D-Expressions: Lisp Power, Dylan Style

Jonathan Bachrach
Artificial Intelligence Laboratory
Massachussetts Institute of Technology
Cambridge, MA 02139
jrb@ai.mit.edu

Keith Playford
Functional Objects, Inc.
86 Chandler Street
Somerville, MA 02144
keith@functionalobjects.com

1 Abstract

This paper aims to demonstrate that it is possible for a language with a rich, conventional syntax to provide Lisp-style macro power and simplicity. We describe a macro system and syntax manipulation toolkit designed for the Dylan programming language that meets, and in some areas exceeds, this standard. The debt to Lisp is great, however, since although Dylan has a conventional algebraic syntax, the approach taken to describe and represent that syntax is distinctly Lisp-like in philosophy.

2 Introduction

The ability to extend a programming language with new constructs is a valuable tool. With it, system designers can grow a language towards their problem domain and enhance productivity and ease of maintenance. A *macro system* provides this capability in a portable, high-level fashion by allowing new constructs to be implemented in terms of existing ones via programmer-defined source-to-source transformations.

Beyond the above, the ability to read, write, and easily manipulate the syntax of a language from within that language can be especially powerful. It can allow the full language to be brought to bear when implementing macros. It can provide a convenient means of saving and restoring structured data in text form. It can form the basis of code analysis tools and be the starting point for experiments with new language processors or into modified language semantics.

2.1 Successes and Failures

Lisp is the only language family (cf [9], [10]) that has succeeded in providing integrated macro systems along with simple and powerful syntax manipulation tools like these. They are considered one of Lisp's unique strengths, perhaps even the most important and distinctive feature of the language. But the key to their viability in Lisp is the simplicity and regularity of its syntax. Recognizing their utility, attempts have been made to provide powerful macro facilities in languages with more conventional syntaxes like those of C or Pascal, but in comparison with what Lisp provides, the solutions have been restrictive, difficult to explain and use, or both. None have been standardized. Further, the utility of syntax manipulation tools independent of the macroexpansion process is typically forgotten.

2.2 Lisp Power, Dylan Style

This paper aims finally to demonstrate that it is possible for a language with a richer, more conventional syntax to provide Lisp-style macro power and simplicity. We describe a macro system and syntax manipulation toolkit designed for the Dylan programming language (cf [13], [7]) that meets our goals as well as, if not better than, those commonly found in Lisps. The debt to Lisp is great, however, since although Dylan has a conventional algebraic syntax, the approach taken to describe and represent its syntax is distinctly Lisp-like in philosophy.

Taking Dylan's already capable rule-based pattern matching and template substitution macro system as a starting point, we explore its underpinnings. Within a broader comparison with other styles, we relate Dylan's approach to the Lisp model and show how their common basis can (and has, in other contexts) be applied to a family of languages whose syntax is based on what we term a *skeleton syntax tree*. We go on to describe a library which provides a code representation (*d-expressions*), source-level pattern matching and code construction utilities, and source code IO. Some models of compile-time evaluation are proposed that are suitable for Dylan, a language in which compiler and run-time are separated. Finally, putting the d-expressions library together with a compile-time evaluation model leads us to a natural procedural macro extension of Dylan's rule-based macro system, one comparable to those available in Lisp.

2.3 Requirements and Goals

A macro facility provides a mechanism to extend a base syntax. Such a facility can be measured along a set of axes, covering such aspects as power, generality, reliability, and ease of use.

Macro facilities differ greatly in their ability to perform syntactic transformations. Their power can be judged by the complexity of their underlying syntactic representations and the extent of their programmability. Macro systems have used input representations such as characters, tokens, syntax, and semantic information, with the latter leading to

more powerful transformations than the impoverished former. First, the underlying syntax representation affects the ability of the macro system to maintain source locations. During debugging, a user should not have to step through macro expanded cruft, no more than they should have to step through code obfuscated by optimizations. Second, the syntax representation determines the ease with which syntax can be manipulated. Ultimately, in order to perform powerful transformations, the underlying representation needs to be at least rich enough to represent recursive structures such as expressions.

A macro facility's programmability is a function of the strength of its transformational engine, starting from a purely substitution system (e.g., the C Preprocessor [11]) all the way up to a full programming language. In between, some macro systems augment substitution with support for repetition and conditionals. The more powerful the transformational engine, the more expressive the macro facility will be. A good test of a macro facility is the extent to which it can represent the base language on which it was built. This helps determine whether a language can be seamlessly extended or whether macro defined constructs will be confined to a limited syntactic space.

A powerful macro facility should provide an ability to read and write syntactic elements. Input facilities are obviously crucial for writing a file compiler, while output facilities are useful for warnings and debugging. Ideally, a macro facility could be provided as a separate library upon which to base compilers and code analysis tools.

Another important axis along which to measure a macro facility is its reliability. Will it produce syntactically correct output and produce correct results in all cases? Can a macro introduce a syntax error by way of an admissible usage? In other words, users should only see syntax errors from the actual code they write. Can syntactic constituents interfere with each other? In some systems (e.g., C Preprocessor [11]) extra parentheses are required to prevent unwanted operator precedence bugs.

From an ease of use point of view as well as a reliability perspective, it is important that a macro facility ensures that variable references copied from a macro call and from a macro definition mean the same thing in an expansion. In particular, the macro system should (1) avoid accidental collisions of macro variables with program variables of the same name regardless of the context in which a given macro is used and (2) maintain consistency when macro variables refer to identically named variables introduced within a given macro definition. (1) is generally referred to as *hygiene* [12] and (2) is generally referred to as *referential transparency*. A violation of either is called a *variable capture error*. Variable capture errors create subtle bugs that are difficult to debug. Much has been written about how to avoid introducing these sorts of errors [8], but we maintain that there is no reason to introduce them in the first place.

Other aspects that improve usability are intuitiveness of a macro's definition and ease with which it's debugged. We assert that a macro facility should enable a programmer to write down what they want the input and output to look like augmented only by substitution parameters. The alternative of constructing and manipulating representations manually or specifying productions in terms of Abstract Syntax Tree classes is too tedious and error prone requiring intimate knowledge of an underlying syntactic representation. Debugging shift/reduce errors is a daunting task for even the best programmers. Furthermore, some form of a trace of the transformation engine and selected expansions is a great help to macro writers.

3 An Overview of Syntax Representations

We split the space of syntax representations into four broad categories: text based, token based, abstract syntax trees, and skeleton syntax trees. Each one in considered in turn in the following sections and evaluated with respect to the requirements set out above.

3.1 Text-Based Representations

The most basic representation of a piece of code is its original text. Such unstructured text is difficult to analyse and manipulate, and the simple string-based pattern matching tools often used are neither syntax aware nor context aware. Even simple things like matching only code, as opposed to text contained within strings or comments, can be problematical.

Repeated application of a full parser, if available, is expensive and may not be possible on subexpressions taken out of context. Also, unless auxiliary data is maintained somehow, source location information is lost by transformations.

Because of these problem and others, a text-based representation clearly does not meet our requirements.

3.2 Token-Based Representations

In a token-based representation, code is manipulated as a sequence of tokens as generated by a lexer. This has a number of benefits over straight text. For example, the lexer typically strips out comments, leaving only code. Also, the problem of accidental token merging when constructing new phrases can be eliminated. The C preprocessor [11] and Pop-11 [1] are examples of languages offering a token-based macro system.

If each token is represented by a structure rather than a simple string, it is possible to record its classification (e.g. literal string, identifier, punctuation) and source location along with it. This information can be maintained through source-to-source transformations.

Processing an unstructured token stream is not very convenient, however. If a macro takes as its arguments a fixed number of tokens following the macro name, it works correctly only when its arguments are single-token elements. If a macro needs to work in terms of expressions, the number of tokens it accepts is variable. Consider the following:

```
define token-macro increment!
    (loc1, comma, loc2) => (replacement-tokens)
  assert(comma = ",");
  concatenate(loc2, ":=", loc1, "+", delta)
end token-macro;

// OK
increment!  n, delta;

// Assertion fails, comma gets bound to "["
increment!  a[i], delta;
```

C macros and one kind of Pop-11 macro actually constrain the shape of macro input to being of function call form. This allows the parser to easily determine the extent of a macro's input and split it into comma-separated chunks before finally passing these on to the macro.

At the other extreme, macros can be given full control over the token stream. While this is the ultimate in versatility, it presents some practical problems. For example, the

extent of forms cannot be determined without performing macro expansion. This can be a problem for development environments and code analysis tools, but also for programmers. The approach lacks modularity. Token macros become, in essence, part of an ad-hoc recursive descent parser and programmers must be aware of the implications of that. They must always take care to respect any macro syntax contained within the subexpressions manipulated because those subexpressios are not isolated within nested structure.

One further problem familiar to most C programmers is inadvertent expression merging. When bringing code together within a macro or when returning replacement code to a macro call point, care must be taken to isolate the inserted code from its surroundings. Wrapping with parentheses where appropriate is normally sufficient to avoid, typically, operator precedence related bugs. Failing to do so is still a common source of bugs, however.

Token-based solutions can clearly be very powerful. However, we feel that the pitfalls and complexities that come with that power make them an unlikely basis for a macro system intended to be as simple and accessible as Lisp's.

3.3 Abstract Syntax Tree Representations

Abstract syntax trees (AST's) are highly structured representations of source code. In an object-oriented implementation, there would typically be one class per definition, statement, or other language construct. Down at the expression level, there might be classes corresponding to function calls, array accesses, literal references, and so on.

Typically with an AST, adding a new syntactic construct requires extending the parser to understand it and adding a new class to represent that construct. How difficult this is is related to the complexity of the parser because there is no intermediate form between tokens and full AST. Worst case, there is no special support at all for modular macro extensions, and new language constructs must be defined as a cooperating element of the full language grammar [3]. It could require extensive knowledge of the parser's intricacies to avoid or resolve problems like shift/reduce conflicts in an LALR parser, for example. We don't consider requiring such knowledge of programmers to be reasonable.

3.4 Skeleton Syntax Tree Representations

The distinguishing feature of a skeleton syntax tree (SST) is that it has few classes and corresponds to a very simple core grammar. That grammar describes "shapes" rather than particular language constructs. Many different constructs may fit within the same basic shape.

The most extreme example of a language based on an SST is, of course, Lisp. Lisp's underlying grammar says nothing about `cond` or `defun`, it simply specifies the single basic shape of all Lisp constructs. Although not a programming language, XML has much the same property.

SST's define the axioms to which all syntax must conform. In Lisp, it is, essentially, that brackets match. In XML, that tags must match. The simplicity of the axioms makes the overall structure of the language easy for people to understand and to remember and also for programs to work with.

Language-level constructs are defined modularly, above the core grammar and without requiring any changes to it. User-defined constructs have the same status as standard syntactic forms.

Referring back to a previous section, another way to look at an SST is as a lightly-structured, token-based representation. SST's have most of the power of general token-based approaches, but without many of the pitfalls discussed. The challenge is to define a simple SST upon which it is possible to build a language with a conventional algebraic syntax.

It is worth noting that although Lisp is SST-based, the SST used in most Lisps would not meet our requirements. Because standard run-time objects (numbers, symbols, lists) are reused to represent the Lisp SST, there is no reliable way to record auxiliary information like source location or hygiene context. Scheme macro systems [5] that implement automatic hygiene have to face this deficiency and work in terms of special-purpose syntax objects rather than lists and symbols.

4 Dylan's Rewrite-Rule Only Macro System

Dylan macros provide a way to define new syntactic forms. Their definitions are called macro definitions and their usages are called macro calls. At compile-time, macro expansion replaces macro calls with other constructs per their definition. This macro expansion process is repeated until the resulting expansion contains no more macro calls.

4.1 Dylan's Rewrite Rules

The macro system defined in the Dylan Reference Manual (DRM) [13] is a rewrite-rule only pattern matching template substitution system. Macros are named with module bindings using the usual Dylan namespace mechanism. Each macro is comprised of a sequence of rewrite rules where the left-hand side of a rule is a pattern that matches a fragment of code and the right-hand side of a rule is a template which is substituted for the matched fragment. Pattern variables in the left-hand side serve as macro arguments. Pattern variables appearing on the right-hand side substitute arguments into the expansion. The rules are attempted in order until a given rule's pattern is found to match, at which time, the matched fragment is replaced with the corresponding template. If no patterns are found to match, then an error is raised.

4.2 Dylan's Skeleton Syntax Tree

Before we delve into the intricacies of the rewrite-rule only macro system, it is important to explore Dylan's SST. Dylan's SST is a richer version of Lisp's s-expression based SST. Lisp's SST can roughly be described as follows:

```
form:  symbol
form:  literal
form:  "(" elements ")"

elements:  form ...
```

Dylan adds to this a few more basic shapes:

```
form:  identifier
form:  literal
form:  "(" elements ")"
form:  "define" modifiers LIST-DEFINITION elements ";"
form:  "define" modifiers BODY-DEFINITION elements "end" ";"
form:  STATEMENT elements "end" ";"

elements:  [form | punctuation ] ...
```

where `modifiers` are a sequence of adjective names. Consult the DRM [13] for the full grammar.

Although, the Dylan rule-based macro system could be made to work with more involved and stricter grammars, a skeleton syntax tree offers a number of advantages. First, an SST modularizes parsing, that is, descending structures without doing an expansion is now possible, changes in a given expansion algorithm don't require redoing a top-level parse, and syntax errors can be kept localized. Second, an SST gives the user a lot of latitude to choose an appropriate grammar for a given macro, that is, the Dylan grammar does not dictate the precise form of templates, but instead very loosely constrains their basic shape.

4.3 Source-level Pattern Matching

Now that we have described Dylan's SST we are in a position to describe the constituents of rules. Patterns appearing on the left hand side of rules provide a "what you see is what you get" (WYSIWYG) mechanism for specifying the desired input of a macro in terms of concrete syntax. A pattern basically looks like the construct that it is to match augmented by pattern variables which match and bind to appropriate parts of the construct. Pattern variables are denoted with a `?` prefixing their names. Pattern variables have constraints that restrict the syntactic type of fragments (based on the Dylan grammar) that they match when appearing in a pattern. Some of the various pattern constraints are token, name, variable, expression, and body. Constraints are denoted with a `:` separated pattern variable name's suffix.

An example pattern (forming the basis of a complement to Dylan's `unless` macro) is the following:

```
{ when (?test:expression) ?body:body end }
```

where the following would be a well-formed `when` macro call:

```
when (close?)  close(stream) end
```

Patterns match their input based on a simple matching procedure. In order to avoid easily constructed ambiguous patterns, patterns are matched using left to right processing combined with a match shortest first priority for matching wildcard pattern variables. A pattern matches if and only if all of its sub-patterns match. For more details, please consult the DRM [13] .

4.4 Source-level Code Generation

Templates appearing on the right hand side of rules provide a WYSIWYG mechanism for specifying the desired output of a macro in terms of concrete syntax. A template basically looks like the construct that it produces augmented by pattern variables that stand in for parts of the construct bound to the same pattern variables in the corresponding left hand side pattern. For example, the full macro containing the example pattern from above would be:

```
define macro when
  { when (?test:expression) ?body:body end }
    => { if (?test) ?body end if }
end macro;
```

where

```
when (close?)  close(stream) end
```

would expand into

```
if (close?)  close(stream) end if
```

with `?test` bound to `close?` and `?body` bound to `close(stream)`.

Templates do not produce a parsed output but instead produce a well formed sequence of tokens whose parsing is delayed, where well-formed means that all occurrences of Dylan brackets match. Delaying their parsing permits fewer inter-macro dependencies thereby allowing macros to be compiled independent of each other and permitting recursive macros without forward declarations.

4.5 Example Macros

We present a number of Dylan macros in order to give a sense of the range of their uses. Graham's book [8] is an excellent source for macro motivation and examples. An example of a function macro is `increment!`:

```
define macro increment!
  { increment!(?place:expression) }
    => { ?place := ?place + 1 }
  { increment!(?place:expression, ?amount:expression) }
    => { ?place := ?place + ?amount }
end macro;

increment!(x);
increment!(height(x), 10);
```

which is the Dylan equivalent of C's `++`.

An example statement macro is `with-open-file`:

```
define macro with-open-file
  { with-open-file (?stream:name, ?options:*) ?:body end }
    => { let ?stream = #f;
         block ()
           ?stream := make(<file-stream>, ?options);
           ?body
         cleanup
           ?stream & close(?stream)
         end block }

end macro;

with-open-file (stream, locator:  "phonenumbers")
  process-phone-numbers(stream);
end with-open-file;
```

where the `*` constraint on the pattern variable `?options` means that options can match any parsed fragment. This macro is typical of a whole range of macros, called `with-` macros, where resources, datastructures, or special variables must be managed (and cleanups performed) or more generally where context needs to be built.

The following define macro:

```
define macro functional-variable-definer
  { define functional-variable ?var:name = ?init:expression }
    => { define variable ?var ## "-value" = ?init;
         define method ?var ()
           ?var ## "-value"
         end method;
         define method ?var ## "-setter" (new-value)
           ?var ## "-value" := new-value;
         end method }
end macro;

?  define functional-variable time = 0;
?  time();
> 0
?  time() := 25;
> 25
```

provides a functional interface to variables, defining accessors for variables, and thus hiding implementation details. The above example demonstrates how to create new identifiers through the `##` concatenation operator.

4.6 Hygiene

Dylan's macro system is hygienic and maintains referential transparency such that variable references copied from a macro call and from a macro definition mean the same thing in an expansion. For an example of hygiene, consider the following macro:

```
define macro or
  { or(?x:expression, ?y:expression)
    => { let tmp = ?x;
         if (tmp) tmp else ?y end }
end define;
```

where a temporary binding, called `tmp`, is introduced in order to avoid multiple evaluations of the first argument, `x`. The macro system ensures that `tmp` introduced by the `or` macro does not collide with visible bindings named `tmp` in the enclosing program. For example, the following

```
begin
  let tmp = 5;
  or(1, 2);
  tmp
end
```

returns 5.

For an example of referential transparency, consider the following `increment!` macro from above. The `+` in the resulting template refers to `+` in the defining macro's namespace regardless of where the macro is used. In other words, even in the face of `+` being renamed or even not imported at all into the namespace of a macro call, the macro system ensures that `increment!`'s `+` is the same and is available.

Sometimes it is necessary to circumvent hygiene in order to permit macros to introduce variables accessible by the lexical context from which the macro was called. For example, imagine writing a loop macro that provides a standard named function, called `break`, for calling when terminating iteration:

```
define macro repeat
  { repeat ?:body end }
    => { block (?=break)
           local method repeat () ?body end;
           repeat();
         end block }
end macro;
```

A user can then write the following:

```
repeat
  if ((input := read(*standard-input*, eof:  #f)) == #f)
    break()
  else
    write(input, *standard-output*)
  end if
end repeat;
```

and have it terminate upon eof by calling the `repeat` macro's provided exit function, `break`. This mechanism can also be used for writing anaphoric macros [8].

4.7 Local Rewrite Rules

Dylan provides a mechanism for defining local named rewrite rule sets which act like local subroutines. These help in breaking a macro up into rewrite rule subproblems and permit a form of modularization. They are invoked after an initial pattern matching using the main rule set and before template substitution. Instead of backtracking, a macro is deemed invalid if none of a given local rewrite rules match.

Consider the following example:

```
define macro defaulted-variable-definer
  { define defaulted-variable ?:name ?default }
    => { define variable ?name = ?default }
default:
  { } => { #f }
  { = ?:expression } => { ?expression }
end macro;
```

Instead of adding more main rules, only extra rules for the varying fragments need to be written.

It turns out though that local rewrite rule sets are particularly useful for macros involving iterating over a sequence of constituent fragments:

```
define macro properties-definer
  { define properties ?kind:name ?properties end }
    => { define variable ?kind = list(?properties) }
properties:
  { } => { }
  { ?prop:name; ...  } => { ?#"prop", ...  }
end macro;
```

where, the substitution `?#"property"` coerces the identifier into a similarly named symbol. The ellipses ... are a shorthand for aux-rule recursion, that is,

```
properties:
  { } => { }
  { ?prop:name; ...  } => { ?#"prop", ...  }
```

is a shorthand for

```
properties:
  { } => { }
  { ?prop:name; ?properties } => { ?#"prop", ?properties }
```

5 The D-Expressions Library

In order to write macros that require more complicated macro expansion processing, we introduce d-expressions and later a full procedural macro facility. The d-expressions library provides a collection of classes suitable for representing fragments of source code in skeleton syntax tree form. If desired, skeleton syntax trees may be constructed and pulled apart manually using the exported interface of these classes.

In addition to the raw, skeleton syntax class hierarchy, source level tools are provided for easier parsing and construction of source code fragments held in that form. These tools are the moral equivalents of the destructuring-bind and backquote macros familiar to Lisp programmers.

The final facility provided is source code IO. An input method is defined that is parameterizable with syntactic context (i.e. a map from identifiers to their syntactic categories) and also, optionally, with a home context (e.g. the code's top level module context) and source location context. An output method is also defined and writes a re-readable text form of any given skeleton tree. These methods are the moral equivalents of Lisp's read and write functions.

5.1 Skeleton Syntax Tree Classes

The basic d-expression AST class hierarchy is agreeably simple:

```
<d-expression> [Abstract]
  <d-leaf> [Abstract]
    <d-identifier>
    <d-literal>
    <d-punctuation>
  <d-compound> [Abstract]
    <d-nested>
    <d-macro-call>
    <d-sequence>
```

As an example, given the input source:

```
f(x, y) + g(z) + h[0];
```

and a syntactic context in which none of the identifiers involved is associated with a macro category, the object tree is:

```
{<d-sequence>
  {<d-identifier> "f"}
  {<d-nested> "(" ")"
    {<d-identifier> "x"}
    {<d-punctuation> ","}
    {<d-identifier> "y"} }
  {<d-identifier> "+"}
  {<d-identifier> "g"}
  {<d-nested> "(" ")"
    {<d-identifier> "z"} }
  {<d-identifier> "h"}
  {<d-nested> "[" "]"
    {<d-literal> "0"} }
  {<d-punctuation> ";"} }
```

On the other hand, if `f` were associated with a function-form macro, this would be the result:

```
{<d-sequence>
  {<d-macro-call> "f"
    {<d-nested> "(" ")"
      {<d-identifier> "x"}
      {<d-punctuation> ","}
      {<d-identifier> "y"} } }
  {<d-identifier> "+"}
  {<d-identifier> "g"}
  {<d-nested> "(" ")"
    {<d-identifier> "z"} }
  {<d-identifier> "h"}
  {<d-nested> "[" "]"
    {<d-literal> "0"} }
  {<d-punctuation> ";"} }
```

Note how the macro call becomes a single syntactic entity at top level, but containing nested structure. This is consistent with our earlier-stated intuition that macros introduce "new kinds of bracket" to the syntax.

5.2 Source-level Tools for Parsing

Working with AST classes directly is tedious and error-prone. Where possible, it is desirable for a programmers to be able to work in terms of the source code shapes they are already familiar with.

As shown earlier, an intuitive and accessible way of expressing a parser is with patterns. This allows a programmer to write out the general shape of input expected, but with pattern bindings in place of the variable parts of the code. The source-level parsing tools provided in the d-expressions library take this approach.

The primary parsing tool offered is `expression-case`. This is a simple example:

```
expression-case (d-expr)
  { always-assert(?test:*) }
    => generate-always-assertion(test);
  { debug-assert(?test:*) }
    => generate-debug-assertion(test);
end expression-case;
```

The input expression, `d-expr` in this case, must evaluate to a valid d-expression. This d-expression is tested against each of the left hand side patterns in turn. If one of the patterns matches, its pattern variables are bound to local variables of the same name and the corresponding right hand side expression is run in the context of those bindings.

Pattern syntax and pattern matching behaviour in `expression-case` are as specified for Dylan's rewrite-rule only macros. To recap, pattern variables are introduced with `?` and have an associated constraint. The most liberal constraint is the wildcard constraint, `*`, which matches anything, as used in the example.

To illustrate how `expression-case` relates to the underlying skeleton syntax classes, consider this hypothetical expansion of the first case in the above:

```
// Fail is bound to an exit procedure that skips and
// moves on to try the next pattern.  If a match-xxx
// function can't match, it invokes its fail argument.
let _elements = d-top-level-elements(d-expr);
let _elements-after
  = match-identifier(#"always-assert", _elements, fail);
let (_elements-inside, _elements-after)
  = match-nested(#"(", #")", _elements-after, fail);
match-empty(_elements-after, fail);
// Matched, so bind the pattern variables.
let test = elements-inside;
// The right hand side code.  The values returned are
// the values returned by expression-case.
generate-always-assertion(test);
```

The utility functions `match-xxx` use the skeleton syntax accessor methods to do their work.

5.3 Source-level Tools for Code Generation

A similarly intuitive and accessible way of generating parameterized code is with templates. These allow a programmer to write out a prototype of the form they want to generate, but with substitutions in place of the variable parts of the code.

The d-expressions library implements the special syntax

generation time. This canonicalization is delayed to execution time when a syntactic context within which to analyse the generated code must have been established.

In particular, the static checking of templates attempted in [14] cannot be usefully applied here. On execution, template identifiers are "read" within a syntactic context established at runtime by the program containing the template, not in the static syntactic context of the program's source code. For example, in the mapping of names to syntactic classifications put in place at runtime, `if` may have no special association, or an association that differs from that of standard Dylan syntax.

Because it matches typical usage patterns well, we have elected to allow syntactic template contexts to be established with dynamic scope. That is, a syntactic context may be established with a single dynamic binding for all templates executed within its scope. Similar dynamic binding forms can be used to establish new, shared hygiene contexts and source location annotations for template-generated code run within their scope.

5.4 D-Expression IO

A method is provided for reading d-expressions from an input stream. Reading requires parameterization with the following:

- A *syntactic context* is a mapping between identifier names and their syntactic category. The set of available categories is fixed and corresponds to the set of broad "shape" categories offered by the skeleton syntax, as described above.

  This mapping is all that is needed for the reader to be able to parse and return a single, complete d-expression. In particular, macroexpansion does not have to be performed during parsing.

  A read identifier remembers its syntactic category. This category is preserved through template substitutions so that it may be re-read correctly if inserted as part of a newly constructed piece of source code (e.g. through macroexpansion).

Reading may be parameterized optionally with the following:

- A *home context*, if meaningful, can be specified for the code read. This context object is remembered by and can be recovered from the identifiers within the returned d-expression. The home context is intended to allow a top level binding context or container to be associated with code. If reading full Dylan code, the home context would typically identify the module in which the code lives and be used when resolving identifiers to module variables.

- A *source location context*, if desired, can be specified for the code read. A context object together with a seed line/column source location are used to record detailed source location information about each element of the d-expression read. The context object would typically identify a file or other physical source container.

A method is also provided for writing d-expressions to an output stream as re-readable text. The only parameterization available on output is involved with specifying the layout of the resulting text and has no semantic impact.

6 Models of Compile-time Evaluation

The preceding sections outline a utility library for reading, writing, and manipulating code representable by our skeleton syntax tree form. As previously suggested, this library is useful for a number of purposes, from simple storage and retrieval of data in text form to parsing for a full-scale Dylan compiler (the Functional Objects Dylan compiler was in fact based on such a library), or indeed a compiler for any language with a Dylan-like syntax.

But in order to meet our goals, the task remaining is to describe a Lisp-strength procedural macro system for Dylan. The source-level code manipulation tools may be available in the form of the d-expressions library as we have seen, but some model of compile-time evaluation is required before they can be used to implement macros. In the following sections we consider some approaches to compile-time evaluation.

6.1 Compile-time is Run-time

It is not uncommon for languages originally or conventionally implemented using an interpreter, or where an interpreter or dynamic compiler is available during program execution, to allow programs to construct and then execute new code at runtime. A macro in such a language can be thought of as a runtime function flagged such that it gets passed the unevaluated source code of its arguments. The code of the macro function rewrites the input source and then, either explicitly or implicitly, re-invokes the interpreter/compiler on the result.

Older dialects of Lisp used to support this approach, calling such functions *fexpr*'s. However, more recent dialects like Common Lisp no longer support them. Fexpr's can have the semantic problem of "losing" the local binding environment of their input expressions in lexically-scoped languages, but more importantly they defeat optimized compilation and impose the overhead of some form of language processor having to be available at runtime along with any of the code rewriting tools used.

Dylan is very much designed with compilability and minimal runtime overhead in mind, so this approach to arbitrary computation during macroexpansion is not acceptable. These days, the fexpr approach tends to be limited to interpreted or dynamically compiled scripting languages like TCL.

6.2 Static Compiler Plugins

Probably the simplest approach to compile-time evaluation, for the language implementor at least, is to admit to the existence of the compiler and allow programmers to write compiler "plugins". In this model, procedural macro code cannot be interleaved with the runtime source - it must be compiled separately and thought of as a compiler extension that will later be applied to the runtime source.

Compile time code and runtime code are therefore very clearly distinguished, being contained within entirely different programs. Compile-time dependencies and run-time dependencies can't easily be confused. A plugin model also opens the door on other interesting compiler extensions and customisations not necessarily related to macros.

Because a macro implemented in a plugin is defined in a context away from its logical home, extra annotation is required to identify the module where it should be bound. For example:

```
Module:  parsergen-plugin

define macro parser-definer in parsergen, parsergen
  { define parser ?:name ?rules:* }
    => let compiled-grammar = compile-grammar(name, rules);
       let tables = generate-grammar-tables(compiled-form);
       let driver = generate-parser-engine(compiled-form);
       { ?tables;
         ?driver; }
end macro;
```

The macro is implemented in the `parsergen-plugin` module, but through the addition of an "in" clause, parsergen-definer will be bound in the parsergen module of the parsergen library. The macro will *not* be bound or available within parsergen-plugin.

The compiler maintains a table of macros defined by plugins which it queries when compiling a module definition in a program under development. If the module matches the "in" clause of a macro, that macro is bound in the module's namespace and becomes available to all code within that module and to the module's clients (assuming the macro binding is exported by the module).

A further implication of the "in" clause is that template code generated while expanding a call to the macro will be resolved as if the code had appeared within the target module. This is important since the namespace and syntactic context of the target module is likely to be quite different from that of the implementing module. Further, details of the target module will not be known until after compilation of the plugin macro, and then may vary from compile to compile of the program under development. The ability to delay interpretation of templates is crucial in enabling this compilation model.

Although managing a separate runtime program and a parallel compiler plugin sounds like a headache, most of the details can be hidden by development environment support. The source of the plugin library can be shown along with the source of the target and the two compiled together as a single action. The plugin is compiled first and the resulting library dynamically loaded into the compiler to ensure that the macros defined are available when compilation of the runtime program begins. Procedural macros that define other procedural macros are possible, with plugins being loaded in order to compile other plugins.

If cross-compiling, two compiler back-ends may need to be loaded simultaneously in order to compile plugins. However, in a situation where the runtime target platform is resource constrained, running large or otherwise resource-hungry compile-time code only on the development host can be a necessity.

There are two main drawbacks to the plugin approach. The first is that the transition from Dylan's rewrite-rule only macro system to procedural macros cannot be seamless. As soon as general compile-time evaluation is used, the macro must be moved out of the runtime source into the plugin source.

The second drawback is the danger involved in running arbitrary user code within the compiler/development environment process. Instability may result.

6.3 Compile Time Loading and Eval-When

To achieve a seamless transition between Dylan's rewrite-rule only macro system and full procedural macros it must be possible to use procedural facilities while still interleaving macros with runtime code.

While such integration is desirable, maintaining as clear a distinction as possible between compile-time and runtime dependencies is also important. Code within a library must be annotated as to whether it is compile-time code, runtime code, or both. The definition of the library itself must distinguish compile-time only dependencies from runtime dependencies.

For simplicity, with the exception of macro bindings themselves, we allow only runtime definitions to be exported from libraries. However, those runtime definitions may be imported into a library for use either during compilation or at runtime or both. For example, if a library defined parsing utilities intended for use during macro expansion, those utilities would be defined and exported as ordinary runtime definitions. In order to make those utilities available to its macros and other compile-time code, a client library would employ a *compile-stage use*. For example:

```
define library parsergen
  use dylan, stage:  both;
  use io; // runtime only by default.
  use parsing-tools, stage:  compile;

  export parsergen;
end library;
```

Code on the right hand side of rules within a macro definition are implicitly *compile-stage* code and so have access only to bindings imported through a compile-stage use. Other top level definitions can be made into compile-stage definitions by wrapping them in a compile-stage form. For example:

```
define macro parser-definer
  { define parser ?:name ?rules:* }
    => let compiled-grammar = compile-grammar(name, rules);
       let tables = generate-grammar-tables(compiled-form);
       let driver = generate-parser-engine(compiled-form);
       { ?tables;
         ?driver; }
end macro;

compile-stage

  define class <compiled-grammar> (<object>)
    // ...
  end class;

  define method compile-grammar (name, rules) => (grammar)
    // ...  calls to parsing-tools ...
  end method;

  // etc.

end compile-stage;
```

Compilation of a library containing both compile-stage and run-stage code can be thought of as proceeding as follows. Any procedure with the same effect is equally valid.

If the library definition has compile-stage uses, a compile-stage version of the library in the form of an application is needed in order to compile the run-stage portion of the library. The compiler processes each top level form of the library in turn. As it encounters run-stage code, it processes it normally. As it encounters macros or explicit compile-stage code, that code is compiled and then immediately dynamically loaded into the compile-stage app. If the compiler finds a call to a procedural macro in run-stage code, a macro call is packaged and sent to the macro function in the compile-stage app where it is expanded before the results are sent back to the compiler. When compilation is complete, two binaries are saved: one for the run-stage code and one for the compile-stage code. The compile-stage aspect will be loaded when compiling client libraries if procedural macros are exported.

One likely variation on this is to dynamically load the compile-stage code directly into the compiler process rather than start a separate application. Also, under certain circumstances it may be possible to determine the compile-stage code of a library by a pre-pass, avoiding the need for incremental compilation of the compile-stage code.

The problem with this approach is clearly its complexity for implementors. In the general case, it requires either incremental compilation or interpretation of compile-stage code interleaved with another ongoing compilation; new and challenging requirements for Dylan implementations, which currently can be as simple as straightforward batch compilers.

7 Status

Our current status is that we have implemented a d-expressions library and have used it to write our compiler. We support the plugin model of compile-time evaluation, but don't yet support the full strength compile-time evaluation model. Nevertheless, we are capable of it since we already support incremental compilation and dynamic update.

The Dylan language is out there and the rewrite-rule only macros are part of the Dylan culture just as much as they are the Lisp culture. There are people out there really writing non-trivial macros in Dylan for themselves. So it has proved to be a very usable system. What's more, there's more than one implementation.

8 Related Work

In this section we survey the most related macro systems.

8.1 Lisp Macros

Dylan was very much inspired by Lisp's destructuring and backquote facilities. Their advent was a quantum leap in macro systems and played a big part in popularizing macros and Lisp itself [2]. We maintain that our patterns and templates are as natural to use as Lisp's destructuring and backquote. In fact, we feel that our splicing operator is an easier to use unification of backquote's unquote (`,`) and splicing (`,@`) operators. The reason this is possible in Dylan is because pattern variables can be bound to the actual elements of a sequence and not the sequence itself. For example, in Lisp, in order to splice in elements to the end of a parameter list, one uses the splicing operator on a whole sequence as follows:

```
(let ((more '(c d e)))
  `(a b ,@more))
```

while in Dylan, one could do the same with the following:

```
begin
  let more = {c, d, e};
  {(a, b, ?more)}
end
```

without the need for a different splicing operator. In order to better relate Dylan's templates to Lisp's backquote, we present the following table showing a loose correspondence between the two:

Name	Lisp	Dylan
backquote	`` ` () ``	{}
unquote	,	?
splicing	,@	?
quote	'	\

For an example of where quoting is necessary, consider the case of nested macros, that is, macros that define other macros. Suppose we find that several macros have a similar form. We can abstract this macro pattern into a macro-defining macro. For example, consider a macro that defines macros that pass a thunk to a given procedure [2]:

```
define macro caller-definer
  { define caller ?abbrev:name ?proc:expression end }
    => { define macro ?abbrev
           { ?abbrev (\?var:name) \?:body end }
             => { ?proc(method (\?var) \?body end) }
         end macro }
end macro;

define caller collect call-with-new-collector end;
define caller catch call-with-current-continuation end;
```

where \ is used to prevent evaluation of pattern variables in the inner macro definition.

Unfortunately, several limitations restrict Lisp macros' ease of use. First, variable capture is a real problem and leads to difficult to debug macros. Second, macro calls are difficult to debug as Lisp macros do not offer a mechanism for correlating between macro expanded code and a user's original source code.

Several other researchers have reported on systems that generalize the backquote mechanism to infix syntaxes. Weise and Crew [14] are discussed below. Engler, Hsieh, and Kaashoek [6] employ a version of backquote to support a form of partial evaluation in C.

8.2 Scheme

Scheme's macro system (i.e., `syntax-case` and `syntax-rules`) comes the closest to offering the power and ease of use of Dylan's macro system. We feel though that the combination of our rewrite-rule only and procedural macro systems provides a much more cohesive whole that gracefully progresses from a self-contained pattern language to the full power of procedural macros. Scheme's system, on the other hand, uses a different notation for their pattern language, `syntax-rules`, than from their procedural macro system, `syntax-case`. In Dylan, the same basic pattern matching language naturally incorporates the full Dylan language when writing procedural macros. In particular, Scheme's system requires a programmer to introduce local pattern variables using `with-syntax` whereas, in Dylan, a programmer merely introduces them with the usual local variable (i.e., `let`) definition syntax. For example in Scheme one must write the following:

```
(lambda (f)
  (with-syntax ((stuff f))
    (syntax stuff)))
```

where in Dylan one could do the equivalent with the following:

```
method (f) { ?f } end
```

This stems from the fact that Scheme requires users to specify reserved intermediate words up front otherwise names occurring within a pattern or template are interpreted as pattern variables. In Dylan, pattern variables have a special notation and thus reserved intermediate words do not need to be declared ahead of time, that is, they're the names without the special notation.

Scheme's system provides a nice solution to hygiene that automatically avoids most variable capture errors. Dylan

improves on this in a couple ways. First, it handles the more general problem of referential transparency in the face of modules (a.k.a., namespaces), ensuring that variables referenced within a macro definition mean the same thing in macro calls regardless of their namespace context. Second, Dylan provides a much more natural mechanism for introducing hygiene escapes using an intuitive notation, `?=`. Consider the Scheme the following version of the Dylan `repeat` macro defined above:

```
(define-syntax repeat
  (lambda (x)
    (syntax-case x ()
      ((k e ...)
        (with-syntax
            ((break (datum->syntax-object (syntax k) 'break)))
          (syntax
            (call-with-current-continuation
              (lambda (break)
                (let repeat () e ... (repeat)))))))))))
```

Notice how Scheme must use a long-winded call and `with-syntax` binding to produce the desired `break` variable.

Although Scheme's repeated pattern matching mechanism (i.e., `...`) is cute, it is unfortunately brittle. Scheme's repeated patterns only win if every input in a given sequence has the same general shape; as soon as this isn't the case, one must resort to general traversal. As an example of why the Dylan approach is more general, consider the case of sequences of heterogeneous input. For example, imagine extending the common `define-structure` example [4] such that constant structure slots can be optionally defined, say, and then compare the code. That is, a Scheme macro call to `define-structure` would look like:

```
(define-structure foo x y (const z))
```

whereas an equivalent Dylan macro call would look like:

```
define structure foo x, y, const z end;
```

The usual Scheme `define-structure` involves the following pattern:

```
(_ name field ...)
```

used in a template in the following fashion:

```
(syntax
  (begin
    (define constructor
      (lambda (field ...)  (vector 'name field ...)))
    ;; ...
    ))
```

will not work for this case. The Scheme code has to emulate Dylan's more general traversal over the input in order to handle this possibility.

One advantage Scheme macros have over Dylan macros is that one can limit a macro's visibility to a lexically local region of code using `let-syntax` and `letrec-syntax`. Dylan macros could be extended to provide local scoping for limited macro shapes. In particular, because of the requirements of an initial skeletal parse, a local macro could not be defined that introduced new end brackets, such as statement macros. It would be possible though to introduce local function macros to Dylan as these are unambiguously parseable.

8.3 Weise and Crew

Weise and Crew [14] describe a macro system for infix syntax languages such as C. Their macro system is programmable in an extended form of C, guarantees syntactic correctness of macro produced code, and provides a template substitution mechanism based on Lisp's backquote mechanism. Their system lacks support for hygiene, but instead requires programmer intervention to avoid variable capture errors.

Unfortunately, their system is restrictive. Their macro syntax is constrained to that describable by what is, essentially, a weak regular expression. In contrast, in our system, within the liberal shapes of an SST, just about anything goes, and further, the fragments can be parsed using any appropriate technique. Finally, because templates are eagerly parsed, it's not clear that forward references within expanders (and so mutual recursion, say) works in their system. It could be made to work, however (by forward declaring the input syntax apart from the transformer), but this would be awkward to use.

Their system requires knowledge of formal parsing intricacies. They say:

> 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.

It is not a reasonable requirement that programmers must understand and solve static grammar ambiguities like this.

We believe that as far as possible, programmers should only have to know the concrete syntax, not the abstract syntax. Unfortunately, their system requires programmers to use syntax accessors to extract syntactic elements, whereas our system allows access through pattern matching concrete syntax.

Their system requires templates to always be consistent. We feel that it's often useful to be able to generate incomplete templates containing macro parameters for instance, that are spliced together at the end of macro processing to form something recognizable. Weise and Crew's insistence that the result of evaluating a template should always be a recognizable syntactic entity defeats that mechanism. Being able to work easily with intermediate part-expressions that have no corresponding full-AST class (e.g. a disembodied pair of function arguments) is a win for the SST approach.

9 Summary

Most of the people involved in Dylan's design were experienced Lisp programmers. Anyone who has worked extensively in Lisp is well aware of the potential of Lisp macros, and despite the challenge presented by Dylan's algebraic syntax, we knew we still wanted full-power macros. As compiler writers, we also realised the value of source code tools as a library. D-expressions are the result.

We have brought together the elements of a Lisp-inspired SST capable of representing Dylan's rich algebraic syntax, best-of-breed source level code manipulation tools, and a model of compile-time evaluation suitable for a language with strong compiler/runtime separation like Dylan, to produce what we feel is the first macro system with Lisp's power and simplicity for a language with a conventional syntax.

Acknowledgements

Much of the initial design work on Dylan macros was done at Apple. The syntax of macro definitions, patterns, con-

straints, and templates used in Dylan's standard macro system was developed by Mike Kahl. Dylan's very first "loose grammar" was due to David Moon, who also designed a pattern matching and constraint parsing model suitable for such a grammar. Earlier, Moon had proposed a model of compile-time evaluation for Dylan in the context of a prefix-syntax macro system from which ours takes some terminology.

Generalizations of Dylan's loose grammar enabling it to describe all Dylan's syntactic forms, along with the first implementation of the macro system, were developed by the authors while at Harlequin Ltd. The procedural macro system was initially designed and implemented as a component of Harlequin's Dylan compiler, now owned by Functional Objects, Inc.

Many other "Dylan Partners" made contributions to the design of Dylan macros, particularly the Gwydion team at CMU.

Dave Moon provided feedback on various drafts of this paper. This paper also benefitted from helpful discussions with Alan Bawden and Tony Mann. Tony Mann gave many insights into the limits of the rewrite-rule only macro system by pushing the macro writing envelope.

References

[1] Barrett, R, Ramsay, A, and Sloman, A. *POP-11: a Practical Language for Artificial Intelligence*. Ellis-Horwood, Chicester, 1985.

[2] A. Bawden. Quasiquotation in lisp. *Proceedings of the ACM Conference on Lisp and Functional Programming*, ?(?), 1999.

[3] Luca Cardelli, Florian Matthes, and Martin Abadi. Extensible syntax with lexical scoping. Technical Report 121, DEC SRC, February 1994.

[4] Kent R. Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[5] R.K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, ?(?), 1993.

[6] E.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'c: A language for fast, efficient, high-level dynamic code generation. In *Proceedings of Symposium on Principles of Programming Languages*, January 1996.

[7] N. Feinberg, S. Keene, R. Mathews, and T. Withington. *Dylan Programming*. Addison Wesley, 1997.

[8] P. Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice-Hall, 1994.

[9] Guy Lewis Steele Jr. *Common LISP: The Language, Second Edition*. Digital Press, Burlington, MA, 1990.

[10] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988.

[12] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygenic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161. ACM, ACM, August 1986.

[13] A. Shalit. *The Dylan Reference Manual*. Addison Wesley, 1996.

[14] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.

A Auxiliary Macros

A number of limitations of local rewrite rules require resorting to top-level auxiliary macros. The first problem is that local rewrite rules do not have access to pattern variables defined in main rule sets. The usual solution is to employ an auxiliary macro which takes those needed variables as extra macro arguments. For example, suppose we want to add a prefix option to allow the creation of properties with a common prefix. We need to introduce an auxiliary macro so that the prefix variable can be in scope when iterating over the properties. For example, consider the following:

```
define macro properties-definer
  { define properties ?kind:name
        prefixed-by ?prefix:name ?properties:* end }
    => { define variable ?kind
           = concatenate
               (prefixed-props (?"prefix") ?properties end) }
  { define properties ?kind:name ?properties:* end }
    => { define variable ?kind
           = concatenate
               (prefixed-props ("") ?properties end) }
end macro;

define macro prefixed-props
  { prefixed-props (?prefix:name) end }
    => { #() }
  { prefixed-props (?prefix:name) ?prop:name; ?more:* end }
    => { list(?prefix ## ?prop),
         prefixed-props (?prefix) ?more end) }
end macro;
```

Auxiliary macros are also needed when pattern variables must be walked in two different ways. Consider a macro, called `iterate`, for creating internally recursive procedures that are as convenient as loops. It has the following basic form:

```
iterate name (variable = init, ...)
  body
end iterate
```

and has the semantics of evaluating the body with the variables initially bound to the inits and such that any call to `name` inside the body recursively calls the procedure with the variables bound to the arguments of the call. In writing a macro for `iterate`, the parenthesized fragments must be walked once to extract variables and another time to extract initial values. Unfortunately, with the current macro system, there is no way to walk the same pattern variable with more than one set of local rewrite rules. Instead, an extra copy of the pattern variable must be made and passed on to an auxiliary macro:

```
define macro iterate
  { iterate ?loop:name (?args:*) ?:body end }
    => { iterate-aux ?loop (?args) (?args)
           ?body
         end }
end macro iterate;

define macro iterate-aux
  { iterate-aux ?loop:name (?args) (?inits) ?:body end }
    => { local method ?loop (?args) ?body end;
         ?loop(?inits) }
args:
  { }
    => { }
  { ?:variable = ?:expression, ...  }
    => { ?variable, ...  }
inits:
  { }
    => { }
  { ?:variable = ?:expression, ...  }
    => { ?expression, ...  }
end macro repeatable-aux;
```

B Dylan Rewrite-Rule Only Macro Extensions

B.1 Template Calls

Both of these reasons for needing auxiliary macros are somewhat artificial because in fact local rewrite rules are really like local functions and should allow extra arguments and should be directly callable on any pattern variable. The problem lies in the fact that the local rewrite rule is artificially tied to one pattern variable by virtue of its name.

In order to overcome this problem, we introduce a direct template call, obviating the need for auxiliary macros is many cases. This leads to a much more elegant solution to these more complicated macros. A template auxiliary rule set call has the following form:

```
?@rule-name{ <arbitrary-template-stuff> }
```

where a new macro punctuation `?@` marks a template call. For example, in the prefixed `properties-definer` macro, we can now directly invoke a local rewrite rule set with extra arguments:

```
define macro properties-definer
  { define properties ?kind:name
        prefixed-by ?prefix:name ?properties:* end }
    => { define variable ?kind
           = concatenate
               (?@prefixed-properties{ ?"prefix"; ?properties }) }
  { define properties ?kind:name ?properties:* end }
    => { define variable ?kind
           = concatenate(?@prefixed-properties{ ""; ?properties } ) }
prefixed-properties:
  { ?prefix:name }
    => { #() }
  { ?prefix:name; ?property:name; ?more:* }
    => { list(?prefix ## ?property),
         ?@prefixed-properties{ ?prefix; ?more } }
end macro;
```

Similarly, `iterate` can now be written without auxiliary macros using two template calls:

```
define macro iterate2
  { iterate2 ?:name (?bindings:*) ?:body end }
    => { local method ?name (?@vars{ ?bindings }) ?body end;
         ?name(?@inits{ ?bindings }) }
vars:
  { }
    => { }
  { ?:variable = ?:expression, ...  }
    => { ?variable, ...  }
inits:
  { }
    => { }
  { ?:variable = ?:expression, ...  }
    => { ?expression, ...  }
end macro;
```

We can also introduce a template macro call

```
?@{ <arbitrary-template-stuff-that-forms-a-macro-call> }
```

which acts as a kind of shorthand for the `:macro` constraint and permits the definition of macros for use as shared rewriting tools. For example:

```
define traced macro mcreverse
  { mcreverse(?list:*) } => { ?list }
list:
  { } => { }
  { ?:expression } => { ?expression }
  { ?:expression, ...  } => { ..., ?expression }
end macro;

define traced macro user
  { user(?stuff:*) } => { list(?@{ mcreverse(?stuff) }) }
end macro;
```

where the traced modifier causes macro expansion to be traced. For example, here's the trace for `user(1, 2, 3)`:

```
{ user } > user(1, 2, 3)
{ mcreverse } > mcreverse(1, 2, 3)
{ mcreverse } < 3 , 2 , 1
{ user } < list (3, 2, 1)
```

Like normal macro calls, a new hygiene context in created for `?@\{ }` calls, so you could define gensym thusly:

```
define macro gensym
  { gensym() } => { gensymed-name }
end macro;
```

C Limits of Rewrite-Rule Only Macro System

Dylan's rewrite-rule only macros are more powerful than they first appear. Unlike a grand tradition in Lisp, replacement phrases are constructed purely by template substitution and not arbitrary computation. A number of inherent properties increase the power of this base macro system.

First, there is a powerful set of built-in atomic features such as hygiene, name concatenation, and pattern constraints which often obviate the need for such procedural facilities. Other macro systems (e.g., Scheme's `syntax-case`) must rely on function calls for these facilities and must resort to the full power of procedural macros.

Second, the rewrite-rule only system allows for powerful control structures, not unlike a mini programming language. Macros can expand into other macro calls including into recursive calls. Although not discussed in this paper, macros can have keyword and optional arguments that can be used to collect a series of properties which can be spliced into the resulting output in bulk. For example, another formulation of the iterate macro making use of this facility would be:

```
define traced macro iterate3
  { iterate3 ?:name (?bindings:*) ?:body end }
    => { ?@generate{ ?name (?@parse{ ?bindings }) ?body } }
parse:
  { }
    => { }
  { ?:variable = ?:expression, ...  }
    => { var:  ?variable, init:  ?expression, ...  }
generate:
  { ?:name (#key ??var:variable, ??init:expression) ?:body }
    => { local method ?name (??var, ...)  ?body end;
         ?name(??init, ...)  }
end macro;
```

where, for example, `??var:variable` matches all keywords arguments with keyword `var:` and binds the pattern variable `??var` to the sequence of their values.

Finally, expanding into calls to runtime functions can often defer calculations that would have otherwise needed to be computed during macro expansion. Furthermore, constant folding and more generally partial evaluation can often be relied on to collapse these residual calls. For example, concatenation can be deferred to runtime as follows:

```
define macro catsyms
  { } => { }
  { catsyms(?items) } => { concatenate(?items) }
items:
  { ?item:name, ...  } => { list(?#"name"), ...  }
end macro;

?  catsyms(a, b, c);
> #(#"a", #"b", #"c")
```

Similarly, counting a sequence of items can be performed as follows:

```
define macro count
  { count() } => { 0 }
  { count(?item:expression, ...)  } => { 1 + ...  }
end macro;

?  count(a, b, c);
> 3
```

Unfortunately, Dylan's rewrite-rule only macros can only count a given input sequence, but can't count from say 0 to some given limit. Thus, it is awkward to construct a macro, for example, that expands into a specified numbers of functions with increasing number of parameters.