

Copyright

by

Donovan Michael Kolbly

2002

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

Abstract	iv
Chapter 1 Introduction	1
1.1 Motivation	1
1.1.1 Our Contribution	2
1.2 Modularity, Reusability, and Extensibility	3
1.3 Syntax-Directed Translation	7
1.4 Summary of Basic Approach	8
1.5 Scope of Work	8
Chapter 2 Macro Systems	10
2.1 Introduction	10
2.2 Background	11
2.2.1 C Macros	12
2.2.2 Lisp Macros	20
2.2.3 Scheme Macros	25
2.2.4 Systems Related to Macros	27
2.3 A Taxonomy of Scheme Macros	29
2.3.1 Call-by-name Inline Procedures	29

2.3.2	Advertent Capture	30
2.3.3	Explicit Intentional Capture	31
2.4	Our Contribution	32
2.4.1	RScheme Macros	32
2.4.2	Type Reflective Macros	33
2.5	Our Implementation	33
2.5.1	Operation	34
2.5.2	Reflection Operators	46
Chapter 3 Extensible Parsing		50
3.1	Introduction	50
3.2	Background	53
3.2.1	What We Would Like to Do	53
3.2.2	Granularity of Grammar Changes	54
3.2.3	Applications of Our Approach	54
3.2.4	Limitations	55
3.3	Contour Sensitivity	56
3.3.1	Major Styles of Environment Passing	56
3.3.2	General Mechanism	57
3.4	Implementing Contour Sensitivity	58
3.4.1	Frequent Grammar Changes	58
3.4.2	Simple Interpretation	60
3.5	Implementation of Continuation Passing Parser	61
3.5.1	General Description	61
3.5.2	Representation of Rules	62
3.5.3	Dynamic Rule Compilation	65

3.5.4	Major Styles in Terms of Mechanism	65
3.5.5	Examples	67
3.5.6	Performance	71
Chapter 4 Extensible Earley Parsing		72
4.1	Introduction	72
4.2	Description of Earley Parsing	72
4.2.1	States	74
4.2.2	State Sets	74
4.2.3	Initial Conditions	75
4.2.4	Processing	75
4.2.5	Prediction	77
4.2.6	Scanning	78
4.2.7	Completion	79
4.2.8	Relationship to Tomita Parsing	80
4.3	Advantages to the Earley Approach	81
4.3.1	Flexibility	81
4.3.2	Extensibility	82
4.3.3	Understandability	84
4.3.4	Complexity	85
4.4	Drawbacks to the Earley Approach	85
4.4.1	Expressiveness	85
4.4.2	Performance	86
4.5	Extensibility	87
4.5.1	Scope Issues	87
4.6	Our Implementation	89

4.6.1	Details	89
4.6.2	Meta-syntax	90
4.6.3	Meaning Computation	92
4.6.4	Performance	95
4.7	Literal Equivalence	97
4.8	Improvements to Basic Earley	101
4.8.1	Conflict resolution	101
4.8.2	Pruning states using FIRST	102
4.8.3	Approximating FIRST	102
Chapter 5 Compiler Extension Framework		104
5.1	Capabilities of Extension Framework	104
5.1.1	Declarative, Pattern-Based Transformation	105
5.1.2	Pattern Matching Synthesized Attributes	105
5.1.3	Procedural Code-Production Mechanisms	106
5.2	Elements of an Extension Framework	106
5.3	Implementation	107
5.3.1	Meta-language	107
5.3.2	Syntax Evaluation	109
5.3.3	Recursive Compilation	110
5.3.4	Pattern Variables	111
5.3.5	Local Grammar Changes	114
5.3.6	In-line Computation	116
5.4	Declarative Transformations	116
5.5	Synthesized Attributes	117
5.6	Procedural Code Production	119

5.6.1	Token Sequences	119
5.6.2	Compilation	119
5.6.3	Environments and Syntax	119
5.6.4	Reflection	120
5.7	Modular Syntax	120
5.8	Full Meta-syntax	121
5.8.1	Syntax Declarations	123
5.8.2	Syntax Rules	124
5.8.3	Syntax Pattern Elements	124
5.8.4	Actions	127
5.8.5	Local Variables	132
5.8.6	Inline Actions	132
5.8.7	Example	133
5.9	Issues and Future Work	135
5.9.1	Substitution Conformance	135
5.9.2	Translation Recursion	136
5.9.3	Meta-syntax Scope	137
5.9.4	Syntax Module Templates	137
Chapter 6 An Application of an Extensible Language		138
6.1	Introduction	138
6.2	Implementation Approach	140
6.2.1	Declaring the Extension	141
6.2.2	Building the Final Meaning	142
6.2.3	Declaring the State-Keeping Variable	145
6.2.4	Declaring the Java Class's Entry Method	146

6.2.5	Declaring the Java Class's Accessor Methods	147
6.2.6	States Within an Automaton	147
6.2.7	Building the State Switcher	151
6.2.8	Declaring transitions	152
6.2.9	Symbolic State Names	153
6.3	Example Use of the DFA Extension	155
6.3.1	Sample Extended-Java File	155
6.3.2	Generated class definition	155
6.3.3	Generated process() method	157
Chapter 7 Final Words		159
7.1	Related Work	159
7.1.1	Syntactic Exposures	159
7.1.2	Term Rewriting	159
7.1.3	Hygienic Macro Expansion	160
7.1.4	Reusable Generative Programming	161
7.1.5	Adaptable Grammars	161
7.1.6	Open Compilers and MOPs	162
7.2	Limitations and Future Work	165
7.2.1	Meta-Object Protocol	165
7.2.2	Error Reporting	165
7.2.3	Synthesized Attributes	166
7.3	Conclusions	169
Bibliography		171
Vita		177

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