Convenient Macros for Languages with Conventional Algebraic Syntax

Jonathan Bachrach

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

http://www.ai.mit.edu

The Problem: How do we provide an easy to use mechanism for extending a programming language with new syntactic forms in a language with conventional algebraic syntax?

Motivation: The ability to extend a language with new syntactic forms is a powerful tool. A sufficiently flexible macro system allows programmers to build from a common base towards a language designed specifically for their problem domain. However, macro facilities that are integrated, capable, and at the same time simple enough to be widely used have been limited to the Lisp family of languages to date. In this project we introduce a macro facility, called the Java Syntactic Extender (JSE) [1], with the superior power and ease of use of Lisp macro systems, but for Java, a language with a more conventional algebraic syntax. The design is based on the Dylan macro system, but exploits Java's compilation model to offer a full procedural macro engine. In other words, syntax expanders may be implemented in, and so use all the facilities of, the full Java language.

Previous Work: Dylan macros were the main inspiration for JSE, but unlike Dylan's rewrite-rule only macro system, JSE exploits Java's compilation model to offer a full procedural macro engine. In other words, syntax expanders may be implemented in, and so use all the facilities of, the full Java language. Because of this, JSE has a simpler pattern matching and rewrite rule engine utilizing standard Java control and iteration constructs for instance.

Lisp's destructuring and quasiquote facilities inspired Dylan's macro system design. Their advent played a big part in popularizing macros and Lisp itself [3]. Macro systems for Scheme (e.g., `syntax-rules`) come the closest to offering the power and ease of use of JSE. As with Lisp's macros, their major limitation is that they are restricted to languages with prefix syntax.

The most popular macro mechanism for languages with conventional syntax is grammar extension macros [4] [5] [2] [8], which allow a programmer to make incremental changes to a grammar in order to extend the syntax of the base language. JSE is less ambitious in that it provides a convenient and powerful mechanism for extending the syntax in limited ways. In particular, it provides only a limited number of shapes and requires that macros must always commence with a name. We feel that this tradeoff is justified because it makes the vast majority of macros easier to write.

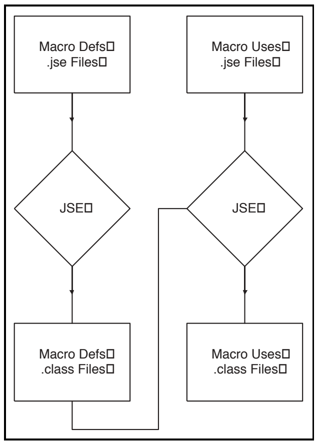
Approach: Macros are user-defineable syntactic extensions. In JSE, parsing is extended to include a recursive top-down macro expansion process which replaces macro uses with their syntactic replacement constructs, until there are no more macro uses in a program. Macros can occur at certain known places in the grammar, have a few particular syntactic shapes, and are identified by name.

Syntax expanders are classes that implement the SyntaxExpander interface and follow an associated naming convention. These classes co-exist with the runtime Java source and class files of a program, but are actually dynamically loaded into the compiler or preprocessor on demand during a build.

Being implemented by classes, syntax expanders can either be defined at top level or live inside other classes. Syntax expanders defined at top level can be referred to unqualified. If defined within another class, the appropriate qualification must be used, as if referring directly to a nested static class.

Macros are first defined and compiled by JSE producing `.class` files containing class represention of the defined macros. An application that uses these macros is then subsequently compiled by JSE augmented by demand-loaded class files corresponding to actually used macros. This then results in the final application class files.

It would be possible to manually write expander classes against the core code fragment manipulation API pro-



vided, but the idea is that a higher level syntax form be used, which provides source-level pattern matching and code generation facilities and which expands into an appropriate class definition.

Impact: Macros also provide an effective vehicle for planned growth of a language (see [4]). In Guy Steele's OOPSLA-98 invited talk entitled, "Growing a Language" [6], he says "From now on, a main goal in designing a language should be to plan for growth." A large part of Lisp's success and longevity can be linked to its powerful macro facility, which, for example, allowed it to easily incorporate both Object-Oriented and Logic programming paradigms. We're hoping that JSE will provide similar planned growth for Java.

Future Work: Many important future directions remain. We would like to see JSE provide more guarantees about macro definitions always producing admissible expansions. We would like to extend JSE to allow for symbol macros and to support generalized variables such as CommonLisp's setf. Furthermore, JSE seems like a very nice substrate for staged compilation as exhibited in 'C (see also [7]). Finally, we think that our approach could be fruitfully applied to other languages such as C and Scheme.

Research Support: This research is supported in part by DARPA under the Express contract F30602-97-2-0013.

References:

[1] Jonathan Bachrach. The java syntactic extender. In *Proceedings of OOPSLA 2001*, October 2001.

[2] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 143–153. IEEE Computer Society Press, 1998.

[3] Alan Bawden. Quasiquotation in Lisp. In *SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12. ACM, January 1999.

[4] Braband and Schwartzbach. Growing languages with metamorphic syntax macros. www.brics.dk/bigwig/research/publications/macro.pdf, 2000.

[5] Daniel de Rauglaudre. Camlp4. http://caml.inria.fr/camlp4/, 2001.

[6] Guy Lewis Steele Jr. Growing a language. *Lisp and Symbolic Computation*, 1998.

[7] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *SIGPLAN workshop on partial evaluation and semantics-based program manipulation*, pages 203–217, 1997.

[8] 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.