Programmable Syntax Macros

Daniel Weise
Roger Crew
Microsoft Research Laboratory

Abstract

Lisp has shown that a programmable syntax macro system acts as an adjunct to the compiler that gives the programmer important and powerful abstraction facilities not provided by the language. Unlike simple token substitution macros, such as are provided by CPP (the C preprocessor), syntax macros operate on Abstract Syntax Trees (ASTs). Programmable syntax macro systems have not yet been developed for syntactically rich languages such as C because rich concrete syntax requires the manual construction of syntactically valid program fragments, which is a tedious, difficult, and error prone process. Also, using two languages, one for writing the program, and one for writing macros, is another source of complexity. This research solves these problems by having the macro language be a minimal extension of the programming language, by introducing explicit code template operators into the macro language, and by using a type system to guarantee, at macro definition time, that all macros and macro functions only produce syntactically valid program fragments. The code template operators make the language context sensitive, which requires changes to the parser. The parser must perform type analysis in order to parse macro definitions, or to parse user code that invokes macros.

1 Introduction

Macros are a meta-programming construct that transforms programs into programs. Macro languages are the world's "second oldest" programming language, having been invented immediately after symbolic assembly language. Macro languages provided abstraction facilities that assembly language lacked, such as procedure call and return, advanced control statements, and data-structuring facilities. The additional programming abstraction provided by a macro system has the advantage of incurring no runtime penalty (although it does incur compile-time penalty).

Thirty to forty years later, macro languages retain the same important use: providing abstraction facilities that a given language lacks. For example, a powerful syntax macro facility for C can seamlessly extend C to provide exception handling, a simple object system, "atomic" resource allocation and deallocation, and new control statements.

Syntax Macros

Syntax Macros operate during parsing. Their actual parameters, which are abstract syntax trees (ASTs), are discovered by the parser. Macros produce ASTs that replace the code of the macro invocation in downstream compiler operations. Syntax macros declare the type of AST they return. This information is used by the parser to ensure macro invocations only occur where their return type is expected. For example, syntax macros that return statements can only occur where statements are allowed by the grammar.

Syntax macros were independently introduced by Cheatham [4] and Leavenworth [9] in the middle 60's. They proposed that the actual parameters to macro invocations be found by a parser, and that the syntactic type of a macros return result be part of a macro definition. Vidart's PhD thesis [14] cleaned up many problems with syntax macros by making the leap to transformations on trees, rather than on token streams. His macro system was substitution based, that is, macro bodies are ASTs whose leaves are formal parameters that are replaced at invocation time with actual parameters. Our use of the term "Syntax Macro" refers to Vidart's use of the term. W. R. Campbell [3] proposed a paper extension to Vidart's work that extended it towards programmability, but in an *ad hoc* fashion. No implementation was reported for Campbell's extension.

The major advantages of syntax macros are *syntactic safety, encapsulation,* and *syntactic abstraction.* Syntactic safety guarantees that a macro user will never see a syntax error introduced by the use of a macro. Because transformations are specified in terms of syntactic constituents that the parser isolates, and because the type system of the base language can guarantee the construction of syntactically valid program fragments, users will only see syntax errors in terms of code they themselves write.

"Encapsulation" refers to non-interference of syntactic constituents. In a token based macro system, a substitution can yield a syntactically valid program that produces the wrong result because of unintended interference. For example, consider a macro with two formal parameters, `A` and B, whose body includes the expression `A * B`. If A and B were bound to `x + y` and `m + n`, respectively, then, in a token based macro system, the expansion contains the string `x + y * m + n`, whose parse is `x + (y * m) + n`, which is not the intended `(x + y) * (m + n)`. CPP macro writers are encouraged to use parentheses liberally to avoid this well known problem. In a syntactic framework, such interference is impossible because substitution is performed at the tree level. The macro writer does not need to be aware of unintended conflict.

"Syntactic Abstraction" refers to the ability to add new elements to existing syntactic domains, and to introduce new concrete syntax for the new elements. A syntax macro system literally adds to existing syntactic domains. (At least one macro system [10] also allows the construction of new syntactic domains.)

Programmable Syntax Macros

Macro systems differ in the power of their transformational engine (Figure 1). The weakest macro system are template based systems, the most powerful macro systems sport a complete programming language. A programmable syntax macro system allows the macro writer to act as a compiler writer. It can be viewed as a portable mechanism for extending the compiler itself. (This will become truer when the syntax macro system is extended into a semantic macro system, that is, one with access to a program's static semantic information.)

The problem of constructing a powerful and simple programmable syntax macro facility for syntactically rich languages has remained open. The fundamental problems to be solved in delivering a programmable syntax macro system for a language such as C are: to have the macro language be an extended version of the base language, to guarantee syntactic correctness of macro produced code, and to provide the convenience and simplicity of the substitution model where needed. This research solves these problems by making the macro language be an extension of the base language, by using the C type system to ensure syntactic correctness when macros and macro functions are defined, and by introducing code template operators to achieve the effect of substitution where needed by the macro writer. The presence of code template operators make the language context sensitive, which requires changes to the parser. The parser must perform type analysis to parse macro definitions, or to parse user code that invokes macros. Also, the parser must be fully re-entrant.

As an example of the type of code that must be written when a programmable macro (or other meta-programming) system lacks code templates or other code substitution facilities, consider a simple macro that abstracts away resource allocation and deallocation. Suppose that some window system required certain painting operations to be bracketed with `BeginPaint` and `EndPaint` statements that allocate then deallocate a "painting resource" (these examples are based upon the Windows API). Without macros, an idiomatic code fragment is:

```
BeginPaint(hDC, &ps);
<stmt1>;
...
<stmtn>;
EndPaint(hDC, &ps);
```

We'd like to capture this idiom in a macro called `Painting`, whose use would appear as:

```
Painting {
  <stmt1>;
  ...
  <stmtn>;}
```

In a straightforward meta-programming system, a function for producing the above code given the statement `{<stmt1> ...  <stmtn> }` might be (where `@stmt` is a type specifier for the AST statement type):

```
@stmt paint_function(@stmt s) {
  return(
   create_compound_statement(
    create_declaration_list(),
    create_statement-list(
     create_function_call(
       create_id("BeginPaint"),
       create_argument_list(
         create_id("hDC"),
         create_address-of(create_id("ps")))),
     s,
     create_function_call(
       create_id("EndPaint"),
       create_argument_list(
         create_id("hDC"),
         create_address-of(create_id("ps")))))));}
```

This style of code plagues meta-programming systems. For example, the meta-programming system of [2] heavily uses this style of coding.

We solved the problem of concisely generating syntactically valid program fragments in a programmable system by adding explicit template operators to the macro language that provide the convenience of substitution semantics. These template operators automatically construct the above style of code from a code template. For example, using the template operator backquote (`), the above function is written

```
@stmt paint_function(@stmt s) {
return(`{BeginPaint(hDC, &ps);
         $s;
         EndPaint(hDC, &ps);})}
```

where `$s` means to substitute the value of `s`. In general, the prefix operator `$` causes evaluation of the expression it prefixes; it is not restricted to just the substitution of identifiers. (Our code template mechanism was modeled after Lisp's [12].)

We have designed, and have mostly implemented, a programmable macro system for C that has the following attributes:

Programmability/Macro Basis	Character	Token	Syntax	Semantic
Full Programming Language	GPM [13]	360 Assembler	MS^2 Lisp, Scheme [6]	Maddox
Repetition & Conditional		Bliss [15] M5		
Repetition			Hygienic Macros [8]	
Substitution	Pre-ANSI CPP	ANSI CPP	Cheatham Vidart	

Figure 1: Two Dimensional Categorization of Macro Systems. Our macro system is MS^2 (Meta Syntactic Macro System). *Character Macros* operate at the character level. They transform streams of characters into streams of characters. We count pre-ANSI CPP as character based because many of them were. *Token Macros* operate on the tokenized representation of a program. They operate after (or in conjunction with) the tokenizer. Token based macro systems are simple to implement. Most assembler macro languages and the well known CPP [1] are token based. *Semantic Macros* are an extension of syntax macros that have access to, and can make decisions based upon, semantic information maintained by the static semantic analyzer. Semantic macros are the most powerful method for extending a language.

---

1. It is fully programmable, the macro language is C plus an extended type system plus additional primitive functions.
2. Macros manipulate Abstract Syntax Trees.
3. Its rich pattern language specifies the concrete syntax and the syntactic types of actual parameters for macro invocations.
4. Macros declare the syntactic types they return.
5. Full type checking during macro processing guarantees syntactically valid transformations.
6. Code templates are supported through the use of backquote forms.
7. Non-local transformations are possible, and are a powerful tool.

This paper has 5 sections. The macro language is presented in Section 2. Section 3 discuss implementation issues for the language. The fourth section presents many uses and examples of programmable syntax macros. The last section provides implementation status, related research, directions for future research, and conclusions.

2 The Macro Language

The macro language is C extended with AST types and operations on ASTs. We call the part of a program that defines macros and macro variables the *meta-program*. The meta-program is fully run during macroexpansion. None of it exists at runtime (except possibly as part of debugging information). Meta-programming constructs and regular programs that invoke macros can either be located in separate files, or mixed together into the same file, as is done now with CPP.

The macro language adds two new *top-level-declaration*s to the grammar of C: **meta-declaration**s and **macro-definition**s. These top-level declarations, which are prefaced by the keywords `metadecl` and `syntax`, respectively, declare elements of the meta-program that define macro transformations. Our syntactic descriptions follow the conventions of [5]. Non-terminals of the original C language appear in *italic type*. Non-terminals in our extensions to C appear in **boldface type**. All concrete tokens appear in `typewriter type`. Alternatives in a production are listed on separate lines. The only meta notation within an alternative is for repetition, which is represented as three dots (...), and which indicate that the previous item may appear one or more times. The macro languages add the following seven meta-tokens to C: `(|`, `{|`, `|}`, `$$`, `$`, `::`, and `@`. The syntactic clauses we present are meant to augment the syntax rules given in [5]. That is, the reader should assume that the syntax clauses of [5] are part of our syntactic definition of the macro language. Our rules either replace, or add to, this base set of rules.

top-level-declaration:
declaration
function-definition
meta-declaration
macro-definition

meta-declaration*:*
`metadecl` *declaration*

macro-definition*:*
`syntax` **macro-header** *compound-statement*

macro-header*:*
ast-specifier *declarator* {| **pattern** |}

Macro definitions have two parts, a header and a body (compound statement). The header specifies the syntactic type of the AST returned by invocations of the macro, the name of the macro, and the macro pattern used for invocation. Pointer and function declarators are not meaningful in this context. The macro pattern specifies concrete syntax and the required AST types of the actual parameters. The macro pattern guides the parser when it discovers the invocation of a macro. The compound statement is the macro body, the value it returns is the result of the macro invocation.

pattern:
pattern-element ...

pattern-element:
token
$$ **pspec** :: *identifier*

pspec:

ast-specifier	
+ **pspec**	list of 1 or more
+ \ *token* **pspec**	list of 1 or more + separator
* **pspec**	list of 0 or more
* \ *token* **pspec**	list of 0 or more + separator
? **pspec**	optional element
? *token* **pspec**	optional preamble + element
. **pattern**	tuple

The pattern parser used to parse macro invocations requires that detecting the end of a repetition or the presence of an optional element require only one token lookahead. It will report an error in the specification of a pattern if the end of a repetition cannot be uniquely determined by one token lookahead. If the ? *token* **pspec** clause is used in the pattern, then the token indicates whether the optional **pspec** is present, if the token is present in the invocation, then the **pspec** must be present. The optional elements are for constructing statements such as loops that accept, for example, optional `step` or `while` clauses.

Example macro headers will be presented after we provide the syntax for **ast-specifier**.

The AST Type Language

The type language for ASTs has as primitives `id`, `stmt`, `decl`, `exp`, `num`, and `type_spec`.

type-specifier:
`@` **ast-specifier**
⋮

ast-specifier:
`decl`
`stmt`
`id`
`exp`
`num`
`type_spec`

The combining types on asts are *tuples* and *lists*. We overload C syntax for declaring tuples and lists: structure declarations define tuples, and array declarations define lists. For example, the declaration `@id id_list[]` defines `id_list` to be a list of identifiers. Every macro pattern has an associated type: the repetition patterns produce lists and patterns produce tuples. The C operators on arrays and structs are overloaded to operate on lists and tuples. For example, `*id_list` is the equivalent of the Lisp instruction `(car id_list)`, and `id_list + 1` corresponds to `cdr`. It is illegal to take the address of either a scalar or structured ast value.

Additional Primitive Functions

The macro language includes primitive functions on ASTs, such as `length`. There are also functions for creating new identifiers, such as `gensym()` and `concat_ids(id1, id2)`. The additional functions add no conceptual novelty to the macro system.

We also have predefined member names for extracting components of ASTs such as `stmt1->declarations` and `declaration3->type_spec`. We were not willing to add pattern-matching binding mechanisms, such as ML or modern functional languages have. While pattern matching would add to the clarity of the macro language, we currently believe that such extra syntactic mechanism should be provided by macros, and not by us. The macro language should be C, with as few syntactic extensions as possible.

The only excursion from this principle is the presence of anonymous functions. As an experiment, the macro language includes anonymous functions that may only be passed downwards. Anonymous functions are written as `(|` *declaration-list expression*`)`. These functions return the value computed by the *expression* without needing a return statement. Because macros perform many list manipulations, anonymous functions are very useful. Future research will indicate whether anonymous functions must remain a special element in the macro language.

Example Macro Headers

As an example, consider the macro that implements the `Painting` abstraction shown in an earlier example. The syntax specifier says that the actual parameter of the macro is expected to be a statement. There are no "buzz tokens." The macro definition is:

```
syntax stmt Painting {| $$stmt::body |}
{return (`{BeginPaint(hDC, &ps);
```

```
        $body;
        EndPaint(hDC, &ps);});}
```

As another example of a macro-header, consider a macro that mimics `enum`, but also automatically provides functions for reading and writing elements of the enumerated type. An invocation of the macro might be

```
new_enum color {red, blue, green};
```

The header for such a macro is:

```
syntax decl new_enum[]
  {| $$id::name { $$+\,id::ids }; |}
```

The macro `new_enum` returns a list of declarations. The macro-pattern `+\,id` matches 1 or more repetitions of identifiers separated by commas. If the macro parser cannot find such a repetition, an informative error message is returned to the user. The pattern contains concrete syntax (e.g., the comma separator) which does not appear in the AST (a list of identifiers) returned by the parser and then bound to `ids`. Also, the trailing semicolon is part of the syntax of the macro, which makes the syntax for `new_enum` consistent with the syntax for other top level declarations in C. The macro returns a list of top level declarations, rather than a single declaration. It returns a list of declarations because it needs to return a `enum` declaration, along with two function declarations, one for reading elements of the new type, and one for writing elements of the new type.

Template Operators

A key innovation of this research is the incorporation of Lisp's backquote operator for explicitly declaring code templates in a syntactically rich language. In a CPP style macro, the body of the macro is the template, and substitution happens to *any* token that matches either a formal parameter or a `#defined` keyword. In this research, the body of a macro is a C compound statement. An explicit code template mechanism, along with an explicit "unquoting" (substitution) operator are used to achieve substitution macro semantics. We use the backquote character, `, to introduce code templates, and use dollar sign, $, for the unquoting operator. The backquote operator is a prefix operator, much as the token "`!`" is, and is therefore in the class of *unary-expressions*.

unary-expression:
  postfix-expression
  ⋮
  predecrement-expression
  backquote-exp-expression
  backquote-stmt-expression
  backquote-decl-expression
  backquote-pattern-expression

The backquote operator returns an AST. This AST may have *placeholders* embedded in it. The first token after the backquote determines the syntactic type of the AST that backquote will return. The open brace, "{", signifies that a statement follows, the open parenthesis, "(", signifies an expression, and an open bracket, "[", signifies a top-level declaration. There is a also a general form of backquote that accepts a pattern that determines the AST(s) to parse.

backquote-exp-expression:
  ` (*expression*)

backquote-stmt-expression:
  ` { *statement* }

backquote-decl-expression:
  ` [*top-level-declaration*]

backquote-pattern-expression:
  ` {| **pspec** :: *template-specified-syntax* |}

The first three backquote forms are convenient shorthand notation for expressions, statements, and declarations that could have been specified by the general backquote form (*e.g.*, `{| exp :: *expression* |}).

Dollar sign, "$", introduces a new syntactic domain: *placeholder*. Dollar sign precedes either an identifier or a parenthised expression. Placeholders may only appear within backquote expressions. A placeholder signifies that its argument is to be evaluated, and the results to be placed within the code (AST) being constructed by the backquote operator.

placeholder*:*
  $ *identifier*
  $ (*expression*)

Many primitive syntactic classes are extended to include placeholders as valid alternatives. For example, placeholders can stand in for statements, expressions, declarations, and identifiers.

When backquote expressions are evaluated during macro expansion, placeholders must expand to AST of the expected type. For example, placeholders that stand in for statements must expand to statement ASTs. That they will do so is ensured by the static type checker when the backquote is parsed. (However, nested backquote causes special problems. Some of the code it creates cannot be analysed until the first level of expansion occurs, so not all checking can occur when nested backquote expressions are parsed.)

Macros and macro functions produce ASTs. Most importantly, code templates also produce ASTs, and placeholders return ASTs. Therefore, our macro system differs from most others because the macro writer may ignore issues of concrete syntax when specifying the code a macro is to produce. This distinction arises when manipulating lists of items that contain concrete separators that irrelevant to the abstract syntax. For example, the separator character for the *init-declarators* of a *declaration* is the comma character. Suppose that the macro writer has a list of identifiers, say, `(red blue green)`, bound to meta-variable `ids`, and wishes to produce the top level declaration

```
enum color red, blue, green;
```

The concrete syntax for this declaration includes the separator commas. The macro writer need not be aware of this concrete syntax, and can simply write the template

```
`[enum color $ids;]
```

Some macro systems that allow for repetition and list handling (*e.g.*, the Bliss macro system) have a set of special case rules for ensuring that the correct separators appear in the produced code. Because our syntax macro system explicitly constructs ASTs, and not concrete code, these extraneous concerns vanish. However, the macro writer does need to be aware of the C abstract syntax used by the macro system. We think that making the macro writer be aware of abstract syntax is a fair price to pay for simplifying the transformational language.

3 The Parser

Macros and template operators make the grammar context sensitive. The presence of placeholders in templates force the parser to employ semantic analysis (*i.e.*, type analysis and checking) that determines the type of AST returned by a placeholder when it is evaluated. This AST type information guides the parsing of templates. The presence of macro invocations also require parse-time semantic analysis for correct parsing of code that contains macro invocations and for checking that macros of a given type only appear where they are allowed. Our parser is a hand-written recursive descent parser at the declaration and statement levels, but a bottom-up precedence parser at the expression level.

Parsing Macro Headers

When a macro keyword is encountered, the parser interprets the macro's **pattern** to guide the parse of the invocation. The specifier indicates where and which tokens are to be found, as well as the required syntactic constituents of the macro invocation. This process is a relatively small part of compiling a program. However, even this process could be accelerated by a routine that compiled a parse routine for each macro's **pattern**. This specialized routine would be associated with the macro keyword and called when needed.

Parsing Code Templates (Backquote)

The AST denoted by a code template must be uniquely determined by information available at macro definition time. It cannot depend on information available only when the macro is invoked. This restriction allows macro expansion to be implemented very efficiently.

The presence of placeholders in templates force the parser to employ semantic analysis of the metacode containing the backquote and the placeholder expression. At any given point during a parse, the parser may be faced with optional constructs and several alternative constructs. Normally, the parser uses concrete tokens such as identifiers and punctuation to thread through the maze of possible parses. These tokens are not in the templates because they won't be supplied until a macro is invoked.

For example, consider the source code template `` `[int $y;] `` in a context where a declaration is expected. There are several possible parses depending on the AST type that `y` will be bound to, as shown in Figure 2.

As another example of parses being dependent upon the AST type returned by placeholder expressions, consider the difficulties of parsing compound statements, which have no explicit markers that separate declarations from statements. For example, the code template `` `{int x; $ph1 $ph2 return(x);} `` has four different possibles parses depending upon the types of `ph1` and `ph2` (Figure 3).

We help the parser disambiguate potential parses by careful design of the token stream that the parser reads from. We introduce a new type of token that we call a *placeholder token*. Upon discovering a `$` token, the "tokenizer" co-routines with the parser to parse the placeholder expression in the semantic context at entry to the containing backquote expression; performs AST type analysis on the expression to determine the type of AST it will return when run; and then wraps the expression and its type into a placeholder token. The different routines of the parser perform lookahead on the token stream to see if the AST they are to parse is represented by the next token. For example, if the routine `parse-statement` finds a placeholder token on the head of token stream that has type `stmt`, then the routine returns the placeholder token as the result of its parse. If it doesn't find such a token, it operates normally.

The AST type analysis performed by semantic analyser is germane. It knows the declared types of meta-variables (both globals and parameters of macros and meta-functions) and the types returned by primitive operations on ASTs. It uses this information to determine the type returned by a placeholder expression.

Parsing Macro Invocations

When the parser encounters a macro keyword, it parses the invocation according the macro's pattern, packages up the macro with its actual parameters for later expansion, then uses the declared type of the macro to decide how to continue the parse. In principle, macro invocations should be allowed to appear wherever placeholders are allowed. Our system, however, currently only allows macro invocations where either declarations, statements, or expressions are expected.

statement:
    expression-statement
    ⋮
    null-statement

AST type of y	Parse
init-declarator[]	⟨declaration (int) y⟩
init-declarator	⟨declaration (int) (y)⟩
declarator	⟨declaration (int) (⟨init-declarator y ⟨ ⟩⟩)⟩
identifier	⟨declaration (int) (⟨init-declarator ⟨direct-declarator y⟩ ⟨ ⟩⟩)⟩

Figure 2: The different parse trees for the source code template `‘[int $y;]` depending upon the AST type of the metavariable `y`. A node of the tree and its children is written ⟨`node-name child1 ... childn`⟩. List elements in the tree are written within parentheses.

ph1	ph2	Parse	
decl	decl	⟨c-s ⟨decl-list(⟨decl "int x"⟩ ph1 ph2)⟩	⟨stmt-list (⟨r-s ⟨exp ⟨id x⟩⟩⟩)⟩⟩
decl	stmt	⟨c-s ⟨decl-list(⟨decl "int x"⟩ ph1)⟩	⟨stmt-list (ph2 ⟨r-s ⟨exp ⟨id x⟩⟩⟩)⟩⟩
stmt	stmt	⟨c-s ⟨decl-list(⟨decl "int x"⟩)⟩	⟨stmt-list (ph1 ph2 ⟨r-s ⟨exp ⟨id x⟩⟩⟩)⟩⟩
stmt	decl	Syntactically Illegal Program	

Figure 3: Parses of the code template `‘{int x; $ph1 $ph2 return(x);}`. For conciseness, we have abbreviated *compound-statement* as *c-s*, *return-statement* as *r-s*, *statement* as *stmt*, *identifier* as *id*, *expression* as *exp*, and *declaration* as *decl*.

placeholder
macro-invocation

primary-expression:
name
literal
parenthised-expression
placeholder
macro-invocation

declaration:
declaration-specifiers initialized-declarator-list ;
placeholder
macro-invocation

macro-invocation:
macro-keyword *syntax-specified-by-macro-template*

Dealing with Context Sensitivity

C is not a context free language. Due to `typedef`, the parse of a program fragment depends upon the context that the fragment appears in. For example, the fragment `foo *i;` should be parsed as a declaration if `foo` has been made type specifier via `typedef`, otherwise the fragment should be parsed as an expression statement. The existence of syntactic macro delcarations introduces further context sensitivity.

Our parser is forced to parse program fragments independently of the context those fragments will appear in. (We aren't completely happy with this approach, and are examining alternatives.) Such fragments appear within backquote expressions inside of macro bodies, and in the actual parameters to macro invocations.

This design choice limits the expressiveness of macros, and leads to some non-intuitive results. Macros that produce `typedefs`, use macro-defined `typedefs`, or are parameterized over `typedef`'ed names must be used carefully. Otherwise, a name will be parsed as if it were a normal *identifier*, instead of as a *type-specifier*.

This design choice also prevents a macro from setting up context in which its actual arguments are to be parsed within. For example, consider a looping macro that wants the special keyword `exit` to have meaning only within its actual arguments. The clean solution is to have the looping macro set up a special `exit` macro that only has effect during the parsing of its actuals. However, the looping macro's arguments are parsed with no knowledge of the invoking macro, other than its template. Therefore the `exit` macro must be global, and invocations of it can't know whether they occur within the arguments to the looping macro. To get around this problem, we must add linguistic constructs to the macro language that would allow an invoked macro to set up a parsing context for its arguments.

Macro Expansion

Because the macro language is C extended with AST datatypes and a few new primitive functions, macro expansion is simply a matter of running a C program on the parsed arguments of a macro invocation. The ease of debugging macros depends upon the quality of the debugger provided by the C programming environment being used.

The present implementation uses an embedded interpreter for a subset of the C language to execute meta-code. We chose this solution for simplicity. Macros perform fairly simple and routine actions where speed is not of tremendous importance, so an interpretive approach suffices. The alternative approach, using fully compiled C routines, would require that the parser be able to dynamically link in compiled C functions. This is certainly possible, but taking the speediest approach for executing macros was beyond the scope of this research. A production system might well opt for a fully compiled approach, rather than an interpreted approach.

4 Uses of Macros

A programmable syntax macro system offers many advantages. First, it provides a framework upon which special purpose preprocessors can be built. Many software projects, especially in the database field, extend a language to incorporate domain specific data types and statements. The first task of these projects is to write a preprocessor, a task that would be trivial if a suitable macro facility were available.

Second, macros form the basis of an extremely low overhead virtual machine. The literature from the 1960's and 1970's is rife with papers that propose the use of powerful macro languages to solve the software portability problem. Although we now have a common system programming language (C) to solve portability problems at the hardware architecture level, in the 90's the divergence of OS API's (Unix SVR4, BSD, SUNOS, MAC/OS, DOS, W/NT)[1] present new and difficult portability problems. There are two solutions to this problem: implement a common virtual machine as an interpreter, which incurs a large performance penalty, or implement a common virtual machine as a series of macros in a programmable macro language, which has other problems, but at least can be very low overhead.

New control constructs, such as specialized looping constructs, and domain dependent control constructs are easily implemented in a programmable syntax macro system. Specialized control constructs raise the abstract programming level. For example, resource allocation/deallocation is a common idiom of the form: Grab a resource, use the resource, release the resource. The `Painting` macro displayed earlier had this structure. A simple macro can capture the allocate/use/deallocate idiom.

We now present examples of macros we have written. Because of space restrictions, we focus on what can be done with macros, rather than the actual mechanisms (*i.e.*, macro definitions). The simpler macros can be implemented with CPP, though not as cleanly, abstractly, or safely as with syntax macros.

Dynamic Binding

Dynamic binding is very important in many applications. One of the most important applications of dynamic binding is the declaration and use of exception handlers, which are, ipso facto, dynamically bound entities. Macros offer a simple and clean mechanism for declaring dynamic binding. For example, the following macro defines a new statement type that modifies an integer variable, executes code, then unmodifies the variable.

```
syntax stmt dynamic_bind {|
  {$$type_spec::type $$id::name = $$exp::init}
  $$stmt::body |}
{@id newname = gensym();
 return(`{$type $newname = name;
          $name = $init;
          $body;
          $name = $newname;})}
```

An invocation of this macro might appear as

[1] Some of these names are trademarks of their respective owners.

```
dynamic_bind {int printlength = 10}
  {print_class_structure(gym_class);}
```

(In a semantic macro system, which has full access to the static semantic analyzer of the base language, the type of `name` would be available to the macro system. In this case, the macro user wouldn't need to declare the type of `name`.)

Exception Handling

We now implement an exception handling system using syntax macros. This is a simple system, where the catch tags are identifiers. The exception system could be made more complex by writing more complex macros, but the example is about using macros, not about a powerful exception system. An exception system needs three items:

1. A method for establishing a handler.
2. A method for invoking a handler.
3. A method for carefully unwinding the stack .

We have invented three new statements, `throw`, `catch`, and `unwind_protect`:

```
syntax stmt throw {| $$exp::value ; |}
{if (simple_expression(value))
   return(
    `{if (exception_ptr == NULL)
       error("No handler for %d", $value);
       else longjmp(exception_ptr,$value);});
    else
      return(
       `{int the_value = $value;
         if (exception_ptr == NULL)
            error("No handler for %d", the_value);
         else longjmp(exception_ptr, the_value);});}

syntax stmt catch
  {| $$exp::tag { $$stmt::handler } { $$stmt::body }
  |}
  {return(
    `{int *old_exception_ptr = exception_ptr;
      int jmp_buf[2];
      int result;
      result = setjump(jmp_buf);
      if (result == 0)
        {exception_ptr = jmp_buf; $body}
      else {exception_ptr = old_exception_ptr;
            if (result == $tag)
               $handler else throw result;}});}

syntax stmt unwind_protect
  {| { $$stmt::body } { $$stmt::cleanup } |}
  {return(
    `{int *old_exception_ptr = exception_ptr;
      int jmp_buf[2];
      int result = setjump(jmp_buf);
      if (result = 0)
        {exception_ptr = jmp_buf; $body}
      exception_ptr = old_exception_ptr;
      $cleanup;
      if (result != 0) throw result;});}
```

(Note, these examples ignore the problem of variable capture caused by the introduction of new names that might match names in substituted code. This problem is solved by having a function `gensym` to create names that cannot appear in user code, but the extra work would have needlessly complicated the examples. We briefly discuss inadvertent name capture in the last section.)

The `catch` statement establishes a handler called `name` during the execution of its `body`. If, during execution, a `throw` statement is executed that throws `name`, then the execution of `body` exits, and `handler` is run in its place. That is, the exception system has "termination" semantics. `Unwind-protect` is a statement that ensures `clean_up` is executed even when a `throw` statement throws out of the execution of its `body`. If no throw occurs during the execution of `body` then `clean_up` is executed. If a throw does occur, the execution of `body` is terminated, `clean_up` is run, then the throw continues unwinding the stack until it reaches the `name` it matches. The macro `unwind-protect` is a very important part of an exception handling system when writing interactive programs where the user may asynchronously abort a computation. We now show these macros in action.

```
myenum error_types
  {division_by_zero, file_closed, using_unix};

int foo(a,b,c)
int a, b;
int *c;
{int z, *y;
 z = a + b;
 catch division_by_zero
   {printf("%s", "You lose, division by zero.");}
   {*c = frob(z, a);}
 unwind_protect {start_faucet_running();}
                {stop_running_faucet();}
 return(z);}
```

The macro `unwind-protect` is especially important for maintaining the allocate/deallocate invariant discussed earlier. Consider again the Painting macro. It should use unwind-protect to ensure the deallocation of the painting structure:

```
syntax stmt Painting {| $$stmt::body |}
{return (`{BeginPaint(hDC, &ps);
           Unwind_protect
            $body
            {EndPaint(hDC, &ps);}});}
```

The user of the Painting macro need not be aware of this behavior, it's just part of the abstraction.

Readers and Writers for Enumerated Types

C does not provide routines for reading or writing elements of user defined types. Syntax macros can automatically provide such routines. In this example we consider a reader and a writer for enumerated types. A macro such as `myenum` could be defined to have the same effect as `enum`, and also write extra code, as shown below.[2]

```
myenum fruit {apple, banana, kiwi};
```

expands into

```
enum fruit {apple,banana,kiwi} ;
```

[2] We have not yet addressed the issue of meta-level namespaces. The artifice of using the name `myenum` rather than the more logical `enum` is to avoid name clashes. A better method would make explicit the notion that a macro package extends language N into language $N+1$, and allow code templates to declare whether the identifiers they employ refer to language N or to language $N+1$. We could then name the macro `enum`, and have its use of `enum` refer to the language being extended, rather than to the language being defined. This problem is independent of standard name capture problems addressed by hygienic macro systems.

```
void print_fruit(int arg)
{switch (arg)
   {case apple: printf("%s", "apple");
    case banana: printf("%s", "banana");
    case kiwi: printf("%s", "kiwi");}}

int read_fruit()
{char s[100];
 getline(s, 100);
 if (!strcmp(s, "apple")) return(apple);
 if (!strcmp(s, "banana")) return(banana);
 if (!strcmp(s, "kiwi")) return(kiwi);}
```

The macro itself is (where the function `map` applies a function parameter to each element of its second parameter to return a new list):

```
syntax decl myenum[]
  {| $$id::name { $$+id,::ids }; |}
  {return(list(

    `[enum $name $ids;],

    `[$(symbolconc("print_",name))(arg)
      {switch (arg)
        $(map((|
               @id id;
               `{case $id:
                  printf("%s", $(pstring(id)));}),
               ids))}],

    `[$(symbolconc("read_",name))()
      {char s[100];
       getline(s,100);
       $(map((|
               @id id;
               `{if (!strcmp(s,$(pstring(id))))
                    return($id);}),
              ids))}]
     ));}
```

Generalizations of this example are quite useful. Persistence code, RPC code, dialog boxes, etc., can be automatically created when data is declared. The problem of automatically constructing such routine code has been addressed before in the meta-programming literature. For example, [2] presents a meta-program for automatically constructing reading and writing functions for Pascal enumerated types. Because that system didn't have code templates, the code for doing so is rather bulky and unwieldy. By using code templates, we achieve a similar effect with far less code that is far more readable.

Code Rearrangement

Frequently code must appear in one place in a program when it would be conceptually simpler to distribute the code throughout the program. For example, in programs that are not object oriented, large dispatch routines are still common. Such code is more perspicuous when each datatype says how it relates to the dispatch, rather than have the information only in the one large dispatch routine. Conversely, frequently code is spread through a program that would be better placed in one or two routines. Many GUI's force the programmer sprinkle responses to different events for a given state of the system throughout her code. It would be more perspicuous to write the code in one place, and have it automatically distributed throughout the code.

The example we are about to show modularizes the construction of a dispatch table in a non-object oriented language. The dispatch procedure will reside in one place in the final code, but we will write the code in a distributed fashion. We will achieve this effect by writing macros that accumulate the code fragments together, and then use a macro to glue them together and emit a full blown window procedure. The concrete dispatch procedure we'll discuss is the main window processing loop in a Windows based application.

We'll use three macros for writing distributed code that is collected into one large dispatch routine (we present just the macro-headers):

```
syntax decl new_window_proc {| $$id::proc_name
  default $$id::default_proc_name; |}
syntax stmt window_proc_dispatch
  {| ( $$id::proc_name, $$id::message_name )
  $stmt::body |}
syntax stmt emit_window_proc
  {| $$id::proc_name ; |}
```

The macro `new_window_proc` establishes a new window procedure with name `proc_name`. The default procedure to be called when this procedure doesn't handle a message is `default_proc_name`. This procedure will collect code fragments created by repeated calls to `window_proc_dispatch`. The macro `window_proc_dispatch` takes the name of a window procedure, the name of a message, and a statement. It associates the message name and the statement with the window procedure. The macro `emit_window_proc` causes to appear in its place in the program the window procedure that was accumulated via `window_proc_dispatch`es that named the window procedure. As a simple example, the program

```
new_window_proc wproc default DefWindowProc;

window_proc_dispatch(wproc, WM_DESTROY)
{KillTimer(hWnd, idTimer);
 PostQuitMessage(0)}

window_proc_dispatch(wproc, WM_CREATE)
{idTimer = SetTimer(hWnd, 77, 5000,
                    (TIMERPROC) NULL);}

emit_window_proc wproc;
```

produces the program

```
int wproc(HWND hWnd, UNIT message,
          WPARAM WParam, LPARAM lParam)
{switch (message) {
    default: {
      return(DefWindowProc(hWnd, message,
                           wParam, lParam));
      break; }
    case WM_CREATE: {
      idTimer = SetTimer(hWnd, 77, 5000,
```

```
                         (TIMERPROC) NULL);
      break; }
    case  WM_DESTROY: {
      KillTimer(hWnd, idTimer);
      PostQuitMessage(0);
      break; }}
 return(NULL);}
```

The three macros use a global meta-variable to communicate. Beside the three macro definitions, there is also metacode for declaring this metavariable, which holds the methods for the different window procedures defined by the macros.

```
metadecl struct w_proc
 {@id name; method_table *mtp; w_procs *next;};
metadecl struct w_proc *proc_list = NULL;
```

Space prevents us from presenting the full macro definitions. The macro `new_window_proc` adds an empty method table to `proc_list`, the macro `window_proc_dispatch` adds a method to a method table, in `proc_list`, and the macro `emit_window_proc` produces a C function based on the contents of the method table of the window procedure to be produced.

5 Related and Future Research

The system is currently undergoing revision. Except for the last macro example, all macros shown in this paper worked in the previous implementation. Once the re-implementation is complete, we will conduct large scale experiments with the macro system.

Besides the systems already mentioned [14, 3, 4, 9], there's current work on new macro systems for syntactically rich languages. Todd Jonker [7] is working on hygienic macro technology for syntactically rich languages. William Maddox at Berkeley is investigating extensions of his Master's thesis [10] that give macros semantic abilities. Dain Samples has recently announced M5 [11], another token-based macro system. It is a very general macro processor that can be programmed to understand the lexical conventions of different languages (*e.g.*, C, C++, Ada, and Tex).

In the Lisp community, "Hygienic Macros" [8] and "syntactic abstractions" have been proposed for making macros easier and cleaner to write. Such macro systems automatically avoid unanticipated capture of free variables. Early hygienic macro system were substitution based and had support for repetition. More recent research [6] has given hygienic macro systems complete programming power. We feel that hygienic macro systems are very important, and are considering methods for making our system be hygienic and referentially transparent. In a sense, this research has brought C up to 1970's Lisp macro technology. More research is required to take it into the 90's.

Another goal is the implementation of *semantic macros*, which are an extension of syntax macros where the macro processor does static semantic analysis (*e.g.*, type checking). Semantic macros will have two new important powers. First, they conditionalize their operation based on runtime types returned by the expressions they manipulate. This will allow, among other things, a form of object oriented dispatch at compile time. It will also simplify the invention of new binding forms, which need access to the runtime type of values they will be manipulating. Second, the macro writer will do all relevant type checking in the macro itself, and not wait for type errors to be found by downstream type checking. This would make macros be completely self contained, and ensure that programmers wouldn't end up having to track down type errors in code they didn't write, as is the case for syntax macros.

We have shown that a programmable macro system can be very simple and very powerful. The major contribution of this paper is the introduction of explicit code template operators and AST datatypes into an existing language to could a powerful macro system for that language. Such an approach can be used for any language as long as the parser can be modified to perform type checking during the parse. An important advantage of this approach is that a new macro language with its own special syntax, operators, statements, and functions does not have to be invented, as has been historically done.

Acknowledgements The authors would like to thank Bjarne Steensgaard, who read and commented on drafts of this paper, and the anonymous referees.

References

[1] *ANSI Standard on C, X3.159-1989,* ANSI, NY, 1990.

[2] Cameron, R. D., "Software reuse with metaprogramming systems," Proceedings of the Fifth Annual Pacific Northwest Software Quality Conference, OR, pp. 223-32, 1987

[3] Campbell, W. R., "A compiler definition facility based on the Syntactic Macro," Computer Journal 21(1), pp. 35-41, 1975

[4] Cheatham, T. E., "The introduction of definitional facilities into higher level programming languages," pp. 623-637, Proc. AFIPS (Fall Joint Computer Conference, 29), 1966

[5] Harbison, S., and Steele, G., *C, A Reference Manual*, Third Edition, Prentice Hall, 1991

[6] Hieb, R., Dybvig, R. K., Bruggeman, C., "Syntactic abstraction in Scheme," University of Indiana Computer Science Technical Report 355, 18 pages, June 1992 (Revised July 1992).

[7] Jonker, Todd, Personal Communication, 1992

[8] Kohlbecker, Eugene, Friedman, Daniel P., Felleisen, Matthias, Duba, Bruce, "Hygienic macro expansion," Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, pp. 151-161, ACM Press, NY, 1986

[9] Leavenworth, B. M., "Syntax Macros and Extended Translation," CACM 9(11), pp. 790-793, 1966

[10] Maddox, William, *Semantically-Sensitive Macroprocessing* Report No. UCB/CSD 89/545, (Master's Thesis), 82 pages, University of California, Berkeley, 1989.

[11] Samples, Dain, M5, Electronic Announcement on comp.compilers newsgroup, 1992

[12] Steele, G., *Common Lisp, The Language,* Digital Press, 1984.

[13] Strachey, C., "A general purpose macrogenerator," Computer Journal, 8(3), pp. 225-241, 1965

[14] Vidart, J., *Extensions syntaxiques dans une contexte LL(1),* University of Grenoble, Th'ese pour obtenir le grade de Docteur de troisi'eme cycle, 1974

[15] Bliss-11 Programmer's Manual, CMU Department of Computer Science, 1974.