Adapting Scheme-Like Macros to a C-Like Language

Kevin Atkinson Matthew Flatt
University of Utah, School of Computing
{kevina,mflatt}@cs.utah.edu

Abstract

ZL is a C++-compatible language in which high-level constructs, such as classes, are defined using macros over a C-like core. ZL's parser and macro expander are similar to that of Scheme. Unlike Scheme, however, ZL must deal with C's richer syntax. Specifically, support for context-sensitive parsing and multiple syntactic categories (expressions, statements, types, etc.) leads to novel strategies for parsing and macro expansion. In this paper we describe ZL's approach to parsing and macros.

1. Introduction

C is a simple language that gives the user considerable control over how source code maps to machine code. For example, structures are guaranteed to be laid out in a particular way, dynamic memory is not allocated unless it is asked for, all function calls are explicit, and the only functions created are the ones that are explicitly defined. For this reason, most low-level system code is written in C. Nevertheless programmers want to use high-level language constructs for various reasons, and such use generally relinquishes low-level control. For example, C++ does not guarantee a particular layout of objects. This lack of control can cause a number of problems, including compatibility problems between different releases of software and between different compilers.

A programmer can regain control over higher-level language feature implementation through an extensible compiler. To provide such extensibility, macros for C are an ideal choice. Macros for C let a programmer build higher-level constructs from a small core, rather than forcing a programmer to accept a built-in implementation. Moreover, since macros elevate language extensions to the level of a library, individual advanced language features are only active when loaded.

A simple macro system, such as the C preprocessor, is not adequate for this purpose, nor is any macro system that acts simply as a preprocessor. Rather, the macro system must be an integral part of the language that can do more than rearrange syntax. In addition, the C base language must be extended to support the necessary primitives for implementing higher-level features. ZL, our new C and C++ compatible systems programming language in development, addresses both of these needs.

One target application of ZL explored in an earlier paper [4] is to help provide stability under software evolution. In particular, a programmer can use the ZL macro system to mitigate the problems of fragile ABIs (Application Binary Interfaces) due to software changes and incompatible ABIs due to compiler changes. This control is possible because ZL allows the programmer to control how an API (Application Programmer Interface) maps to an ABI.

This paper focuses on the ZL parser and macro expander. For relatively simple language extensions, ZL supports pattern-based macros similar to Scheme's `syntax-rules` [26]. In addition, ZL supports parser extensions that change the tokenization (roughly) of the input source, so that macro uses need not have the restricted form that Scheme's macro system imposes. Even with such extensions, pattern-based macros are limited. Therefore, in the same way that Scheme provides procedural macros via `syntax-case` [14], ZL supports procedural macros. ZL's API for procedural macros includes support for reflective tasks such as partial expansion, getting the value of a macro parameter, or determining whether a symbol is currently defined.

Our contribution in this paper is to demonstrate the adaptation of Scheme-style, hygienic macros to C-style syntax. Dealing with C's idiosyncratic syntax introduces complexities that are not solved by simply converting the original text into an S-expression intermediate format. Instead, parsing of raw text must be interleaved with the expansion process, and hygiene rules must be adapted carefully to actions such as accessing structure members.

The rest of this paper is organized as follows. Section 2 gives an overview of ZL parser and expander. Section 3 introduces macros and give details of the ZL parser. Sections 4 and 5 go into more detail on the macro transformer and give several example macros. Section 6 provides details of the macro implementation, and Section 7 discusses the status of our ZL implementation.

2. Parsing and Expanding

To deal with C's idiosyncratic syntax while also allowing the syntax to be extensible, ZL does not parse a program in a single pass. Instead, it uses an iterative-deepening approach to parsing. The program is first separated into a list of partly parsed declarations by a Packrat [18, 19] parser that effectively groups tokens at the level of declarations, statements, grouping curly braces, and parentheses. Each declaration is then parsed. As it is being parsed and macros are expanded, sub-parts, such as code between grouping characters, are further separated.

ZL's iterative-deepening strategy is needed because ZL does not initially know how to parse any part of the syntax involved with a macro. When ZL encounters something that looks like a function call, such as `f(x + 2, y)`, it does not know if it is a true function call or a macro use. If it is a macro use, the arguments could be expressions, statements, or arbitrary syntax fragments, depending on the context in which they appear in the expansion. Similarly, ZL cannot directly parse the body of a macro declaration, as it does not know the context in which the macro will ultimately be used.

More precisely, the ZL parsing process involves three intertwined phases. In the first phase *raw text*, such as `(x+2)`, is parsed. Raw text is converted into an intermediate form known as a *syntax object*, but which can still have raw-text components. (Throughout this paper we show syntax objects as S-expressions, such as `("()" "x+2")`.) In the second phase, the syntax object is expanded as necessary and transformed into other syntax objects by expanding macros until a fixed point is reached. In the third phase, the fully expanded syntax object is compiled into an *AST*.

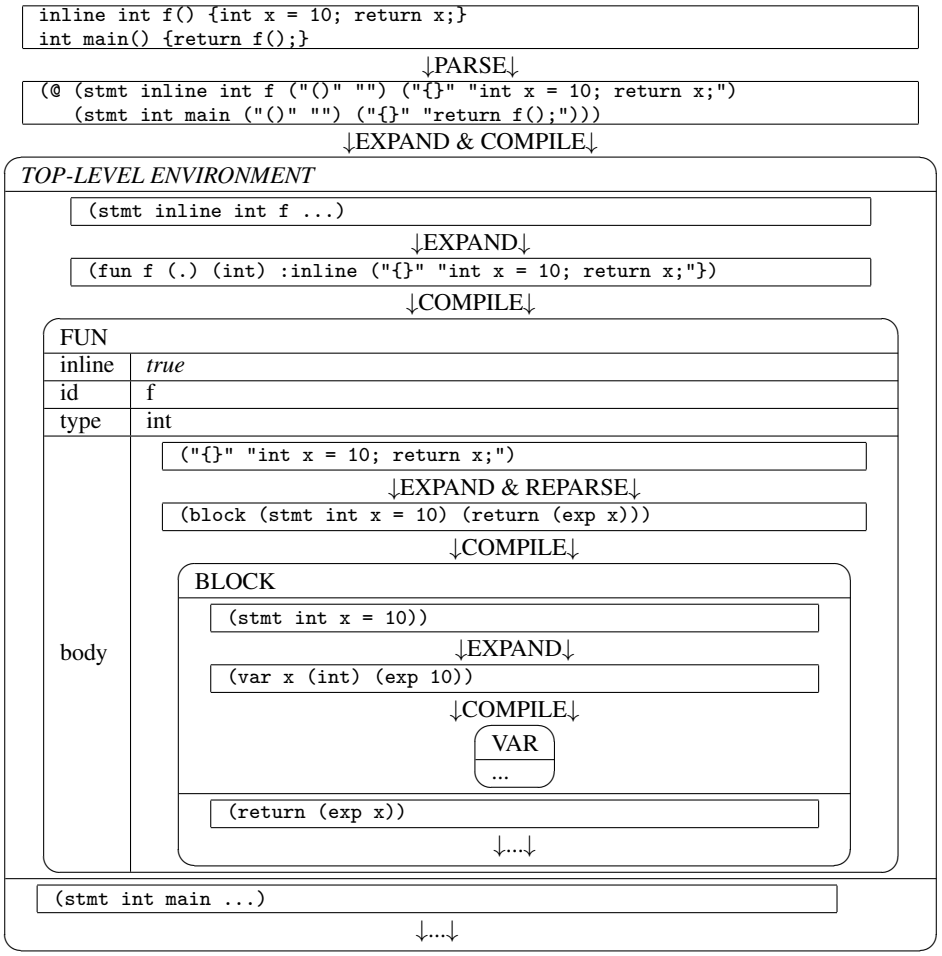

Figure 1. How ZL compiles a simple program. The body of `f` is reparsed and expanded as it is being compiled.

Figure 1 demonstrates ZL's parsing and expansion process. The top box contains a simple program as raw text, which is first parsed. The result is a *syntax list* (internally represented as a `@`) of `stmt`'s where each `stmt` is essentially a list of tokens, as shown in the second box. Each statement is then expanded and compiled in turn, and is added to the top-level environment (which can be thought of as an AST node). The third box in the figure shows how this is done, which requires recursive parsing and expansion. The first `stmt` is compiled into the `fun f`, while the body of the function is left unparsed. Next, `fun` is compiled into an AST (shown as a rounded rectangle). During the compilation, the body is expanded. Since it is raw text, this process involves parsing it further, which results in a `block`. Parsing the block involves expanding and compiling the sub-parts. Eventually, all of the sub-parts are expanded and compiled, and the fully parsed AST is added to the top-level environment. This process is repeated for the function `main`, after which the program is fully compiled.

As the figure illustrates, different parsing and expansion processes are interleaved in the expansion of a program, and they are all interrelated. A programmer who is extending ZL with new syntactic forms, however, need not imagine it all at once. ZL supports simple pattern-based macros and Parsing Expression Grammar (PEG) [19] based lexical extensions for simple cases. For more sophisticated extensions, ZL supports procedural macros. Finally, for the most sophisticated extensions such as implementing a class system, programmers must understand the details of ZL's model of lexical scope and reparsing. We take each of these layers in turn in the following sections.

3. Pattern-Based Macros and Lexical Extensions

A *pattern-based macro* is the most familiar kind of syntactic extension: it is simply a transformation of one piece of syntax to another. Since the expander is a fixed-point algorithm, macros can expand to other macros.

The simplest macros in ZL define new syntactic forms that take the shape of function calls. For example, consider an `or` macro that behaves like C's `||` operator, but instead of returning true or false, returns the first non-zero value. Thus, `or(0.0, 6.8)` returns `6.8`. To define it, one uses ZL's `macro` form, which declares a pattern-based macro:

```
macro or(x, y) { ({typeof(x) t = x; t ? t : y;}); }
```

In ZL, as in GCC, the `({...})` is a statement expression whose value is the result of the last expression, and `typeof(x)` gets the type of a variable. Like Scheme macros [14], ZL macros are hygienic, which means that they respect lexical scope. For example,

the `t` used in `or(0.0, t)` and the `t` introduced by the `or` macro remain separate, even though they have the same symbol name.

The `or` macro above has two *positional* parameters. Macros can also have *keyword* parameters and *default values*. For example:

```
macro sort(list, :compar = strcmp) {...}
```

defines the macro `sort`, which takes the keyword argument `compar`, with a default value of `strcmp`. A call to sort will look something like `sort(list, :compar = mycmp)`.

As another example, consider a macro that iterates over an STL-like container. Assuming an object `con` that contains integers, a use of `foreach` to print its contents is:

```
foreach(x, con, {printf("%d\n", x);});
```

The macro defining foreach is:

```
macro foreach (VAR, WHAT, BODY) {
  typeof(WHAT) & what = WHAT;
  typeof(what.begin()) i=what.begin(), e=what.end();
  for (; i != e; ++i) {
    typeof(*i) & VAR = *i;
    BODY;
  }
}
```

Notice how the `foreach` macro creates a new binding (`x` in the above example) that is visible in `BODY`.

3.1 Extending the Parser

The syntax of the `foreach` macro above is a bit ugly. It would be nice if we could instead write something like:

```
foreach (x in con) printf("%d\n", x);
```

which does not have the shape of a function call. ZL lets us do this by modifying[1] the `STMT` production in the grammar for the parser (from raw text to syntax objects) to recognize the new form:

```
<foreach> "foreach" "(" {ID} "in" {EXP} ")" {STMT}
```

In this grammar, anything between `{}` becomes a sub-part of the syntax object that is named between the `<>`. We must pair this modification with a macro for the new syntax form. The definition of the new `foreach` macro is identical to the function-call one except that `smacro` is used instead to declare that the macro works with a syntax object produced by the parser.

Support for both styles of macros (function call and syntax) is important, because not every macro warrants support in the parser. For example, since the `or` macro from Section 3 has limited usefulness, it probably does not warrant adding a new operator. Furthermore, function-call macros are typically sufficient in generating boilerplate code. In contrast, new general-purpose forms typically merit a parser extension.

3.2 The Parser

The ZL grammar is specified through a PEG [19], but with a few extensions to the usual PEG notation, and a Packrat [18] parser is used to convert strings of characters to syntax objects. A simplified version of ZL's initial grammar is shown in Figure 2. For readers not familiar with PEGs, the two most important things to note are that PEGs work with characters rather than tokens, and the `/` operator defines a prioritized choice. A prioritized choice is similar to the `|` operator used in Backus-Naur Form, except that it *unconditionally* uses the first successful match. For example, given the rule "`A = 'a' / 'ab'`" the string `ab` will never match because the first choice is always taken. The PEG specification more closely

[1] More modular lexical extensions that do not requiring modifying the full grammar is future work. (See Section 9.)

```
TOP = <top> SPACING {STMT}+;

STMT = <<mid PARM>> {MID} ";"
     / <if> "if" "(" {EXP} ")" {STMT} ("else" {STMT})?
     / <while> "while" "(" {EXP} ")" {STMT}
     / <break> "break" ";"
     / <return> "return" {EXP} ";"
     / {BLOCK}
     # other statements ...
     / <stmt> ({TOKEN_}+ {PAREN} {BRACE} / {TOKEN}+ ";");

EXP = <exp> {TOKEN}+;

BLOCK = <block> "{" {STMT}* "}";

TOKEN_ =  <<mid PARM>> {MID} / {BRACK} / {CONST} /
          {ID} / {SYM};
TOKEN = TOKEN_ / PAREN;

PAREN = <()> "(" {RAW_TOKEN*} ")";
BRACE = <{}> "{" {RAW_TOKEN*} "}";
BRACK = <[]> "[" {RAW_TOKEN*} "]";

CONST = <f> ...  / <l> ... / # float, numeric literal
        <s> ... / <c> ...    # string, character

ID = <<mid>> {MID} / {[@$\a_][\a_\d]*} SPACING;

SYM = {'...' / '==' / '+' / ...} SPACING;

RAW_TOKEN = STRING / CHAR / SYM / BRACE / PAREN /
            BRACK / COMMENT / [^\)\]\}];

STRING = '"' ('\\'_/[^"])+ '"' SPACING;
CHAR   = '\'' ('\\'_/[^'])+ '\'' SPACING;

SPACING = [\s]* COMMENT?;

COMMENT = ...;
```

Figure 2. Simplified PEG grammar.

resembles regular expression syntax (as used in `grep`) than it does Backus-Naur Form. The `()`, `[]`, `?`, `*`, `+`, and `_` (otherwise known as `.`) operators are all used in the same manner as they are in regular expressions. Anything between single quotes is a literal string. The double quote is like the single quote, except that special rules make them behave similarly to tokens. For example, `"for"` will match the `for` in `for(`, but it won't match the prefix of `foreach`. The `{}` and `<>` are extensions to the standard PEG syntax and are used for constructing syntax objects in the obvious ways. The special `<<mid>>` operator and `MID` production are explained later in Section 6.2.

A nice property of PEGs is that the associated Packrat parser uses memoization to guarantee linear performance [19]. This memoization is also used to avoid quadratic parsing times with ZL's frequent reparsing of strings. For example, when parsing `(x*(y+z))` as `("()" "x*(y+z)")`, the `PAREN` production is used on `(y+z)`, since ZL must recognize the grouping. When `("()" "x*(y+z)")` is expanded, the same `PAREN` production is used. Therefore, by keeping the memoization table for the `PAREN` production after the initial parse, there will be no need to reparse `(y+z)`.

3.3 Built-in Macros

The grammar serves to separate individual statements and declarations, and to recognize forms that are convenient to recognize using a Packrat parser. As such, it creates syntax objects that need additional processing before they can be compiled into an AST. The

```
1 Syntax * or(Syntax * p, Environ *) {
2   Match * m = match(NULL, syntax (_, x, y), p);
3   return replace(syntax
4                    {({typeof(x) t = x; t ? t : y;});},
5                  m, new_mark());
6 }
7 make_macro or;
```

Figure 3. Procedural macro version of the `or` macro from Section 3.

```
1  Syntax * foreach (Syntax * syn, Environ * env) {
2    Mark * mark = new_mark();
3    Match * m = match_args(0, syntax(VAR,WHAT,BODY), syn);
4    Syntax * what = match_var(m, syntax WHAT);
5    if (!symbol_exists(syntax begin, what, mark, env) ||
6        !symbol_exists(syntax end, what, mark, env))
7      return error(what,
8                   "Container lacks begin or end method.");
9    UnmarkedSyntax * repl = syntax {
10     typeof(WHAT) & what = WHAT;
11     typeof(what.begin()) i=what.begin(), e=what.end();
12     for (;i != e; ++i) {typeof(*i) & VAR = *i; BODY;}
13   };
14   return replace(repl, m, mark);
15 }
16 make_syntax_macro foreach;
```

Figure 4. Version of `foreach` that returns a helpful error message if the container does not contain the `begin` or `end` methods.

expander has several built-in macros for this purpose: `stmt`, `exp`, `()`, `[]`, and `{}`.

The `stmt` macro recognizes declarations and expressions. It first tries the declarations expander, which is a handwritten parser designed to deal with C's idiosyncratic syntax for declarations. If the declarations expander fails, then the expression expander is tried, which is an operator-precedence parser [17]. The `exp` macro is like the `stmt` macro, but only the expression expander is tried.

The macros `()`, `[]`, and `{}` are used for reparsing strings. The `()` and `[]` macros reparse the string as an expression using the `EXP` production in the grammar, where as the `{}` generally reparses the string as a block using the `BLOCK` production.

4. Procedural Macros

So far, we have shown only pattern-based macros that simply rearrange syntax. Some macros, however, must take action based on the input. Examples include providing better error messages (as in Figure 4) or performing some sort of action based on the value of one of the parameters (as will be used in the example of Section 5). For these situations, ZL provides *procedural macros*, which are functions that take a syntax object and an environment and return a transformed syntax object.

Figure 3 demonstrates the essential parts of any procedural macro. The macro is defined as a function that takes a syntax object and environment, and returns a transformed syntax object. Syntax is created using the `syntax` form. The `match` function is used to decompose the input while the `replace` function is used to rebuild the output. Finally, `make_macro` is used to create a macro from a function. More interesting macros use additional API functions to take action based on the input; for example, the `foreach` macro in Figure 4 uses `error` (line 7) to return an error if the container lacks the proper methods. Figures 5 and 6 define the key parts of the macro API, which we describe in the rest of this section and the next section.

Types: `UnmarkedSyntax`, `Syntax`, `Match`, and `Mark`

Syntax forms:

```
new_mark() — returns Mark *
syntax (...)|{...}|ID — returns UnmarkedSyntax *
raw_syntax (...) — returns UnmarkedSyntax *
make_syntax_macro ID [ID];
make_macro ID [ID];
```

Callback functions:

```
Match * match(Match * prev,
              UnmarkedSyntax * pattern, Syntax * with)
Match * match_args(Match * prev,
              UnmarkedSyntax * pattern, Syntax * with)
Syntax * match_var(Match *, UnmarkedSyntax * var);
Syntax * replace(UnmarkedSyntax *, Match *, Mark *)
```

Figure 5. Core Macro API (described in Section 4.1).

4.1 Core API

Syntax is created using the `syntax` and `raw_syntax` forms. The different forms create different types of code fragments. In most cases, the `syntax {...}` form can be used, such as when a code fragment is part of the resulting expansion; the braces will not be in the resulting syntax. If an explicit list is needed, for example, when passed to `match` as in Figure 3 (line 2), then the `syntax (...)` form should be used (in which the commas are part of the syntax used to create the list). Neither of these forms create syntax directly, however; for example, `syntax {x + y;}` is first parsed as `("{}" "x + y;")` before eventually becoming `(plus x y)`. When it is necessary to create syntax directly, the `syntax ID` form can be used for simple identifiers. For more complicated fragments the `raw_syntax` form can be used in which the syntax is given in S-expression form.

The `match` function decomposes the input. It matches pattern variables (the second parameter) with the arguments of the macro (the third parameter). If it is successful, it prepends the results to `prev` (the first parameter) and returns the new list. If `prev` is `NULL`, then it is treated as an empty list. In the match pattern a `_` can be used to mean "don't care." The match is done from the first part of the syntax object. That is, given `(plus x y)`, the first match is `plus`. Since the first part is generally not relevant, ZL provides `match_args`, which is like `match` except that the first part is ignored. For example, `match_args` could have been used instead of `match` in Figure 3.

The `replace` function is used to rebuild the output. It takes a syntax object (the first parameter, and generally created with `syntax`), replaces the pattern variables inside it with the values stored in the `Match` object (the second parameter), and returns a new `Syntax` object.

The final argument to `replace` is the *mark*, which is used to implement hygiene. A mark captures the lexical context at the point where it is created. Syntax objects created with `syntax` do not have any lexical information associated with them, and are thus *unmarked* (represented with the type `UnmarkedSyntax`). It is therefore necessary for `replace` to attach lexical information to the syntax object by using the mark created with the `new_mark` primitive (the third parameter to `replace`).

Match variables exist only inside the `Match` object. When it is necessary to access them directly, as done in lines 5 and 6 of Figure 4, `match_var` can be used; it returns the variable as a `Syntax` object, or `NULL` if the match variable does not exist.

Once the function for a procedural macro is defined, it is necessary to declare it as a macro using one of `make_macro`

```
bool symbol_exists(UnmarkedSyntax * symbol,
                   Syntax * where, Mark *, Environ *)
Syntax * error(Syntax *, const char *, ...)
size_t ct_value(Syntax *, Environ *)
Syntax * partly_expand(Syntax *, Position pos,
                       Environ *)
Context * get_context(Syntax *)
Syntax * replace_context(UnmarkedSyntax *, Context *)
UnmarkedSyntax * string_to_syntax(const char * str)
Environ * temp_environ(Environ *)
Syntax * pre_parse(Syntax *, Environ *)
```

Figure 6. Additional Macro API Functions (described in Sections 4.2–4.4).

or `make_syntax_macro`. The first creates a function-call macro, while the second creates a syntax macro.

4.2 Beyond Match and Replace

The `foreach` macro in Figure 4 uses `symbol_exists` (lines 5 and 6) to check if the `begin` and `end` symbols exist in the container type. The symbol name is passed in as the first argument and the container as the second. If the second argument is `NULL` then the current environment will be checked instead. The third argument provides the context in which to look up the current symbol, and finally the last argument is the environment to use.

If the symbol is not found, the `foreach` macro uses `error` (line 7) to return an error. The `error` function creates a syntax object that will result in a syntax error when it is parsed. The first argument is used to determine the location where the error will be reported; the location associated with this syntax object is used as the location of the error.

The expansion of `foreach` depends on whether a symbol is present in its input. Often, expansion depends instead on the compile-time value of an expression in the input. The `ct_value` function takes a syntax object, expands the expression, parses the expansion, and evaluates the parsed expression as an integer to determine its value.

Also, the `foreach` macro did not need to decompose the syntax for the container passed in. If decomposition were necessary, the syntax object (representing the container) would need to be expanded first because, at the point the macro was called, the container is likely still represented as a generic `exp`, which is just a list of tokens. For example, if the container were the identifier `c`, the syntax object for the container would be `(exp c)` instead of `(id c)`. The `partly_expand` function partly expands a syntax object to allow further decomposition. The `pos` parameter tells ZL what position the syntax object is in; common values are `TopLevel` for declarations, `StmtPos` for statements, and `ExpPos` for expressions.

4.3 Controlling Visibility

Often it is necessary to bend normal hygiene rules. For this ZL provides two different mechanisms: the `get_context` and `replace_context` macro API functions and the `fluid_binding` primitive.

Replacing Context. The `get_context` and `replace_context` functions are used to bend hygiene in a very similar fashion to `datum->syntax-object` in the `syntax-case` expander [13]. For example, a macro defining a class needs to create a vtable that is accessible outside of the macro creating the class. The `get_context` function gets the context from some symbol, generally some part of the syntax object passed in, while `replace_context` replaces the context of the symbol with the one provided. For example, code to create a symbol `_vtable` that can be used later might look something like:

```
...
Match * m = match_args(0, raw_syntax (name ...), p);
Syntax * name = match_var(m, syntax name);
Context * context = get_context(name);
Syntax * _vtable = replace_context(syntax _vtable,
                                   context);
...
```

Here `name` is the name of the class that is passed in as `m`. The `name` symbol is extracted into a syntax object so that it can be used for `get_context`. The `replace_context` function is then used to put the symbol `_vtable` in the same context as `name`. Now `_vtable` will have the same visibility as the `name` symbol, and thus be visible outside the macro.

Fluid Binding. The `get_context` and `replace_context` functions are one way to bend hygiene. The other is to use `fluid_binding`, which allows a variable to take its meaning from the use site of a macro rather than the macros's definition site, in a similar fashion to `define-syntax-parameter` in Racket [16, 7].

A prime example of the need for `fluid_binding` is the special variable `this` in classes. Variables in ZL are lexically scoped. For example, the code:

```
int g(X *);
int f() {return g(this);}
int main() {X * this = ...; return f();}
```

will not compile because the `this` defined in `main` is not visible in `f`, even though `f` is called inside `main`. Normal hygiene rules preserve lexical scope in a similar fashion, such that:

```
int g(X *);
macro m() {g(this);}
int main() {X * this = ...; return m();}
```

will also not compile. Attempts to make this work with `get_` and `replace_context` will not compose well [7]. What is really needed is for `this` to be scoped based on where it is used when expanded, rather than where it is written in the macro definition. This can be done by marking the `this` symbol as `fluid` using `fluid_binding` at the top level and then using `fluid` when defining the symbol in local scope. For example:

```
fluid_binding this;
int g(X *);
macro m() {g(this);}
int main() {X * fluid this = ...; return m();}
```

will work as expected. That is, the `this` in `m` will bind to the `this` in `main`.

4.4 Other API Functions

When it is necessary to create syntax on the fly the `string_to_syntax` function can be used. Its usage and the usage of the `temp_environ` and `pre_parse` functions are demonstrated in the example presented in the next section.

5. An Extended Example

To get a better idea of how procedural macros work, this section gives the code of a macro that fixes the size of a class. Fixing the size of a class is useful because changing the size often breaks binary compatibility, which forces code using that class to be recompiled. Additional examples of how ZL can be used to mitigate the problem of binary compatibility are given in our previous work [4].

The macro to fix the size of the class is shown in Figure 7. To support this macro the grammar has been enhanced to support fixing the size. The syntax for the new class form is:

```
1  Syntax * parse_myclass(Syntax * p, Environ * env) {
2    Mark * mark = new_mark();
3    Match * m = match_args
4        (0, raw_syntax(name @ (pattern ({...} @body))
5                       :(fix_size fix_size) @rest), p);
6    Syntax * body = match_var(m, syntax body);
7    Syntax * fix_size_s = match_var(m, syntax fix_size);
8
9    if (!body || !fix_size_s) return parse_class(p, env);
10
11   size_t fix_size = ct_value(fix_size_s, env);
12
13   m = match(m, syntax dummy_decl,
14             replace(syntax {char dummy;}, NULL, mark));
15   Syntax * tmp_class = replace(raw_syntax
16       (class name ({...} @body dummy_decl) @rest),
17                                m, mark);
18   Environ * lenv = temp_environ(env);
19   pre_parse(tmp_class, lenv);
20   size_t size = ct_value
21       (replace(syntax(offsetof(name, dummy)), m, mark),
22        lenv);
23
24   if (size == fix_size)
25     return replace(raw_syntax
26                      (class name ({...} @body) @rest),
27                    m, mark);
28   else if (size < fix_size) {
29     char buf[32];
30     snprintf(buf, 32, "{char d[%u];}", fix_size - size);
31     m = match(m, syntax buf,
32             replace(string_to_syntax(buf), NULL, mark));
33     return replace(raw_syntax
34                      (class name ({...} @body buf) @rest),
35                    m, mark);
36   } else
37     return error(p,"Size of class larger than fix_size");
38 }
39 make_syntax_macro class parse_myclass;
```

Figure 7. Macro to fix the size of a class. All ... in this figure are literal.

```
class C : fix_size(20) { ... };
```

which will allow a macro to fix the size of the class `C` to 20 bytes. The enhancement involved modifying the `CLASS` production to support the `fix_size` construct. A simplified version of the new production is as follows:

```
CLASS = <class> "class" {ID/}
            (:<fix_size> ":" "fix_size" "(" {EXP} ")" )
            {<{...}> "{" {STMT}* "}" }?;
```

Most of the syntax is already described in Section 3.2. The only new thing is `:<>`, which constructs a property to be added to the parent syntax object, which in this case is `class`. The `{...}` (in which the ... are literal) is the name of the syntax object for the class body.

The macro in Figure 7 redefines the built-in `class` macro. It works by parsing the class declaration and taking its size. If the size is smaller than the required size, an array of characters is added to the end of the class to make it the required size.

The details are as follows. Lines 2–7 decompose the class syntax object to extract the relevant parts of the class declaration. A `@` by itself in a pattern makes the parts afterward optional. The `pattern` form matches the sub-parts of a syntax object; the first part of the object (the `{...}` in this case) is a literal[2] to match against, and the other parts of the object are pattern variables. A `@` followed by an identifier matches any remaining parameters and stores them in a syntax list; thus, `body` contains a list of the declarations for the class. Finally, `:(fix_size fix_size)` matches an optional keyword argument; the first `fix_size` is the keyword to match, and the second `fix_size` is a pattern variable to hold the matched argument.

If the class does not have a body (i.e., a forward declaration) or a declared `fix_size`, then the class is passed on to the original class macro in line 9. Line 11 compiles the `fix_size` syntax object to get an integer value.

Lines 13–22 involve finding the original size of the class. Due to alignment issues the `sizeof` operator cannot be used, since a class such as "`class D {int x; char c;}`" has a packed size of 5 on most 32 bit architectures, but `sizeof(D)` will return 8. Thus, to get the packed size a dummy member is added to the class. For example, the class `D` will become "`class D {int x; char c; char dummy;}`" and then the offset of the dummy member with respect to the class `D` is taken. This new class is created in lines 13–17. Here, the `@` before the identifier in the replacement template splices in the values of the syntax list.

To take the offset of the dummy member of the temporary class, it is necessary to parse the class and get it into an environment. However, we do not want to affect the outside environment with the temporary class. Thus, a new temporary environment is created in line 18 using the `temp_environ` macro API function. Line 19 then parses the new class and adds it to the temporary environment. The `pre_parse` API function partly expands the passed-in syntax object and then parses just enough of the result to get basic information about symbols.

With the temporary class now parsed, lines 20–22 get the size of the class using the `offsetof` primitive.

Lines 24–37 then act based on the size of the class. If the size is the same as the desired size, there is nothing to do and the class is reconstructed without the `fix_size` property (lines 24–27). If the class size is smaller than the desired size, then the class is reconstructed with an array of characters at the end to get the desired size (lines 28–35). (The `string_to_syntax` API function simply converts a string to a syntax object.) Finally, an error is returned if the class size is larger than the desired size (lines 36–37).

The last line declares the function `parse_myclass` as a syntax macro for the `class` syntax form.

6. Macro Implementation

This section describes the implementation of ZL's macro system. We first describe the basic macro-expansion algorithm without the reparsing steps to focus on the hygiene system. For simplicity, we first assume that macro parameters and syntax forms are fully parsed. The next subsection explains the details.

6.1 Basic Expander and Hygiene System

ZL's hygiene system is similar to the `syntax-case` system [14]. However, the data structures are different. A mark holds a lexical environment, and marks are applied during `replace` rather than to the input and result of a macro transformer. Special lookup rules search mark environments in lieu of maintaining a list of substitutions.

The Idea. During parsing, ZL maintains an environment that maps from one type of symbol to another. Symbols in the environment's domain correspond to symbols in syntax objects, while each symbol in the environment's range is generated to represent a particular binding. Symbols in syntax objects (and hence the environment domain) have a set of marks associated with them. The

[2] ZL matches literals symbolically (i.e., not based on lexical context). Matching sensitive to lexical context is future work.

```
float r = 1.61803399;

Syntax * make_golden(Syntax * syn, Environ * env) {
  Mark * mark = new_mark();
  Match * m = match_args(0, syntax (A,B,ADJ,FIX), syn);
  UnmarkedSyntax * r = syntax {
    for (;;) { float a = A, b = B;
               float ADJ = (a - r*b)/(1 + r);
               if (fabs(ADJ/(a+b)) > 0.01) FIX;
               else break; }
  };
  return replace(r, m, mark);
}
make_macro make_golden;

int main() {
  float q = 3, r = 2;
  make_golden(q, r, a, {q -= a; r += a;});
}
```

Figure 8. Example code to illustrate how hygiene is maintained. The `make_golden` macro will test if `A` and `B` are within 1% of the golden ratio. If not, it will execute the code in `FIX` to try to fix the ratio (where the required adjustment will be stored in `ADJ`) and then try again until the golden ratio condition is satisfied.

set of marks are considered part of the symbol's identity. A mark is created with the `new_mark` primitive and applied to symbols during the replacement process. During this process, each symbol is either replaced, if it is a macro parameter, or marked. A mark also has an environment associated with it, which is the global environment at the site of the `new_mark` call.

When looking up a binding, the current environment is first checked. If a symbol with the same set of marks is not found in the current environment, then the outermost mark is stripped and the symbol is looked up in the environment associated with the stripped mark. This process continues until no more marks are left.

An Illustrative Example. To better understand this process, consider the code in Figure 8. When the first binding form "`float r = ...`" is parsed, `r` is bound to the unique symbol `$r0`, and the mapping `r => $r0` is added to the current environment. When the function `make_golden` is parsed, it is added to the environment. When the `new_mark()` primitive is parsed inside the body of the function, the current global environment is remembered. The `new_mark()` primitive does not capture local variables, since it makes little sense to use them in the result of the macro. Next, "`make_macro make_golden`" is parsed, which makes the function `make_golden` into a macro.

Now the body of `main` is parsed. A new local environment is created. When "`float q = 3, r = 2`" is parsed, two unique symbols `$q0` and `$r1` are created and corresponding mappings are added to the local environment. At this point, we have:

```
float $r0 = 1.61803399;
[make_golden => ..., r => $r0]
int main () {
  float $q0 = 3, $r1 = 2;
  [r => $r1, q => $q0, make_golden => ..., r => $r0]
  make_golden(q, r, a, {q -= a; r += a;});
}
```

The expanded output is represented in this section as pseudo-syntax that is like the input language of ZL with some additional annotations. Variables starting with `$` represent bound symbols. The `[...]` list represents the current environment in which new binding forms are added to the front of the list.

Now, `make_golden` is expanded and, in the body of `main`, we have:

```
...
[r => $r1, q => $q0, make_golden => ..., r => $r0]
for (;;) { float a'0 = q, b'0 = r;
           float a = (a'0 - r'0*b'0)/(1 + r'0);
           if (fabs(a/(a'0+b'0)) > 0.01)
             {q -= a; r += a;}
           else break; }
'0 => [r => $r0]
```

where `'0` represents a mark and `'0 => [...]` is the environment for the mark. Notice how marks keep the duplicate `a` and `r`'s in the expanded output distinct.

Now, the statement "`float a'0 = q, b'0 = r`" is compiled. Compiling the first part creates a unique symbol `$a0` and the mapping `a'0 => $a0` is added to the new environment inside the `for` loop. The variable `q` on the right-hand-side resolves to the `$q0` symbol in the local environment. A similar process is performed for the second part. We now have:

```
...
for (;;) { float $a0 = $q0, $b0 = $r1;
           [b'0 => $b0, a'0 => $a0, r => $r1,
            q => $q0, ...]
           float a = (a'0 - r'0*b'0)/(1 + r'0);
           ...}
'0 => [r => $r0]
```

Next, the statement "`float a = ...`" is compiled. A unique symbol `$a1` is created for `a` and the associated mapping is added to the local environment. Then the right-hand-side expression must be compiled. The variables `a'0` and `b'0` resolve to `$a0` and `$b0`, respectively, since they are found in the local environment. However, `r'0` is not found, so the mark `'0` is stripped, and `r` is looked up in the environment for the `'0` mark and resolves to `$r0`. We now have:

```
...
for (;;) { ...
           float $a1 = ($a0 - $r0*$b0)/(1 + $r0);
           [a => $a1, b'0 => $b0, a'0 => $a0,
            r => $r1, q => $q0, ...]
           if (fabs(a/(a'0+b'0)) > 0.01)
             {q -= a; r += a;}
           else break; }
'0 => [r => $r0]
```

Next, the `if` is compiled. The marks keep the two `a` variables in the expression `a/(a'0+b'0)` distinct, and everything correctly resolves. Thus, we finally have:

```
float $r0 = 1.61803399;
int main() {
  float $q0 = 3, $r1 = 2;
  for (;;) { float $a0 = $q0, $b0 = $r1;
             float $a1 = ($a0 - $r0*$b0)/(1 + $r0);
             if (fabs($a1/($a0+$b0)) > 0.01)
               {$q0 -= $a1; $r1 += $a1;}
             else break; }
}
```

Hence, all symbols are correctly bound and hygiene is maintained.

Multiple Marks. The symbols in the expansion of `make_golden` only had a single mark applied to them. However, in some cases, such as when macros expand to other macros, multiple marks are needed. For example, multiple marks are needed in the expansion

```
macro mk_plus_n (NAME, N) {
  macro NAME (X) { ({int x = X; x + N;}); }
}

static const int x = 10;
mk_plus_n(plus_10, x);

int main() {
  int x = 20;
  return plus_10(x);
}
```

Figure 9. Example code to show how hygiene is maintained when a macro expands to another macro.

of `plus_10` in Figure 9. In this figure, `mk_plus_n` expands to

```
macro plus_10 (X'0) { ({int x'0 = X'0; x'0 + x;}); }
```

where the first mark `'0` is applied. A second mark is then applied in the expansion of `plus_10(x)` in `main`:

```
{ ({int x'0'1 = x; x'0'1 + x'1;}) }
```

In particular, a second mark is added to `x'0`, making it `x'0'1`. This symbol then resolves to the `x` local to the macro `plus_10`. In addition, `x'1` resolves to the global `x` constant[3] and the unmarked `x` resolves to the `x` local to `main`. Thus, hygiene is maintained in spite of three different `x`'s in the expansion.

Structure Fields. Normal hygiene rules will not have the desired effect when accessing fields of a structure or class. Instead of trying to look up a symbol in the current environment, we are asking to look up a symbol within a specialized sub-environment.

For example, the following code won't work with normal hygiene rules:

```
macro sum(q) {q.x + q.y;}
struct S {int x; int y;}
int f() {
  struct S p;
  ...
  return sum(p);
}
```

The problem is that `sum(p)` will not be able to access the fields of `p` since it will expand to "`p.x'0 + p.y'0`" with marks on `x` and `y`. The solution is to use a special lookup rule for structure fields. The rule is that if the current symbol with its sets of marks is not found in the structure, strip the outermost mark and try again, and repeat the process until no more marks are left. This process is similar to the normal lookup rule except that the sub-environment associated with the mark is ignored since it is irrelevant. In the above example, `p.x'0` in the expansion of `sum(p)` will resolve to the structure field `x` in `struct S`.

Replacing Context. The `get_context` and `replace_context` functions (see Section 4.3) can be used to bend normal hygiene rules. A *context* is simply a collection of marks. Thus `get_context` simply gets the marks associated with the syntax object, while `replace_context` replaces the marks of a syntax object. If a syntax object already has any marks associated with it, they are ignored.

Fluid Binding. The `fluid_binding` form (see Section 4.3) bends hygiene by allowing a variable to take its meaning from the use site rather than from the macros's definition site. It changes the scope of a marked variable from lexical to *fluid* and is used together with the `fluid` keyword, which temporarily binds a new symbol to the fluid variable for the current scope.

The `fluid_binding` form inserts a *fluid-binding* symbol into the environment that serves as an instruction to perform the lookup again. The symbol consists of the instruction and a unique symbol name to perform the second lookup on; the name is constructed by taking the symbol name and applying a fresh mark to it (with an empty environment). For example, "`fluid_binding this`" inserts the mapping `this => fluid(this'0)` into the environment, where the fluid-binding symbol is represented as `fluid(SYMBOL'MARK)`. The "`fluid VAR`" form then replaces the variable `VAR` with the unique symbol name associated with the fluid binding. This has the effect of rebinding the `fluid_binding` variable to the current symbol for the current scope. For example, "`X * fluid this`" becomes "`X * this'0`" and `this'0` gets temporarily bound to the local symbol `$this0`. Finally, whenever a symbol resolves to something that is a fluid binding the symbol will be resolved again, this time using the unique symbol name in the fluid binding. For example, `this` will first resolve to `fluid(this'0)`, which then resolves to `$this0`.

To see why this method works, consider the parsing of the example from Section 4.3:

```
fluid_binding this;
int g(X *);
macro m() {g(this);}
int main() {X * fluid this = ...; return m();}
```

The `fluid_binding` form is first parsed and the mapping `this => fluid(this'0)` is added to the environment where `'0` is an empty mark. The declaration for `g` and the macro `m` is also parsed and we now have:

```
[m => ..., g => ..., this => fluid(this'0)]
int main() {X * fluid this = ...; return m();}
```

Now `main` is parsed. Because the `this` variable has the `fluid` keyword, the symbol `this` is looked up in the environment and "`fluid this`" becomes `this'0` giving:

```
int main() {X * this'0 = ...; return m();}
```

The `this'0` variable is then added to the environment and rest of the body of `main` is expanded (which includes the expansion of `m`):

```
int main() {
  [this'0 => $this0, ...]
  return g(this'1);
}
'1 => [..., this => fluid(this'0)]
```

The body of `main` is now parsed. The variable `this'1` (from the expansion of `m`) first resolves to the fluid symbol `fluid(this'0)`, which temporarily becomes `this'0` and then resolves to `$this0`. The rest of `main` is also parsed giving:

```
int main() {return g($this0);}
```

Hence, the `this` variable in the macro `m` gets resolved to the `this` variable in `main` as intended.

6.2 The Reparser

Supporting Scheme-style macros with C-like syntax turns out to be a hard problem for two reasons. The primary reason, as mentioned in Section 2, is that ZL does not initially know how to parse any part of the syntax involved with macros. The other and less obvious reason is that when given a syntax form such as "`syntax (x * y)`", ZL does not know if `x` and `y` are normal variables or pattern variables until the substitution is performed. If they are normal variables, then it will be parsed as `(exp x * y)`, but if they are pattern

[3] In pattern based macros there is an implicit call to `new_mark` at the point where the macro was defined; hence, the `'1` mark captures the environment where `mk_plus_10` (expanded from `mk_plus_n`) is defined, which includes the global constant `x`.

variables, it will be parsed as `(exp (mid x) * (mid y))` where `mid` (macro identifier) is just another name for a pattern variable. ZL solves the former problem by delaying parsing as much as possible, which works nicely with ZL's hygiene system by reducing the complexity of macro explanation from quadratic to linear. ZL solves the latter problem by installing special hooks into its Packrat parser.

The Idea. As already established, the `syntax ()` and `syntax {}` forms create syntax objects with raw text that cannot be parsed until ZL knows where the syntax object will ultimately be used. Thus `replace` is unable to perform any replacements. Instead, `replace` annotates the syntax object with with a set of instructions to apply later that includes two bits of information: (1) the mark to apply, and (2) the substitutions to apply.

For example, given the code:

```
int x;
Syntax * plus_x(Syntax * syn, Environ * env) {
  Match * m = match_args(0, syntax (y), syn);
  return replace(syntax (x + y), m, new_mark());
}
make_macro plus_x;
```

the call `plus_x(z)` results in `("()" "x + y"){'0; y => (parm "z")}` where the `{}` represents the annotation and `parm` is a built-in macro (see Section 3.3) to indicate the need to reparse. The first part of the annotation is the mark and the second is the substitution to apply. Thus the substitution is delayed until ZL knows where the call to `plus_x` will be used.

Eventually, the annotated syntax object will need to be parsed, which requires two steps. First the raw text needs to be parsed using the Packrat parser. Second the instructions in the annotations need to be applied.

Parsing the raw text creates a problem since ZL does not know which identifiers are pattern variables. Solving this problem involves a special hook into the Packrat parser, which is the purpose of the special `<<mid>>` operator shown in the grammar (Figure 2). The relevant bits of the grammar (with some extra required productions) are these:

```
EXP = <exp> {TOKEN}+;
TOKEN_ = <<mid PARM>> {MID} / {ID} / ...
MID = {[@$\a_][\a_\d]*} SPACING;
PARM = {STMT} EOF / {TOKEN} EOF / {EXP} EOF;
```

The `<<mid>>` operator is a special operator that matches only if the identifier being parsed is in the substitution list. When a `MID` matches, and the pattern variable is of the type that needs to be reparsed (i.e., matched with a `syntax` form), the parser adds a note as to how to reparse the macro parameter. This is either the production where it matches or the production as given in the `<<mid>>` instruction. For example, when parsing

```
("()" "x + y"){'0; y => (parm "z")}
```

as an expression, the parser is able to recognize `x` as an identifier and `y` as a `mid`. During the parsing of `x` the `MID` production is tried but it is rejected because `x` is not a pattern variable, yet when `y` is tried, it matches the `MID` production since `y` is a pattern variable. Thus the result of the parse is:

```
(exp x + (mid y PARM)){'0; y => (parm "z")}
```

After the raw text is parsed, the instructions in the annotation are applied to the sub-parts; if the syntax object represents raw text then the instructions are simply pushed down rather than being directly applied. In the above example this process will result in:

```
(exp'0 x'0 +'0 z)
```

That is, marks are applied and `(mid y PARM)` becomes `z`. During the substitution, the string `z` is reparsed using the `PARM` production noted in the second argument of `mid`. Hence, the string `z` becomes the identifier `z`.

The results of the reparse are then expanded and parsed as before. Marks are used as described in Section 6.1, but with the additional rule that if no marks are left and a symbol is still not found then it is assumed to be associated with a primitive form. For example, `exp'0` is assumed to represent the built in `exp` macro, since `exp` is not in the current environment. Since the result is an `exp`, it will be expanded again to become

```
(plus x'0 z)
```

which will then be converted into an AST.

Additional Examples. In the previous example, the result of the reparse is a fully parsed string, but this is not always the case. For example, if the macro `plus_x` were instead `plus_2x`, and the call `plus_2x(z)` expanded to:

```
("()" "2*x + y"){'0; y => (parm "z")}
```

the result will first parse to:

```
(exp ("()" "2*x") + y){'0; y => (parm "z")}
```

with `"2*x"` left unparsed. Applying the annotations will then result in:

```
(exp'0 ("()" "2*x"){'0; y => (parm "z")} + z)
```

That is, since the `"()"` syntax objects represents raw text, the instructions are pushed down on that object rather than being directly applied.

Also, in the same example, the macro parameter was just an identifier and the special `PARM` production is not needed, as it would be correctly parsed as a `TOKEN`. However, this is not always the case. For example, if the call to `plus_x` were instead `plus_x(z + 2)` the string "`z + 2`" would need to be parsed as a `PARM` since it is not a token.

Matching and Replacing with the `raw_syntax` Form. As the lazy substitutions of macro parameters and the reparsing are coupled, lazy substitution only applies to syntax forms that are to be reparsed, such as the `()` and `{}` forms. Syntax created with `raw_syntax` is fully parsed, and thus `replace` performs the substitutions eagerly.

7. Implementation Status

ZL is a C++-compatible language that provides a C-like core and leaves most of the higher level C++ features to be defined via the macro system. In particular, most of the class implementation in ZL is left to macros. However, since classes are an integral part of the C++ type system, ZL still needs to have some notion of what a class is. Thus, in addition to supporting basic C constructs and macros, ZL also provides *user types*, which are ZL's minimal notion of classes. The details of user types and ZL's implementation of classes are explored in our previous work [4].

The current ZL prototype supports most of C and parts of C++. For C, the only major feature not supported is bitfields, mainly because the need has not arisen. For C++, we support classes with single inheritance, but currently do not support multiple inheritance, exceptions, or templates.

As ZL is at present only a prototype compiler, the overall compile time when compared to compiling with GCC 4.4 is 2 to 3 times slower. However, ZL is designed to have little to no impact on the resulting code. ZL's macro system imposes no run-time overhead.

The ZL compiler transforms higher-level ZL into a low-level S-expression-like language that can best be described as C with

Scheme syntax. Syntactically, the output is very similar to fully expanded ZL as shown in Figure 1. The transformed code is then passed to a modified version of GCC 4.4. When pure C is passed in we are very careful to avoid any transformations that might affect performance. The class macro currently implements C++ classes in a way that is comparable to a traditional compiler's implementation, and hence should have no impact on performance.[4]

8. Related Work

ZL's design philosophy is closely related to Scheme's [26] design philosophy of providing a small core language and letting everything else be defined as macros. Parts of ZL were previously described in our earlier paper [4]. However, the focus of that paper was on how ZL can be used for to mitigate ABI compatibility problems. More details of ZL and the ABI compatibility problem is also the topic of the first author's dissertation [3].

Other Macro Systems. There are numerous other macro systems for various languages, but apart from Scheme, few have the goal of allowing a large part of the language to be defined via macros. As such, they are either a macro system built on top of an existing language, or they lack procedural macros for general compile-time programming.

Maya [6] is a powerful macro system for Java. Maya macros (known as Mayans) support lexical extensions by extending Java's LALR(1) grammar. Like ZL's macros, Mayans are procedural and hygienic. Unlike the current version of ZL, Mayans are modular; however, since they extend the LALR(1) grammar, conflicts may well arise when combining them. OpenJava [27] and ELIDE [11] are similar to Maya but less advanced. Neither of these systems support hygiene, and they do not support general syntax extensions.

A procedural and hygienic macro system based on the Earley [15] parser is described in Kolbly's dissertation [22]. His system is similar to Maya in that macro expansion is part of the parsing process, yet more powerful as the Earley parser can handle arbitrary grammars rather than just the LALR(1) subset. His macro system is also used in the RScheme [1] dialect of Scheme.

Fortress [2] is a new language with hygienic macro support and the ability to extend the syntax of the language. Like ZL, it uses a Packrat parser to support lexical extensions. In addition and unlike the current version of ZL, the lexical extensions are modular and thus can be combined. Fortress macros support recursive and mutually recursive definitions unlike some other macro systems. However, macros cannot expand to other macros, and they are not procedural.

The Dylan [25] language has support for hygienic macros. However, unlike ZL, one cannot really extend the grammar as macros are required to take one of three fixed forms: `def`, `stmt`, and `fun` call macros. The JSE system [5] is a version of Dylan macros adapted to Java.

MS^2 [29] is an older, more powerful macro system for C. It essentially is a Lisp `defmacro` system for C. It offers powerful macros since they are procedural, but like Lisp's `defmacro` lacks hygiene. In addition, like Dylan but unlike ZL, macros are required to take one of several fixed forms; no mechanism for general syntactic extensions is provided.

The `<bigwig>` [9] language support pattern-based macros and lexical extension. However, and unlike ZL, its macros are limited in power because recursion is explicitly forbidden. By limiting the power of the macro system `<bigwig>` can support type safety and termination of the macro-expansion process.

[4] We verified that run-time performance was the same by compiling several real world programs with both ZL and GCC 4.4. More details of this process are given in our previous work [4].

MacroML [20] has similar aims to `<bigwig>` in that it limits what macros can do to ensure safety. While MacroML supports recursion, it does not support lexical extensions. In addition, macros are not allowed to inspect or take apart code. However, these restrictions allow macros to be statically typed. This guarantees that macro definitions are well formed and thus always produce valid code.

Extensible Compilers. Macros are one approach to providing an extensible compiler, but a more traditional approach is to provide an API to directly manipulate the compiler's internals, such as the AST. On the surface this approach may seem more powerful than a macro system, but we believe a macro system can be equally powerful with the right hooks into the compiler.

Xoc [12] is an extensible compiler that supports grammar extensions by using GLR (Generalized Left-to-right Rightmost derivation) parsing techniques. Xoc's primary focus is on implementing new features via many little extensions, otherwise known as plugins. This approach has an advantage over most other extensible compilers in that the extensions to be loaded can be tailored for each source file. As such, Xoc provides functionality similar to that of traditional macro systems.

METABORG [10] is a method for embedding domain-specific languages in a host language. It does this by transforming the embedded language to the host language using the Stratego/XT [28] toolset. Stratego/XT supports grammar modifications using GLR parsing techniques.

Polyglot [24] is a compiler front-end framework for building Java language extensions; however, since it uses an LALR parser, extensions do not compose well. JTS [8], is a framework for writing Java preprocessor with the focus on creating domain-specific languages. Stratego/XT [28] is a compiler framework whose primary focus is on stand-alone program transformation systems; it also supports grammar modifications using GLR parsing techniques. CIL [23] focus in on C program analysis and transformation, and as such, does not support grammar modifications.

9. Conclusion and Future Work

ZL is a C++-compatible language in which high-level constructs are defined using macros over a C-like core language. The ZL macro system is unique in that it offers the full power and safety of Scheme's hygienic macros while also handling C's richer syntax. We have described the design and implementation of ZL's parser and macro implementation with its novel parsing and expansion strategy. While ZL macros are similar to Scheme's in many ways, C's richer syntax presents unique challenges that have been solved by ZL.

Future work includes supporting modular grammar extensions so that it is no longer necessary to modify the grammar specification file to support new lexical extensions. By using techniques from the Rats! [21] parser, ZL will allow grammar modifications to be scoped in a similar manner that variables and other identifiers are. Currently, ZL macros are dynamically typed, where a type is a grammar production. Another possibility is to support statically typed macros in addition to dynamically typed ones. This will allow for better error reporting and the ability to compile macros into templates for efficient expansion.

For the current implementation of ZL, see the ZL web page available at `http://www.cs.utah.edu/~kevina/zl/`.

Acknowledgments

We thank Carl Eastlund, Eric Eide, Gary Lindstrom, Ryan Culpepper, and Jon Rafkind for feedback on various incarnations of this paper. We also thank the anonymous reviewers for their corrections and comments.

References

[1] RScheme web site. `http://www.rscheme.org/rs/`.

[2] Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Proc. Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2009.

[3] Kevin Atkinson. *ABI Compatibility Through a Customizable Language*. PhD thesis, University of Utah, 2011. Expected.

[4] Kevin Atkinson, Matthew Flatt, and Gary Lindstrom. ABI compatibility through a customizable language. In *Proc. Generative Programming and Component Engineering (GPCE)*, pages 147–156, Eindhoven, The Netherlands, 2010.

[5] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proc. OOPSLA*, pages 31–42, Tampa Bay, FL, 2001.

[6] Jason Baker and Wilson C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proc. PLDI*, pages 270–281, Berlin, Germany, 2002.

[7] Eli Barzilay, Ryan Culpepper, and Matthew Flatt. Keeping it clean with syntax-parameterize. In *Workshop on Scheme and Functional Programming*, Portland, OR, 2011.

[8] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proc. Intl. Conf. Software Reuse (ICSR)*, page 143, 1998.

[9] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 31–40, Portland, OR, 2002.

[10] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. OOPSLA*, pages 365–383, Vancouver, BC, Canada, 2004.

[11] Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. Explicit programming. In *Proc. Conf. Aspect-Oriented Software Development (AOSD)*, pages 10–18, Enschede, The Netherlands, 2002.

[12] Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 244–254, Seattle, WA, 2008.

[13] R. Kent Dybvig. Syntactic abstraction: the syntax-case expander. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 25, pages 407–428. O'Reilly and Associates, June 2007.

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

[15] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[16] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[17] Robert W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10(3):316–333, 1963.

[18] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proc. Intl. Conf. Functional Programming (ICFP)*, pages 36–47, Pittsburgh, PA, 2002.

[19] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proc. POPL*, pages 111–122, Venice, Italy, 2004.

[20] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in macroml. In *Proc. Intl. Conf. Functional Programming (ICFP)*, pages 74–85, Florence, Italy, 2001.

[21] Robert Grimm. Better extensibility through modular syntax. In *Proc. PLDI*, pages 38–51, Ottawa, Ontario, 2006.

[22] Donovan Kolbly. *Extensible Language Implementation*. PhD thesis, Univ. of Texas, Austin, 2002.

[23] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. Conf. Compiler Construction*, pages 213–228, 2002.

[24] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proc. Conf. Compiler Construction*, pages 138–152, 2003.

[25] Andrew Shalit, David Moon, and Orca Starbuck. *Dylan Reference Manual*. Addison-Wesley, 1996.

[26] Michael Sperber (Ed.). The revised^6 report on the algorithmic language Scheme, 2007.

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

[28] Eelco Visser. Program transformation with Stratego/XT. Rules, strategies, tools, and systems in Stratego/XT 0.9. In Lengauer et al., editor, *Domain-Specific Program Generation*, Lecture Notes in Computer Science, pages 216–238. Spinger-Verlag, June 2004.

[29] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. PLDI*, pages 156–165, Albuquerque, NM, 1993.