

# Dissertation Proposal

Version 2.0

Jon Rafkind

June 14, 2011

# 1 Introduction

Thesis: Semantic macro systems are feasible and useful for typed languages with rich syntactic structure.

Many existing languages are accumulating features they were not initially designed with. C++0x, Python 3, and Java 7 are all adding language features that require significant changes to their respective base implementations. Many changes, such as lambda support for C++0x and Java 7, consist almost entirely of syntactic transformations to existing features. Lisp [3]-derived languages have a long history with supporting similar types of extensions, some even allowing large systems to be implemented as libraries [? ].

Languages that support syntactic abstraction are able to do so due to the regularity of their syntactic grammars. S-expressions in particular are a simple mechanism for which transformations can be reasoned about easily. Other forms of syntax designed to be practical often have complex grammars with many rules. This level of complexity presents a challenge to both designers of syntactic abstraction mechanisms and users of those systems. The former group must balance the number of implementation details about the grammar that are exposed while simultaneously providing an expressive system to the latter group.

Even though s-expressions have existed for many decades language designers continue to create languages with rich syntactic structure. Syntactic abstraction mechanisms have evolved and matured in s-expression based languages and the technology developed there should be applied to other languages. Some features have been applied to existing non-s-expression languages, notably pattern matching and hygiene, while other features such as procedural transformations and observation of concrete syntax have had less success.

S-expressions mimic the structure of abstract syntax trees which makes their concrete syntax straight forward to analyze. Complex grammars usually have many nodes that represent the abstract syntax tree and often lack a direct correspondence back to their concrete versions. Syntactic abstraction systems that allow programmers to access the abstract syntax tree through these nodes quickly become unwieldy.

Transformations built on top of other each other cannot communicate effectively at the abstract syntax tree level because a higher level transformation cannot know how a lower level transformation will eventually be parsed and thus will not know how to construct its abstract syntax tree. Concrete syntax is an appropriate manifestation of the program for transformations to consume and produce so that transformations can be built in terms of each other.

Grammars bridge the gap between concrete syntax and abstract syntax and have been chosen as the model to extend by some systems [? ]. Grammar rules become public types that must exist forever or risk breaking all code that relies on its specific structure. For example

```
declaration = <type> <identifier> <expression>
```

A new macro could be written that expands into a declaration form.

sql-declaration: declaration = sql <identifier> <sql expression>

Now the original grammar cannot be altered in such a way that the 'declaration' rule is removed. Furthermore, there are often many such rules and finding the right ones to expose to the programmer can be problematic.

Syntactic transformations that deal with concrete syntax must somehow produce valid concrete syntax and have typically done so with template systems. Ellipses notation has served s-expression based languages well but often is not directly applicable to other syntactic systems. Delimiters and infix notation in particular require extra support from templating systems.

Layered transformations have an additional burden in languages with types. They must generate code that can be type-checked properly while enforcing the high level constraints of any new constructs it introduces. Evaluation of syntactic transformations occur before type checking and this provides an opportunity for transformations to have some influence over the types of the program.

## 1.1 Motivation

Just as in the past with Algol and Pascal, many language designers are trying to create useful macro systems for languages to give them the ability to grow beyond their initial implementation. While a variety of languages have incorporated some features of proposed macro systems none of the popular languages initially developed without a macro system has successfully had a macro system added on later. There are a number of reasons for this:

- Hygiene is not considered early enough in the design
- The existing parser/grammar technology is too difficult to make extensible
- A separate tool is required

Macro systems that lack hygiene cannot cope with numerous language extensions all working together. The fundamental problem is the primary semantics, that is the semantics as it looks like a form should have, will not equal the expansion semantics, the semantics given to a program derived from a series of macro transformations.

Many parsing technologies in use today, such as LR, are not flexible enough for non-expert programmers to modify them. Moreover, some parsing technologies do not compose thus preventing two different extensions to be used in the same program. Parsing expression grammars and generalized LR parsers allow grammars to be composed in sensible ways but still require programmers to make modifications at a low level. Systems such as MS2 and Myans for Java use inflexible parsers.

Language extensions that are constructed outside the normal toolchain of a language have typically had a difficult time finding widespread use. Multiple language extensions may not be com-

possible because the output of one tool may conflict with the input of another. Often times language extension tools are only used if they are part of the native toolchain. Dylan and Ocamlp4 both contain systems that are built into the default distribution of software and have seen some use. On the other side, the Java Syntax Extender was built as a separate preprocessor and does not seem to have received much support from the Java community.

## 1.2 Design of Macro Systems

The main goal of a macro system is to provide abstractions over the syntax of a language. Ultimately the system should produce syntax that a programmer might have otherwise written. Allowing programmers to write syntax at the same level they wish to work in is a natural solution. Systems that support freely generated syntax can easily go wrong by breaking lexical scope unintentionally. An alternative design would be to force macros to produce syntax that adheres to a strict set of rules which can be statically checked for consistency. Hygienic macro systems allow programmers to use concrete syntax while maintaining lexical consistency.

With our work we hope to provide three improvements to current language extension technology. First we will develop a framework for defining languages using macros that can recognize and produce complicated syntaxes such as those that support infix operators. Second we will extend the templating system with rich operators for constructing new syntax. Finally we will add an extra property to macros that allow them to specify typing expressions to the system in a composable manner.

## 2 Related work

Extensible language mechanisms have come in many forms. Systems such as Xoc [? ], Xtc [? ], Stratego [? ], Caml4p [? ], and Rscheme [? ] have developed compiler frameworks that expose core aspects of the system. Programmers must learn intricate details about how those systems work to utilize the provided API's even though often times the given API hides many implementation details.

Systems that provide syntactic abstraction can broadly be separated into term rewriting systems and fully programmable systems. Term rewriting systems such as those found in R5RS Scheme [2], Myans [? ] for Java, and Dylan [? ] have enjoyed reasonable success but are limited in their power. Racket and other systems that implement Dybvig's [? ] **syntax-case** allow macros to compute arbitrary expressions.

The Java Syntax Extender [? ] provides programmable macros while also allowing new syntax to be written concretely. While this achieves many of the goals we hope for, the JSE was not designed with hygiene at the outset. They hypothesized that hygiene could be later added to the system but no additions have been forthcoming. Furthermore JSE restricts macro invocation to places where it can determine the syntactic extent by forcing where delimiters can be placed.

An important piece of macro systems that use concrete syntax is the ability to construct new syntax using templates. MS2 [? ] was an early system that incorporated Lisp's quasiquote into a templating system for C. However MS2 does not have the facilities to expand syntax that correspond to infix syntax or any other complex scheme.

Improvements in parsing technology are slowly merging into macro systems. GLR [? ] and [? ] are relatively recent parsing formalisms that are easier to use than traditional parsers. GLR parsers are similar to LR parsers but will return possibly more than one parse tree instead of failing to generate an ambiguous LR parser. GLR grammars can also be combined with other GLR grammars in a composable way. **Xoc** is a recent attempt at using GLR for language extension. PEG parsers are also composable and continue to be developed towards having an extensible grammar. **Ometa** has added new features to the PEG formalism that make it a good fit for future syntactic extension.

Extensible compilers, such as Xoc and Polyglot [? ], have been developed to give programmers an easier path to extend a base language but are overly complex for the kinds of problems that syntactic abstraction usually solves. The key ingredient to building a useable macro system is maintaining a strong link with the language being extended. Meta languages and extra API's, while powerful, hinder the understanding of the system as a whole and eventually fall into disuse.

[? ] has implemented a library, `syntax/parse`, for Racket that provides some of the same features as Ometa. In particular `syntax/parse` provides a backtracking pattern matcher where patterns can be higher order. Furthermore, `syntax/parse` can add arbitrary properties to syntax patterns.

Recently [? ] has developed a system that allows levels of a language tower to extend the type system by means of open classes. In his system, macros are classes that can have extra methods

defined on them and can be overridden at any point to give the macro writer a chance to observe macros being invoked and modify their transformation. Our system will be similar in spirit to Fisher's but using a different mechanism that hopefully is more composable.

Kohlbecker [?] and others have outlined various principles of macro design that we adhere to.

- Macros should be free form and not require special characters to warn the parser that a macro is about to be invoked.
- Hygiene must be a goal from the initial design.
- New syntaxes should not be constrained due to the nature of the macro system.

We believe these principles are what makes the Scheme and Racket macro systems so attractive.

## 3 Proposed Solution

We build on top of existing macro technology developed in Scheme but extend them to support more general syntaxes. Currently macros consist of two stages: parsing and transcription. First we modify parsing by allowing macros to define their own extent rather than force syntactic boundaries. Second we modify transcription such that pattern variables can control their expansion.

Retaining hygiene throughout the process is a matter of invoking the proper functions at the lowest level. Before a macro is invoked, its syntax is marked as per Dybvig and the returned syntax is marked again thus fulfilling the macro's contract to keep track of newly introduced identifiers.

### 3.1 Modified Parsing

Idea: Use **pattern matching to extend macro grammars**.

Many macro systems use an extensible grammar to achieve its goals. Extending the grammar is cumbersome and errorprone so we choose to disallow grammar extension in the usual sense. Instead we imagine a core grammar with a small number of nonterminals where each nonterminal can invoke a macro.

Macros are invoked when they are observed by the parser. When the parser recognizes a macro the input stream is passed to the macro which can consume as much input as it wants. Any unconsumed input is processed by the parser as normal. This method does not constrain syntax to use specific delimiters; each macro can decide how its form will be terminated.

We believe pattern matching is the most convenient mechanism for macros to use to convert its input into something usable. Like many other systems we allow macro writers to define their own syntactic classes which can be used as pattern matchers. We can build on the syntax/parse library developed by Culpepper which provides a set of tools for abstracting over syntactic forms.

Syntax patterns only constrain the shape of syntax rather than force a certain path of nonterminals to be taken through some grammar. This is different from other approaches that try to create extensible grammars due to its looser constraints.

Similar to other systems, such as JSE and Rscheme, we allow user defined syntactic classes. Each syntactic class recognizes either a specific shape of syntax or a primitive form. For instance, if a syntax class named **loop** is defined and some syntax pattern uses **loop** then as long as the input syntax can be transformed into something that **loop** accepts the pattern matcher will succeed.

Consider writing an SQL extension where sub-expressions are constrained to the SQL language. A top level clause might look like

```
clause = sql select SQL-expression-select* from SQL-expression-table SQL-  
select-rest
```

```
SQL-expression-select = identifier
                        count(identifier)
                        max(identifier)
```

Where **SQL-expression-select\*** means invoke a macro that produces syntax that will eventually match the underlying syntax class **SQL-expression-select**.

While syntactic classes are similar to BNF-style non-terminals we don't use syntactic classes to generate a parser, instead the syntactic classes act as parsers for us. This reduces the semantic gap that many programmers face when using language extension tools.

Not all elements of a pattern need to be specified by a syntactic class. In fact forcing a specific class may invoke a macro too early. If the macro being defined does not wish to impose any restrictions on its input then it should not perform macro expansion too early otherwise syntactic constructs that were meant for templates may be lost. For example in the following macro ultimately the code within the brackets will be inserted in a position following some other macro. That other macro may wish to inspect its arguments before macro expansion has occurred.

```
macro handler { stuff ... } = syntax{subhandler(0){ stuff ... }};

handler {macro_call x};
}
```

Here, **subhandler** may wish to inspect its body for variables that it may bind such as **macro\_call** but if **macro\_call** is also bound to a macro then when **handler** goes to parse its body it will inadvertently invoke **macro\_call** instead of passing it as-is to the template.

## 3.2 Modified Transcription

Idea: **Create low-level transcription primitives and macros that can use them.**

The algorithm laid out by Kohlbecker and Wand in *Macros that Work* cannot easily be applied to non-s-expression based syntaxes. Consider a simple for loop:

```
for (x start max; ...) body
=>
begin
  var x = start; ...
  while (x < max && ...)
    body
end
```

Which should bind the variables 'x' and loop through the values from 'start' to 'max' running 'body' on each iteration. Presuming we can parse the pattern we will have trouble dealing with the template. We would like to repeat such templates as

```
var x = start
```

for each identifier that 'x' is bound to but the ellipses does not know to operate on the entire expression 'var x = start'. Ad-hoc rules such as searching through the syntax for a delimiter, such as ';', could work but would limit the system to a fixed number of syntactic constructs that can be used with ellipses.

A possible solution to this issue is to use a more complex template language which provides operators and special delimiters designed to deal with complex syntax. We believe this is a step in the right direction and would like to generalize this tactic further.

We propose to use some set of operators on the template syntax combined with an extensible parser that gives macro writers a great deal of control over how macro expansion works. Pattern variables in the template will be more like objects that have a number of methods which can be called by the operators.

For instance, in Scheme we normally use something like the following expression

```
(syntax (+ 1 2 x ...))
```

Where the 'syntax' function is given its input after having been processed by the standard 'read' parser. We hypothesize a new function, which might be created by a macro writer, that can use a different parser.

```
syntax2(1 + 2 + x ...)
```

Which would produce a tree such as {item(1) item(+) item(2) item(+) ellipses(x)}. The 'item' and 'ellipses' identifiers are the operators on the individual parts of the template and can be chosen freely by the 'syntax2' parser.

Expansion would then consist of executing each operator in sequence, possibly creating a complex data structure that future operators accept as input. In this way the ellipses(x) call can look at the data so far, see '+' right before it, and expand in such a way that makes sense for the infix operator '+'. For example if 'x' was bound to (3 4 5) the result would be

```
1 + 2 + 3 + 4 + 5
```

Even if this is not the final syntax, once a set of core primitives for the template language exists we can develop multiple higher level syntaxes that a user is free to select from.

Language extensions must be able to create their own transcription syntaxes because no single transcription syntax will suffice for all syntaxes. Transcription syntaxes themselves can be created with macros that utilize the low level primitives we will define.

### 3.3 Extensible Types

Idea: **Attach properties to syntax objects that some type system can consume.**

We can view a type system as two components: first a set of attributes on a syntax tree and secondly a recursive descent algorithm that consumes the attributes and produces a new syntax tree or a failure.

During a macro transformation we can attach new type properties to the syntax objects. As a first step we will provide a core set of type properties that higher levels can combine to create custom types. Macros can control the type constraints on syntax objects it creates by removing or adding properties.

The typing pass will necessarily be defined at a language level instead of being defined in an ad-hoc manner along with macros. Only the top-level language is capable of making sense of the type properties that will eventually be attached via macros.

In Racket a module is wrapped with a special form, `%%module-begin`, that languages can define and use to provide extra support such as whole module analysis. A similar hook can be installed for other languages at the appropriate place.

### 3.4 Applications

Many systems could be built as language extensions but lacking the ability to do so instead opt to re-use features of the host language. Ruby on Rails is a popular framework for web servers that uses the flexible Ruby syntax to create a mostly declarative language. Ruby on Rails cannot re-use any static typing facilities because Ruby only provides dynamic assertions. While Ruby on Rails has mostly fulfilled its goal as a domain specific language for writing web applications it does not allow the user to specify robustness using types. Our system could potentially allow Ruby on Rails to insert the proper type specification.

JQuery is an extension to Javascript built as a library of functions designed to simplify many common tasks in web development. It also uses the somewhat flexible syntax of Javascript but not always in the cleanest manner. JQuery is clearly leaning towards being its own language built on top of Javascript but is burdened by its syntax. Extensions to JQuery itself would have to deal with the underlying Javascript language instead of being written explicitly in terms of JQuery. This may be a pain point for future library writers.

Scripting languages, such as python and lua, are finding applications in many products due to their flexibility and ease of use for end users. Most applications write a foreign interface between their native code and the language runtime and some provide a set of libraries that can be used with that language, but few customize the language itself to fit in with the application. Many times the application designers hope that providing an API geared towards their application will be enough but complex protocols and objects can make using the API not entirely straight forward. Language customization would be a great boon to application designers but most of the scripting languages

available do not support it.

## **4 Research Plan**

The major goals can be broken into a few smaller tasks

### **4.1 Pattern Matcher**

1. Use syntax/parse to develop a pattern matching system for complex syntax. This is already partially implemented.

### **4.2 Templating System**

1. Develop a set of primitive constructs that expand syntax. 2. Create high level syntax that can be transformed into the primitive constructs.

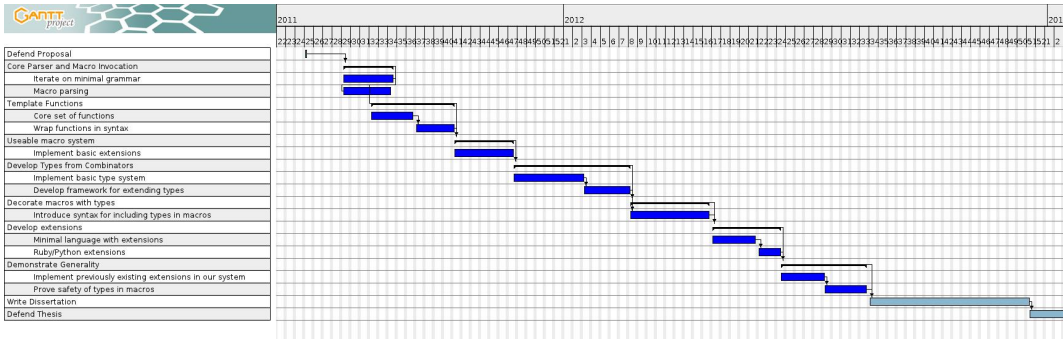
### **4.3 Type System**

1. Add properties to syntax objects that can be observed in later passes of compilation.

## Tasks List

Name	Start	End	Milestor	%	Resources	Notes
Defend Proposal	6/20/11	6/21/11	false	0		
Core Parser and Macro Invocation	7/19/11	8/26/11	false	0		
Iterate on minimal grammar	7/19/11	8/26/11	false	0		
Macro parsing	7/19/11	8/24/11	false	0		
Template Functions	8/9/11	10/11/11	false	0		
Core set of functions	8/9/11	9/10/11	false	0		
Wrap functions in syntax	9/12/11	10/11/11	false	0		
Useable macro system	10/11/11	11/25/11	false	0		
Implement basic extensions	10/11/11	11/25/11	false	0		
Develop Types from Combinators	11/25/11	2/21/12	false	0		
Implement basic type system	11/25/11	1/17/12	false	0		
Develop framework for extending types	1/17/12	2/21/12	false	0		
Decorate macros with types	2/21/12	4/21/12	false	0		
Introduce syntax for including types in macros	2/21/12	4/21/12	false	0		
Develop extensions	4/23/12	6/14/12	false	0		
Minimal language with extensions	4/23/12	5/26/12	false	0		
Ruby/Python extensions	5/28/12	6/14/12	false	0		
Demonstrate Generality	6/14/12	8/18/12	false	0		
Implement previously existing extensions in our system	6/14/12	7/17/12	false	0		
Prove safety of types in macros	7/17/12	8/18/12	false	0		
Write Dissertation	8/20/12	12/19/12	false	0		
Defend Thesis	12/19/12	1/22/13	false	0		

# Gantt Chart



## **Bibliography**

[r5rs] “R5RS.”

## **References**

- [1] <http://groovy.codehaus.org/>.
- [2] Richard Kelsey, William Clinger, and Jonathan Rees (Eds.). Revised<sup>5</sup> report of the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
- [3] Guy Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 225 Wildwood St. Woburn, MA 01801, 1994.