

A comparison of macro systems for extending programming languages

Dirk Gerrits, René Gabriëls

30 March 2005

Abstract

With the ever increasing complexity of software, more and more projects are employing code generation techniques. Usually a stand-alone program is used to mechanically produce large volumes of source code that are then compiled along with the rest of the project's code just as if it had been written by a human.

Another approach, that hasn't been given as much attention in recent years, is the use of *macro systems*. With a macro system it is possible to extend the syntax of a programming language in a controlled way, thereby integrating such code generation more closely into the programming environment.

In this essay we will describe three increasingly advanced macro systems: textual macros, syntactic macros, and syntactic macros that respect lexical scoping rules. For each macro system we sketch its underlying algorithm, discuss the system's merits and pitfalls, and contrast it with other systems considered in this essay.

Contents

1	Introduction	3
2	Textual macros	4
2.1	History	4
2.2	Workings	5
2.3	Analysis	6
2.3.1	C macros	6
2.3.2	Operator priority problems	8
2.3.3	Variable capture	9
2.3.4	Multiple evaluation	9
2.3.5	Resolving errors	10
2.4	Examples	10
3	Syntactic macros	12
3.1	History	12
3.2	Workings	12
3.3	Analysis	13
3.3.1	Common Lisp macros	13
3.3.2	Advantages	14
3.3.3	Variable capture	14
3.3.4	Free identifiers	16
3.4	Lexical scoping	17
3.5	Remaining issues	18
3.6	Some final examples	19
4	Concluding remarks	21

1 Introduction

Although *macro systems* have many uses, for this essay we consider the purpose of a macro system to be to allow the programmer to extend a programming language's syntax by defining *macros*. A macro describes the syntax of a new type of expression or statement and how it transforms into the programming language's *base syntax*. A use of such a macro in the program text is called a *macro call*.

To compile a program using macros means simply to *macroexpand* all macro calls into base syntax, and then compile the resulting program using a compiler for that base syntax. There are numerous choices on the time in the compilation process at which to do macro expansion, which gives us an important classification criterion:

- *Textual macros* are expanded before, during, or just after tokenization. They take in a stream of characters or tokens and then produce another such stream which is fed to the parser.
- *Syntactic macros* are expanded during or after the context-free parsing process. They operate on abstract syntax trees, rather than text.
- *Semantic macros* are expanded during or after context analysis. They operate on abstract syntax trees extended with context attributes.
- *Computation macros* are expanded during the code generation process. They serve as a mechanism to extend or enhance the code generating capabilities of a compiler.

The situation is illustrated in Figure 1.

In this essay we will limit ourselves to textual and syntactic macros. For more information on semantic and computation macros we refer you to [4] and [3, Ch 1.10], respectively.

For each type of macro systems we will give an overview of its history, followed by a description of its general workings, and conclude with an examination of an existing programming language that uses such a system by considering the merits and pitfalls of that system and contrasting it with other systems we present in this essay.

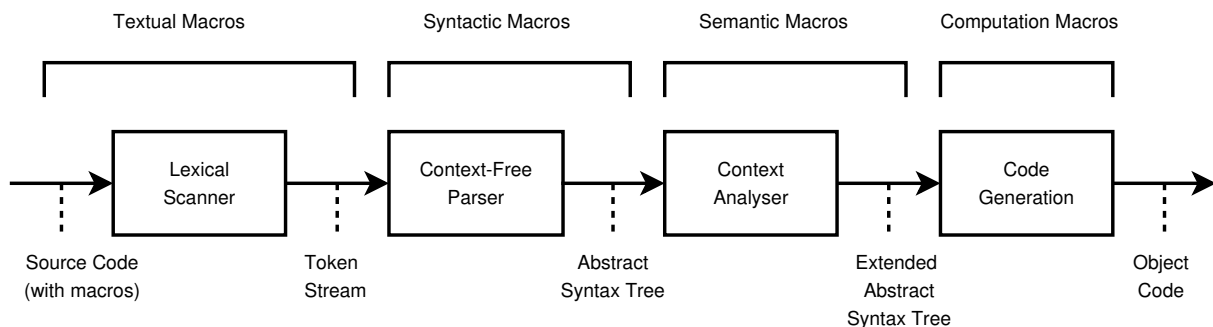


Figure 1: Structure of a compiler with macro system(s)

2 Textual macros

2.1 History

Textual macros are the oldest form of macros. They were invented in the realm of assembly languages in the 1950s, to allow the programmers to extend the operations (i.e. opcodes) that were supported by the specific architecture. This is useful because assembly code usually contains lots of repetitions of a small number of patterns, that are either too small to justify their own subroutine, or are not well suited for procedural abstraction. Building an abstraction around the code repetition was a logical step, and so assemblers were extended with macro facilities that allowed programmers to define new operations in terms of lower level operations. Here's a simple example, using an abstract assembly language, that abstracts a method for copying a value:

```
COPY MACRO A, B
    LOAD  A
    STORE B
ENDMACRO
```

Another useful application of macro facilities in assembly languages was to achieve almost portable assembly code. For example, an assembler for an architecture that didn't have floating point operations could simulate them by using macros to expand the floating point operations to supported integer arithmetic.

The first macro facilities for assembly languages used a technique known as *template filling*, in which parameter occurrences in the macro body were substituted for the actual parameters in the macro call, after which the resulting instantiated macro body was substituted for the macro call.

A problem that arose was that of interfering labels. When a fresh label was used in the macro body, programmers either had to supply the name of it to the macro call (which resulted in leaky abstraction) or never use the macro more than once and never use the label anywhere else (which is even worse than leaky abstraction). So the assemblers had mechanisms for generating unique fresh labels when instantiating a macro. A simple example, showing the use of this mechanism, is the MAX macro which uses the pseudo-operation GEN, which is supplied by the assembler:

```
MAX MACRO A, B, C      ; C := MAX(A,B)
    GEN  L1, L2      ; Generate unique labels
    LOAD A
    LOAD B
    SUB
    JN   L1           ; Jump when negative
    LOAD A
    GOTO L2
L1  LOAD B
L2  STORE C
ENDMACRO
```

Programmers soon realized that by adding a high-level macro-time language, they could control the macro expansion by inspecting the actual parameters. As a result, powerful high-level macro-time languages were created, with their own variables, arithmetic operations and control structures. This enabled, among other things, the practice of *conditional compilation*, where certain pieces of source text are either omitted or retained based on the value of a conditional.

A different development was that the matching process became more flexible. Instead of defining new assembly operations, programmers could design their own high-level statements. With this functionality, programmers could almost create a complete high-level language using powerful macros with assembly language as the base language.

When compiler-based high-level languages (like Algol and Fortran) gained in audience, interest in macro systems to extend assembly languages with higher level concepts diminished. However, macro facilities are also useful for high-level languages, for introducing new types of statements or expressions for example. As a result, powerful stand-alone macro systems were introduced (like ML/I), that could be used as a *preprocessor* for any high-level language. Some high-level languages even had their own specialized preprocessor (like PL/I and later C).

We will use textual macro processors for high-level languages as the basis for the rest of this discussion, and leave assembly language behind. A coverage of macro systems for assembly languages, can be found in chapters one through five of [2]. The above examples were based upon the material presented there.

2.2 Workings

A textual macro processor is basically a text transformer. The rules for the transformation (the actual macros) are usually stated in the source text itself. These rules consist of two parts: a pattern and a body. The macro processor's task is to scan the text for occurrences of patterns of known macros, and substitute instantiated bodies for the found patterns. When a new macro definition is found, it's added to the collection of known macros, until it goes out of scope. There are several different algorithms to do macro processing, we will describe two of them:

Scheme 1 Walk through the source text from the beginning to the end, and do the following:

1. When recognizing a new macro definition, add it to the *environment*, which represents the known macro definitions.
2. When recognizing a pattern of one of the known macros:
 - (a) Instantiate the body of the macro with the actual parameters.
 - (b) Substitute the instantiated body for the macro call.
 - (c) Continue the algorithm at the beginning of the expansion.

Scheme 2 Walk through the source text from the beginning to the end, and do the following:

1. When recognizing a new macro definition, add it to the *environment*, which represents the known macro definitions.
2. When recognizing a pattern of one of the known macros:
 - (a) Recursively call the algorithm on the actual parameters, to obtain fully macro-expanded actual parameters in the base language.
 - (b) Instantiate the body of the macro with the fully macro-expanded actual parameters.
 - (c) Recursively call this procedure on the instantiated body, to obtain a fully macro-expanded body in the base language.
 - (d) Substitute the instantiated body for the macro call.
 - (e) Continue the algorithm at the end of the macro expansion.

The essential difference here is whether we recursively process the macro expansion in isolation or after substituting it into the source text. We arrive at two more, slightly different schemes, by changing Scheme 1 to recursively process macro arguments before substituting them into the macro body, or changing Scheme 2 not to do that. We won't consider those options any further though, as Scheme 1 and Scheme 2 as presented are closest to the actual macro systems we examine in this essay.

The environment is necessary to implement *scoping rules*. The scoping rules usually include that macro definitions can be nested, and that they can be redefined. The former can be done by deleting macro definitions from the environment when they go out of scope (this is more difficult in the first scheme than in the second). The latter can be done by allowing the same pattern to occur more than once, and scan from new to old definitions.

The pattern matching can be done in several ways, depending on how flexible it has to be. Some systems use a rigid syntax that looks much like procedure definitions and calls in ordinary programming languages. But there are also systems that use powerful regular-expression-like patterns.

2.3 Analysis

We will use the C preprocessor (often called `cpp`) in our analysis of textual macros. The reason for this is that it is by far the most popular textual macro system still in use for a programming language. Even derivatives such as C++ still use it.

2.3.1 C macros

The C preprocessor is a custom tailored program for the C programming language, but it could also be used for other programming languages that have some superficial similarities with C. In that case, its output would not go to a C compiler, but to a text file that is then processed by a compiler for another language.

On a C program the C preprocessor does everything that needs to be done before syntactic analysis. It combines a lexical scanner, a macro processor, and several other things in one program. Its functionality is standardized along with the language itself in the ANSI C standard of 1999 (see [11]). The steps it needs to perform are standardized as well:

1. Read the file line by line into memory.
2. Replace trigraphs with their corresponding symbols.
3. Join lines ending with a backslash with the next line.
4. Tokenize the input into preprocessor tokens and spaces. Comments (`/* */` and `//`) are replaced by a single space token in this process.
5. Do the real work, that means performing the following steps simultaneously:
 - Execute preprocessing directives (starting with a `#` sign). When an `#include` directive is found, execute all the steps until this one recursively on the named file, and substitute the resulting token stream for the `#include` directive.
 - Expand macro calls (using Scheme 2 of the previous section).
 - Execute `_Pragma` unary operator expressions.
6. Delete all remaining preprocessing directives.
7. Replace escape sequences in character constants and string literals by their corresponding symbol.
8. Concatenate adjacent string literal tokens.
9. Convert preprocessing tokens to compiler tokens, deleting whitespace.

Macros are defined by a `#define` directive. The first parameter is the pattern, the second the body: a very simple macro would be to replace the identifier `pi` with some approximation of π :

```
#define pi 3.14159
```

Any occurrence of the identifier `pi` will, from this declaration on, be replaced by the defined value. Now it is clear why knowledge about the programming language is useful for a (textual) macro processor. It should only expand the identifier (as defined by the programming language) `pi`, and not the word “pi” appearing in a string literal like `"What is pi again?"` or as a substring in the name of an identifier like `pizza`.

The above example is what C calls an *object-like macro*, because its pattern is just a simple data object. C also supports a mechanism for a function definition like syntax, with arguments separated by commas between parentheses. A macro belonging to that category is called a *function-like macro*. An example of the latter is a macro defining a `max(a, b)` operation:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

Redefining a macro is not allowed in C, unless it's exactly the same macro definition. What *is* allowed, is undefining a macro by using the `#undef` directive, followed by a new definition of the same identifier. For example, after the following directive is found, no further replacements of `pi` shall be made:

```
#undef pi
```

Recursive calls are ignored. So if the expansion of a call to a macro named `foo` contains a new macro call to `foo`, that new macro call is not expanded any further but is left verbatim in the macro expansion.

2.3.2 Operator priority problems

To save parentheses, mathematicians have invented operator priority rules. A multiplication operator (`*`) should always have a higher priority than an addition operator (`+`) for example. Most programming languages have copied this priority system to make expressions more readable. This however gives rise to semantic problems when using macros. Consider the following macro definition:

```
#define mul(a,b) a*b
```

This works fine as long as we call it with subexpressions in which the main operators are of a higher priority than `*`. However, when we don't do this (and it is reasonable that we don't), the `*` may swallow operands that should belong to other operators. Consider the following macro call and its expansion:

```
mul(w+x,y+z)
w+x*y+z
```

`x*y` is evaluated before the addition operators. To solve this problem, a macro body should use parentheses wherever such errors might occur. But even the following definition is not correct:

```
#define mul(a,b) (a)*(b)
```

At the place where this macro is called, it might be surrounded by operators of even higher priority, which would steal away operands from the multiplication operator. Consider the following macro call and expansion:

```
~mul(x,y)
~(x)*(y)
```

Here the bitwise complement is applied before the multiplication. A correct definition for the `mul`-macro would be:

```
#define mul(a,b) ((a)*(b))
```

2.3.3 Variable capture

Variable Capture occurs when a macro body contains temporary variables which have the same name as one of the variables in the actual parameters. Consider the following example:

```
#define swap(a,b) { typeof(a) temp = a; a = b; b = temp; }
```

This macro works fine,¹ as long as none of its parameters is the variable named `temp`. Consider the following macro call and its expansion:

```
swap(x,temp);
{ typeof(x) temp = x; x = temp; temp = temp; };
```

A solution would be to let the macro processor generate unique identifiers for temporary variables. Unfortunately the C preprocessor doesn't support this. The only "solution" is to use unlikely identifiers, like ones whose names end with an underscore:

```
#define swap(a,b) { typeof(a) a_ = a; a = b; b = a_; }
```

2.3.4 Multiple evaluation

A subtle problem that only shows up in high-level languages which allow side effects, is the problem of multiple evaluation of expressions with side effects. A programmer that calls the macro expects, for example, that the expression is evaluated only once, while the macro definition actually evaluates it multiple times. The same holds for statements, but in practice that is not so much of a problem, because programmers know very well that every statement has side effects. An example of this problem is the `square` macro:

```
#define square(x) ((x)*(x))
```

As long as it is called with expressions without side effects, it will product the correct result. However, when the expression has side effects, there is a possibility that the wrong result is computed. Now consider the following macro call and expansion:

```
square(readint(f));
((readint(f))*(readint(f)));
```

In which `readint(f)` is a function which returns the next integer from file `f`. The expansion shows that the semantics of the generated piece of code is clearly not what was meant. This case is solved by introducing a new temporary variable ²:

¹The macro uses the non-standard `typeof` operator. It is supported by several C and C++ compilers, but if it is not available, a simple `int` or `double` will do. The macro will then just be less reusable.

²The `{...}` construct is a non-standard extension supported by the GNU Compiler Collection (see [13]) and others, to make a compound statement that serves as an expression, returning the value of the last expression

```
#define square(x) ({ typeof(x) x_ = x; (x_)*(x_); })
```

In this case, there is no problem, because both programmers know what a square function should do. However, in more complicated cases, this might result in misunderstandings between the programmer creating the macro, and the one calling it. The programmer creating the macro should make clear what the evaluation rules of the macro are to anyone calling it. In the case of `square` these are probably obvious, but in the general case a comment would not be misplaced.

2.3.5 Resolving errors

Error messages that the compiler generate as a result of a piece of malformed code, are difficult to relate to the original source text, because the compiler works on the fully macroexpanded text. So programming errors in macro definitions which results in malformed code, are harder to find than ordinary errors.

Run-time errors are a more subtle version of this problem that occurs when a macro expansion generates code which is correct as seen from the compiler, but does the wrong thing. Such errors will show up only at run-time and will be even more difficult to track back to the responsible code in the source text.

2.4 Examples

The C macro definitions we have seen thus far are quite simple. In fact, the majority of macro definitions used in real programs are simple ones, like the ones for conditional compilation and constant declaration. More complex macros are quite difficult to write, or not possible at all. In this section however, we will discuss some examples of useful, more complex macros.

We could, for example, write a `foreach` macro, which does something for every element of a linked list. Because C doesn't have a linked list type built-in, we will start by defining one:

```
typedef struct node {
    int data;
    struct node *next;
} list_node;
```

Now, we would define the `foreach` construction in terms of the built-in `for` construction as follows:

```
#define foreach(list) \
    for (list_node* node_ = list; node_ != 0; node_ = node_->next)
```

The `foreach` construction is a substitute for a header: it is an *open ended* macro, which expands in an illegal form, if calls are not directly followed by a body (a statement). For example, to print every element in the list `mylist`, we could write:

```
foreach(mylist) {  
    printf("%d\n", node_>data);  
}
```

The problem in this example is that the variable `node_` leaks from the macro definition. To abstract this variable away, we could write another macro that functions as a substitute:

```
#define current node_>data
```

We can now print the list, using this macro call:

```
foreach(mylist) {  
    printf("%d\n", current);  
}
```

3 Syntactic macros

3.1 History

The history of syntactic macros seems to have started somewhere in the 1960's. One of the earliest papers on the subject is from 1966 [1], wherein B.M. Leavenworth describes a simple macro system for Algol-like languages that is somewhat more high-level than textual macros. This development was in response to a perception of the deficiencies of textual macros.

As early as 1963 a crude form of syntactic macros made its way into LISP 1.5 as described in [9]. This however was not in response to inadequacies of textual macros, but was meant to cleanup the syntactic code generation facilities LISP 1.5 already possessed. This eventually evolved into the `defmacro` system now present in ANSI Common Lisp [8], making that one of the first (if not the first) standardized languages to have a syntactic macro system.

3.2 Workings

As noted earlier, syntactic macros operate on programs in the form of abstract syntax trees rather than text. This still allows at least two different macro expansion mechanisms. One would be to perform parsing first and then do macro expansion on the resulting abstract syntax tree. Another would be to do macro expansion during the construction of the abstract syntax tree. At first glance the difference between these two choices may just seem to be a matter of efficiency, but things are more subtle than that.

If we want to build a complete abstract syntax tree before macro expansion, then all macro calls will need to have a very similar structure so that they can be parsed correctly without knowing the definition of the macros yet. This is true even if we would attempt to record macro definitions during parsing, because a macro call might reasonably expand into a macro definition. So we'll either have to have a very regular structure of macro calls in such a scheme, or disallow macro defining macros. Common Lisp, and its predecessors, take the former approach because their base syntax was already very regular. For the syntactically richer languages it is more common to interleave the building of the abstract syntax tree with macro expansion so that the macro calls themselves can have a richer syntax while still allowing macro defining macros.

If we create a full abstract syntax tree first, macro expansion would work as follows.

Scheme 1 After the abstract syntax tree is created, it is traversed in depth-first order, and for each node it is determined in which of the following cases it falls.

1. It's a macro definition. In that case, we record the definition.
2. It's a macro call as described by one of the recorded macro definitions. In that case, replace the node with a node resulting from substituting the arguments of the call in the macro's expansion. Then continue the algorithm on that newly created node.

3. It's base syntax. In that case, the processing of the current node is now done. The algorithm now proceeds to process any or all of the node's children, depending on the syntactic category of said node. For example, if the node describes a block introducing local variables, the forms describing the values of the new variables should be processed, but the variable names shouldn't. Also, the body of the block should be processed, but with an environment extended with the new variables.

Alternatively, we could recursively macroexpand the arguments in step 2 before substituting them, similarly to the way the C preprocessor works. That option will not be considered here any further.

When we interleave the creation of an abstract syntax tree with macro expansion, we arrive at a slightly different approach.

Scheme 2 Given the token stream created by the tokenizer:

1. Create a parser for the context free grammar of the base syntax with added macro definition syntax.
2. Start parsing the token stream.
3. When a macro definition is encountered, extend the grammar with new grammar rules for the macro, update the parser for this new grammar, and continue parsing.

Such an approach has the advantage that the syntax of macro calls can be more elaborate. The approach is also language neutral: to accommodate a new programming language it will just need to be fed a different starting grammar.

3.3 Analysis

Syntax macros are in principle not as powerful as textual macros because every abstract syntax tree should correspond to some textual representation but not vice versa. However, we'll see that syntax macros can be significantly easier to use than textual macros. We will do this with some examples of syntactic macros as supported by ANSI Common Lisp. [8].

3.3.1 Common Lisp macros

Common Lisp macros are defined with `defmacro`. Its first argument is the name of the macro being defined, its second argument the formal parameter list of the macro, and its remaining arguments describe the macro expansion. Here is a (trivial) example:

```
(defmacro div (a b)
  `(/ ,a ,b))
```

This defines a macro named `div` which accepts two arguments. Its macro expansion is defined with simple template filling, indicated by the backquote (```). Everything inside the backquoted form will appear verbatim in the macro expansion unless it is preceded by a comma (`,`). Something preceded by a comma causes its *value* to be substituted into the expansion. So in this example, the macro expansion will be `(/ 3 4)` when `a=3` and `b=4`. Such a macro call looks like `(div 3 4)`, so a Common Lisp macro call looks exactly like a Common Lisp function call: the name of the function or macro, followed by any arguments separated by spaces, all enclosed in parentheses.

3.3.2 Advantages

This macro is of course utterly useless, as it's just a poor replica of the built-in division function, but there is already a small advantage over textual macros here. The operator priority problems described in subsection 2.3.2 simply cannot occur, for example. There arguments were passed in as strings, and the macro writer had to use parentheses liberally to preserve proper operator priority behavior. Here the arguments are passed in as abstract syntax trees, which know no parentheses or operator priority.

Another (related) advantage is that there doesn't need to be a separator between macro arguments. Just the knowledge of the syntactic categories is enough to make the parser do the right thing. In Common Lisp this knowledge is implicit because macro arguments are all *s*-expressions, which are parsed exactly alike. In a more syntax-rich language macro definitions are slightly more verbose, as it is necessary to state the syntactic categories of macro arguments explicitly, as well as the syntactic category of the macro's expansion. Nevertheless, the advantage in the *usage* of such macros remains.

A more important advantage is that errors in the use of syntactic macros are more easily correlated with the actual source code. Because the arguments to macros are parsed before (or during) macro expansion, any syntax errors in them can be reported before the macro is actually expanded.

3.3.3 Variable capture

Despite their advantages, syntactic macros as described above do not solve all the problems of textual macro systems. Let's look at a slightly more useful example:

```
(defmacro swap (a b)
  `(let ((temp ,a))
     (setq ,a ,b)
     (setq ,b temp)))
```

Common Lisp's `let` operator introduces new local variables and `setq` is the assignment operator, so this defines a simple swap using a temporary variable. When called as `(swap x y)` we get the macro expansion:

```
(let ((temp x))
  (setq x y)
  (setq y temp))
```

which does what we would expect. However, when called as `(swap foo temp)`, we get:

```
(let ((temp foo))
  (setq foo temp)
  (setq temp temp))
```

Here the temporary variable introduced by the macro expansion shadows the variable that the user passed in as an argument because they have the same name. As a result, the macro will not do the right thing.

A non-solution is to use more “unlikely” temporary variable names when writing macros. A true solution would be to use temporary variable names that are guaranteed to be unique. In Common Lisp such a variable name can be created with the `gensym` function. We can thus fix our `swap` macro as follows:

```
(defmacro swap (a b)
  (let ((temp (gensym)))
    `(let ((,temp ,a))
      (setq ,a ,b)
      (setq ,b ,temp))))
```

Here we create a local variable named `temp` inside the macro definition’s body whose value is a unique variable name. That name is then substituted in the backquote template through `,temp`. Now the macro call `(swap foo temp)` will expand into:

```
(let ((#:g1 foo))
  (setq foo temp)
  (setq temp #:g1))
```

where `#:g1` indicates a variable name that can not be forged by the programmer in any way.

Hence, through proper discipline Common Lisp macros can be written so that they do not unintentionally capture the programmer’s variables. As will be discussed later on, it is also possible to change the macro system so that this problem doesn’t exist in the first place. It must be noted though, that variable capture is not always unintentional. So called *anaphoric macros* [5] can occasionally be very useful, for example. Consider the following:

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
    (if it
        ,then-form
        ,else-form)))
```

This defines an anaphoric variant of the standard `if` expression named `aif`. It takes a “test-form”, a “then-form”, and an optional “else-form”. It has the effect of evaluating the test-form and binding that value to a local variable named `it`. If that value is true, the then-form is evaluated, otherwise the else-form is evaluated. Because the then- and else-form are placed inside the `let`, they can access the value of the test-form through `it`. This is useful in any language that has multiple values that can be interpreted as true or false. Such languages include Common Lisp, C, C++, and Python, to name a few.

3.3.4 Free identifiers

Another problem that syntactic macros “inherit” from textual macros is that of the scope of free identifiers in macro expansions. Consider the following (contrived) code using the `swap` macro of the last subsection:

```
(flet ((setq (set-a set-x)
        (print (union set-a set-b))))
  (let ((set1 '(1 2 3))
        (set2 '(2 3 4)))
    (setq set1 set2)
    (swap set1 set2)
    (setq set1 set2)))
```

This defines a local function named `setq` that prints the union of two sets. In that scope two local variables are declared holding sets. The local `setq` function is called on the two sets, then the variables are intended to be swapped, and the `setq` function called again. What will actually happen is that the `setq` function is called four times and nothing is swapped! That is because the macro expansion contains `setq` (and `let`) as a free identifier that get substituted in the macro expansion verbatim.

This problem seems similar to the one discussed in the previous subsection, but is more serious because the macro writer cannot prevent it! Because of this it seems paradoxical that Common Lisp’s macro system can be effectively used at all in large software development projects. However Common Lisp has some other distinguishing features that alleviate the problem somewhat.

First of all, identifiers reside in *packages*. If a macro and the objects denoted by the free identifiers in its macro expansion are all defined in a package A, the use of that macro in package B will not have the discussed problem. Further more, variables and functions live in separate namespaces in Common Lisp. So even if a macro is used in the same package it is defined in, the problem will rarely if ever occur. Local functions or macros with the same name as a global function or macro are hardly ever written, and even so will not interfere with variables of the same name.

There is another approach the designer of a syntactic macro system could take and this will get rid of the problem, and that of the previous subsection altogether. This will be discussed in the next section.

3.4 Lexical scoping

The two problems discussed in the previous section have a common cause: macro expansion doesn't respect *lexical scoping* rules. It doesn't have to be this way however. We can change both the schemes presented in section 3.2 to respect lexical scoping rules.

First we must record the environment e in which a macro is defined along with its definition. Next, whenever a call to the macro is processed, a new environment e' is created that is an extension of e , and the macro's expansion is inserted as if it had been written in that environment e' . In this way, free identifiers in the macro expansion refer to objects in the environment in which the macro was defined, and identifiers introduced by the macro expansion cannot capture variables of the same name from elsewhere, not even from a nested call of the same macro. Making this change to scheme 1 from section 3.2 is straightforward:

Scheme 1' After the abstract syntax tree is created, it is traversed in depth-first order, and for each node it is determined in which of the following cases it falls.

1. It's a macro definition. In that case, we record the definition along with the current environment.
2. It's a macro call as described by one of the recorded macro definitions. In that case, create a new environment e' that is an extension of the environment e stored with the macro definition. Then replace the current node with the node that would result from writing the macro's expansion in environment e' . Continue the algorithm on this newly created node.
3. It's base syntax. In that case, the processing of the current node is now done. The algorithm now proceeds to process any or all of the node's children, depending on the syntactic category of said node. For example, if the node describes a block introducing local variables, the forms describing the values of the new variables should be processed, but the variable names shouldn't. Also, the body of the block should be processed, but with an environment extended with the new variables.

A scheme like this could be used in the implementation of the Scheme programming language [10] and its macro system(s).

The adaptation of Scheme 2 is a bit more involved. To record environment information with macro definitions we'll have to associate environments with pieces of grammar.

Scheme 2' Given the token stream created by the tokenizer:

1. Associate a top-level environment with the context free grammar of the base syntax with added macro definition syntax. The grammar will need to have at least 3 non-terminals for identifiers to indicate that it is a binding occurrence, an applied occurrence, or an identifier that does not participate in block structure at all. ([7] calls these Binding, Var, and Label, respectively.) Create a parser for this grammar.

2. Start parsing the token stream. When grammar rules are applied, new environments are created inheriting from the environment up to that point. Environments can get enriched by occurrences of Binding identifiers.
3. When a macro definition is encountered, extend the grammar with new grammar rules for the macro and associate them with the current environment. Update the parser for this new grammar, and continue parsing.

The above is just a rough sketch of the scheme presented in [7] and we refer to that paper for more details.

3.5 Remaining issues

Even with a syntactic macro system with lexical scope, there remain some issues for macro writers to be aware of. Again, we'll discuss these with some examples. Rather than introducing a macro system with lexical scope (such as Scheme's [10]) we'll save space by pretending that Common Lisp's macro system does respect lexical scoping rules.

One remaining problem is that of unintended multiple evaluations. This problem cannot not just be eliminated though, because multiple evaluations are sometimes needed. When we write a *for*-loop as a macro on top of, say, `goto` and `labels`, we'll want the body to be evaluated multiple times. That's the entire point of the macro! So, as hinted at before, the evaluation semantics of a macro should be part of its interface; the number of times each argument is evaluated, as well as the order in which these evaluations should occur. The authors are not currently aware of a macro system that formalizes this idea, but discussing the evaluation rules in a macro's documentation is usually adequate.

A more serious problem is that of getting compile-time or run-time error messages in terms of the expanded source code instead of the actual source code typed in by the programmer. We noted in section 3.3.2 that this problem is diminished for syntactic macros because macro arguments are parsed, but it still exists. For example, if we pass the `div` macro defined in subsection 3.3.1 a non-numeric argument as in `(div "foo" "bar")`, the resulting type error will inevitably be in terms of `/` instead of `div`. If we used a statically typed language instead of Common Lisp, we might be able to fix this by checking the types of the expressions passed as arguments to `div`. However that will still not catch the error in something like `(div 15 0)`. The only real solution (if we want error messages in terms of the actual source code) is to place (run-time) checks in the macro expansion:

```
(defmacro div (a b)
  '(let ((tempa ,a) (tempb ,b))
      (if (and (typep tempa 'number) (typep tempb 'number)
              (/= tempb 0))
          (/ tempa tempb)
          (error "Both arguments to div should be numbers and the second ~
argument should be non-zero: ~S ~S" tempa tempb))))
```

Without going into details, it is clear that our simple one-line macro has grown immensely to supply correct error messages. (Without lexical scoping the code would be one line longer still.) In practice therefore, macros do only the most basic error checking. If a programmer happens to make an error not covered by those checks, the error message will be in terms of the expanded code, and not the code he or she typed in. To lessen the programmer's pain, usually some way is provided to look at the expanded code to track down such errors.

Finally, with the syntactic macro systems described here it is not possible to do the kind of conditional compilation mentioned in section 2.1. It might be possible to change Scheme 2 or 2' in such a way that this becomes possible again, but certainly not Scheme 1 or 1': conditional compilation has to be done before the full abstract syntax has been constructed, that's the whole point!

3.6 Some final examples

We'll conclude this section with somewhat more useful examples of what syntactic macro systems are capable of.

Syntactic macros can be used to introduce new control structures into a language. One of the most useful kinds of control structures in imperative languages is iteration, and hence many iterative control structures have been proposed. In a language with a syntactic macro system, it is not necessary to provide all or even many of them, because users can define them for themselves, in terms of more low-level iterative control structures. One could define a for-statement in terms of jumps and labels, for example. In Common Lisp that would look as follows:

```
(defmacro for ((var init limit &optional (step 1)) &body body)
  (let ((loop (gensym)) (end (gensym)))
    `(let ((,var ,init))
      (tagbody
        ,loop
        (if (> ,var ,limit) (go ,end))
        ,@body
        (setq ,var (+ ,var ,step))
        (go ,loop)
        ,end))))
```

The first argument to `for` should be a list of a variable name, an initial value, a final value, and an optional step (which defaults to 1 when omitted). Apart from that, `for` can have an arbitrary number of extra arguments that will form the `body` of the for-“statement”. The expansion of `for` is based on `tagbody` which is Common Lisp's way of introducing a scope for labels that can be jumped to (with `go`). Two labels are used, and their names are created with `gensym` to avoid any conflicts. We could use `for` like this:

```
(for (x 1 5)
  (print x)) ; prints the numbers 1, 2, 3, 4, 5
```

```
(for (x 0 50 10)
  (print x)) ; prints the numbers 0, 10, 20, 30, 40, 50
```

If so desired, the definition could be altered slightly so we would have to write:

```
(for (x = 1 to 5)
  (print x)) ; prints the numbers 1, 2, 3, 4, 5
```

```
(for (x = 0 to 50 by 10)
  (print x)) ; prints the numbers 0, 10, 20, 30, 40, 50
```

In a similar fashion we could define a while-“statement”, to be used like this:

```
(while (input-available)
  (print (read-input)))
```

And something like Haskell’s list comprehensions, to be used like this:

```
(collect (* x x) for x = 1 to 10 if (prime x))
```

Etcetera, etcetera. All perfectly achievable and reasonable goals. At some point however, such exercises may become repetitive. One might then attempt a more ambitious macro that encompasses all these iteration macros. Indeed, several such attempts have been made in the Common Lisp community, and one of them (the `loop` macro) made it into the ANSI standard. With it, the above examples could be written as follows:

```
(loop for x from 1 to 5
  do (print x))
```

```
(loop for x from 0 to 50 by 10
  do (print x))
```

```
(loop while (input-available)
  do (print (read-input)))
```

```
(loop for x from 1 to 10
  if (prime x)
  collect (* x x))
```

`loop`’s definition is too large and complex for the scope of this essay, but the point is that it and similar macros *can* be written by a normal programmer, so there is no need for programmers to beg the language designers to put their favorite iteration construct in the next standard. In a sense, macro systems bring the power of language design to the ordinary programmer.

4 Concluding remarks

We hope to have given you some insight into some of the advances made in macro systems over the past decades. We compared and contrasted several different approaches and existing systems, their workings and their uses. We would like to conclude by saying that we have merely scratched the surface of a very vast topic.

We have only given very few, very simple examples using the physical macro systems we discussed. To see what the C preprocessor is capable of, the interested reader is encouraged to check out the free Boost.Preprocessor library [12]. In the Common Lisp examples we have only done simple template filling, but it is actually possible to use the full Common Lisp language to compute a macro expansion from macro arguments. The book *On Lisp* [5] is not only a tutorial on Common Lisp and its macro system, but also gives a good overview of what such a macro system is capable of.

Also, we haven't discussed the possibility of multiple macro systems for a single programming language. Because the macro systems illustrated in figure 1 operate in separate stages of compilation, it would be possible to have them all in a single programming language. Indeed, Common Lisp doesn't just have the syntactic macro system (based on s-expressions) discussed above, it also has a textual macro system (based on characters). Similarly, [6] describes a syntactic macro system for C and such a system *could* be used in addition of the C preprocessor, rather than in place of it.

Of the four types of macro systems illustrated in figure 1 we have only discussed textual and syntactic macros. Another very interesting type of macros is described in [4]. Such *semantic macros* can process context information, making macro expansion a more non-local process. Chapter 1.10 of [3] briefly mentions *computation macros*, and hints at further reading material on them.

Lastly, the lines between these 4 types of macro systems are not always very clear. Common Lisp's semantics are so simple, and its syntactic macro system so powerful, that it is possible to mimic many semantic macros. A powerful enough textual macro system could do parsing and gain some of the benefits of syntactic macros.

References

- [1] B.M. Leavenworth, *Syntax macros and extended translations*, Communications of the ACM, 9(11):790-793, 1966. 35
- [2] M. Campbell-Kelly: *An Introduction to Macros*, ISBN: 0 356 04388 6, Macdonald & Co, 1973
- [3] P.J. Brown, *Macro processors and techniques for portable software*, ISBN: 0 471 11005 1, Wiley, January 1974.
- [4] W. Maddox, *Semantically-sensitive macroprocessing*, Master's thesis, The University of California at Berkeley, Computer Science Division (EECS), Berkeley, CA 94720, December 1989.
- [5] P. Graham, *On Lisp*, ISBN: 0130305529, Prentice Hall, 1993. Available online at: <http://www.paulgraham.com/onlisp.html>
- [6] D. Weise, R.F. Crew, *Programmable Syntax Macros*, ACM SIGPLAN Notices, 1993, 28, (6), pp. 156-165. Available online at: <http://citeseer.ist.psu.edu/weise93programmable.html>
- [7] L. Cardelli, F. Matthes, M. Abadi, *Extensible Syntax Macros with Lexical Scoping*, SRC Research Report 121, Digital Equipment Corporation Systems Research Center, 21 February 1994. Available online at: <http://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/abstracts/src-rr-121.html>
- [8] ANSI/INCITS X3.226-1994, *American National Standard for Information Systems - Programming Language - Common Lisp*, 1994. An unofficial HTML version is available online at: <http://www.lispworks.com/reference/HyperSpec/Front/index.htm>
- [9] G.L. Steele Jr., R.P. Gabriel, *The Evolution of Lisp*, published in: T.J. Bergin, R.G. Gibson, R.G. Gibson Jr., *History of Programming Languages, Vol. 2*, ISBN: 0201895021, Addison-Wesley Professional, 12 February 1996. Available online at: <http://www.dreamsongs.com/NewFiles/Hopl2.pdf>
- [10] R. Kelsey, W. Clinger, J. Rees (eds.), *Revised⁵ Report on the Algorithmic Language Scheme*, Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998. Available online at: <http://www.schemers.org/Documents/Standards/R5RS/>
- [11] ISO/IEC 9899 Second Edition 1999-12-01 *Programming Languages - C*
- [12] V. Karvonen, P. Mensonides, *The Boost Library Preprocessor Subset for C/C++*, Available online at: <http://www.boost.org/libs/preprocessor/doc/index.html>
- [13] Free Software Foundation *GCC 3.4.3 manual*, Available online at: <http://gcc.gnu.org/onlinedocs/>