define this cost domain, and the assignment of performance constraints, money, time, knowledge, and semantic articulation which it implies; we have developed some rules of thumb to help determine when a project is within the DSL Cost Domain.

1.1 Domain Specific Languages

DSLs have existed at least since the beginning of interactive computer interfaces. We narrow our survey to the meta-level—techniques and tools for developing DSLs. Kienle defines a DSL as:

> a small formal (programming) language whose expressive power covers precisely a certain application domain and does not include many of the features found in general-purpose languages[18].

A pattern language for the discussion of DSLs is given by Spinellis[**?**], who also explores the economic constraints of the DSL Cost Domain. Spinellis states that the economic constraints influence the design techniques for DSLs. He emphasizes the role that incremental development has and the importance of rapidly delivering results. Hudak[19] also discusses the relationship between language development costs and the need for incremental design. He describes the implementation of several embedded DSLs, which he implements in Haskell through the use of higher-order functions and user-defined infix operators. Aycock[20] states that frequently a fast compiler is unnecessary, particularly if input programs are small on average. The issues that dominate in such a situation are development time, maintainability, and mutability of the language.

1.2 Three Rules of Thumb

1. Is the cost of processing the language non-critical?
2. Is it important that the language's definition remain flexible?
3. Will the number of the language's users differ from the number of the language's developers by less than two orders of magnitude?

These rules can guide implementation decisions for language projects. If, for a given language project, the answers are all "yes", then the project is clearly within the DSL Cost Domain. These rules are not necessary to place a project in the DSL Cost Domain, but they are sufficient. Judgment must be exercised in other cases.

Consider the case when, though a language's user community is large, it is expected to continue to experience rapid evolution and development. If language

processing is not cost critical, it may still be that the most cost effective application of development resources is to apply techniques from the DSL Cost Domain to the problem. If at some future time the language's development rate were to decrease, it might then become cost effective to migrate the language implementation to more sophisticated implementation techniques in order to achieve processing efficiencies.

Additionally, while an entire project might not fall within the DSL Cost Domain, some of its deliverables may. Candidate deliverables include initial language prototypes, where the flexibility of the language's definition is more important than the language's processing cost; and small glue languages needed to direct the behavior of larger sub-systems (such as the loading of extension libraries).

1.3 XML Not Harmful

A long-held tenet of faith of some in the compiler community is that XML is harmful. It has been said to be too expensive a form, in both processing and representation costs, for serious use in AST manipulation. Though this may have been true when XML arrived on the scene in 1998, the costs of processing XML have not increased dramatically since then, while computing power / cost has increased 16 fold as of the writing of this paper. For many applications, and for the DSL Cost Domain, XML should no longer be considered harmful.

As an example, Ying Zou and Kostas Kontogiannis are a pair of researchers at the University of Waterloo, active in the areas of AST representation in XML[2], and dramatic code refactorings built upon such representations[1]. In [1], they describe some experiments in which various C source bases were represented in XML, and report expansions of:

- **AVL Library** - 164,401B C $\Rightarrow$ 1,660,167B XML
- **Bash** - 628,919B C $\Rightarrow$ 25,421,443B XML
- **Tcsh** - 930,644B C $\Rightarrow$ 47,444,861B XML

These researchers are actively performing source analysis and transformation on code bases in the 50Mi size range. On a modern desktop machine, 50Mi of XML should take about 5 seconds to parse, which will mostly be the cost of disk IO, and the resulting parse tree should easily fit in memory. In cost domains where the flexibility of the language definition is more important than the language's processing cost, such as the DSL Cost Domain, we could quite easily accept such performance numbers.

2 What is LPT?

LPT is a source to source transformation approach. It is not without its limits, limits which show up in the face of transformations which require deep semantic

analysis, such as data-flow analysis. However, when analysis is properly partitioned, *deep analysis* is seldom needed; and LPT tools are easy to develop and work with, while deeper tools tend to require hundreds of man years to develop.

LPT is simply shorthand for:

1. **Lex** the character stream into terminals;
2. **Parse** the terminal stream into a node tree; and
3. **Transform** the resultant node tree.

This captures the basic partitioning of an LPT system, both at the conceptual and coding layers. LPT breaks the transformation task into well defined pipeline components which lexically analyze, parse, and transform input.

2.1 lex, yacc, and XSLT

There exist several large, relatively distinct classes of approaches to lexing, parsing, and transformation. The approach outlined in this paper makes a collection of high-level choices amongst these classes. Specifically, **LPT With lex, yacc, and XSLT** uses some **lex**-alike lexer generator, some **yacc**-alike parser generator, and some XML/XSLT[13][15] processing environment. **lex**[3] and **yacc**[3] are the most copied of the lexer and parser generators, and have clones for any environment or target language which you might seriously wish to use.

XML and XSLT The choice which shapes this approach most is that our Transform engine will be an XSLT processing environment. XSLT is an XML language for describing XML node tree transformations, and virtually all XML environments include an XSLT processor. XSLT is commonly used to yield HTML, PostScript, PDF, and WAP views of XML documents; it is used to transform between database formats in enterprise environments; and it is supported natively by all third-generation browsers (including Mozilla and IE). Thus XML and XSLT are rapidly consuming the design space for data interchange.

Since we will be using XSLT for transformations, lex and parse nodes will be XML nodes. Our XML/XSLT environment will provide XML node data structures, so we need not implement any node or tree data structures ourselves; a fact which affirms our choice of XML and XSLT.

lex and yacc Many projects exist to re-implement these generators in various environments. For **lex**, these include flex++ (C++)[6], jflex (Java)[7], alex (Haskell)[8], aflex (Ada)[9], TP lex (Turbo Pascal)[10], and many others. For **yacc**, these include C++ (in the case of Yacc++[12]) and Java (in the case of JavaCC[12]). The examples in this paper have been confirmed to work in flex, a lex clone, and bison, a GNU yacc clone, but you should be able to find a flavor for any environment which also possesses **lex** and an XML/XSLT environment.

3 Jargon

The **yacc** family of parser generators work in context-free grammars. Since productions in context-free grammars have only a single non-terminal on their left-hand side; versus the multiple non-terminals permitted in context-sensitive grammars, the jargon of **yacc** discussion tends to be imprecise. Little distinction is made between the set of all right-hand sides matching a given left-hand side, and the definition of a non-terminal. For the purposes of this paper, not only will we continue the practice of ignoring this distinction; we will also require that all productions which yield a given non-terminal on their left-hand side are collected as part of the single definition of that non-terminal.

4 Lispy

Many of the examples in this paper will play with a simplistic, Lisp-like language (called “Lispy”), which possesses applications, identifiers, numbers, and strings. A basic transformation would be to re-write a Lispy statement from its parenthetical notation to a more common parameter format, like so:

```
(foo bar (baz 'abc' 1)) ⇒ foo(bar, baz('abc', 1))
```

This paper will describe how to build up an LPT transform tool to perform tasks like this. We will examine the Lex, Parse, and Transform steps, in order; explaining each step in terms of our Lispy transformation example.

5 Semantic Trees

Computer languages deal with streams of characters. These characters have an oppositionally defined uniqueness (an 'a' is defined in as much as it is not a 'b'), and have a position within the stream. Most interpretations of an input combine characters into larger oppositionally and positionally defined units - tokens. This is the syntax level of a language, and is the area where most tools concern themselves. The literature of parser generation and transformation talks mainly in terms of *Abstract Syntax Trees* (*AST*s), that capture the syntactic information structurally.

The term AST is something of a misnomer, as most reifications of the AST idea have a structure that is built using more than syntactic analysis of the input. In addition to the definitional transformation of removing unnecessary syntactic elements (such as many terminal symbols), ASTs typically collapse the parse tree using internalized knowledge of the semantic meaning of the language being parsed. When a tree represents oppositionally and positionally defined units which result from the application of a signification code to a Syntaxtic Tree, we promote the use of the term *Semantic Tree*. This term of art is based upon the Theory of Semiotics[17].

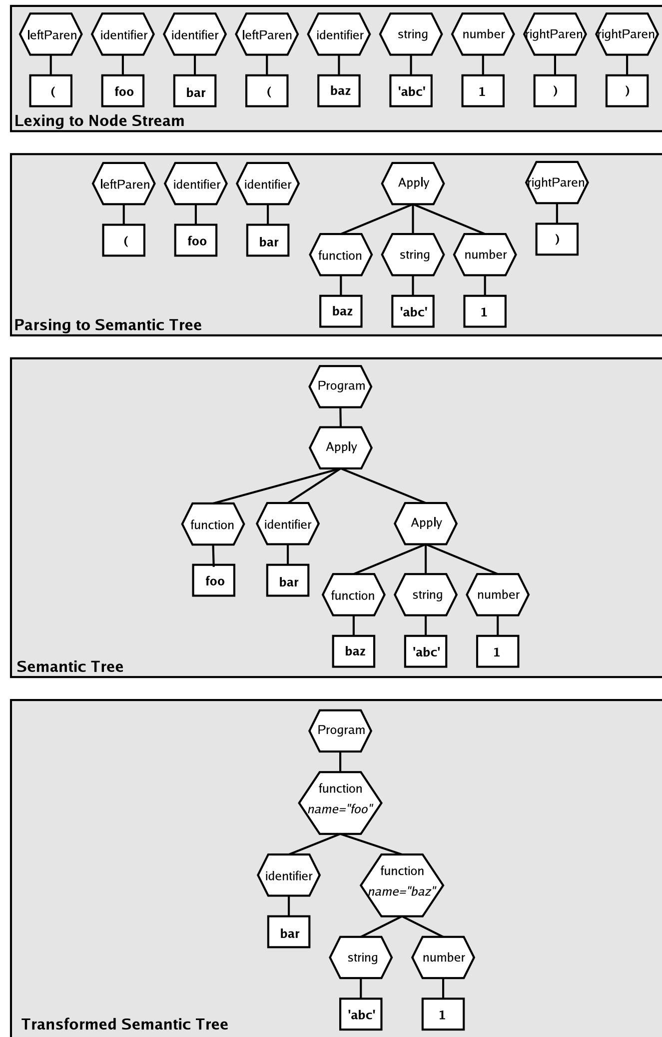

Fig. 1. Intermediate Forms

6 Lexer Generation

Lexers scan input character streams, and yield terminal streams (represented as some node or token data structure) to a parser. They perform this task by scanning characters in the input stream, and selecting sub-strings which match certain patterns. Various actions can be performed on these sub-strings, such as the construction of a terminal node to be passed on to the parser. For a given lexical match, our lexer will construct an XML sub-tree which consists of an XML element root node with the same name as the terminal, that contains an XML text node whose value is the lexical content of the terminal. Figure 1 shows the terminal node stream for our example Lispy program.

The rules used by **lex**-family lexer generators associate regular expressions with action code. These expressions are built into the generated lexer's state machine, and when their patterns match, the rule's associated action code is executed. While different lexer generators support different dialects of regular expressions (and thus different patterns must be developed for different lexer generators), there is enough similarity between regex dialects that you should have little difficulty adapting a pattern from one lexer generator to another.

In an effort to reduce the work done in parse and processing steps, many complex and subtle features have been developed for the various lexer generators available. Since we are using a full transform environment (XSLT), we have little need for such cleverness at the Lex stage, and so we cast **lex** actions into three distinct and limited classes:

1. **match** rules - which produce terminal nodes for lexical tokens;
2. **ignore** rules - which consume characters, but take no other actions; and
3. **error** rules - which specify patterns which should yield lexical errors.

We represent these rules in XML, using a language with elements **match**, **ignore**, and **error**; as shown in Figure 2. The regex patterns for these rules are the text contents of the elements; and a **match** rule's **term** attribute sets the name of the terminal node produced.

```
<match term="leftParen">"("</match>
<match term="rightParen">")"</match>
<match term="identifier">[[:alpha:]][[:alnum:]]*</match>
<match term="number">[[:digit:]]+</match>
<match term="string">'[^']*'</match>
<ignore>[^[:print:]]|[\n[:blank:]]</ignore>
<error>.</error>
```

Fig. 2. Lispy lexer rules

6.1 Lexer Action Code

Ignore Rules The **ignore** rules usually produce relatively trivial action code. These rules merely need to match and consume strings, and not to yield tokens. With **flex**, ignore rules rules yield:

```
PATTERN {
    /* Ignore */;
}
```

Error Rules The **error** rules require a bit more work. When they match, it means that the input is lexically malformed, and we want to halt the transformation. To do this, you'll need some form of error token, separate from all tokens used in your terminal or non-terminal sets. This token holds no lexical value, so we don't build it an XML node; it only serves to indicate to the parser that we are in a failure mode. With **flex**, error rules yield:

```
PATTERN {
    /* Error */
    yylval = NULL;
    return lispy_lex_error;
}
```

Match Rules The **match** rules capture the lexeme as a text node, and create an element node for the terminal, which contains the lexeme text node as a child. XML libraries vary, but for **flex** and **libxml**[4], our match rules yield:

```
PATTERN {
    /* Match */
    yylval = xmlNewNode(NULL, "TERMINAL_NAME");
    xmlAddChild(yylval, xmlNewText(yytext));
    return lispy_sym_TERMINAL_NAME;
}
```

Line and Column Numbering It is frequently useful to annotate these terminal nodes with line and column attributes. Some lexer generators provide environments which track line and column numbers; but in others, you will need to create custom purpose rules which match at the beginning of the rule sets to track these numbers. In **flex**, this is often accomplished in the following manner:

```
.    ++column_number; REJECT;
\n   ++line_number; column_number = 1; REJECT;
```

In our environments, we have made this annotation a run-time option; so our true match action code is closer to this:

```
PATTERN {
    /* Match */
    yylval = xmlNewNode(NULL, "TERMINAL_NAME");

    if (track_term_position) {
        char buf[32];
        snprintf(buf, 32, "%d", line_number);
        xmlSetProp(yylval, "line", buf);
        snprintf(buf, 32, "%d", column_number);
        xmlSetProp(yylval, "column", buf);
    }

    xmlAddChild(yylval, xmlNewText(yytext));
    return lispy_sym_leftParen;
}
```

6.2 String De-escaping

The act of complex string decoding (such as interpreting '\n' in a C-style string), is usually handled by special purpose code in the lexer. Most string manipulations can be implemented with XPath 1.0 functions and XSLT 1.0 templates,

with a mild amount of cleverness. Since these implementations should behave identically in different XSLT environments; and since the 2.0 working drafts of XSLT and XPath contain powerful regex and string mapping functionality; we believe that all string manipulations should be delayed until the transformation stage.

7 Parser Generation

The input to a parser is a terminal node stream (provided by a lexer). Parsers scan these streams for sets of symbols which match productions in their grammar (for our purposes, symbol sets which match some production for a non-terminal). Upon matching a symbol set, the set is replaced in the stream by the non-terminal's node, and parsing continues. Our parser will construct XML nodes for its non-terminal productions, which contain the nodes from the matched symbol set as children.

The rules used by **yacc**-family parser generators associate non-terminal productions with action code. These productions are built into the generated parser's state machine, and when they match, the rule's associated action code is executed.

Since most systems built with parser generators consume their data at parse time, rather than construct a full tree; many complex and subtle features are available. We will be using XSLT to perform the Transform stage, so most of these features will not be needed by our parser, and the action code for our productions will fall into two classes:

1. **non-terminal** productions - which consume symbols, and produce a non-terminal node; and
2. **epsilon**, or empty, productions - which permit a non-terminal to match the empty set.

We also address some of the tedious tasks of grammar construction and transformation here in the parser, using two techniques: **Production Time Node Re-Write**, and **List Resolution**. Our parse rules are represented in XML, using a language with top-level elements **nonterm** and **list** (see Figure 3); and the first rule is the start symbol.

7.1 Non-Terminal Rules

In our XML parser language, non-terminal rules are represented by **nonterm** XML elements. These elements have two attributes:

1. **name** - which specifies the name of the produced non-terminal, and is required; and
2. **empty** - which specifies if this non-terminal is permitted an epsilon production, and defaults to "false".

```
<list name="Program" empty="true">
  <production>
    <symbol>Apply</symbol>
    </production>
  </list>

<nonterm name="Apply">
  <production>
    <symbol action="drop">leftParen</symbol>
    <symbol rename="function">identifier</symbol>
    <symbol action="content">ApplyList</symbol>
    <symbol action="drop">rightParen</symbol>
    </production>
  </nonterm>

<list name="ApplyList" empty="true">
  <production>
    <symbol>Apply</symbol>
    </production>
  <production>
    <symbol>identifier</symbol>
    </production>
  <production>
    <symbol>number</symbol>
    </production>
  <production>
    <symbol>string</symbol>
    </production>
  </list>
```

Fig. 3. Lispy parser rules

In most environments, non-terminals need to have names which are legal identifiers. While we could have used an empty ***production*** *to specify epsilon productions, we felt that the* ***empty*** *attribute was clearer.*

The **nonterm** element nodes have one or more children, which are **production** element nodes; these define the various productions for a given non-terminal. Each **production** element contains one or more **symbol** element nodes, which define the grammatical symbols of the production.

symbol element nodes have 2 optional attributes:

1. **action** - which specifies the re-write action to take on the symbol's node, and defaults to "preserve"; and
2. **rename** - that, if present, specifies a new name for the symbol's node.

7.2 Production-Time Node Re-Write

When a production matches, the various families of parser generators provide different means of accessing the semantic content of each symbol in the production. In our case, this will be the XML sub-tree for a symbol. We have developed a collection of basic re-write actions which can be applied to a symbol's sub-tree in the action code of a production. The root of a symbol's sub-tree will always be a single element node, since the action code of all of our non-terminal and terminal productions yield rooted XML trees.

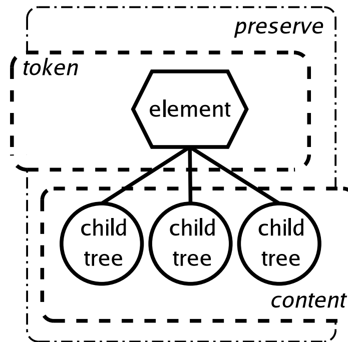

Fig. 4. Node Re-Write Selections

Our re-write method consists of four basic selections which can be performed on a sub-tree:

1. **preserve** - which keeps the symbol's entire sub-tree as a child of the produced non-terminal's node;
2. **token** - which keeps only the root of the sub-tree;
3. **content** - which keeps all of the sub-tree, except for the root node; and
4. **drop** - which discards the symbol's sub-tree entirely.

These selections are illustrated graphically in Figure 4.

Our re-write method also supports renaming the root node, using the **rename** attribute on a symbol; but it should be noted that renaming is, for obvious reasons, not compatible with the "content" or "drop" node selections.

7.3 Parser Action Code

The code for these action rules is trivial, so examples should suffice. The main thing you need to know about **bison** actions is that the semantic value of the non-terminal being produced is stored at location `$$`, and the semantic value of the `1...N` symbols for a given production are stored at locations of the form `$1,$2,...,$N`.

The general case for a non-terminal production is to create a new XML node with the same name as the non-terminal, and to yield that node as the semantic content of the production. With **bison** and **libxml**[4], our production code was of the general form:

```
$$ = xmlNewNode(NULL, "NONTERMINAL_NAME");
/* Insert Re-Write Code for Each Symbol */
```

Rename Re-Write for Symbol 1

```
xmlNodeSetName($1, "NEW_NAME");
```

Preserve Re-Write for Symbol 1

```
xmlAddChild($$, $1);
```

Token Re-Write for Symbol 1

```
xmlAddChild($$, $1);
if ($1->children) {
    xmlFreeNodeList($1->children);
    $1->children = NULL;
}
```

Content Re-Write for Symbol 1

```
if ($1->children) {
    xmlAddChildList($$, $1->children);
    $1->children = NULL;
}
xmlFreeNode($1);
```

Drop Re-Write for Symbol 1

```
xmlFreeNode($1);
```

List of Terminals For most parser generators, it will be necessary to provide the generator a list of all terminals which will be yielded by the lexer to the parser. This list can be extracted by transformation on the XML rules for the lexer.

7.4 List Rules

Now, what are lists? In most grammars, matching a list of elements requires that we construct a recursive non-terminal (left-recursive for LALR(1) grammars) which matches the various productions of the list, and has an epsilon production. For a list to match the symbols A and B, a **nonterm** rule would look like this:

```
<nonterm name="LIST" empty="true">
  <production>
    <symbol>LIST</symbol>
    <symbol>A</symbol>
    </production>
  <production>
    <symbol>LIST</symbol>
    <symbol>B</symbol>
    </production>
  </nonterm>
```

Unfortunately, given a symbol stream `A B A A`, this yields the containment `(((() A) B) A) A)`, rather than the more useful `(A B A A)`. We would likely need to restructure this tree at transform time in order to actually make any use of it; and since this action (matching lists of elements) is virtually guaranteed to come up while writing an LPT tool, there is some real value in having a higher level model for dealing with this. The smgn tool[18] flattens recursive lists, but does not handle delimited lists, and provides no means of controlling the nature of the collapse.

We have developed a higher-order list abstraction defined in terms of our non-terminal abstraction. The meaning of the list abstraction is captured in

terms of left recursion and node re-write (particularly the "content" action) in the produced non-terminals. These methods can be applied to yield lists of this type for any LALR(1) parser generator, no matter its features; and they are relatively easy to adapt to other parser generation models. Our **list** rules yield **nonterm** rules, not action code. Since this is a transformation approach based around XML and XSLT, and since our grammar rules are themselves XML, it is only natural for us to define the resolution of **list** elements into **nonterm** elements in terms of XSLT templates.

We'll lay out the skeleton of the productions generated by a **list** rule as loose BNF (without discussing the node re-writes acting upon symbols). In our BNF, Φ will denote the set of productions for elements of the list, Δ will denote the set of productions for delimiters of the list, and $\bowtie$ will denote the join operation, as suggested for delimited lists in *Advanced Compiler Design and Implementation*[16]. Additionally, know that if Δ is ϵ, we don't have any delimiters, and we ignore all parts of the BNF within $\langle\rangle$. Informally, a delimited list of this form:

$$LIST ::= \Phi \bowtie \Delta$$

becomes the following set of non-terminals:

$$\begin{aligned} LIST &::= LIST_item\ LIST_rlist \\ LIST_rlist &::= LIST_rlist\ \langle LIST_delim \rangle\ LIST_item \\ &\quad\ \mid \epsilon \\ LIST_item &::= \Phi \\ \langle LIST_delim &::= \Delta \rangle \end{aligned}$$

Our list abstraction provides for delimiters, named items (in which case a matching list production is held in an item element with the given name), unnamed items (in which the symbols of the production are direct children of the list), and a collection of complex re-writes upon both items and delimiters. Our **list** rules have seven attributes:

1. **name** - specifies the name of the produced non-terminal, required;
2. **empty** - specifies if this non-terminal is permitted an epsilon production, defaults to "false";
3. **item_name** - specifies if this list's items should be named, and what the name should be;
4. **item_action** - specifies the action to be taken for this list's items (defaults to "content" if **item_name** is not set, and "preserve" if it is);
5. **item_empty** - specifies if there is an epsilon production for the list item, note that there is a grammatical conflict if both the list items and the list itself are given epsilon productions;
6. **delim_name** - specifies if this list's delims should be named, and what the name should be; and
7. **delim_action** - specifies the action to be taken for this list's delimiters (defaults to "drop" if **delim_name** is not set, and "preserve" if it is).

The rules have one or more **production** children, which work the same as those of **nonterm** rules. These productions describe the various productions of a list's items. **list** rules may have **delimiter** children, which behave the same as **production** rules, save that they describe the productions of a list's delimiters.

A Quick List Example Let's examine a more complex list example. Suppose we are working with a language whose functions have comma delimited parameter lists (where the value of a given parameter is given by some Expression), and parameters may be set by keyword. The grammar for such a language might appear as:

$$\begin{aligned} ParamList ::= (\ & Expression \\ & \mid identifier\ equals\ Expression\ ) \\ \bowtie\ & comma \end{aligned}$$

We wish the ST for parameter lists to be a **ParamList** node, containing a collection of **Param** nodes, each containing one parameter. We also wish to perform some re-write on keyword parameters, by renaming the **identifier** symbol to **keyword**, and by dropping the **equals** symbol. Additionally, we wish to permit epsilon productions on the list, which would be difficult to express in our current BNF without more formalism there. All of this can be accomplished with the following **list** rule:

```
<list name="ParamList" item_name="Param" empty="true">
  <delimiter>
    <symbol>comma</symbol>
    </delimiter>
  <production>
    <symbol rename='keyword'>identifier</symbol>
    <symbol action='drop'>equals</symbol>
    <symbol>Expression</symbol>
    </production>
  <production>
    <symbol>Expression</symbol>
    </production>
  </list>
```

Which resolves to the following set of **nonterm** rules:

```
<nonterm name="ParamList" empty="true">
  <production>
    <symbol rename="Param"
       action="preserve">ParamList__item</symbol>
    <symbol
       action="content">ParamList__rlist</symbol>
    </production>
  </nonterm>

<nonterm name="ParamList__rlist" empty="true">
  <production>
    <symbol action="content">ParamList__rlist</symbol>
    <symbol action="drop">ParamList__delim</symbol>
    <symbol rename="Param"
       action="preserve">ParamList__item</symbol>
    </production>
  </nonterm>

<nonterm name="ParamList__item" empty="false">
  <production>
    <symbol rename="keyword">identifier</symbol>
    <symbol action="drop">equals</symbol>
    <symbol>Expression</symbol>
      </production>
  <production>
    <symbol>Expression</symbol>
      </production>
  </nonterm>
```

```
<nonterm name="ParamList__delim" empty="false">
  <production>
    <symbol>comma</symbol>
    </production>
  </nonterm>
```

Formal List Resolution Formally, the resolution of **list** rules into **nonterm** rules is defined by the following XSLT transformations, which produce a **nonterm** rule, with **production** children, for each non-terminal appearing in our informal BNF. The **symbol**s used in these **production**s are provided by three helper templates, which act to realize the various re-writes our list abstraction requires.

Initially, understand that every **list** rule produces three (or four, if delimiters are defined) **nonterm** rules. If the name of the **list** is "LIST", then these **nonterm**s will be:

1. "LIST" - which is the top-level non-terminal for the list, and the symbol that should be used in productions;
2. "LIST__rlist" - which defines the non-terminal for the recursive lists, which are completely consumed by node re-write actions;
3. "LIST__item" - which matches the various list items of the list rule; and
4. "LIST__delim" - which matches the various delimiters of the list rule, if there *are* delimiters.

```
<xsl:template match="list">
<nonterm name="{@name}" empty="{string(@empty = 'true')}">
  <production>
    <xsl:call-template name="list-item-symbol"/>
    <xsl:call-template name="list-rlist-symbol"/>
    </production>
  </nonterm>
<nonterm name="{concat(@name, '__rlist')}" empty="true">
  <production>
    <xsl:call-template name="list-rlist-symbol"/>
    <xsl:if test="delimiter">
      <xsl:call-template name="list-delim-symbol"/>
      </xsl:if>
    <xsl:call-template name="list-item-symbol"/>
    </production>
  </nonterm>
<nonterm name="{concat(@name, '__item')}"
    empty="{string(@item_empty = 'true')}">
  <xsl:for-each select="production">
    <xsl:copy-of select="."/>
    </xsl:for-each>
  </nonterm>
<xsl:if test="delimiter">
<nonterm name="{concat(@name, '__delim')}" empty="false">
  <xsl:for-each select="delimiter">
    <production>
      <xsl:copy-of select="*"/>
      </production>
    </xsl:for-each>
  </nonterm>
  </xsl:if>
  </xsl:template>
```

Every place where we use the **__rlist** symbol, it has the “content” action. This causes the recursive lists to collapse, so that the top level list non-terminal gets only the **__item** children of all the recursive lists, in the proper order.

```
<xsl:template name="list-rlist-symbol">
  <symbol action="content">
    <xsl:value-of select="concat(@name, '__rlist')"/>
    </symbol>
  </xsl:template>
```

We do a bit more work for list items. Though the non-terminal’s name remains “LIST__item”, if **item_name** is set for the **list**, the **__item** symbol gains a **rename** action to rename it’s sub-tree’s root node to the given name. The default action to perform on **__item** symbols is “content”, though this changes to “preserve” if **item_name** is set, and can be set explicitly using **item_action**.

```
<xsl:template name="list-item-symbol">
  <symbol>
    <xsl:if test="@item_name">
      <xsl:attribute name="rename">
        <xsl:value-of select="@item_name"/>
        </xsl:attribute>
      </xsl:if>
    <xsl:attribute name="action">
      <xsl:choose>
        <xsl:when test="@item_action">
          <xsl:value-of select="@item_action"/>
          </xsl:when>
        <xsl:when test="@item_name">preserve</xsl:when>
        <xsl:otherwise>content</xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
    <xsl:value-of select="concat(@name, '__item')"/>
    </symbol>
  </xsl:template>
```

__delim symbols are just like **__item** symbols, except that their default action is “drop”, and not “content”.

```
<xsl:template name="list-delim-symbol">
  <symbol>
    <xsl:if test="@delim_name">
      <xsl:attribute name="rename">
        <xsl:value-of select="@delim_name"/>
        </xsl:attribute>
      </xsl:if>
    <xsl:attribute name="action">
      <xsl:choose>
        <xsl:when test="@delim_action">
          <xsl:value-of select="@delim_action"/>
          </xsl:when>
        <xsl:when test="@delim_name">preserve</xsl:when>
        <xsl:otherwise>drop</xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
    <xsl:value-of select="concat(@name, '__delim')"/>
    </symbol>
  </xsl:template>
```

8 Transformation

Working from the lexer and parser rules given for Lispy in Figures 2 and 3, any Lex and Parse implementations based upon this approach will parse the Lispy string:

```
(foo bar (baz 'abc' 1))
```

into the following XML tree:

```
<Program>
  <Apply>
    <function>foo</function>
    <identifier>bar</identifier>
    <Apply>
      <function>baz</function>
      <string>'abc'</string>
      <number>1</number>
      </Apply>
    </Apply>
  </Program>
```

As we discussed earlier, a tree such as this one goes beyond simply capturing syntactic structure of the input. It captures the semantics of the input, and is a Semantic Tree. The STs this approach generates will be XML trees, and the full power of XML environments can be leveled against them. ***transform*** *elements are synonyms for* ***stylesheet*** *elements in XSLT[15]* Returning to our example transform from Section 3, the following XSLT **transform** uses the ST generated by the parse stage to emit output with the same semantic meaning as the Lispy input, but a different syntax, in this case the more parameter oriented syntax from earlier:

```
<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="text"/>

<xsl:template match="Apply">
  <xsl:value-of select="function"/>
  <xsl:text>(</xsl:text>
  <xsl:for-each select="*[name() != 'function']">
    <xsl:if test="position() != 1">, </xsl:if>
    <xsl:apply-templates select="."/>
      </xsl:for-each>
  <xsl:text>)</xsl:text>
  </xsl:template>

  </xsl:transform>
```

Fig. 5. Lispy transform rules

This results in the desired output:

```
foo(bar, baz('abc', 1))
```

There are many classes of transformation which can be accomplished entirely in XSLT, especially those which resolve some domain language into some implementation or expression language. While there are certainly transformations beyond the scope of XSLT structural manipulations, such as some forms of optimizations; even these transformations, destined to be taken care of during another processing stage, may be aided by intermediate transformations which can be accomplished with XSLT.

Consider the process of implementing a Lispy interpreter. This interpreter could consume the XML output of a Lispy LPT process, and drive calls to C functions based upon the structure of the generated trees. Even though the XML ST contains all the semantic information that the tree interpreter mechanism would need, we could reduce the complexity of the call driver significantly by absorbing **Apply/function** pairs into named **function** nodes. The following transform does just that:

```
<xsl:transform version='1.0'
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>

<xsl:output method="xml"/>

<xsl:template match="*" priority="1">
  <xsl:copy>
    <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

<xsl:template match="Apply" priority="2">
  <xsl:element name="function">
    <xsl:attribute name="name">
      <xsl:value-of select="./function"/>
      </xsl:attribute>
    <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>

<xsl:template match="function" priority="3"/>

</xsl:transform>
```

Fig. 6. "Parametric" ST transform

The tree resulting from this transform, shown below, will be much easier to interpret for driving a series of function calls.

```
<Program>
  <function name="foo">
    <identifier>bar</identifier>
    <function name="baz">
      <string>'abc'</string>
      <number>1</number>
      </function>
    </function>
  </Program>
```

With the tree above, the function call driver could be easily realized by a simple SAX parser. This brings up another advantage of the XML based approach

that effects work beyond the transformation stage. Just as the use of XML libraries means that programmers can avoid writing their own data structures, it also means that programs can process STs with very little manipulation code. In many ways these savings are large enough that, combined with gains in other stages of implementation of a parser; the LPT approach may be advantageous even if the simplicity of the ST or its intended use means that a transform stage can be completely forgone.

9 Conclusion

There are two main advantages to the LPT approach. The first is that it stakes the hard work on the already common XML/XSLT environment, an environment which has more native programmers every year. Each year XML/XSLT becomes yet more powerful and ubiquitous. The second advantage is that producing an LPT tool of this form should be very straight forward, even mechanical, for most programmers.

We have a strong reason to believe in the straight-forwardness of the LPT approach. The very mechanical nature of the LPT approach is exploitable, and LPT tools can be generated easily from their XML descriptions. We have used this process to develop XML descriptions of LPT tools, and then, for a given environment, the tools themselves. We have then used the source code of these tools, and the XML grammar descriptions which guided their production; and developed XSLT transforms capable of generating a tool's source from it's description. This automation of the entire LPT process was itself straight forward (for **flex**, **bison**, and **libxml**, our transform was roughly 500 lines of XSLT); and will be the topic of a future paper.

The LPT approach is neither code nor cognition intensive. While there are real limits to how far such a shallow approach can go, it appears that there are sufficient application domains to make the approach a real win. We believe that it has great potential to make tool production significantly faster, and can thus help drive down the cost of developing new domain languages.

References

1. Ying Zou, and Kostas Kontogiannis *A Framework for Migrating Procedural Code to Object-Oriented Platforms.* In *the 8th IEEE Asia-Pacific Software Engineering Conference (APSEC), Macau SAR, China, pp. 408-418*, December 2001. `citeseer.ist.psu.edu/494242.html`
2. Ying Zou, and Kostas Kontogiannis *Towards a portable XML-based source code representation.* In *XML Technologies and Software Engineering (XSE2001)*, 2001. `citeseer.ist.psu.edu/500234.html`
3. John Levine, Tony Mason, and Doug Brown *lex & yacc.* O'Reilly, 2nd Edition, 1992. ISBN: 1-56592-000-7
4. Daniel Veillard *libxml: The XML C parser and toolkit of Gnome.* `xmlsoft.org`

5. M. E. Lesk, and E. Schmidt *Lex - A Lexical Analyzer Generator.*
   `dinosaur.compilertools.net/lex/index.html`
6. *Bison++ and Flex++.*
   `www.idiom.com/free-compilers/TOOL/BNF-15.html`
7. *JFlex - The Fast Scanner Generator for Java.*
   `jflex.de/`
8. *Alex: A lexical analyser generator for Haskell.*
   `www.haskell.org/alex/`
9. *Aflex and Ayacc.*
   `www.ics.uci.edu/~arcadia/Aflex-Ayacc/aflex-ayacc.html`
10. *Turbo Pascal Lex/Yacc.*
    `www.musikwissenschaft.uni-mainz.de/~ag/tply/tply.html`
11. Wikipedia *Yacc.* 2003.
    `en.wikipedia.org/wiki/Yacc`
12. Computer Science Department, Stevens Institute of Technology *Lexer and Parser Generators.*
    `www.cs.stevens-tech.edu/~badri/cs616/lexparse.html`
13. World Wide Web Consortium *Extensible Markup Language (XML) 1.0 (Second Edition).* 2000.
    `www.w3.org/TR/REC-xml`
14. World Wide Web Consortium *XML Path Language (XPath) Version 1.0.* 1999.
    `www.w3.org/TR/xpath`
15. World Wide Web Consortium *XSL Transformations (XSLT) Version 1.0.* 1999.
    `www.w3.org/TR/xslt`
16. Steven S. Muchnick *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, Inc., 1997. ISBN: 1-55860-320-4
17. Umberto Eco *A Theory of Semiotics.* Indiana University Press, 1979. ISBN: 0-253-35955-4
18. Holger M. Kienle *Using smgn for rapid protoptyping of small domain-specific languages.* ACM Press, 2001.
    `doi.acm.org/10.1145/609769.609779`
19. Paul Hudak *Modular Domain Specific Languages and Tools.* In *Proceedings: Fifth International Conference on Software Reuse, pp. 134–142*, IEEE Computer Society Press, 1998.
    `citeseer.ist.psu.edu/hudak98modular.html`
20. J. Aycock *Compiling little languages in Python.* In *Proc. 7th Int. Python Conf., pp. 69–77*, Foretec Seminars, Reston, VA., November 1998.
    `citeseer.ist.psu.edu/aycock98compiling.html`