2007

The Topes Format Editor and Parser

Christopher Scaffidi
Carnegie Mellon University

Brad Myers
Carnegie Mellon University

Mary Shaw
Carnegie Mellon University, mary.shaw@cs.cmu.edu

The Topes Format Editor and Parser

Christopher Scaffidi, Brad Myers, and Mary Shaw

May 2007
CMU-ISRI-07-104 / CMU-HCII-07-100

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213

Abstract

It is currently difficult and time-consuming to validate and manipulate data in web applications, so we have developed an editor and a parser to simplify these tasks. Our editor enables end-user programmers to create and debug reusable, flexible data formats without learning a complex new language. Our parser uses these formats to turn strings into structured objects and to report its level of confidence that each string is a valid instance of the format. End-user programmers can use our system to create validation code that takes a graduated response to slightly invalid data. We evaluate our system's expressiveness by defining formats for commonly-occurring web data.

This work was funded in part by the National Science Foundation under ITR grant CCF-0325273 (via the EUSES Consortium) and by the National Science Foundation under ITR grants CCF-0438929 and CCF-0324861.

Keywords: end user programming, end user software engineering, data validation, data manipulation, data formats

1. Introduction

End-user programmers currently must store data using basic primitives such as strings, integers, and floating-point numbers, due to limitations of existing end-user programming tools for creating spreadsheets, web macros, and web applications, and other programs. These supported primitives are at a much lower level of abstraction than the problem domain, which typically involves categories of short, human-readable pieces of information such as phone numbers, state names, and product identifiers, each of which may appear in multiple formats.

The problem with this mismatch is that end-user programming tools can provide only limited support for validating program input. To perform custom validation, the end-user programmer typically must write a regular expression (or custom code in an even more complex language), which the application uses to check values in data fields.

Unfortunately, end-user programmers consider regular expressions "confusing and difficult to read" [2]. One reason is that regular expressions use exotic characters (such as |), and another reason is that regular expressions are cumbersome for expressing negation (e.g.: US phone numbers' area codes never end with "11") as well as numeric ranges (e.g.: the parts of an IPv4 address are in the range 0 through 255). Even experienced programmers recognize that regular expressions are hard to read, write, and maintain. For example, the top few results in a Google search for the phrase "regular expressions" includes comments such as, "Do not worry if the above example or the quick start make little sense", and, "Sometimes you have a programming problem and it seems like the best solution is to use regular expressions; now you have two problems."

Consequently, many programmers prefer not to write regular expressions and sometimes omit validation entirely, which can significantly degrade data quality. We saw an example of this in interviews of six creators of "person locator" sites after Hurricane Katrina [17]. Some interviewees omitted validation because implementing validation would have taken too long. Others thought it so important to collect any data—even slightly erroneous data—that they did not want to put up any barriers to the data input process. This decision resulted in numerous data errors, some of which were subtle. For instance, one web application user put "12 Years old" into an "address" field, which would be hard to distinguish from a valid value such as "12 Years St" using existing data validation mechanisms.

Ideally, in order to reduce data errors without erecting considerable barriers to data collection, it would have been helpful if this Hurricane Katrina web application had taken a graduated response to "slightly invalid" inputs rather than simply accepting all inputs. For example, the application could have given the end user a warning such as, "This does not look like a valid street address. Are you sure that you want to enter '12 Years old' as a street address?" Implementing this approach would have been difficult, since regular expressions specify binary recognizers: either a value matches the expression, or it does not. Thus, expressions cannot describe soft constraints (such as the fact that a street abbreviation usually is "St.", "Rd.", "Dr.", "Ave.", or certain other well-known values). Expressions cannot detect "slightly invalid" data and cannot trigger a graduated response.

In this paper, we present two main contributions that simplify specifying formats and to using formats for validating data. First, we present a direct manipulation user interface that helps users define formats without learning unnatural notation. Our editor internally transforms our human-readable notation into an augmented context-free grammar (ACFG). Second, we present a parser that builds structured objects from input strings and returns a number between 0 and 1 to indicate the parser's confidence in each string's validity.

These contributions have three ancillary benefits.

First, our editor presents formats in human-readable notation, which should help users to check their work. This should also facilitate sharing of formats, since one end-user programmer could save a format to a file and email it to a co-worker, and then the co-worker could review the format to see if it is correct and if it is useful for that co-worker's needs.

Second, our editor allows users to incrementally add, remove, and change parts of the format, which should also facilitate sharing: one end-user programmer could create a generic format with just the basic structure, then share it with other users who could customize the format as needed to make it more precisely match their needs.

Finally, our system allows what we call "soft constraints"—statements about data that are often but not always true. These enable programmers to create an application that responds in a graduated manner to slightly invalid inputs.

For example, if an input only violates one soft constraint, the application could display a warning rather than rejecting the input. If the user confirms the input, then the application can accept it.

We our editor and parser are part of a future system that will enable users to specify families of related formats, as well as the operations to transform from one format to another. For example, one family might describe all US phone number formats, as well as the transformations among those formats. Unlike typical programming primitives such as floats and strings, each family of formats has a natural *place* in end-user programmers' problem domains. Consequently, we refer to each family of formats as a "tope," the Greek word meaning "place." For the details of our data model and our implementation plan, refer to [14].

After describing an illustrative example of how our system greatly lowers the difficulty of validating web data, we describe our system in detail. We evaluate its expressiveness by implementing several formats for data observed in use, and we implement a mashup using several more sophisticated formats. We close by comparing this research to related work and by discussing our own future work, which will include an evaluation of the system's usability.

2. Data validation example

When constructing a web form for a web application, an end-user programmer must create code that validates inputs from end users. Some IDEs such as phpClick [12] or Microsoft Visual Studio[1] can automatically generate this code from a regular expression specified by the end-user programmer. Below, we examine this approach and its deficiencies, then present an improved approach based on a reusable JavaScript library API that we have built on top of our editor and parser.

Suppose that an end-user programmer wants an application to verify that inputs to a single "person name" field follow "Lastname, Firstname" format. In the Visual Studio IDE, the end-user programmer drags a textbox from the toolbox onto the web page, then drags a `RegularExpressionValidator` from the toolbox and drops it beside the textbox. The next step is to click on the validator's icon and type a regular expression such as `[A-Z][a-z]+,\s[A-Z][a-z]+` Last, the end-user programmer enters an error message (e.g.: "Please enter a valid name") that will appear for invalid input. The IDE generates code that checks inputs against the regular expression at run-time. For example, if a user forgets a comma, then the error message appears.

This approach has two deficiencies. First, a straightforward regular expression like this one is too constraining. For example, it would reject "d'Gates, Bill" and "Gates-Billings, Bill". Specifying a more accurate expression would be much more work and probably beyond most end-user programmers' abilities. Second, this approach is inflexible: as we saw in our Katrina study, it is sometimes desirable to let users enter slightly invalid data, but to warn them so they can correct input if desired [17].

To address these deficiencies, we have implemented a new validator library API (called a `PatternValidator`) on top of our editor and parser to replace the old `RegularExpressionValidator`. This improved validator enables programmers to specify formats without resorting to regular expressions.

First, the end-user programmer drags a textbox from the toolbox onto the web page, then drags our `PatternValidator` and drops it beside the textbox. Clicking on our validator brings up our format editor, which is shown in Figure 1 and discussed in more detail in Section 3.

Using the editor, the end-user programmer names the format, specifies its parts, tests it if desired, and then saves the format in a file in a subdirectory of the website (alongside any images, stylesheets, scripts, and other supplementary content comprising the website). Our validator works with the IDE to generate the necessary code for validating inputs according to the format. At run-time, if an end user's input violates the format, then our parser generates an error message, and our validator library shows this message to the end user.

To accommodate slightly invalid input, our parser assigns a number between 0 and 1 to represent the parser's confidence in each input's validity. The validator always rejects any input with a confidence of 0 and always accepts any input with a confidence of 1.

---

[1] Microsoft Visual Studio is primarily used by professional programmers, though there is an "express" version for "hobbyist, novice, and student developer" programmers: http://msdn.microsoft.com/vstudio/express/

If the confidence is *between* 0 and 1, the validator displays a confirmation popup rather than rejecting the input. If the user confirms the input's validity, then the validator accepts it. For example, the popup would appear if the user entered "von Gates, Bill" because the format contains a soft constraint that the last name often starts with 1 uppercase letter. (The end-user programmer can also configure our validator to always reject any input with confidence less than 1, thus dispensing with the popup.)

In addition to the validator described above, we used our parser to implement other JavaScript functions so that programmers who know JavaScript can customize validation still further. For example, with a few lines of code, an application can accept both "Firstname Lastname" and "Lastname, Firstname" inputs and reformat the former into the latter so only a single canonical format is submitted by the form to the server.

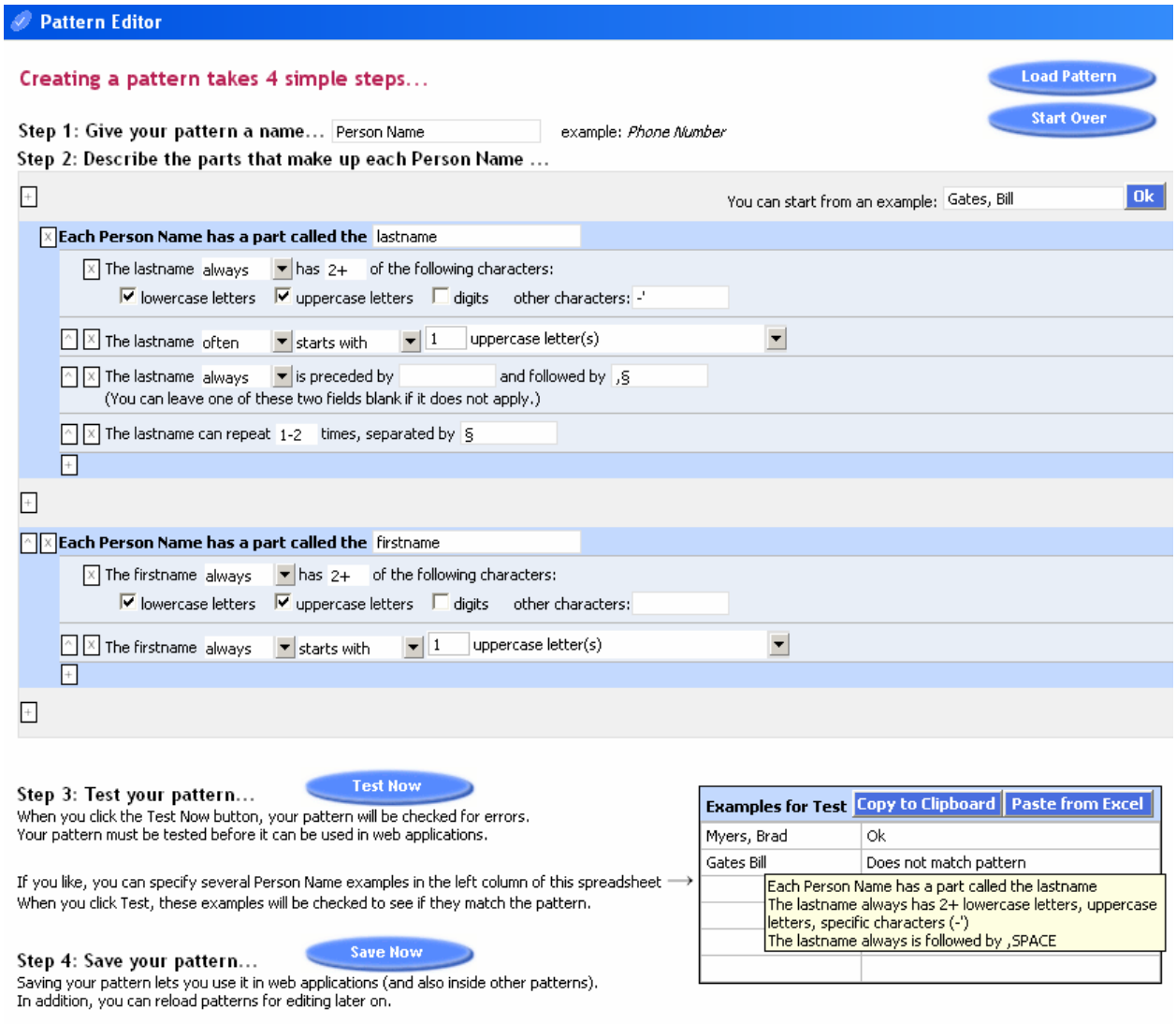


Figure 1: Editing a format for a person name. The end user programmer specifies the format's parts ("lastname" followed by "firstname") and various facts about each part. The [+] buttons add parts/facts, the [x] buttons remove parts/facts, and the [^] buttons change the order of parts/facts in the format. Each § symbol indicates a space. During Step 3, the programmer can specify some sample values; if these do not match the format, then a tooltip with a targeted error message appears. See Section 3 for more discussion of our editor.

3. Defining formats in our editor

Creating a format with our editor involves four steps as shown in Figure 1.

First, the end-user programmer specifies the data format's name, such as "phone number" or "person name". Prompting the end-user programmer to name the format enables the editor to concretely refer to the data by name in the next three steps.

Second, the end-user programmer specifies the format's parts. For example, a US phone number has three parts: area code, exchange, and local number. The end-user programmer can add, remove, and reorder parts, and can specify constraint-like facts on parts. Table 1 shows the available types of facts.

Table 1: Facts that can be stated about parts.

Type of fact	Description
Pattern	Specifies the characters in a part and how many of them may appear
Literal	Specifies that the part must equal one of a certain set of options
Numeric	Specifies a numeric inequality or equality
Substring	Specifies that the part starts or ends with a certain literal string, or a number of certain characters
Wrapped	Specifies that the part is preceded or followed by a certain string
Optionality	Specifies that the part may be missing / blank
Repeat	Specifies that the part may repeat, with possible separators between repetitions
Tope	Specifies that the part must match another format

The editor does not allow the end-user programmer to specify more than one *Literal* fact (since more than one option can be listed in the same *Literal* fact), more than one *Optionality* fact (since that would be redundant), or more than one *Tope* fact (in preparation for future work where a single *Tope* fact can reference a family of formats).

The *Pattern, Literal, Numeric,* and *Substring* facts can be marked as "never", "often", or "always" true. *Wrapped* facts can be "often" or "always" true. (We may add additional levels of likelihood, such as "sometimes", in future versions.) If a string violates a fact that is often true, then our parser returns a score between 0 and 1. Users have trouble understanding mixtures of conjunction and disjunction [11], so all facts here are conjoined. Disjunction is expressed as two facts with parallel structure that are each often true, and if a string violates both facts, then its validity is downgraded toward 0 at parse-time (e.g.: when a part "often starts with 'a'" and the part also "often starts with 'b'", then that part must rarely start with something other than 'a' or 'b').

To help users get started, our editor includes a textbox where users can type an example of the data. The editor identifies non-alphanumeric characters in the example and treats these as separators between parts. It then initializes a part in the editor for each part of the example and looks for a few basic facts on the parts (e.g.: noticing that a word-like part starts with an uppercase letter).

To further help users create formats, we have developed a prototype inference system that examines multiple examples of data and then initializes our editor with a single format describing the majority of those examples. The end-user programmer can then review, test, and customize this format as desired. Because the inferred format covers most but not necessarily all of the examples, it can be used to find outliers in the example data (by running the parser with the inferred format on each string in the sample data, then looking for strings that have low confidence). We have integrated this inference and outlier-finding subsystem into Excel, then evaluated and described it in a separate paper [15].

In the third step of the editor, the end-user programmer can enter additional examples that will be tested using the specified format. To perform these tests, the editor first does basic sanity checks on the format and then converts the format into an ACFG, as described in Section 4.2. Next, the editor uses the parser to attempt to parse each example. The editor displays a targeted error message alongside each test example that fails (essentially showing a list of the constraints violated by the test string). The end-user programmer can then iteratively debug the format.

Finally, the editor saves the format to a file. This file contains the format in ACFG notation as well as an XML representation of the format. If the format will be used to validate or manipulate data in a web application, then the file is stored in a subdirectory of the web application (where it can be referenced at run-time by the application's JavaScript). This file can be re-edited later, if desired.

The main feature of our editor's interface is its use of sentence-like screen prompts to present constraint-like facts. One key to making our sentences readable is allowing users to specify numeric ranges in textboxes that accept an integer. For example, users can specify that a part starts with "1", "2-3", or "4+" uppercase letters. One somewhat unintuitive aspect to our editor is that it makes spaces visible by representing them with a special symbol, §. If an end-user programmer types a space into a textbox, the editor automatically changes the space into this special symbol. Though this slightly reduces readability, it is preferable to having spaces that the end-user programmer cannot see or debug. (In error messages, spaces appear as `SPACE` instead of § to avoid font-related problems.)

Another feature of our editor is that it facilitates format reuse in two ways. First, users can specify that a particular part must match another, so users can create hierarchies of formats. Second, users can load an existing format file, tweak it, and save it as a new file, thereby creating a tope (a family of related formats).

4. Algorithms

The editor algorithm converts a format to an ACFG, and the parser algorithm parses a string according to the ACFG. The parser algorithm is available in Java as well as C#, so end-user programmers can use formats in Java, JavaScript, and Microsoft COM-centric environments (e.g.: Excel).

4.1. Internal ACFG notation

Below, we very briefly summarize our ACFG notation so that the operation of our algorithms will be clear when we discuss them in the following sections.

Rationale for designing the ACFG notation

There are several important reasons why we chose ACFGs as the underlying representation.

First, any format that can be expressed as a regular expression (the most commonly supported format language in end-user programming tools) can also be expressed as a CFG, and any format that can be expressed as a CFG can be expressed as an ACFG. Therefore, ACFGs have at least as much expressiveness as regular expressions. (Some of this power might actually not be needed in future versions of the system. In particular, CFGs can contain pairs or sets of productions that reference each other in mutual recursion, but we have not yet seen a single data format that would require mutual recursion among productions. Removing support for this language feature from the algorithm might improve efficiency without limiting expressiveness in practice.)

Second, the ability to associate constraints with grammar productions makes it straightforward to represent common data formats that are cumbersome to express as regular expressions. As an example, here is one of the simplest possible regular expressions for specifying a phone number in `###-###-####` format for which area code and local exchange must not start with 0 or 1, and must not end with 11…

```
[2-9]([023456789][0-9]|1[023456789])\-[2-9]([023456789][0-9]|1[023456789])\-\d\d\d\d
```

An equivalent (and much more readable) ACFG constraint is discussed on the next page.

Third, a number of algorithms exist for parsing CFGs, and in Section 4.3 we describe our extension of one algorithm to handle our ACFG. Selecting an ACFG representation lets us take advantage of these algorithms and will create opportunities for us to further tailor these algorithms to users' needs.

Fourth, a CFG-driven parser generates a tree, which is a data structure familiar to many programmers. This is important in situations where an end-user programmer will use the parser's output in order to manipulate the data, as is the case in our mashup example in Section 5.2.

Finally, a CFG associates a name with each part of the format, so that each part can be referenced. This is essential for data manipulation, and it will be essential for future work aimed at helping end-user programmers specify how to transform one format into another.

Short summary of notation

Each line of the ACFG specifies one or more productions. For example, a US phone number format file might contain the following ACFG:

```
#NUMBER : #AREA – #EXCH – #LOCAL
#AREA : #D #D #D
#EXCH : #D #D #D
#LOCAL : #D #D #D #D
#D : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Any term starting with a hash `#` is a variable. Other terms are literals, which can contain escaped Unicode characters (which is useful for non-English formats).

Boolean expressions can be attached to a line to constrain that line's productions. This example constraint prevents area codes from starting with 0 or 1:

```
#AREA : #D #D #D : #AREA>=200
```

Expressions can contain numeric comparisons, arithmetic operators, Boolean negation, conjunction, disjunction, and parentheses. The notation also includes four functions that accept strings and return a number:

- `CCOUNT`: for counting occurrences of character sets. Counting can be restricted to occurrences adjoining the string's start or end
- `IN`: searches for a string in a set of options, returning a number > 0 if the string is in the set, -1 otherwise
- `LEN`: returns the length of a string
- `SFIND`: searches for one string in another. The search can be restricted to the containing string's start or end. Returns a number > 0 if found, -1 otherwise

Variables in a production and literal strings enclosed in left-apostrophes can be passed to these functions. This example forbids area codes that end with `11`:

```
#AREA : #D #D #D : SFIND(#AREA,`11`,`end`)==-1
```

Sometimes, a constraint statement is not always true. As discussed in Section 4.3, if an input violates such a constraint, the parser continues parsing but annotates the resulting parse tree with indications that the input might be invalid. This makes it possible to perform the flexible validation demonstrated earlier in Section 2.

To indicate when a constraint is not always true, a "confidence" in the range of 0 and 100 is appended to the constraint (where 100 indicates that the constraint is always true, and 0 indicates that it is never true). For example, this constraint is "often" true:

```
#AREA : #D #D #D : #AREA>=200 {c=60}
```

4.2. Converting a format to an ACFG

To specify a format, end-user programmers use our editor, which displays formats with sentence-like screen prompts. But to parse inputs, our parser uses an ACFG. Consequently, the editor must convert the human-readable format to an ACFG. This ACFG is stored along with the format in a format file.

In our user interface, a format consists of one or more parts, each of which contains one or more constraint-like facts. The simplest approach to generating an ACFG would be to create a top-level production of the form `#format: #part1 #part2 #part3 ...` and then to generate additional productions for each part. Facts about parts could then be used to generate ACFG constraints on the productions for each part.

In practice, our algorithm is more complex than this simple approach due to several difficult questions:

- What ACFG productions should the editor generate for the structure within each part?
- In particular, what productions should the editor generate when the end-user programmer specifies no facts about the part (or very few)?
- How should the editor represent facts that have no equivalent ACFG constraint: *Wrapped*, *Repeat*, *Optionality*, and *Tope* facts?
- How should the presence of "never", "often", and "always" modifiers affect the constraints and productions generated?
- How should the presence of two alternative facts that are "often" true be converted into a disjunction?

We deal with these issues according to three principles. First, any facts that *cannot* be represented as ACFG constraints are instead used to generate ACFG productions. Second, if no such facts are available, then any necessary ACFG productions are generated using facts that *can* be represented *either* as constraints or as productions. Finally, ACFG constraints are also used to represent all facts that can be represented as constraints (including facts that *must* be represented as ACFG constraints).

Generating productions

The editor uses a hierarchy of productions to represent the *Wrapped* and *Repeat* facts (which, respectively, indicate that the part is preceded/followed by text, and that the part can repeat as a list with certain separators):

```
#format: #part1_wrap #part2_wrap ...
#part1_wrap: leftText #part1_rep rightText
#part1_rep: #part1_core | #part1_core sep
#part1_core: #part1 : <constraints>
```

The editor generates one `#part1_wrap` production for each *Wrapped* fact and one `#part1_rep` production for each *Repeat* fact. The "core" variable represents an instance of the part that conforms to all remaining facts.

This leaves two additional facts that cannot be represented as constraints, *Optionality* and *Tope* (which, respectively, indicate that a part can be missing, and that the part must match another format). Again, we use productions to represent these in the ACFG.

If an *Optionality* fact is present, the editor creates a production allowing the "core" variable to be blank:

```
#part1_core :
```

If a *Tope* fact is present, then the editor recursively generates productions for that format and its parts:

```
#part1 : #subformat
#subformat: #recursePart1 #recursePart2
```

If no *Tope* fact is specified, the editor must still generate productions for `#part1`. In this case, if a *Literal* fact is specified (indicating a list of valid options for the part), then the editor generates a production of the form:

```
#part1 : option1 | option2 | ...
```

If no *Tope* or *Literal* fact is present, then the editor looks for a *Pattern* and/or *Numeric* fact (which, respectively, indicate that the part contains certain characters, and that the part meets a numeric equality/inequality). If these are present, then the editor identifies the list of characters that could be present in the part (uppercase letters, lowercase letters, digits, or other characters), and then represents the `#part1` variable as a list of those characters. (The presence of a *Numeric* fact indicates that the part can contain digits.)

```
#part1 : #part1_char | #part1_char #part1
#part1_char : char1 | char2 | ...
```

If no *Tope, Literal*, *Pattern*, or *Numeric* facts are present, then the editor generates productions that let the `#part1` contain a sequence of any characters.

Generating constraints

Pattern, Literal, Numeric, and *Substring* facts that are always true can be expressed as constraints in a straightforward manner, then attached to the "core" production. Facts that are never or often true require extra steps.

Four facts might "never" be true: *Pattern*, *Literal*, *Numeric*, and *Substring*. The editor converts such a fact to an ACFG constraint and prepends a Boolean negation operator.

The editor allows five types of facts to "often" be true: *Pattern*, *Literal*, *Numeric*, *Substring*, and *Wrapped*.

The first four of these can be represented as ACFG constraints, so the editor converts these to constraints as usual and associates them with the "core" production. Because such constraints are often true, the editor assigns them a confidence of 60 (rather than 100). This value was chosen because in many studies, end-user programmers assigned a probability of approximately 60% to the word "often" [9]. The Pattern, Literal, Numeric, and Substring facts can be marked as "never," "often," or "always" true. Wrapped facts can be "often" or "always" true.

If a *Pattern*, *Literal*, or *Numeric* fact is often true, then it is not used to generate any productions, since doing so might cause the parser to reject valid inputs. For example, if a *Pattern* fact specifies that the part often contains 3-5 digits, then it is possible that the part contains more or fewer digits, or other characters altogether. As another example, if a *Literal* fact indicates that a part often is "A" or "B" or "X", then the part could possibly be "T" or "S". Thus, when a fact is often true, it is not used to restrict the productions, but it instead is just used to generate constraints on the "core" production, as described above.

As described earlier, *Wrapped* facts are used to generate productions. When all *Wrapped* facts are often true, then the part might not be preceded or followed by any separators. So in this case, the editor generates an implicit production `#part1_wrap : #part1_rep`

4.3. Parsing strings using the ACFG

Our parsing algorithm is similar to GLR [19], which parses input strings according to a CFG that is permitted to be ambiguous. GLR runs in linear time (with respect to the length of the input string) when the grammar has low ambiguity, as is generally the case with ACFGs generated by the algorithm in Section 4.2.

GLR operates by maintaining an acyclic directed graph of parse states as it reads through the input string. Each state represents a set of productions that are partially complete, and the nodes of the input's parse tree are the links between states in this graph. As each token is read, any productions that are "waiting" for that token are advanced and included in new states. If a variable `v` is produced because a production is completed, the algorithm goes back to see what other productions were waiting for `v`, and these waiting productions are then advanced and included in new states. That instance of `v` is used as a link in the graph from the states of the waiting productions to each new state.

The parse completely fails if at any point there are no states remaining (i.e.: all productions were waiting for an input that never arrived at the appropriate point in the input). The parse succeeds if a production for the top-level variable is completed exactly at the last input token. In this case, the parse tree can be read off from the state graph, as the nodes of the parse tree (variable instances) are the links between states in the graph.

We adapt GLR by incorporating confidences into the parsing process. When a variable `v` is produced because a production is completed, GLR automatically treats this variable as being valid and looks back to see what other productions were waiting for `v`. In contrast, we associate a confidence with each instance of a variable (the nodes in the parse tree), ranging from 0 through 1. This value is usually 1, since most nodes are valid, but our parser downgrades the confidence of a node if it was produced through "questionable" means.

That is, when a production p for a variable `v` is completed, the parser checks to see if p has any constraints. If so, it evaluates those constraints. If any is violated, then the parser multiplies the confidence of `v` by 1-`conf`*0.01. For example, if a production's constraint that is "often" true (`conf=60`) is violated, then the parse node produced by that production will have a confidence of 1-60*0.01=0.40. If a second constraint is violated, then the confidence becomes 0.40*0.40=0.16.

In addition, if a variable instance with downgraded confidence is later used on the right hand side of a production, then the parser similarly downgrades the confidence of the variable on the left-hand side of the production. (Thus, the confidence of each node in the parse tree also depends on the confidence of each of its child nodes.) For example, if the variable instances on the right-hand side of a production have confidences of 0.2, 0.4, and 1.0, then the variable instance on the left side has a confidence equal to 0.2*0.4*1.0=0.08.

In short, these multiplications use confidence of violated constraints to produce a confidence estimate for each node, including the root. That way, when the tree is complete, the parser can return a value between 0 and 1 for confidence in the input's validity as a whole. In cases of ambiguity, the parser selects the parse with the highest confidence. This multiplication of factors and selection of parses with highest confidence resembles the approach used in stochastic context-free parsing algorithms, though those require normalization of factors that we call "confidences." [1]

There are two additional wrinkles to our algorithm.

First, an ambiguous parse results if one part is more precisely specified than an adjacent part. For example, one part might be produced from digits and letters (because of the presence of a *Pattern* fact), whereas an adjacent part might be produced from any characters (because no facts were specified about that part). We would like more specific parts to be more "greedy" than less specific parts. Therefore, our parser internally very slightly downgrades a variable in-

stance (parse node) if it is produced through an "any character" production, so that when ambiguous parses appear, we select the one with the least use of "any character" productions. (This slightly reduced confidence is only used internally and is raised back to 1.0 if an otherwise-correct parse tree is returned.)

In addition, if a constraint with `conf=100` is violated, the parser does not reduce the node's confidence to 0, but instead retains a small positive value. That way, our parser only returns a confidence of 0 if the parse completely fails.

5. Expressiveness

We evaluate expressiveness in two ways. First, we implemented sample formats for nearly 6000 data values logged on end users' machines over 3 weeks, revealing the need for one new format primitive (a constraint on where periods can occur in a string) as well as better testing support in the editor (a feature which we have subsequently added, as discussed in Section 3). Second, we implemented more complex formats required to implement a sophisticated mashup example, thus demonstrating the suitability of the editor for implementing more complicated formats.

5.1. Expressiveness for information workers' data

To evaluate expressiveness, we recorded the time required to create sample formats, as well as the formats' correctness. As this was an evaluation of expressiveness, not usability, the formats were implemented by an expert user, this paper's first author. We will perform additional studies in the future to evaluate usability.

To identify test data, we ran logging software for 3 weeks on 4 administrative assistants' computers. They were heavy users of accounting pages on our university intranet. When a user filled out a web form in Internet Explorer (these users' preferred browser), the logger recorded the form fields' HTML names, as well as some text near to each field (so we could see the human-readable labels associated with each field).

For each field, the logger recorded a regular expression describing the value that the user entered. (We recorded a regular expression rather than the literal value in order to protect users' privacy.) To generate regular expressions, we converted each lowercase letter to the regular expression `[a-z]`, uppercase to `[A-Z]`, and digits to `[0-9]`, then concatenated expressions. Repeated expressions were coalesced and represented with a length indicator (e.g.: "cscaffid0@CMU.EDU" became `[a-z]{8}[0-9]@[A-Z]{3}.[A-Z]{3}`).

We manually examined many of the 5897 logged values. We then wrote scripts to gather fields into semantic families such as "email" and "currency" based on HTML field names and the human-readable text near to the fields. Of the 5897 values, 5527 (93.7%) fell into one of 19 semantic families shown in Table 2.

Using the regular expressions as a reference, we then created formats for these 14 families asterisked in Table 2.[2] We did not include 3 families (justification, description, and posting title) because each would have simply required a sequence of any characters. That is, it is doubtful that these fields have any semantics aside from "text." We also omitted usernames and passwords because we wanted to post formats online and did not want to reveal formats of our users' authentication credentials.

Finally, after creating the formats, we carefully went through them and looked for errors. We also used a command-line version of our parser to test the formats with sample values, which we generated by referring to the regular expressions and by using any personal knowledge of the format's semantics (such as what might constitute a likely email address). Table 3 summarizes our results.

These experiences revealed strengths and weaknesses of our system on several dimensions.

Editor expressiveness

For the most part, there was no difficulty expressing families as formats. Some required two formats to cover the data, but this was reasonable, as the formats would not be used in place of one another in a given piece of software without some transformation between the formats. That is, for example, we needed a format for dates in the form "10/12/2004" and another for dates in the form "12-Oct-2004". It would generally be unacceptable for users to put "10/12/2004" in fields where software expected the other format (unless some explicit transformation step takes

[2] These format files are available online at http://www.cs.cmu.edu/~cscaffid/software/formatsamples.zip

place), so representing these as different formats seemed appropriate. This good expressiveness suggests that our ACFG is adequately powerful for representing the data formats that users commonly enter into web forms.

An opportunity to improve expressiveness involves cases when a part contains a certain character, but only in the first or last position. For example, a street type (e.g.: "Rd.") can contain a period, but often only in the part's last position. The editor currently offers no precise way to express this about a part, so we may add a new type of fact to the editor. No new ACFG functions will be required, however, as the constraint is expressible with `SFIND` and `CCOUNT`.

Table 2: Semantic families gathered from user data. Asterisked groups were used for testing.

Family name	# values	Example regular expression
project number *	821	[0-9]{5}
justification	820	Very long
expense type *	738	[0-9]{5}
award number *	707	[0-9]{7}
task number *	610	[0-9][A-Z]
currency *	489	[0-9]\.[0-9]{2}
date *	450	[0-9]{2}\/[0-9]{2}\/[0-9]{4}
sites *	194	[a-z]{3}
password	155	Several characters
username	121	Several characters
description	96	Very long
posting title	65	Very long
email address *	50	[a-z]{8}@[a-z]{7}\.[a-z]{3}
person name *	48	[A-Z][a-z]{5}\s[A-Z][a-z]{8}
cost center *	41	[0-9]{6}
expense type *	41	[0-9]{5}
address line *	37	[0-9]{3}\s[A-Z][a-z]{5}\s[A-Z][a-z]{4}
zip code *	28	[0-9]{5}
city *	16	[A-Z][a-z]{8}

Table 3: Summary of test formats created.

Family name	# formats created	Total # minutes to create	# values left uncovered
project number	1	<≈ 1	1
expense type	1	<≈ 1	
award number	1	<≈ 1	
task number	2	<≈ 1	
currency	2	<≈ 1	6
date	2	5	2
sites	1	<≈ 1	
email address	2	7	7
person name	2	3	
cost center	1	<≈ 1	
expense type	1	<≈ 1	
address line	1	<≈ 1	6
zip code	1	<≈ 1	
city	1	<≈ 1	

Time to create formats

Creating formats for most families required less than a minute. Only three families (date, email, and person name) took over a minute, mostly because there were a large number of distinct regular expressions that we had to mentally synthesize into a single format. Still, the resulting formats required only minutes to create, and they were the most reusable of the 14 families, so spending this time is probably reasonable.

Correctness of formats

When testing our formats, we found four errors. Three were cases where we failed to mark a part as optional. The fourth error was an apparent slip of the mouse, in which we indicated that a fact was often true rather than always true. The version of the editor that we used for this evaluation did not have the testing feature (Step 3 in Figure 1). We noted that these four errors probably could have been found if we had been able to test our formats when we created them. Therefore, after our evaluation, we added the editor's testing feature.

Usefulness for validation

After correcting the errors discussed above, our formats covered 99.5% of the 4250 test values (as shown in Table 3). Of the 22 uncovered values, 17 appear to have been typos by the end users, indicating that formats are powerful enough to express the differences between valid and invalid input, and that our formats therefore can be used to validate input.

The remaining 5 uncovered test values included 4 cases that probably were not typos by the end users (that is, they were intentionally typed), but we suspect that they may have been invalid inputs for the fields, nonetheless. Specifically, these 5 values included 1 case where a user entered a list of addresses rather than a single address, 2 cases where users entered a month and year rather than a full date, and 1 case where a user entered a currency with a comma in the dollar amount.

The final uncovered test value was the case mentioned earlier where a street type had a trailing period, and the editor offered no way for us to express that a street type may contain a period but only in the last position.

5.2. Expressiveness for mashup data

We created a sample mashup to evaluate the expressiveness of the system for data involved in a fairly sophisticated programming task. In a client-side web mashup, JavaScript on a web page dynamically grabs data from within the web page, then uses this data to silently request additional data from the server. The script combines this new data with the original data, and it may use the combined data to issue additional requests for data. Ultimately, the script dynamically updates its web page to display some or all of the data [5]. Currently, writing mashups is difficult, even for professional programmers. The main reason is that grabbing data from pages requires breaking apart unstructured or semi-structured data in the HTML, and this is brittle without a great deal of code to handle exceptional cases.

Implementing this sample mashup not only demonstrated that the system was adequately powerful for creating the necessary formats, but it also helped us to identify a list of "helper" functions that would be beneficial for simplifying the process of creating mashups using our editor and parser. These helper functions included functions for calling our parser to turn a string into a parse tree, for accessing nodes within the parse tree, and for using a format to find strings of text within HTML.

Overview of the sample mashup task

Our sample task aimed to enhance a web page that shows citations by adding JavaScript to display author contact information when an end user clicks a citation of the form shown in Figure 2. This task was mainly selected because it involves several kinds of data (person names, email addresses, and phone numbers) that information workers often need to manually copy from site to site [16], as well as one hierarchical kind of data (citations).

Chris Scaffidi, Mary Shaw, Brad A. Myers. Estimating the Numbers of End Users and End User Programmers, 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), 20-24 September 2005. pp. 207-214.

Figure 2: Sample input citation read by the script

Figure 3: Sample output tooltip generated by the script

Implementing the requisite functionality involved three steps. First, we created formats for the author names and for the citation as a whole. Second, we created formats for the contact information retrieved from an existing web application. Finally, we wrote JavaScript to run our parser on strings, using the formats that we had created.

Step 1, creating formats for authors and citations

As shown in Figure 2, some authors had middle initials on this particular page of citations. However, the script needed to send each author's name to a web application that expected full names in "Firstname Lastname" format. Therefore, we had to use our editor to define a format for a person name's inner structure, so the script could reference first and last names. We then reused this person name format inside of a larger citation format.

To define the needed format, we started with the person name format defined in Figure 1. Adapting this format required swapping the first name and last name (with a single button click), inserting a part for the middle initial, and tweaking the separators between parts (for example, removing the comma after the last name and adding a space between parts).

Next, we used our editor to define a format for the citation as a whole. The first part of this citation format was defined by reusing the format defined above to create an "author" part, which could repeat with a comma and space between authors. In the same format, we specified ", and" as well as " and " for allowed separators between authors. The resulting part of the format is shown in Figure 4.

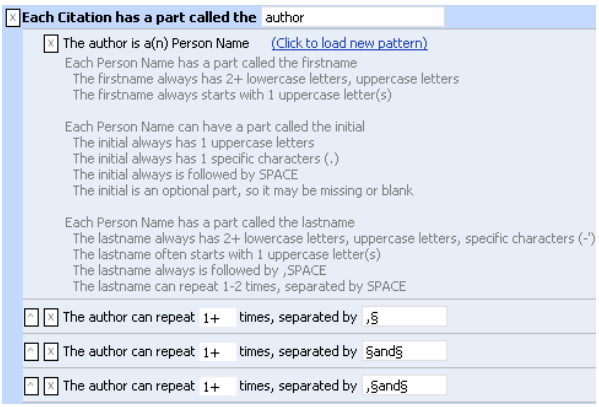


Figure 4: Reusing the person name format for an author part inside the citation format.

Since there was no need to parse the citation's other parts, such as paper title and date, we simply added a part called "otherstuff". Leaving that part unspecified allowed it to consume whatever characters in the citation would not be picked up by the author list.

We saved the citation format in a file called "citation.xml" located in a subdirectory of the web site, where the script could retrieve this file at run-time and pass it to our parser via the mashup library.

Step 2, creating formats for contact information

We created two additional formats for parsing author contact information that would be grabbed from HTML sent back by a web application (as in Figure 5).

Figure 5: Contact information returned by server

The script needed to grab the phone number and email address. The relevant snippet of the HTML is:

```
<b>Office</b><br>
412-268 3564<br>
412-268-2338 (fax)
<p>cscaffid @ andrew.cmu.edu<br>

<a href = "http://www.cs.cmu.edu/~cscaffid ">Home Page</a><p>
```

Note the extra spaces in the email address and the use of a space rather than a hyphen between the phone number's exchange and local number. Before presenting these values as output, we wanted the script to reformat them, which required referencing parts of the phone number and email address. Therefore, we created a format for each category of data and stored them in files on the server alongside the citation format file.

Step 3, writing JavaScript to call the parser

Finally, we needed to write JavaScript to retrieve author names out of the citation, to pass those author names to an existing web application, and to retrieve contact information from the HTML returned by the web application. Appendix A shows the mashup script. The following is a recap of the highlights.

The script passed the text from the selected citation into our library's `parseStructure` function, which used our parser to parse the string into a structured object according to the citation format file. For example, parsing the string shown in Figure 2 yielded the structure shown below. The parser automatically annotates each node in the parse tree with a label (such as "firstname") read from the format.

```
citation
├─author
│  ├ firstname       = "Chris"
│  └ lastname        = "Scaffidi"
├─author
│  ├ firstname       = "Mary"
│  └ lastname        = "Shaw"
├─author
│  ├ firstname       = "Brad"
│  ├ initial         = "A."
│  └ lastname        = "Myers"
└─otherstuff         = ". Estimating ..."
```

The script used our mashup library's `nodes` function to get an array of the author nodes (3 in this case), then called our mashup library's `textFor` function to retrieve the text for each author's "firstname" and "lastname" nodes. With the author names in hand, the script then requested each author's contact information from the web server (using our mashup library's `callWhenReady` function).

Finally, our script used our mashup library's `grabStructure` function to locate each author's phone number and email address in the HTML from the web application. The script then used our library's `conf` function to verify that the phone number and email address were indeed present (which would not be the case when the database does not contain the person, or there are multiple people with the same name). Concatenating the contact information with author names yielded the desired text, which the script displayed as a tooltip on the citation.

Expressiveness

There was no difficulty at all in expressing the necessary formats using the editor. In particular, there was no difficulty creating a hierarchical format for the citation.

Line-count

Without our mashup library (which relies on our editor and parser), the mashup would have required dozens or hundreds of lines of code. In contrast, the basic implementation shown in Appendix A required 4 format files and approximately 2 dozen lines of JavaScript (plus basic input/output, event-handling and error-handling code, such as try/catch for network connectivity errors, that would have been required regardless of whether we used our parser and formats or whether we used another approach such as regular expressions). This gratifyingly low line-count was possible because of the flexibility of formats compared to other approaches such as regular expressions.

Performance

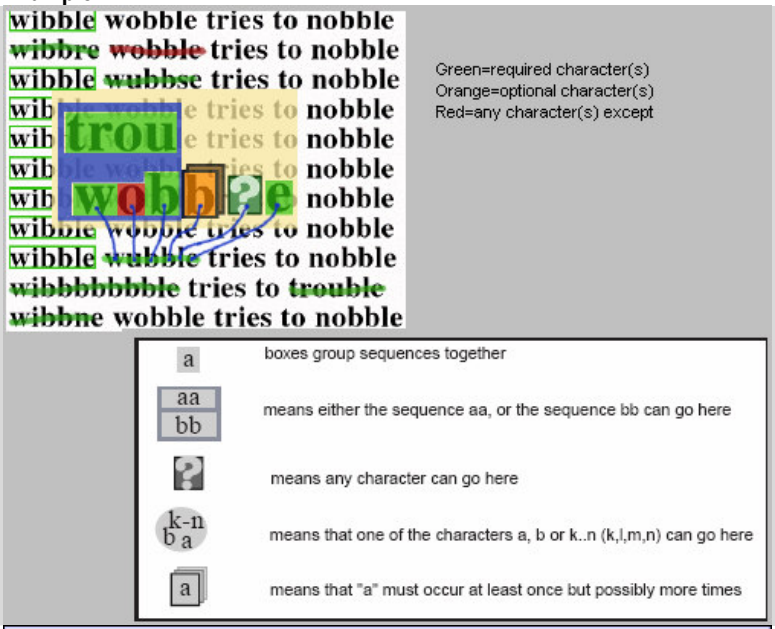
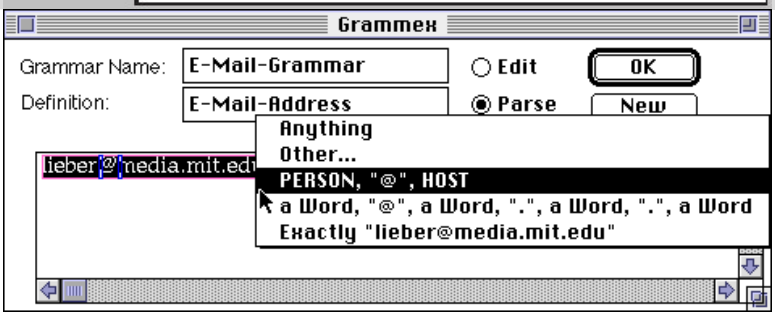
One of the problems with a mashup in general is that the client code must wait for the web application server to return HTML in response to each URL request. Some servers (including the one used in this mashup) can take a few seconds to respond. Our first implementation of the `callWhenReady` library function performed these requests *serially*, resulting in ten to fifteen seconds of latency. Redesigning this function to perform asynchronous requests resulted in a much more tolerable latency of only a few seconds.

6. Related work

Several systems aim to make it easier for people to use regular expressions and context-free languages to define the structure of data. Table 4 shows several examples. SWYN is a visual language that enables end-user programmers to manipulate symbolic pictures that are represented internally as regular expressions [2]. Grammex is an editor for context-free grammars, which end-user programmers define by identifying the grammar non-terminals, and then specifying the form of each non-terminal (for instance, as an integer or a word, or as a list of terminals or non-terminals) [6]. Apple data detectors are also context-free grammars, though in this case, end-user programmers define a grammar by typing the productions for non-terminals, which (for convenience) can take the form of a regular expression [10]. Lapis data patterns are similar to context-free grammars, again typed by end-user programmers, though primitives are also provided for common forms (such as numbers, words, and punctuation) [7][8].

These systems have two limitations. First, data formats in these systems specify binary recognizers: either a value matches the format, or it does not. Thus, these formats cannot support testing for soft constraints. Second, many of the systems above rely on regular expressions, which are cumbersome (though not impossible) to use for expressing commonly-occurring constraints on data, as discussed in Section 1. As a result, several of the formats shown in Table 4 would probably be difficult for end-user programmers to understand.

Table 4: Example data patterns in SWYN, Grammex, Apple data detectors, and Lapis

System	Example	Comments
SWYN [2]		It appears that no screenshots of SWYN expressions for common data such as dates or email addresses have been published. The example provided in [2], shown to the left, corresponds to `(trou | w[a-n,p-z]b) b*[a-z]e`
Grammex [6]		In Grammex, the end-user programmer gives examples and shows how to break the examples into parts that must match a certain form. In the screenshot, `HOST` and `PERSON` were already defined (in popups), and the end-user programmer specifies a grammar for an email address.

Apple data detector [10]	`Date =`
	`{`
	`  (MonthNumber [/\-]+ Day [/\-]* Year?)`
	`}`
	`MonthNumber =`
	`{`
	`  [1-9],`
	`  ([1] [012])`
	`}`
	`Day =`
	`{`
	`  [1-9],`
	`  ([12] [0-9]),`
	`  ([3] [01])`
	`}`
	`Year =`
	`{`
	`  ([1-9] [0-9]),`
	`  ([1-9] [0-9] [0-9] [0-9])`
	`}`
		Apple data detectors are based on a textual language that mixes regular expressions and context-free grammars.
		
		The Apple data detector and Lapis examples have been simplified to conserve space.

Lapis [7][8]	`@DayOfMonth is Number equal to /[12][0-9]|3[01]|0?[1-9]/`
	`  ignoring nothing`
	
	`@ShortMonth is Number equal to /1[012]|0?[1-9]/`
	`  ignoring nothing`
	
	`@ShortYear is Number equal to /\d\d/`
	`  ignoring nothing`
	
	`Date is flatten @ShortMonth`
	`  then @DayOfMonth`
	`  then @ShortYear`
	`  ignoring either Spaces`
	`  or Punctuation`
		Lapis is a textual language with a context-free structure blended with regular expressions and new primitives for matching and skipping common forms (such as strings of spaces or punctuation).

Internally, our system represents formats with an ACFG, meaning that CFG productions can be annotated ("augmented") with constraints [1]. However, our ACFG differs from prior ACFGs that did not allow soft constraints: if a constraint is violated, then the parse is forbidden [1][3][19]. In contrast, our ACFG notation can express soft constraints that allow a slightly invalid parse but cause the parser to downgrade its confidence in the parse's validity.

Stochastic CFGs use probabilities to quantify confidence in parses, but these numbers are associated with productions themselves rather than with constraints on the productions [1]. Through machine learning, some parsers can be trained to apply different probabilities based on context in natural language [4][13][18], but this is still not as expressive as letting users specify arbitrary numeric penalties on arbitrary constraints.

7. Future work

Future usability evaluation with actual end users will probably reveal additional opportunities for enhancing our system. In particular, we suspect that we may need to add additional functionality to help end users test the formats that they create. For example, we may enhance the editor to let end users save example data for use in regression testing when formats are modified.

As noted in Section 5.1, we sometimes needed to define multiple formats for one semantic family. Future versions of our system will enable users to demonstrate how to transform one format into another within the same family, or "tope." Our plan is described in more detail in [14].

Finally, it may be desirable to design a more sophisticated editor that also generates ACFG output. This would enable end-user programmers to tap into more of the power of ACFGs. A more sophisticated editor might allow advanced users to enter textual spreadsheet-like Boolean expressions for constraints. For example, we could enable users to write a conditional expression to forbid the `DAY` in a date from exceeding 28 if the `MONTH` is 2. While this is expressible in our current ACFG notation, we might add an additional function to make the constraint more succinct.

8. Acknowledgements

This work has been funded in part by the EUSES Consortium via the National Science Foundation (ITR-0325273) and by the National Science Foundation under Grants CCF-0438929 and CCF-0613822. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

9. References

[1] J. Allen. *Natural Language Understanding*, Benjamin/Cummings Publishing Co., 1995.

[2] A. Blackwell, SWYN: A Visual Representation for Regular Expressions. *Your Wish is My Command: Programming by Example* (H. Lieberman, ed), Morgan Kauffman, 2001, 245-270.

[3] T. Briscoe, J. Carroll. Generalized Probabilistic LR Parsing of Natural Language (Corpora) with Unification-Based Grammars. *Computational Linguistics*, *19*, 1 (1993), 25-59.

[4] M. Jones, J. Eisner. A Probabilistic Parser and Its Applications. *AAAI Workshop on Statistically-Based NLP Techniques*, 1992, 20-27.

[5] R. Lerner. At the Forge: Creating Mashups, *Linux Journal*, *2006*, 147 (Jul. 2006), 10.

[6] H. Lieberman, B. Nardi, D. Wright. Training Agents to Recognize Text by Example, *Auton. Agents and Multi-Agent Systems*, *4*, 1 (Mar. 2001), 79-92. See http://www.miramontes.com/writing/add-cacm/index.html for screenshots.

[7] R. Miller. *Lightweight Structure in Text*. Technical Report CMU-CS-02-134, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2002.

[8] R. Miller, B. Myers. Lightweight Structured Text Processing. *Proc. 1999 USENIX Annual Technical Conf.*, 1999, 131-144.

[9] F. Mosteller and C. Youtz. Quantifying Probabilistic Expressions. Statistical Science, Vol. 5, No. 1, 1990, pp. 2-12.

[10] B. Nardi, J. Miller, D. Wright. Collaborative, Programmable Intelligent Agents. *Comm. ACM*, *41*, 3 (Mar. 1998), 96-104.

[11] J. Pane, B. Myers, L. Miller. Using HCI Techniques to Design a More Usable Programming System. *Proc. 2002 Symp. Human Centric Computing Lang. and Environments*, 2002, 198-206.

[12] J. Rode, et al. As Easy as "Click": End-User Web Engineering, *Proc. Intl. Conf. Web Eng.*, 2005, 478-488.

[13] C. Rose, A. Lavie. Balancing Robustness and Efficiency in Unification-Augmented Context-Free Parsers for Large Practical Applications. *Robustness in Language and Speech Technology*. Kluwer, 2001.

[14] C. Scaffidi. A Lightweight Model for End Users' Domain-Specific Data. Graduate Consortium abstract in *Proc. 2006 Symp. Visual Languages and Human-Centric Computing*, 2006, 242-243.

[15] C. Scaffidi. Unsupervised Inference of Data Formats in Human-Readable Notation. *Proceedings of 9th International Conference on Enterprise Integration Systems (ICEIS'07)*, 2007, to appear. Available online at http://www.cs.cmu.edu/~cscaffid/papers/eu_20060612_topei.pdf

[16] C. Scaffidi, A. Cypher, S. Elbaum, A. Koesnandar, and B. Myers. *The EUSES Web Macro Scenario Corpus, Version 1.0*. Technical Report CMU-HCII-06-105, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2006.

[17] C. Scaffidi, B. Myers, M. Shaw, Trial By Water: Creating Hurricane Katrina "Person Locator" Web Sites, *Leadership at a Distance* (S. Weisband, ed), Lawrence Erlbaum, 2006, to appear.

[18] A. Stolcke. An Efficient Probabilistic Context-Free Parsing Algorithm That Computes Prefix Probabilities. *Computational Linguistics*, *21*, 2 (1995), 165-201.

[19] M. Tomita. An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics*, *13*, 1 (1987), 31-46.

Appendix A: Mashup code

The following code was used to implement the sample mashup discussed in Section 5.2. For conciseness, error-handling code has been removed. Grayed out lines deal with basic event-registration and input/output that would be required regardless of whether we used our parser/editor or whether we used another approach such as regular expressions.

```
var MYBASE = "_topep/"; // base href in relative URLs
document.write("<s"+"cript src='"+MYBASE+"topep.js'></"+"script>"); // Load the library
document.attachEvent("oncontextmenu", dolookups); // Event registration for right-click

function dolookups() {
  /*** basic input code follows ***/
  var el = window.event.srcElement; // the clicked citation element
  if (el == null) return;

  // show an "I am busy" message
  window.status = "Looking up CMU authors...";
  window.event.returnValue = false;

  var names = new Array(), urls = new Array(); // build up arrays of author names and urls
  var p = parseStructure(el.innerText, MYBASE+"citation.xml"); // parse citation into tree
  var authors = p.nodes("author"); // get author nodes from the parse tree

  for (var i = 0; i < authors.size(); i++) { // for each author
    var author = authors.get(i); // get the author node
    var lname = author.textFor("last"); // get the author's last
    var fname = author.textFor("first"); //        and first names

    var url = "http://people.cs.cmu.edu/cgi-bin/search?name="; // build the query string
    url += (fname.length > 2 ? escape(fname+" "+lname) : escape(lname));

    urls.push(url); // and add the url to the list of urls that we will retrieve
    names.push(fname+" "+lname); // also record the person name that goes with that url
  }

  callWhenReady(urls, showContact, names, el); // call me when the urls are downloaded
}

// A reference to this function was passed to callWhenReady.
// So callWhenReady will callback this function
//   with the array of htmls
//   and the other callWhenReady parameters
//   when the urls are done downloading.
function showContact(htmls, names, el) {
  var msg = ""; // this will be the message that we show to the end user

  var SMARK = "<b>Office</b>", EMARK = "<a"; // these marks delimit contact info
  for (var i = 0; i < names.length; i++) { // for each url/author requested...
    var html = htmls[i]; // this is the html from that author url

    var q = grabStructure(html, SMARK, EMARK, MYBASE+"email_address.xml"); //parse email
    var r = grabStructure(html, SMARK, EMARK, MYBASE+"phone_number.xml");  //parse phone

    if (q.conf() > 0.1 && r.conf() > 0.1) { // if it looks like both are somewhat valid
      var email = q.textFor("username")+"@"+q.textFor("server"); // reformat email
      var phone = r.textFor("area_code")
                    +"-"+r.textFor("exchange")
                    +"-"+r.textFor("local"); //reformat phone

      /*** output formatting code ***/
      if (msg.length > 0) msg += "\r\n";
      msg += names[i] + " " + email + " " + phone; // and then append to the message

    }
  }

  /*** output ***/   // show the message to the user
  window.status = msg;
  el.title = msg;
}
```