The Dissertation Committee for Donovan Michael Kolbly

Certifies that this is the approved version of the following dissertation:

Extensible Language Implementation

Committee:

Gordon Novak, Supervisor

Don Batory

Don Fussell

Calvin Lin

Robert Strandh

Extensible Language Implementation

by

Donovan Michael Kolbly, B.S.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2002

Extensible Language Implementation

Publication No. _____

Donovan Michael Kolbly, Ph.D.
The University of Texas at Austin, 2002

Supervisor: Gordon Novak

This work presents several new approaches to the construction of extensible languages, and is the first system to combine local, dynamically extensible context-free syntax with the expressive power of meta-level procedures. The unifying theme of our system is that meaning should be computed relative to local context.

We show how this theme is manifest in an implementation of a Scheme macro system which achieves hygienic macro expansion without rewriting. Additionally, our Scheme macro system makes available compile-time meta-objects for additional power in writing macros; macros that pattern match on compile-time types for optimization at macro-processing time are one example. This approach is currently in use in our RScheme implementation of Scheme.

We also show the how this approach is applied to languages with conventional syntax, using Java as an example. We present a dynamically extensible parser based on the Earley parsing algorithm. This approach is practical as

well as flexible; a straightforward implementation in C parses a 600-line (2777 token) file in about 44ms on an 866MHz Pentium III.

We also describe a language extension framework that makes possible an extensible variant of Java, in which new syntax can be supplied by the casual programmer with only limited knowledge of the underlying compiler implementation or approach. This finally makes available to Java programmers the easy access to structured macro facilities that Lisp programmers find so powerful. Finally, we demonstrate this framework by constructing a deterministic finite automaton language extension to Java.

Contents

Chapter 1

Introduction

The advantages of extensible languages have long been realized by the Lisp community. The ability to easily adjust the language to fit the application, rather than to always adjust the application to fit the language, is at the heart of what Lisp programmers consider the deep power of Lisp [17, 33]. In this work, we show how that power can be made more accessible and powerful even in Scheme, as well as available to programmers in languages with conventional syntax such as Java. The tree-structured transformations of Lisp macros integrated with an extensible parser allow the concepts to be unified. In addition, a well-structured compiler meta-object protocol exposes relevant aspects of the compilation process and provides powerful programmable hooks for extending the language to fit the application.

1.1 Motivation

This work is motivated by the fact that much existing work in extensible languages is either insufficiently expressive in the kinds of extensions that

are permitted (*i.e.*, function libraries) or expressed at the wrong level or in the wrong ways (*e.g.*, purely procedural transformations operating over text strings).

That is, we are primarily motivated by:

- Ease of language implementation
- Ease of language extension
- Ease of re-engineering language implementation (*e.g.*, to change performance tradeoffs to deal with new technologies or new usage patterns)

We recognize the following themes:

- Extensible languages should have extensible compilers.
- Meaning should be expressed naturally through context, especially through *contour sensitive* contexts, which preserve lexical scoping throughout transformation.
- An extensible compiler should have a friendly interface and be integrated with languages using conventional grammars.
- Objects used during the front-end processing of a program should be reified, and be the domain objects of meta-programming.

1.1.1 Our Contribution

With these themes and motivations in mind, in this work we describe an approach that provides:

- Context-local and dynamically modifiable concrete syntax,

- Full context-free syntactic power, and
- A means for defining procedural meta-level code for arbitrary computation at compile-time.

Furthermore, we illustrate this approach in a system with Java as a base language.

1.2 Modularity, Reusability, and Extensibility

The primary means of creating large and complex software systems has been by building relatively simple *program modules*, and composing those modules into larger, more complex software systems. The process for developing the large software system can then be decomposed into the development of smaller program modules. Smaller program modules are easier to understand, develop, and test, and well-constructed modules can be reused to build other software applications.

A refined use of modularization is for *program layering*. Program layering arranges modules into layers whose role is to transform the program concepts at a higher level of abstraction to those at a lower level of abstraction. A layer is then *extending* the language below the layer, creating a new (possibly superset) language.

Figure 1.1 shows how we draw the relationship between the language L0 below the layer, the module layer, and the language created by the layer, L1.

Layering is a powerful structuring tool, and has been used in systems from the Basic Linear Algebra Subroutines to the 7-layered OSI protocol stack