Modern Extensible Languages

Daniel Zingaro, McMaster University, Hamilton, Ontario, Canada

April 11, 2007

Abstract

Extensible languages are programming languages which allow a user to modify or add syntax, and associate the new syntactic forms with semantics. What are these languages good for? What kinds of features are easy to add, and which are not? Are they powerful enough to be taken seriously? In this survey we will attempt to answer such questions as we consider procedural, object-oriented, functional, and general-purpose extensible languages. We are primarily interested in expressive power (regular, context-free), associated caveats (unhygienic, ambiguity) and ease of use of the various mechanisms.

1 What is an Extensible Language?

Before beginning, it is essential to have an operational definition of what, exactly, constitutes an extensible language; this will dictate the flow of the remainder of the paper. Standish [23] gives a rather broad definition, stating simply: an extensible language allows users to define new language features. These features may include new notation or operations, new or modified control structures, or even elements from different programming paradigms. To facilitate discussion, it is helpful to adopt Standish's nomenclature, and divide this spectrum of features into three classes: *paraphrase*, *orthophrase* and *metaphrase*. Paraphrase refers to adding new features by relying on features already present in the language; a prototypical example is defining macros, which of course ultimately expand into the programming language's standard syntax. Orthophase refers to adding features not expressible in terms of available language primitives; for example, adding an I/O system to a language that lacks one. Finally, metaphrase refers to changing the interpretation of existing language elements, so that expressions are parsed in new ways. In 1975, Standish painted a rather bleak picture of the success

of metaphrase techniques; we will see shortly that thirty years of experience have certainly improved the situation.

There are several classes of programming languages (imperative, functional, object-oriented) for which we may attempt language extension. The remainder of the paper is thus organized as follows. We begin with a simple macro definition facility for an imperative language (C), then move into more sophisticated mechanisms available in a (mostly) functional language (OCaml). We then look at two rather different extension ideas for an object-oriented language (Java), investigate the body of work related to hygienic macro expansion (Scheme), and conclude with two powerful general-purpose extension languages. Throughout, we will draw comparisons between the parsing mechanisms used and the expressiveness of the techniques.

2 Unhygienic Macros in C

We shall begin where we inevitably must begin any survey of programming languages: with C. The C preprocessor includes a mechanism whereby macros can be defined; prior to compilation, these definitions are expanded so that the compiler can focus on C proper. (We are therefore looking at a paraphrastic mechanism.) Provisions are made for two types of macros: object-like, and function-like [16]. Object macros simply associate an identifier with a corresponding replacement (typically valid C code). The most common use for this basic text substitution is introducing mathematical constants such as π, or program-specific constants such as $ARRAYSIZE$. Function macros are similar to standard C functions in many ways. They can accept arguments, which can be used in their corresponding replacements, and can additionally be variadic (i.e. can accept a variable number of arguments). A common example of such a macro is the following, which computes the minimum of two numeric values (taken from [16]):
`#define min(X, Y)  ((X) < (Y) ? (X) : (Y))` After this point in the source file, min is available as (what looks like) predefined C syntax, and can be invoked like any other C function.

In passing, we should note the other main use of the C preprocessor: allowing inclusion of other C source files. This can be thought of as a feature in the orthophrase category because, for example, it can allow adding features (like strings and input/output) which do not belong to the base C language.

2.1 Discussion

What we have just described is certainly not a very convincing argument for the claim that C is an extensible language. Most of the functionality allowed by the object-like macros could be directly incorporated in a typesafe way in a language supporting constant definitions. Function macros are expanded inline — an optimization that admits a certain speedup at the cost of increased code size. However, this is typically available as a compile-time option anyway, so these macros seem to offer little more than standard C functions. Even the syntax used when calling these "functions" still looks like C. Is this really extensibility?

The utility of the preprocessor is further marred by its so-called unhygienic nature. This means that innocuous-looking macros can unwittingly hide variables which should be acted upon, or – via subtle parenthesization issues – change precedence rules as to get an undesired effect. Most of these problems stem from the fact that, in contrast to standard functions, direct substitution of actual arguments for formal arguments is made. As a concrete example, a macro to swap two arguments may look like `#define swap(a, b) {int t; t = a; a = b; b = t;}` If a variable `t` happens to occur in a lexical scope where this macro is expanded, it will be hidden by the new scope of the macro, and so a swap involving `t` will do nothing. Additionally, if a function invocation is used as an argument to this macro, the function will be executed twice, because it is expanded twice within the macro body [16]. Should the function have any side effects, the results will again be unexpected.

Do seasoned C programmers really make these types of mistakes? Surely not, especially in light of well-known techniques for minimizing the aforementioned issues, such as parenthesizing the entire macro body and all of the individual variables therein [16]. Ernst et al. [13], however, analysed 26 C programs comprising 1.4 million lines of C code and found quite the opposite. Of all macro definitions, 23% of them included latent errors (of the sort we have described), waiting to strike when called with just the right identifier names, in just the right lexical environment. In summary, then, not only is this macro facility not much of an extension to C, it is a dangerous one at best. We will now consider far safer, more powerful alternatives.

3 Camlp4

Camlp4 is a preprocessor for the multi-paradigm language Ocaml; that is, it performs transformations on code, much like the C preprocessor described

above. Vastly different, however, is the method by which Camlp4 achieves its results. First, instead of acting at a textual (search and replace) level, it performs transformations on Ocaml abstract syntax trees (ASTs) [9]. Second, it allows for the grammar of the Ocaml language to be augmented with new rules, modified rules, or the deletion of rules. To effect a syntax transformation, one first modifies the Ocaml grammar to include the desired syntax. Next, to transform this new syntax into Core Ocaml, transformations are specified in order to create a new AST from the old one. Creating ASTs node-by-node would be tedious [9], and so Camlp4 provides a mechanism called quotations to ease the task. Specifically, quotations allow one to describe an AST by using Ocaml concrete syntax — since it is unambiguous, it can be directly translated to the corresponding AST. While standard Ocaml concrete syntax would suffice, the designers of Camlp4 took this opportunity to introduce a modified syntax for use inside quotations. The thinking was that they would clean up the rough parts of the grammar, making it easier to understand, parse and pretty-print [9]. (As with any language that tries to remain backward-compatible for many years, Ocaml's syntax is bogged down with legacy constructs, unnecessary to preserve for use in quotations.) As a case in point, standard Ocaml syntax allows the programmer to declare what is effectively a global variable: `let magicnum = 42`. It also allows let-bindings to be introduced with the same let keyword: `let magicnum = 42 in magicnum` 5+. These two constructions result in different types of AST nodes; the one to create is not known until the word `in` is (or is not) seen. Using `value` for the first case, and continuing to use `let` for the second, dispenses with this parsing difficulty. Other changes include using C-like curly braces in imperative constructions, eliminating duplicate constructs for referring to boolean operations and assignments, and requiring parentheses around tuples [9].

With these pieces in place, we can now present a concrete example, taken from [9], of extending Ocaml's syntax with a C-style for-loop. As specified above, we first extend the Ocaml base grammar; the following achieves this:

```
extend expr: LEVEL "expr1"
  [ [ "for"; v = LIDENT; iv = expr LEVEL "simple";
   wh = expr LEVEL "simple"; nx = expr LEVEL "simple";
   "do"; e = expr; "done" ->
    gen_for loc v iv wh nx e ] ]
```

Notice that the new for-loop extends the `expr` nonterminal; that is, it is an expression, just like `while`, `if`, and the original (restricted) `for`. The LEVEL annotations are used by Camlp4 to disambiguate concrete syntax, and is used for imposing precedence and associativity [9]. The keywords

in quotation marks, such as `for` include their text exactly in the concrete syntax; the other parts of the definition associate symbolic names with pieces of the syntax tree, such as expressions or identifiers. The following, then, is a valid for-loop according to this new syntax:

```
# for c 0 (c<5) (c+1) do print_int c; done
```

The final step is to translate from the new syntax back into core Ocaml to be compiled; it should be clear that `gen_for` should carry out this task. In [9], it is defined as:

```
let gen_for loc v iv wh nx e =
  let loop_fun = gensym "iter" in
  <:expr<
    let rec $lid:loop_fun$ $lid:v$ =
      if $wh$ then do { $e$; $lid:loop_fun$ $nx$ } else ()
    in $lid:loop_fun$ $iv$ >>
```

`Gensym`, not shown (but see Appendix A for a sample), returns what it hopes is a unique identifier, by using a counter plus an obfuscated text string; its result is captured in `loop_fun`. The syntax `<:...>>` defines a quotation (an AST node) - in this specific case, an expression node. The quotation defines a recursive function which takes one parameter (the starting value of the for-loop variable), and proceeds to test the first expression (`iv`) of the for-loop. If it is true, it executes the loop expression `e` and recurses with the new value of `iv` resulting from the third expression. The dollar-signs allow ASTs (such as expressions) to be inserted in bigger ASTs (i.e. the ones created by quotations).

Many new constructs can be added via Ocamlp4; `camlp4tut` additionally contains a Pascal repeat-until loop and C-like `#defines`; [17] implements a `try...finally` construct; and the present author has added a simple form of type ascription in the spirit of the one presented in Chapter 11 of [21] (see Appendix A).

3.1 Discussion

Camlp4 gives a glimpse into the power of metaphrastic techniques, since it allows old rules of the grammar to be substituted for new ones. The Ocaml grammar is specified in such a way as to make it parseable by a recursive descent parser [8]. The changes one makes to the grammar should therefore not violate this property. To make this more feasible, Camlp4 automatically left-factorizes rules of the same precedence level [8], but we are still left with only a subset of context-free grammars at our disposal. Furthermore, like C macros, the Camlp4 substitutions are unhygienic in general, as

is evident from the techniques used to generate (hopefully) conflict-free identifiers. However, because we are now operating on ASTs, precedence issues and correct parenthesization become non-issues. The types of modifications we can make are certainly more powerful as well: they "look" different than core Ocaml syntax. The grammar extension idea, then, seems like a powerful technique to effect extensibility, and so we will pursue the study of related attempts below.

4 Java Syntactic Extender

The Java Syntactic Extender (JSE) is to Java what Camlp4 was to Ocaml. It is again a syntactic preprocessor, which interoperates with the code of the target language [1]. As evidenced by the previous section, there are several advantages to this approach, as opposed to a specialized substitution language. Most notably, no such language has to be created in the first place, and the full power and expressivity of the target language (Ocaml or Java) is available. For manipulating fragments of source code, JSE uses skeleton syntax trees (SSTs). Similar to Camlp4's ASTs, SSTs work on the abstract syntax of the language; the difference is that SSTs contain only the shapes most useful for macro processing [1]. Such SSTs can be created manually, or can be created using concrete syntax via code quotes - similar, even in name, to Camlp4's quotations. A pattern matching technique is used to add new syntactic forms to the Java grammar. SST structures (represented as code quotes) are specified as the patterns to match, and identifiers are bound to specific parts of the tree for use in the associated translation. Rules for a specific construct are examined, top to bottom, until a match is found; if there are multiple possible matches, only the first is used. Additionally, there are no provisions for any ambiguity introduced into the grammar by such modifications. When source code written in an extended syntax is to be converted to standard Java code, the macro expander works top-down, expanding macros according to the above rules, until none remain. For every identifier that could be macro-expanded, JSE looks for a class file with a predefined name, prefixed by the identifier it is trying to expand. In this way, syntax expanders take on an object-oriented flavor to compliment the rest of the language; note, though, that this requires that all macros begin with a name (i.e. an identifier).

It should not be surprising that the types of transformations possible with this syntax extender (foreach loops, syntax for enumerations, synonyms for existing keywords, etc.) are similar in power to those possible with Camlp4 [1] [2]. Hygienic macro expansion is, once again, not implemented,

and measures must be taken to prevent the now well-known problems this entails. For example, if we try to write a macro to add a `foreach` statement to Java (presumably for versions prior to 1.5) in terms of `for` and `iterators`, we must pick a name for the iterator. If nested `foreach` statements are encountered, though, the inner iterator name will bind the outer one [2]. We are forced, yet again, to settle for an identifier generator, with a statement such as `Fragment i = IdentifierFragment.genSym("i")` to generate fresh identifiers.

5 OpenJava

We've already illustrated why the textual replacement macro techniques are unsuitable as general language extensibility features, and found that AST-based approaches seem to rectify many of their shortcomings. Can we do better than this, using some other structure for code representation?

To begin, observe once again that syntax trees are just a parsed representation of textual source code. Node types are introduced to represent the abstractions of the language, and further phases of the compiler operate on this intermediate representation rather than raw source code. However, in terms of information about context or program structure, the syntax tree says no more than the original code [24]. What this means is that, while working at the AST level can help us with precedence issues and make it easier to perform substitutions, it cannot assist when other information — such as types — is necessary for proper macro functioning. As a specific example, consider the observer design pattern, where objects designated as observers of subject X are notified by X of changes they should be aware of [24, 14]. Such observers may have to adhere to an interface specifying various notifications that they may be sent; should they wish to ignore an event, the corresponding method body is intentionally left blank [24]. This opens the possibility for introducing a macro to fill in these empty methods, assuming that they were not included because observers plans to ignore them.

With our existing ideas, it should be obvious that this task is not easily accomplished. To begin trying, we should look for syntax in our class that is to be replaced by new syntax which defines the missing methods. This immediately leads to problems. Which missing methods are we talking about? Naturally, they would be defined in an interface, not even present in the source file we are manipulating. Assuming we somehow knew what these methods were, what do we match our class text against to determine which methods are missing? Perhaps we could pattern match against the entire

syntax tree, extract the method names, determine which methods from the interface do not exist, then augment the AST with these new methods. This involves much knowledge about the syntax trees of Java and, to be sure, things would be easier if a more useful representation of the source program was available. If we instead had a list of methods that the class was required to implement, and a means to insert new methods into this collection, writing the macro would be reduced to a trivial exercise. The idea is that we would get a list of methods present in the class and determine which of these methods included the *abstract* modifier. This would give a list of methods that were left out of the class source, so we can simply add these methods and give them empty bodies: no explicit AST, no complicated pattern matching [24]. This line of thinking encapsulates the main ideas of OpenJava—a powerful metaphrastic system for Java.

The central idea is the class meta-object — an object representing the logical structure of a Java class [24]. These objects provide streamlined access to the class they represent through method calls, which are used by macros to modify the class [24]. The classes for such meta-objects are called metaclasses, and all inherit from a common metaclass (`OJClass`) which requires the existance of a `translateDefinition` method to perform the actual macro expansion. Including an instantiates clause in a class definition tells OpenJava to associate an object of the specified metaclass with that Java class.

`TranslateDefinition` can make use of member methods of `OJClass` [24]. For example, `getMethods` can be used to get a list of the Java class's methods; `getModifiers` can be used on these methods to obtain their modifiers (like public or private); `getReturnType` can be used to obtain the return type of methods; `OJMethod` is used to create new method metaobjects; and `addMethod` can add new methods to Java classes. We can now present `translateDefinition` for the previously described observer pattern, taken from [24]:

```
void translateDefinition() {
  OJMethod[] m = this.getMethods(this);
  for(int i = 0; i < m.length; ++i) {
    OJModifier modif = m[i].getModifiers();
    if (modif.isAbstract()) {
      OJMethod n = new OJMethod(this,
        m[i].getModifiers().removeAbstract(),
        m[i].getReturnType(), m[i].getName(),
        m[i].getParameterTypes(), m[i].getExceptionTypes(),
        makeStatementList("return;"));
```

```
      this.addMethod(n);
    }
  }
}
```

Note that `OJMethod`'s constructor takes a list of arguments corresponding to the current method's modifiers, return type, name, parameters, throwable exceptions, and body, all trivially derived from the abstract method `m`, which exists in the interface implemented by a Java class acting as an observer. Note also that we do not even have to mention superclasses or interfaces, let alone look at their code or structure. This representation of Java syntax admits other advantages as well. For example, as the authors note [24], if we want to change the name of a class via a macro, we certainly want to change the name of the class's constructors as well, to correspond with their new class. Since this is a required change, understood by the macro processor, OpenJava can do this automatically. (We can imagine having to find all these nodes and change them in AST-based approaches.) As a second example, we do not have to be concerned with the order of methods in a class, or the order of class or method modifiers: the lack of explicit AST makes this inconsequential, and abstracting away this detail makes preprocessing far simpler than dealing with all possible combinations.

While the above discussion was phrased in terms of `caller-side` translations (i.e. translations within a class), OpenJava also supports `callee-side` translations. For instance, we can modify all places in the code where a specific class is instantiated, which helps write macros for other design patterns including flyweight.

Since preprocessing begins under the premise that we have a valid Java class structure, we might think that extending the syntax in arbitrary or unusual ways would pose significant challenges. For example, how is OpenJava supposed to give us a list of methods if we introduce alternative syntax for method declarations? If we really obscure the syntax of Java classes, and since we do not yet have a means of extending the Java grammar, certainly OpenJava can't return meaningful structures for us to operate on. Even so, OpenJava does allow modest forms of syntax extension, which nicely fit into the model we have described. For example, modifiers on methods and classes are just keywords appearing prior to the class or method name. OpenJava therefore allows arbitrary modifiers to be added to the language [24]. There are also specific points in a class file where new clauses can be added; for example, prior to the definition of the first member declaration, or prior to the body block of such declarations. When the parser finds a word it does not understand, it invokes a user-defined method which returns an object

representing a grammar production to parse. Parsing is achieved through an $LL(k)$ technique, so has comparable power to the recursive descent parser used by Camlp4.

6 Hygienic Macros in Scheme

Much research has been undertaken in the implementation of macro systems for the Lisp family of languages, including Scheme. For this reason, it should not be surprising that most work on hygienic macro expansion (one of our goals outlined at the outset) presents itself in such languages; we therefore devote this section to the topic.

How can we be unhygienic in a language like Scheme? There are essentially two ways. The first involves expanding the macro with unlucky identifier names, as in the C swap example given earlier; in other words, the new bindings capture references to identifiers of the same name [12]. The second involves referring to identifiers within macro bodies in lexical scopes where the identifier names have been unpredictably bound [12]. In C, we can witness this by having a macro call what we think is a procedure, but, because of a local definition, is actually a variable [5]. In Scheme, contriving an example is no more difficult: we can write a macro that thinks it is using the `if` operator, and call it from within a scope where `if` has been defined as an integer variable [12]. Like C programmers, schemers have come up with ways of coping with these problems (including using generated identifier names) which we know are innefective because they rely on explicit programmer intervention.

An algorithmic solution to the problem was first proposed by Kohlbecker [18], and has subsequently been used (adapted or otherwise) in other Scheme macro incarnations [12, 5]. The core idea of the algorithm is to take advantage of alpha-equivalence, renaming bound variables to avoid unwanted variable capture. (The same technique is used when performing beta-reductions in the lambda calculus, since this does not affect the terms that are being rewritten [21].) Kohlbecker notes that one might initially try to just rename variables right after a macro expansion introduces them. However, some such variables should not be renamed; for instance, those which are free in the macro (and should be captured by bindings in the enclosing lexical scope) [18]. This problem is exagerated via pyramiding, where macro expansions are based on previously defined macros: in this case it is not even clear which variables are to remain free. Defining a transcription as one step in the macro expansion process, the goal of the algorithm is to prevent bindings introduced in one such transcription from capturing bindings from

other transcription steps or user-defined identifiers [18]. This is achieved through a four-step process. First, all variables are replaced by so-called marked identifiers, which associate variables with timestamps (initially all the same). Next, macro expansion is performed, but after every transcription, the generated variables are replaced with marked identifiers carrying the same timestamp [18, 15]. In phase 3, we replace the bound, time-stamped identifiers with unstamped identifiers which preserve the property that they do not capture variables generated by other transcriptions. The final phase simply removes the timestamps on the remaining (free) variables, and we are left with a normal unstamped term again.

As a concrete example to show some of the algorithm's workings, consider the following term, adapted from [15]:

```
(let ((t yes)) (or #f t))
```

This uses a let-binding to associate the string "yes" with the identifier `t`, which stays in scope throughout the body of the let. It then passes the terms `#f` and `t` to `or`. `Or` is a procedure which continues to evaluate its arguments until it finds one which is true, and this becomes its return value. If all provided arguments are false, or if there are no arguments, `or` returns false. (In Scheme, everything besides `#f` is true.) The term should therefore evaluate to "yes". Let's assume that `or` is a derived form [1] which expands as follows (simplifying to the case where only two arguments can be supplied): verb+(or e1 e2) -¿ ((lambda (t) (if t t e2)) e1)+. Abstracting on the first expression here is necessary to avoid computing its value twice, which is the wrong behavior if it has side-effects.

Using this definition of `or`, our term rewrites as follows:

```
(let ((t "yes"))
((lambda (t) (if t t t)) #f))
```

This reduces to `#f` - the wrong result, and all because the lambda-bound variable in the expansion of `or` was `t`! Using Kohlbecker's algorithm, au contraire, gives the following:

```
(let ((t:1 "yes"))
  ((lambda (t:3) (if t:3 t:3 t:1)) #f))
```

[1] It would also be possible to rather easily define `or` directly in Scheme as a function, and one might wonder why a macro is necessary. The reason lies in the call-by-value semantics of Scheme, where all arguments to functions are evaluated prior to being used; this does not correspond with the `or` semantics.

Substituting "yes" for `t:1`, and `#f` for `t:3`, the term reduces (correctly) to "yes". This shows that the variable `t` was overloaded to refer to two distinct entities with the naive expansion, and rectified using Kohlbecker's algorithm. The complete algorithm is given in [18].

While the algorithm has proven very effective, there are some drawbacks that have been addressed in later work. For example, recall the structure of the algorithm, which we can summarize as a naive macro expansion followed by re-visiting the expanded code to mark the new identifiers. This is an $O(n^2)$ algorithm, compared to $O(n)$ for a naive expander [5]. In [5], this problem is rectified by marking the identifiers when they are first introduced by the expander, avoiding the costly re-scan. In order to ascertain which identifiers are newly introduced, though, macros must be restricted to a high-level, and hence more restrictive, language. In [12], Kohlbecker's algorithm is once again adapted, this time avoiding the quadratic time complexity and also allowing macros to be written more expressively. Provisions are also in place to escape the usually-desired hygienic features, which can sometimes be useful.

With these algorithms as the backdrop, we can now focus on how to achieve macro extension in Scheme. We use `define-syntax`, which associates a transformer with a keyword [11]. This is similar in spirit to what was done in the Java Syntactic Extender, and again, there are syntactic forms available for easing the creation of these transformers, including syntax-rules and the more powerful syntax-case. To use syntax-rules, we provide a list of literals, and a list of clauses. Clauses are pattern-template pairs: whenever the pattern is found, it is transformed according to the corresponding template [11]. Identifiers in patterns are pattern variables, unless they appear in the list of literals, in which case they are treated as auxiliary keywords. An example of such a keyword is `else`, required by a macro implementing an if-then-else construct. We can also use ellipses in patterns, to match zero or more repetitions of the adjacent subpattern. In syntax-case, transformers are procedures of one argument, taking and returning syntax objects. We again specify a list of clauses consisting of patterns and expressions, the latter describing how to rewrite syntactic forms that matched the associated pattern. Syntax-case also allows the presence of fenders, which can impose additional constraints on the pattern-match, independent of syntactic structure. For example, fenders can be used to determine if the introduction of a new binding would capture references to other identifiers [10]. (Since macros in syntax-case are hygienic, this can only happen if the two identifiers have the same name, and were both introduced at the same transcription step.)

A simple, concrete example of syntax-case may help put this dialogue into perspective. We present a macro expansion for `when`, which takes expression

$e0$ and one or more expressions $e1, e2, \cdots, em$, and executes $e1, e2, \cdots, em$ if $e0$ is true [10]. Notice how ellipses are used to specify "zero or more" expressions.

```
(define-syntax when
 (lambda (x )
  (syntax-case x ()
   ((_ e0 e1 e2 ... ) (syntax (if e0 (begin e1 e2 ... )))))))
```

7 General Extensibility Frameworks

TXL, the Turing Extender Language, was initially invented to quickly allow modifications to be made to the Turing language's syntax. The reason was that Turing was developed by assessing how user's expected it to work, and then modifying it to fit this perception [6]. TXL was thus conceived as a rapid prototyping environment with which to test new additions to the language without having to modify any component of the compiler [6]. It has since grown into a powerful language-independent extension facility, and in this way deviates substantially from language-dependent extenders of the flavor we saw above. We start with a base language grammar, then define new rules or redefine old ones, and assign semantics to the new syntactic forms. The original grammar, and new definitions, are presented in EBNF; that is, we can specify that something be optional or can be repeated zero or more times, in addition to using BNF-style sequencing and alternation. Grammar rules are represented as lists, and parsing decisions are made based on the first rule that matches [6]. In this way, ambiguity is implicitly resolved via the order of productions given in the grammar text. When adding an alternative to a rule, we can decide whether it should be the first or the last rule examined, providing an easy way to give it high or low priority. A top-down (theoretically exponential) backtracking algorithm is used to facilitate parsing of any context-free grammar. Unfortunately, this exponential worst-case can easily occur with left-recursive grammars. TXL detects this special situation and uses a bottom-up parser at these points.

Besides matching on a single level of syntactic category, TXL provides deconstructors which allow pattern variables to be matched against more specific patterns. For example [6], assume we have bound a variable `x` to an if-statement, and we want to remove it if the if-condition is literally false. We can deconstruct the if-statement in `x` into its components, one of which is the if-condition. We can further deconstruct this into its lexical representation; if it is "false", it shouldn't appear in the translation of the code, so it is replaced by the statements following it.

TXL also includes provisions for making transformations dependent on context information from other parts of the code, and hence on other variables bound to remote parse trees. We use subrules for this, passing them the variables on which they may depend. To exemplify this — and also the general flavor of a TXL definition — we present an example that replaces constants by their values in source text [6]. Note how `C` binds the identifier name after the string "const", while `V` binds its associated expression. ReplaceByValue is the subrule which finds occurrences of `C`, replacing them by (parenthesized) `V`.

```
rule resolveConstants
  replace [repeat statement]
  const C [id] = V [expression];
  RestOfScope [repeat statement]
by
  RestOfScope [replaceByValue C V]
end rule

rule replaceByValue ConstName [id] Value [expression]
  replace [primary]
  ConstName
by
  ( Value )
end rule
```

The types of transformations possible with these facilities are far-reaching; for example, [7] defines a generics instantiator, which allows the definition of generic structures (procedures, modules, variables) in Turing.

Since its inception, TXL has adopted the attitude which drove Turing's development: modify the language to suit user expectations. One feature added as a result of this was guards, implemented via "where" clauses [6]. In the spirit of destructors above, we often want to impose further constraints on a rule, even after it successfully matches on some part of the input. This is accomplished using condition rules, which are like normal TXL rules, except they do not perform any substitution. They may succeed or fail; failure results in the failure of the calling rule. Other features had to be added to TXL as it branched off from its Turing-centric beginnings. For example, Turing's lexical conventions were once built-in to TXL, but now it is user-configurable [6]. This additionally lets users work at the lexical level (supporting, for example, scannerless parsing), should it prove more convenient.

A similarly general extension mechanism has been developed by Cardelli et al. with their work on extensible grammars [3]. Compared with TXL, these extensible grammars are parsed as $LL(1)$, and so we have to be more careful when crafting and extending grammars. Instead of beginning with a context-free grammar of the base language, we instead start with its abstract syntax. This is described by giving the *sorts* (the types of tree nodes) and *constructors* (the valid ways of making these nodes). For example, the nodes in a lambda-calculus syntax will naturally include `term`, and the constructors would include application (taking two terms and returning one). There are three predefined sorts for identifiers, for specifying that an identifier is being used as a binder, that it is within a binding expression or that it is not scoped. We can then provide a context-free grammar and associated translation rules for translating a concrete syntax into this abstract syntax. In order to avoid unwanted variable capture, the binding information present in the abstract syntax is used to rename variables. As in TXL, further abstractions can then be added by extending the context-free grammar in various ways, such as adding new nonterminals, adding new productions to existing nonterminals or completely redefining the productions of nonterminals. Productions are associated with actions, which can use variables bound in the matching process as arguments to constructors. This corresponds to creating ASTs node-by-node and, as we've come to expect, there is also a pattern mechanism which allows the constructors to be implicitly called via previously defined concrete syntax.

8 Other Extensible Languages

There are several other extensible languages which we only briefly mention, not because they are unimportant, but because our cross-cutting survey has already covered their main ideas.

8.1 OpenC++

OpenJava was actually developed after its author had worked on a similar system for C++ called OpenC++. It similarly uses a meta-object protocol, which refers to having the metaobjects control compilation [4]. Instead of the more object-oriented view of the class structure used in OpenJava, OpenC++ uses what is conceptually still a parse tree [24], tending to make some elementary examples more complicated.

8.2 Seed7

Seed7 is syntactically similar to Pascal and Ada, but also includes object-oriented features and extension mechanisms [20]. An extension includes two parts: a syntax definition, giving a template for the new syntactic form (including, as in Camlp4, associativity and precedence information), and what looks like a standard Seed7 function, used to superimpose a semantics. The function takes the parameters of the new syntax (variables, types, statements) and can use them in the body of the function to effect the transformation. Like Camlp4, the Seed7 grammar with macro extensions is parsed via recursive descent [19]. Seed7 does more work to detect ambiguities at compile-time, though, and alerts the user appropriately. As an example, defining an infix operator `in` of higher priority than a `for-loop` which uses `in` as one of its keywords causes the `for-loop` to never be recognized. This is detected at compile-time and a `priority error` is given.

8.3 Felix

Felix is a procedural language with static typing, first-class functions and garbage collection [22]. The extensibility features come in several flavors: a C-like preprocessor, the ability to add new infix operators by associating them with prefix-style functions, and a mechanism for extending the grammar with new rules or new nonterminals.

9 Conclusion

Why are extensible languages useful? This question should now be easy to answer, in light of what we have already presented. Far more interesting is to consider why the pioneers of the idea thought extensible languages were useful. Consider Cheatham's [25] explanation which relies on problem domains. A programming language may be employed by different programmers to solve scientific, data manipulation, and system's programming tasks [25]. These different areas have their own idioms, units of data and programming styles. In scientific software, we deal with vectors and matrices, with operators often written in infix; in system's programming, we often work at the bit level, and require low-level operators interfacing more directly with the hardware. We can, from one perspective anyway, consider an extensible language as allowing us to express our required operations and notations as language primitives. Of course, there are alternatives: we may instead decide to create new languages to facilitate working in the various areas. Cheatham

and others (cited in [23]) scoff at this idea, though, and allude to an extensible, super-language which would alleviate the use of all other languages. We now know that this lofty goal was fated from the start, as various programming paradigms now flourish as distinct programming languages. Perhaps, then, we should think of extensibility as a way to add expressiveness to a base language, not as a way for mutating a language into something it is not. Nothing we have presented above makes it clear how we could easily adapt an existing language to deal with entirely new problem domains. Instead, we can start with a language that "mostly" suits our goals, and then (via the powerful extensibility mechanisms we've seen) fill in the gaps to add the missing pieces.

A Type Ascription in Camlp4

Type inferencing algorithms, such as the one employed by Ocaml, conveniently allow programmers to leave out the types of terms and automatically reconstruct necessary type information. Sometimes, though, it would be convenient to explicitly include this information in source text [21]. For example, it can serve as in-code documentation, clarifying the intended type of a specified term. When subtyping is present, it can also instruct the compiler to treat a term as a given (super) type of the most general type that was inferred. Since Ocaml allows type annotations to be associated with lambda-bound terms, the macro expansion technique for type ascriptions in general is very simple. The idea is to take syntax of the form `t as T`, and rewrite it as `(%x:T . x) t`. If the type ascription is incorrect, a compile-time error will result; if it is correct, then one beta-reduction removes it and evaluation continues. The Camlp4 code realizing this idea follows.

```
open Pcaml;;

let unique =
  let n = ref 0 in
  fun () -> incr n; "__pa_ascribe" ^ string_of_int !n

EXTEND
  expr: LEVEL "expr1"
  [[e1 = expr; "as"; e2 = ctyp ->
   let newName = unique () in
    <:expr< (fun ($lid:newName$:$e2$) ->
     $lid:newName$) $e1$ >>]]
     END
```

References

[1] Jonthan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 31–42, New York, NY, USA, 2001. ACM Press.

[2] Jonthan Bachrach and Keith Playford. Java syntactic extender. `http://jse.sourceforge.net`, 2003.

[3] Luca Cardelli, Florian Matthes, and Martin Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Systems Research Center, 1994.

[4] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, New York, NY, USA, 1995. ACM Press.

[5] William Clinger and Jonathan Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162, New York, NY, USA, 1991. ACM Press.

[6] J. R. Cordy. TXL – a language for programming language tools and applications. In ACM 4th International Workshop on LTDA, volume 110 of Electronic Notes in Theoretical Computer Science, pages 3–31. SpringerVerlag, Dec. 2004.

[7] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: a rapid prototyping system for programming language dialects. *Comput. Lang.*, 16(1):97–107, 1991.

[8] Daniel de Rauglaudre. Camlp4 manual. `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`, 2003.

[9] Daniel de Rauglaudre. Camlp4 tutorial. `http://caml.inria.fr/pub/docs/tutorial-camlp4/index.html`, 2003.

[10] R. Kent Dybvig. Writing hygienic macros in Scheme with syntax-case. Technical Report TR 356, Indiana University, 1992.

[11] R. Kent Dybvig. *The Scheme Programming Language, Third Edition*. The MIT Press, 2003.

[12] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp Symbolic Computation*, 5(4):295–326, 1992.

[13] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170, 2002.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[15] Peter S. Housel. An introduction to macro expansion algorithms. `http://www.cs.indiana.edu/pub/scheme-repository/doc/misc/macros-02.txt`, 1993.

[16] Free Software Foundation Inc. The C preprocessor. `http://gcc.gnu.org/onlinedocs/cpp/`, 2005.

[17] Martin Jambon. How to customize the syntax of OCaml, using Camlp4. `http://martin.jambon.free.fr/extend-ocaml-syntax.html`, 2005.

[18] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM Press.

[19] Thomas Mertes. Personal Communication, 2007.

[20] Thomas Mertes. Seed7 homepage. `http://seed7.sourceforge.net`, 2007.

[21] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[22] John Skaller. Felix homepage. `http://felix.sourceforge.net`, 2004.

[23] Thomas A. Standish. Extensibility in programming language design. *SIGPLAN Notices*, 10(7):18–21, 1975.

[24] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for Java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133, London, UK, 2000. Springer-Verlag.

[25] Jr. Thomas E. Cheatham. Motivation for extensible languages. *SIGPLAN Notices*, 4(8):45–49, 1969.