

AN ABSTRACT OF THE THESIS OF

Rajeev K. Pandey for the degree of Doctor of Philosophy in Computer Science
presented on August 28, 1998.

Title: LacEDAemon: A Programming Environment for the
Multiparadigm Language Leda.

Abstract approved: _____

Timothy A. Budd

Multiparadigm programming languages are a recent development in the realm of programming languages. A multiparadigm programming language allows the use of multiple, differing programming paradigms without departing from a single, unified linguistic framework. Multiparadigm programming languages are claimed to have benefits to both pedagogy and complex application creation. The beneficial claims of multiparadigm languages have yet to be validated. The availability of a programming environment would encourage and expedite academic and industrial validation.

Creating a programming environment is considered an extremely labor-intensive activity. Further complications arise from the fact that programming environment creation is an experimental activity: the component mix that best expedites program development in a new programming language cannot be predicted in advance. As a result, few new languages are ever verified in the context of a supportive programming environment. Leda, a unique programming language

that includes the functional, imperative, logic and object-oriented paradigms, is at this juncture.

This thesis describes the structure of an environment framework that allows for experimental study of the necessary components of a multiparadigm programming language environment. New tools and techniques, as well as changes to traditional tools and techniques are required to allow programmers to abstract effectively across paradigms. This research examines the topic by creating LacEDAemon, a testbed programming environment for the multiparadigm programming language Leda, within the framework of a variety of integrated, cohesive tools. LacEDAemon relies on a hypertool-based toolkit integration framework architecture that affords both loose and tight control integration, as well as data integration, using existing, off-the-shelf tools written in a variety of programming languages.

Along with demonstrating the viability of hypertool integration as a low-cost approach for constructing programming environments, LacEDAemon provides a vehicle for: determining an effective multiparadigm programming toolset, studying multiparadigm program design, conducting studies of multiparadigm program visualization, exploring different strategies for software reuse, and examining the merits of conducting all programming activity within the database-centered environment approach. This environment also provides support for investigations in the areas of multiparadigm algorithms, multiparadigm software metrics, and multiparadigm program comprehension. Various techniques for evaluating integrated environments are also applied to LacEDAemon.

LacEDAemon: A Programming Environment for the
Multiparadigm Language Leda

by

Rajeev K. Pandey

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed August 28, 1998
Commencement June 1999

Doctor of Philosophy thesis of Rajeev K. Pandey presented on August 28, 1998

APPROVED:

Major Professor, representing Computer Science

Chair of the Department of Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Rajeev K. Pandey, Author

Approved by Committee:

Major Professor (Timothy A. Budd)

Committee Member (Bella Bose)

Committee Member (Curtis R. Cook)

Committee Member (Paul Cull)

Graduate School Representative (Rod A. Harter)

Date thesis presented August 28, 1998

ACKNOWLEDGMENT

Without the contribution of a number of people, this work would have not have been possible. First and foremost, the members of my Ph.D. committee. I am very grateful for the support, patience and encouragement of my advisor Timothy Budd, who taught me much about both languages and life. To Paul Cull, for always being there for me. To the other members of my committee—Bella Bose, Curtis Cook, and Rod Harter—for their assistance. Other faculty members whose help and encouragement was invaluable include Margaret Burnett, Lawrence Crowl and David Sandberg. Michael Poole at Montana Tech started me out on this journey.

A number of individuals at my various employers over the years provided much mentoring and opportunity for growth. At Hewlett-Packard: Sankar Chakrabarti, Dennis Harms, Bob Miller, Jacek Walicki and Ted Wilson. At Intel: Clark Nelson and Peter Plamondon. At Sun Microsystems: James Kempf, Jim Mitchell, and John Rose. At Western Oregon University: John Marsaglia.

To Tom Bowler, Timothy Justice, Brian Maillard and Nabil Zamel, for being the best of friends, colleagues, and participants in long discussions.

Finally, to my family. My wife Sheri has been a constant source of love and encouragement throughout my graduate career. To the stress-reducing presence of our feline and canine children Puffin, Leopold, Sunkist, Opus, Nova and Katie. To my brother Sunjeev, his wife Arlene, and children Sonya and Sophia, for providing me with a home away from home. To my mother, for her love and unwavering belief in me. To my father, for his teachings and example.

May I someday be worthy of such love and support.

TABLE OF CONTENTS

	<u>Page</u>
1 INTRODUCTION	1
1.1 Multiparadigm Programming Languages	3
1.2 The Multiparadigm Programming Language Leda	5
1.3 Programming Environments	8
1.4 The LacEDAemon Programming Environment	10
1.5 Tcl/Tk, the Hypertool Concept and Software Reuse	12
1.6 What's In A Name: LacEDAemon and Leda	14
1.7 Thesis Outline	15
2 MULTIPARADIGM PROGRAMMING AND LEDA	17
2.1 Never Mind <i>the</i> Paradigm: Multiple Paradigms vs Multiparadigm....	17
2.1.1 The Imperative Paradigm	18
2.1.2 The Object-Oriented Paradigm	19
2.1.3 The Functional Paradigm	22
2.1.4 The Logic Paradigm	24
2.2 Where Should We Teach a Multiparadigm Language?.....	26
2.2.1 The Graduate Level	27
2.2.2 The Compiler Construction Course	27
2.2.3 The Programming Languages Course	29
2.2.4 The CS1/CS2 Courses	30
2.3 Conclusions	31

TABLE OF CONTENTS (Continued)

	<u>Page</u>
3 PROGRAMMING ENVIRONMENTS	32
3.1 Classification of Environments	33
3.2 Language-Centered Environments	34
3.3 Structure-Oriented Environments	35
3.4 Toolkit Environments	37
3.5 Method-Based Environments	39
3.6 AI Programming Environments	46
3.7 Constructing Programming Environments	47
3.8 Communication-Based Environments	47
4 AN OVERVIEW OF LACEDAEMON	53
4.1 Introduction	53
4.2 Integration	53
4.3 Tool Integration Frameworks	55
4.4 Tasks and Tools	58
4.5 The Hypertool Architecture Of LacEDAemon	59
4.5.1 Messaging In Tcl/Tk	61
4.5.2 Embedding Interpreters in Tools	63
4.5.3 Combining Message and APIs	68
4.6 LacEDAemon Messages	68
4.6.1 Request Messages	70
4.6.2 Response Messages	72

TABLE OF CONTENTS (Continued)

	<u>Page</u>
4.6.3 Notification Messages	72
4.7 Extending LacEDAemon.....	75
5 THE LACEDAEMON PROGRAM DATABASE	77
5.1 Introduction: PostgreSQL	77
5.2 Using Postgres as the LacEDAemon repository.....	78
5.2.1 Representing LacEDAemon Participant Information	79
5.2.2 Representing Leda Programs	80
5.3 The Leda Program Information Extractor	83
5.4 An Example	85
5.5 A Revision Control System	86
5.6 Persistent Objects	87
6 THE PRETTYPRINTER.....	89
6.1 Prettyprinting.....	89
6.2 First Class Functions in Leda.....	90
6.3 The Leda Prettyprinter	90
6.4 An Example	92
6.5 Database Directed Prettyprinting	94
7 THE PROGRAM ANIMATOR.....	95
7.1 Leda does the Polka	95

TABLE OF CONTENTS (Continued)

	<u>Page</u>
7.2 Using Polka/Samba to Present Leda Execution	97
7.3 Mediators: Addressing Impedance Mismatch in Tools	98
7.4 Algorithm Animation in LacEDAemon	101
8 THE PROGRAM EDITORS AND BROWSERS	104
8.1 A Customized Editor for Leda	105
8.2 A Smalltalk-Like Browser	107
8.3 Depicting Class Hierarchies	109
9 LITERATE MULTIPARADIGM PROGRAMMING	113
9.1 CWEB	115
9.2 Adapting Literate Programming to LacEDAemon	115
10 DEBUGGING MULTIPARADIGM PROGRAMS	117
10.1 Debuggers	117
10.2 Augmenting The Leda Interpreter	119
10.2.1 Adding Breakpointing	120
10.2.2 Single-Stepping Through Programs	120
10.2.3 Examining Symbol Values	121
10.2.4 Watchpointing	121
10.2.5 Fault Detection	123
10.3 Composing Interactions With LacEDAemon Tools	123

TABLE OF CONTENTS (Continued)

	<u>Page</u>
11 EVALUATION	125
11.1 Evaluation of LacEDAemon	125
11.1.1 The SMP Model and IFCS Taxonomy	125
11.1.2 Tool Integration in LacEDAemon	127
11.1.3 The EBI Framework	129
11.1.4 Comparing Inter-Tool Communication	130
11.1.5 The CEARM Model	131
11.2 Future Directions	137
12 CONCLUSIONS	139
12.1 Contribution of This Work	139
12.2 Assessing Reuse in LacEDAemon	144
12.3 Conclusions	146
BIBLIOGRAPHY	147
APPENDICES	164
APPENDIX A Leda Grammar	165
APPENDIX B Leda Program to Simulate a Turing Machine	173

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1.1 The general-purpose multiparadigm language Leda.	6
2.1 Imperative Programming In Leda.	18
2.2 Object-Oriented Programming In Leda.	21
2.3 Functional Programming In Leda.	22
2.4 Logic In Leda.	25
2.5 A Multiparadigm Compiler Model.	28
3.1 The ECMA/NIST “Toaster” Model.	43
4.1 The Three Dimensions of Integration.	55
4.2 Idealized Integrated Tool.	56
4.3 A Generalized LacEDAemon Hypertool.	60
4.4 The lc.tcl Script.	62
4.5 The send_msg.tcl Script.	62
4.6 Embedding a Tcl Interpreter.	64
4.7 The Spinbox Application.	65
4.8 Leda Code for the Spinbox.	66
4.9 Leda Code for Graphics Library.	67
4.10 Messaging in the LacEDAemon Environment.	69
6.1 Currying In Leda.	91
6.2 Currying Example, Prettyprinted.	93
7.1 An Example Leda Program.	97
7.2 Leda Statement Trace Output.	99
7.3 Samba/Polka Animation Commands.	100
7.4 Execution Visualization in LacEDAemon.	101

LIST OF FIGURES (Continued)

<u>Figure</u>		<u>Page</u>
7.5	Execution Visualization using Samba.	102
8.1	The Leda Major Mode in Emacs.	106
8.2	The Leda Menu in Emacs.	107
8.3	Prototype of a Smalltalk-like Browser.	108
8.4	Storing Class Hierarchy Information.	110
8.5	Graphical View Of Class Hierarchies.	111
8.6	Class Hierarchy Browser.	112
10.1	Single Stepping in Leda.	122
11.1	Mapping LacEDAemon Tools on the Three Dimensions of Integration.	129
11.2	The Request_send event in LacEDAemon.	132
11.3	The Response_send event in LacEDAemon.	133
11.4	The Response_receive event in LacEDAemon.	133
11.5	Conceptual Environment Architecture Reference Model.	136
11.6	Mapping of LacEDAemon to CEARM.	136

LIST OF TABLES

<u>Table</u>		<u>Page</u>
4.1	LacEDAemon Request Messages	71
4.2	LacEDAemon Response Messages	73
4.3	LacEDAemon Notification Messages	74
5.1	The LacEDAemon Participant Relation	80
5.2	The Project Relation	80
5.3	The File Relation	80
5.4	The Location Relation	81
5.5	The Includes Relation	81
5.6	The Source Relation	81
5.7	The Class Relation	81
5.8	The Declaration Relation	81
5.9	The Declaration Category Call	82
5.10	The Declaration Category Function	82
5.11	The Declaration Category Member	82
9.1	The Documented Project Relation	114
9.2	The Document Relation	114
9.3	The Commented Source Relation	114
11.1	LacEDAemon Explicit CEARM Mapping	135
12.1	Reuse in LacEDAemon: Lines of Code	145

LACEDAEMON: A PROGRAMMING ENVIRONMENT FOR THE MULTIPARADIGM LANGUAGE LEDA

1. INTRODUCTION

Since the advent of programming languages, programmers have been creating *tools* to expedite the programming process. These tools have evolved into collections of tools, or *programming environments*. The next logical step is to gain amplification in the power of a tool via the presence of other tools, or *integration*. Integrated programming environments attempt to provide a cohesive toolset to aid the programmer in constructing, correcting and comprehending programs. Having environment support available has become requisite in both academic and industrial settings, albeit for different reasons.

However, tools are software, and creating software is a difficult process. “Software Engineering Environments are large and complex systems and current experience dictates that it is unwise to build them from scratch” [141]. Indeed, Knudsen et al. [102] note that “developments of environments are very large tasks, comparable more with developing operating systems than compilers...” One viable approach is to create an *integration framework*, within which a variety of *existing* tools can be integrated. While some instances of these integration frameworks exist, they have been applied to few languages. In most part, this is due to the large programming effort required to adapt existing tools to these complex integration frameworks. How to build integrated software development environments is still an open and very active area of research [102].

In the past decade, a radical new approach in programming languages, called multiparadigm programming languages, has integrated disparate programming paradigms within a common linguistic framework. No environment for these languages exists, and the value of multiparadigm languages is still in question. Many other experimental languages are to be found in the same situation: without a suite of tools to provide programmer support, their application in either industrial or academic settings remains a difficult issue¹.

Here we describe and demonstrate a new method of creating a tool integration framework, as well as integrating existing tools into a framework, for a programming environment for a multiparadigm programming language called Leda. The somewhat unique position of the Leda language in the realm of multiparadigm languages is indicated via a short introduction to Leda also described here.

The primary contribution of this dissertation to the field of software engineering is to describe and demonstrate a new approach to creating a tool integration framework. By embedding scripting language interpreters within existing tools (which are then called *hypertools*), we provide a powerful and low-cost approach for various forms of integration. The primary contribution of this dissertation to the field of programming languages is to demonstrate the hypertool integration framework approach in creating a programming environment for the multiparadigm programming language Leda, enabling more detailed investigation of the implications of multiparadigm programming than would otherwise be possible. LacEDAemon, the resulting hypertool integration framework-based programming environment for the multiparadigm programming language Leda, also provides a testbed for the design

¹The Language List, version 2.4, January 1995 (<http://cuiwww.unige.ch/langlist>), identifies over 2350 different programming languages.

and deployment of new programming tools. The general hypertool approach also provides a mechanism for other experimental programming languages to more easily develop an associated programming environment than otherwise possible. The integration of a wide variety of tools to form a single application via a messaging framework and common graphical interface is also applicable to domains other than programming environments.

In the next few sections we briefly examine some of the key aspects to this thesis: multiparadigm programming languages, the multiparadigm language Leda, programming environments, and the hypertool concept. We conclude with an outline of the thesis.

1.1. Multiparadigm Programming Languages

The term *paradigm* was popularized by Thomas Kuhn, whose historical account on scientific progress *The Structure of Scientific Revolutions* [109] employed the term to describe a prevailing view of the world held by scientists. Peaceful interludes of scientific progress were punctuated by paradigm shifts—revolutions in scientific thought that occurred when the prevailing model could no longer support new discoveries, and a new paradigm had to be originated to incorporate these new observations.

The term *paradigm* was introduced to computer science via R.W. Floyd’s 1978 Turing Award Lecture [61] in which he noted “programming languages typically encourage use of some paradigms and discourage others.” When used in the context of programming languages, paradigm also refers to a view of the world, but in this case the world view is imposed on the user of the language due to language features. In the realm of computing, multiple, differing paradigms coexist in the sense that

the use of different languages requires the user to view the world differently, and in terms of constructs provided by the language.

A multiparadigm programming language is a system that incorporates two or more of the conventional programming paradigms [77], or a linguistic framework which does not force the programmer into thinking or working in only one model [24]. The January, 1986 issue of *IEEE Software* was a special issue devoted to multiparadigm languages, and there is presently considerable research interest in the area.

Several approaches have been pursued in the creation of multiparadigm languages:

- *augmented languages*

Augmented languages add additional paradigms to an existing language to permit users to utilize a new programming style without learning a completely new language. The additional paradigm usually represents a natural progression or evolution of the language, based on experience. For example, C++ [179] extends C [100] with support for object-oriented programming.

- *hybrid languages*

Hybrid languages typically extend an existing functional or logic programming language by embedding other paradigms. The primary motivation for these languages is to provide a wide range of standard programming and knowledge representation paradigms for solving complex problems in areas such as artificial intelligence [121]. Loops [176] is an example of this type of language.

- *general-purpose languages*

General-purpose multiparadigm languages seek to find the “ideal” blending of

several major paradigms in order to provide a more expressive programming vehicle for general problem solving. Leda [33] exemplifies this approach.

Most existing multiparadigm programming languages are not general-purpose, but belong to the augmented or hybrid variety, and are a result of the lack of very specific language features deemed necessary for the solution of a specific problem. Rather than attack the problem with an inelegant solution, the language is extended in one of a variety of ways. Here there is a close affinity between hybrid multiparadigm programming languages and a particular problem, since the language is essentially designed with the specific application in mind. While these languages have definite advantages in the context of a specific problem, they may not easily generalize to solving different problems. Combining the foundational paradigms to create languages like Leda has been identified as the next step in general-purpose language evolution [67]:

All the major programming language styles - procedural, functional and logical - have application domains where they are particularly effective. This suggests that general-purpose programming languages must embrace a number of these different approaches. Consequently, it seems likely that the future of the major general-purpose programming languages will be as multi-paradigm languages.

1.2. The Multiparadigm Programming Language Leda

Leda is a strongly typed, general-purpose multiparadigm programming language designed by Budd [33]. Programming paradigms supported by Leda include the imperative, object-oriented, functional, and logic. The constraint programming paradigm is also facilitated in recent versions of Leda [182, 200].

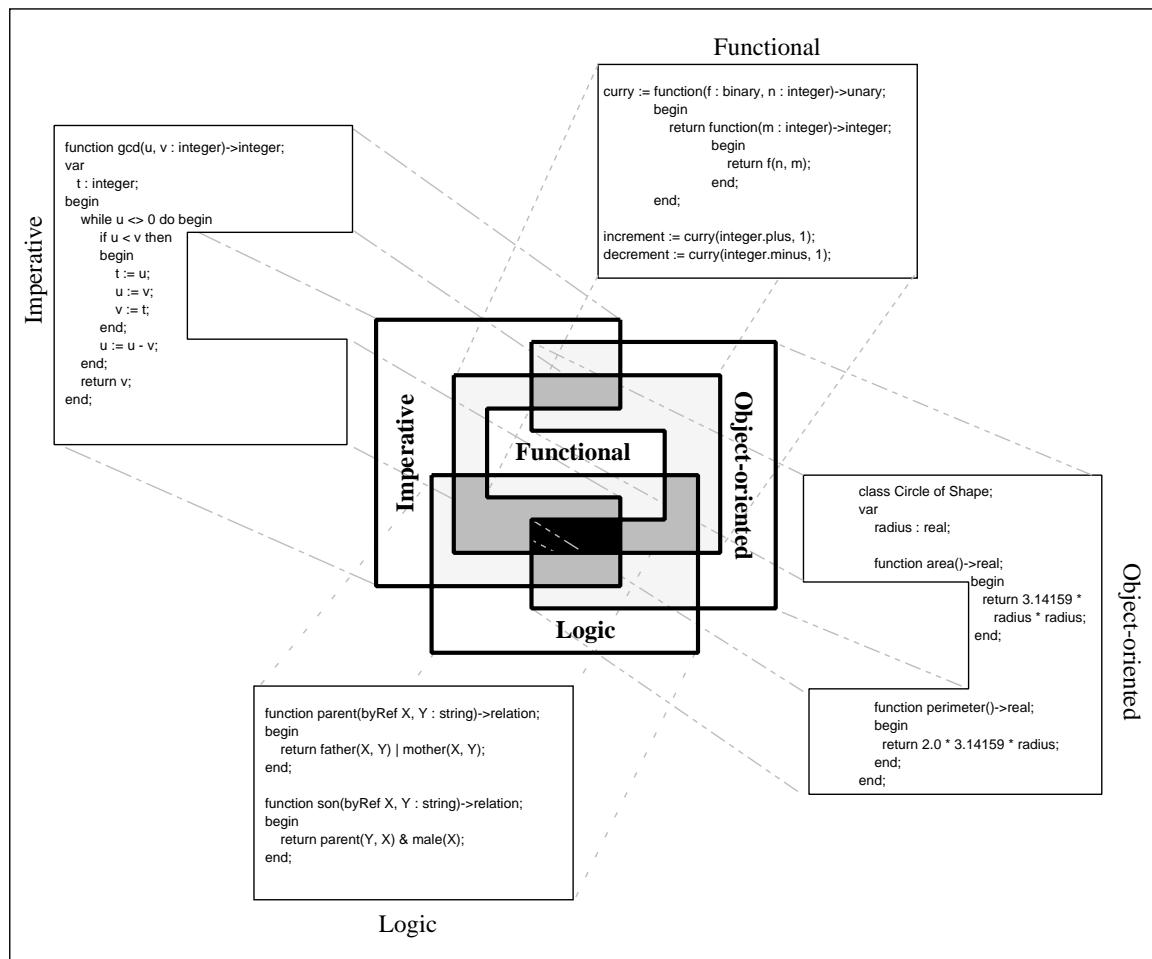


FIGURE 1.1. The general-purpose multiparadigm language Leda.

Leda combines all four foundational paradigms enumerated above. Retaining conciseness while combining the four paradigms was a design goal of Leda. While Leda adds two more paradigms than C++, the resultant language is actually smaller than C++². By emphasizing the essential features—the *exemplars*—of the constituent paradigms, Leda facilitates blendings at various levels of granularity, from individual statements to complete modules. Figure 1.1 illustrates our view of general-purpose multiparadigm languages. This view permits the programmer to apply any of fifteen possible paradigm blendings to a single problem. The figure further illustrates Leda’s articulation of this view.

Leda was designed and implemented in order “to provide a vehicle for experiments in multi-paradigm programming” and to “explore the advantages of using various programming paradigms on assorted problems” [30]. Leda is an evolving research language. For more information on the programming language aspects of Leda, see [24] and [32]. Chapter 2 provides more information on Leda, and Appendix A describes the syntax of Leda.

²For example, Leda has 54% fewer keywords and 67% fewer operators than C++

1.3. Programming Environments

Providing tools designed to aid the programmer in constructing programs has been a topic of research nearly as long as there have been programming languages³. Collecting these tools into a coherent framework is an idea that is over 30 years old. These tool collections have been known by many names:

- Programming environment (PE) – a collection of tools that supports only the coding phase of the software development lifecycle.
- Integrated Development Environment (IDE) – same as PE, a common term in the realm of personal computers.
- Software Development Environment (SDE) – a collection of tools that augments or automates all the activities comprising the software development lifecycle.
- Integrated Project Support Environment (IPSE) – a set of tools covering the entire software lifecycle.
- Software Engineering Environment (SEE) – same as IPSE.
- Computer-Aided Software Engineering (CASE) – same as IPSE.

³As Floyd wryly observed in his Turing Award lecture [61]: “I would rather write programs to help me write programs than write programs.”

There are also “integrated” SEE and CASE environments (ISEE and ICASE). The terminology is often confusing or misleading. For example, terms such as tool, workbench, toolset, and environment are given very different meanings and interpretations [64]. Since the main thrust of LacEDAemon is to provide tools to support the coding of Leda programs, we will use the term “programming environment” throughout.

Dart et al. [53] proposed a taxonomy of environments (more to clarify trends rather than categorize):

- Language-centered environments – specially built to support a specific language
- Structure-oriented environments – arose from syntax-directed editors, now implies manipulation of program structures
- Toolkit environments – collections of small tools
- Method-based environments – support of development methods for certain phases of the software development cycle, or support for methods of managing the development process (now called process-centered or process-oriented environments) enforced by the tools

Language-centered environments tend to get too big for a single person to comprehend or extend, and their tight coupling inhibits reuse of tools. Structure-oriented environments are only capable of generating editors (and other tools) for

syntactic checking, not semantic checking. Process centered environments require a well-articulated methodology to be in place.

There are three basic strategies that have been employed to create programming environments:

- build as a single, monolithic system - such as Interlisp [186], Cedar [185], and CPS [184].
- build as a collection of independent tools - such as Multics [134] and PWB/UNIX [56].
- build as a set of related tools with a communication mechanism - FIELD [151], HP SoftBench [44] and DEC FUSE [80]

LacEDAemon provides a variation on the communication mechanism-based toolkit environment. Programming environments are examined in more detail in Chapter 3.

1.4. The LacEDAemon Programming Environment

Multiparadigm programming languages have been an active research area for over a decade. Much of the research effort has been devoted to developing implementation techniques for languages spanning several paradigms, as well as experimenting with varying the blended paradigms. Recently, attention has turned to the applicability of these languages to real-world problems. Some problems that

have been touted as suitable for multiparadigm solution include a programming language compiler [24], a stock market exchange [89], and a telephone network simulation [202]. Use of the language Leda in solving these problems has been examined as well [95, 36]. The scale of these problems necessitates the availability of a programming environment. Pedagogic applications of multiparadigm languages have also been recently identified [146, 38]. For broad-based penetration of the computing curricula, an environment supporting multiparadigm languages is also necessary.

A programming environment provides the user of a language with a collection of cooperating tools that expedite one or more aspects of program development. Programming environments can be structured in a variety of ways. One approach consists of tightly integrated, custom built tools seamlessly (albeit rigidly) connected together. Most earlier environments such as Interlisp, developed by Warren Teitelman at XEROX [186] employed this technique. Another approach, called the control integration toolkit model, involves a flexible framework of loosely-coupled, “off-the-shelf” tools that request information and services by exchanging messages. Field [151] originated the control integration model. Industry examples utilizing this approach include DEC FUSE, HP SoftBench, and Sun SPARCworks.

LacEDAemon is a prototype programming environment for Leda which includes tools to aid in the creation, correction, and comprehension of Leda programs. LacEDAemon is also designed to serve as a testbed for experiments in toolset com-

position, since development of tools and methodologies for multiparadigm programming is an ongoing activity. The architectural basis for LacEDAemon is the hypertool concept, which allows for the integration and customization of existing tools along with newly constructed tools in a cohesive framework. The hypertool concept and the Tcl/Tk scripting language provides control integration (and possibly presentation integration) at low cost, while providing considerable flexibility by embedded Tcl interpreters within the tools. Construction of LacEDAemon is expedited by the ability to reuse and customize existing tools, while the flexibility and extensibility of Tcl/Tk also allows for experiments in toolset composition.

The three primary activities that LacEDAemon supports, namely creation, correction, and comprehension are intertwined, implying the tools must be interconnected and be able to request information and services from one another. An overview of the architecture of LacEDAemon is provided in chapter 4.

1.5. Tcl/Tk, the Hypertool Concept and Software Reuse

Tcl (“tool command language”) [136] is an embeddable scripting language for controlling and extending applications. Tcl provides generic programming facilities such as variables, loops, and procedures. The Tcl interpreter is a library of C procedures that can be incorporated into applications, with each application extending the core Tcl features by providing additional commands for that application.

Tk [138] extends Tcl by providing a toolkit for the UNIX X Window System⁴. Together, Tcl and Tk provide a programming system for developing and using graphical user interface (GUI) applications. Tk extends the core Tcl facilities with commands that allow the construction of Motif-like user interfaces within Tcl scripts instead of C code. Tk, like Tcl, is implemented as a set of C procedures, allowing extensive application. Base Tk features can also be extended through the creation of user-interface widgets and geometry managers written in C.

Together, Tcl and Tk provide four major benefits, according to Ousterhout [137]:

- rapid development
- applications provided a powerful scripting language
- an excellent “glue language”
- user convenience

Scripting languages allow rapid development of gluing-oriented applications. Recently, Ousterhout [135] provided anecdotal support for the claim that the difference (in terms of code and development time) between scripting languages and system programming languages was a factor of five to ten. It is this difference in code and development time that allows LacEDAemon to be created by a single tool

⁴Tcl and Tk have also been ported to other platforms, including Windows 95/NT and Macintosh.

integrator in short order. The hypertools provide the system code to be reused (the “bricks”), while the Tcl language provides the glue and system substrate (the “cement”).

Ousterhout [137] describes the hypertool concept while describing the Tcl construct `send`:

`send` is intended to encourage the development of small reusable applications called *hypertools*. Many of today’s windowing applications are monoliths that bundle several different packages into a single program. For example, debuggers often contain editors to display the source files being debugged...Unfortunately, each of these packages can only be used from within the monolithic package that contains it.

With `send` each of these packages can be built as a separate stand-alone program. Related programs can communicate by sending commands to each other. For example, a debugger can send a command to an editor to highlight the current line of execution...With `send` it should be possible to reuse existing programs in unforeseen ways...The term “hypertools” reflects this ability to connect applications in interesting ways and to reuse them in ways not foreseen by their original designers.

LacEDAemon embodies Ousterhout’s [137] advice regarding hypertools: “When designing Tk applications, I encourage you to focus on doing one or a few things well; don’t try to bundle everything in one program. Instead, provide different functions in different hypertools that can be controlled via `send` and reused independently.”

1.6. What’s In A Name: LacEDAemon and Leda

The name for the language Leda arose from early example programs that implemented a genealogical database of mythological figures. Leda was the wife of

King Tyndareus, and mother of the famed Helen of Troy. The name Leda was also thought to be ephoneous and unique⁵. Some effort has been made by other Leda-related projects to continue the mythological naming style (Electra [199], Argo [93]).

Lacedaemon is actually the second name proposed for the Leda programming environment. The environment name was originally Sparta, which is the name of the kingdom where Leda and her husband Tyndareus lived. An unintended implication of this name was the fact that the environment might be spartan. As a result, the environment name was changed to Lacedaemon (las-e-dē'mon), which is another name for the kingdom of Sparta, with the letters that spell “Leda” capitalized for emphasis.

1.7. Thesis Outline

Chapter 2 introduces Leda, and collects various thoughts on the uses of multiparadigm languages in vocational and educational settings. Chapter 3 surveys programming environments, providing a historical, taxonomic and developmental context within which to place the LacEDAemon programming environment. Chapter 4 provides a detailed architectural overview of the LacEDAemon programming environment, describing the hypertool integration framework and messaging mech-

⁵Other projects that now share the same name include the LEarning Design Assistant project [83], a visual programming project [126], a C++ data structures project (Library of Efficient Data structures and Algorithms) [125], and a French electronic design automation company, LEDA, S.A.

anism. Chapter 5 describes the LacEDAemon program database, which serves both as a repository of program information as well as definitions of event-based tool interactions. Chapters 6 through 10 describe some components of the LacEDAemon framework that comprise both a useful collection of tools and an illustration of the viability of the integration mechanism. Chapter 6 describes the Leda pretty printer, which utilizes program database information to enhance presentation. Chapter 7 describes program execution illustration in LacEDAemon. Chapter 8 describes program browsers—Smalltalk-style as well as traditional text editor-style, along with tools for the graphical depiction of class and file inclusion hierarchies—adapted for use in LacEDAemon. Chapter 9 describes how the collection of tools cooperate to provide program debugging support. Chapter 10 explores adapting literate programming systems to LacEDAemon, and in doing so illustrates the possibility of scaling up the hypertool integration approach to create software development environments. Chapter 11 provides comparisons of LacEDAemon to other environments as well as an evaluation of the contributions of this thesis. Chapter 12 concludes this dissertation.

2. MULTIPARADIGM PROGRAMMING AND LEDA

2.1. Never Mind *the* Paradigm: Multiple Paradigms vs Multiparadigm

The title of this section is inspired by Paul Luker's article "Never Mind the Language, What About the Paradigm?" [117]. While examining language selection in introductory Computer Science courses, Luker observed:

If it is too early for our current paradigm to change, we should, at least, produce students who will be in a position to adapt to any paradigm shift that occurs....This comes back to a solid theoretical foundation and a broad treatment of programming, which examines the strengths and weaknesses of different approaches, with practical experience to reinforce the issues.

Educators are beginning to recognize that selecting and teaching *the* paradigm may not be the correct approach, although having the particular paradigm chosen be one which coincides with present industry trends certainly has vocational, if not educational, value. It is not clear that one specific paradigm can be chosen as the clear winner. Many newer languages employ a hybrid approach, combining more than one "traditional" programming paradigm in some fruitful way.

In order to prepare our students for any future paradigm shifts, it is certainly inadequate to embrace one programming paradigm as *the* paradigm, and it may not even be enough to employ a hybrid language that incorporates more than a single paradigm. What we would like to do is give students exposure to all of the programming paradigms considered important today. With the current trend of

```
{ Euclid's GCD from Sedgewick }

while u <> 0 do
begin
  if u < v then
  begin
    t := u;
    u := v;
    v := t;
  end;
  u := u - v;
end;
return v;
```

FIGURE 2.1. Imperative Programming In Leda.

languages which include several paradigms, the approach used should emphasize the complimentary aspects of paradigms, not just their competing aspects.

2.1.1. The Imperative Paradigm

The imperative paradigm is the most common view of the computer. The emphasis is on variable locations, values, and legal operations that can be performed on these locations. Locations can be grouped, and execution is structured. The imperative programming paradigm exhibits the following characteristics:

- “*word-at-a-time*” *processing*—computation consists of many individual movements and computations of small items of data.

- *programming by side-effect*—computation proceeds by changing the “state” of the machine via assignment.
- *machine-oriented structure*—language structures, both data and control, are *very* close to the underlying machine architecture.

Examples of the imperative programming paradigm include the languages Pascal [90] and C [100]. Leda provides the standard imperative constructs, as illustrated in Figure 2.1. Encoded here is Euclid’s algorithm for computing GCD, taken from Sedgewick [165].

The correspondence between imperative constructs and the underlying machine is both a strength and a limitation. Petre and Winder [145] observe:

This closeness of language to machine model is reflected in practical ways, e.g., the good control afforded by many imperative languages over machine aspects such as memory allocation and I/O. However, the strength of correspondence means that imperative languages embody hardware-based restrictions, so that pragmatics may intrude upon expression.

2.1.2. The Object-Oriented Paradigm

Object-oriented programming is a very popular and distinctive approach to problem solving. The object-oriented metaphor views programming as *simulation*. Programs model aspects of the real world: objects with properties. The object-oriented programming paradigm exhibits the following characteristics:

- *interacting agents*—objects encapsulate state and behavior, interacting via *message passing*.
- *hierarchical organization*—objects are hierarchically classified according to similarities in their behavior via a *class structure*, and can be successively refined via *subclassing*, *inheritance* and *overriding*.
- *attribute-based response*—objects interpret messages based on their attributes: both particular properties (*instance variables*) and general properties (*class*).

Figure 2.2 illustrates classes, subclassing, inheritance, message passing, and overriding in Leda. Here the classes `Square` and `Circle` are derived as subclasses of `Shape`. Examples of the object-oriented programming paradigm include the languages Smalltalk [70, 29] and C++ [179, 25].

C++ has become a very popular language in computing curricula, overtaking Pascal as the introductory language of choice. C++ has also become very popular in industry, and this enthusiastic embracing of C++ by industry makes teaching it at some point in the undergraduate curriculum almost imperative. But is C++ the answer, *the* paradigm? In the section titled “Living in a Multi-Paradigm Universe” of the popular book *C++ Primer* Lippman cautions:

Object-oriented programming is an evolutionary advance in the design and management of large software systems. It is not, however, the *deus ex machina* come forth to resolve the many ills of the software industry. Good code is difficult to produce under any paradigm and, once in production, bugs continue to be uncovered. This is unlikely to change for quite some time yet.

```
const pi := 3.1415;

class Shape;
  function area() -> real;
  begin end;
end;

class Square of Shape;
  var
    side : real;

  function area() -> real;
  begin
    return side * side;
  end;
end;

class Circle of Shape;
  var
    radius : real;

  function area() -> real;
  begin
    return pi * radius * radius;
  end;
end;

var
  s : array [Shape];
  i : integer;

begin
  s := newArray[Shape](1,2);
  s.atPut(1,Square(2.5));
  s.atPut(2,Circle(3.8));
  for i := 1 to 2 do
    print(s.at(i).area());
  end;
```

FIGURE 2.2. Object-Oriented Programming In Leda.

```
type
  binaryFunc : function(integer, integer)->integer;
  unaryFunc  : function(integer)->integer;

function curry (boundFun : binaryFunc,
               boundValue : integer)->unaryFunc;
begin
  return function (item : integer)->integer;
    begin
      return boundFun(item, boundValue);
    end;
end;

var
  triple : unaryFunc;

begin
  triple := curry (integer.times, 3);
  print ("triple of 7 " + triple(7) + "\n");
end;
```

FIGURE 2.3. Functional Programming In Leda.

Clearly the software industry may experience future paradigm shifts. We must remember Luker's advice on preparing students for such paradigm shifts.

2.1.3. The Functional Paradigm

While object-oriented programming addresses the behavior of objects in time, functional programming concentrates on timeless mathematical relationships. The

essential idea behind functional programming is to treat functions as values. The functional paradigm has the following characteristics:

- “*first class*” *functions*—functions are assignable, passable, and returnable values.
- *functional composition*—a rich set of functions can be assembled via *functionals* or *higher-order functions*, i.e., functions which take other functions as arguments.
- *referential transparency*—functions whose values can be determined solely by the value of their arguments (pure functions).

Figure 2.3 illustrates a higher-order Leda function that binds one argument of a binary integer operation to some fixed value (in this case the binary function is `integer.times` and the value is 3, yielding `triple`). Examples of functional languages include Haskell [54] and ML [195]. MacLennan [119] covers functional programming in general.

In a recent paper on functional programming, Pountain [147] observes: “The functional paradigm is unlikely to displace C++ anytime soon, but as programmers become more aware that object orientation is not a perfect panacea, there should be room for both, or—dare I suggest it?—for some kind of hybrid approach.”

Bruce MacLennan [120] notes that “it may very well be the case that function-oriented programming and object-oriented programming are *complemen-*

tary, rather than *competing* ... programming language technologies.” Leda represents an example of the hybrid approach that Pountain and MacLennan suggest.

2.1.4. The Logic Paradigm

The logic programming paradigm views computation as a deductive process dealing with facts, rules, and queries. Queries are resolved based on facts and rules of inference. The logic paradigm exhibits the following characteristics:

- *nonprocedural emphasis*—the programmer states *what* is desired, not *how* to accomplish it.
- *lack of directionality in relations*—bound arguments allow information to flow into relations while unbound arguments allow information to flow out of relations.
- *standard deduction view of execution*—programs are propositions that assert the existence of the desired result, with a theorem-proving mechanism attempting to construct the desired result to determine the veracity of the proposition.

Leda provides constructs that allow for the definition of facts and inference rules, as well as mechanisms to perform unification and backtracking. The basis for logic programming in Leda is a boolean procedural abstraction called the *relation*. In Figure 2.4 we describe the inference rule for the `sibling` relationship. Assuming the existence of facts defining the `child` relationship (and inference rules such as

```

const
  names := ["Leda", "Castor", "Helen", "Zeus", "Tyndareus"];
  eq := unify[string]; { shorthand notation for unify function }

function child(byRef name, mother, father:string)->relation;
begin
  return eq(name, "Helen") & eq(mother, "Leda") & eq(father, "Zeus")
  | eq(name, "Castor") & eq(mother, "Leda") & eq(father, "Tyndareus");
end;

:
function fatherOf(byRef dad, kid:string)->relation;
var
  mom : string;
begin
  return child(kid, mom, dad);
end;

function parentOf(byRef parent, kid:string)->relation;
begin
  return fatherOf(parent, kid) | motherOf(parent, kid);
end;

function sibling(byRef left, right: string)->relation;
var
  parent: string;
begin
  return parentOf(parent, left) &
         parentOf(parent, right) & (left <> right);
end;

var x,y : string;

begin
  x := NIL; y := NIL;
  for sibling(x,y) do
  begin
    print(x); print(" and "); print(y); print("\n");
  end;
end;

```

FIGURE 2.4. Logic In Leda.

parentOf allowing the derivation of other facts), this program allows for a variety of derivations depending on the bindings of `x` and `y`. The function `eq` determines the equality of defined variables, or provides a binding that can be undone using a reversible assignment operator if the first argument to `eq` is unbound.

Prolog [49] is one of the most popular logic programming languages. Details of other logic programming languages can be found in Hogger [86].

Bobrow [15] suggests a multiparadigm approach, instead of just using the logic programming paradigm:

Logic programming provides a non-procedural representation of knowledge, combined with a powerful database search facility...although this combination is very powerful, it is inappropriate for some problems. By having a number of other programming paradigms as well, one can build more understandable programs more quickly. No single paradigm is appropriate to all problems, and powerful systems must allow multiple styles.

2.2. Where Should We Teach a Multiparadigm Language?

The activity in multiparadigm language research makes it apparent that many new languages will include multiple programming paradigms. The pedagogic value of these languages has also started to be recognized. Placer observes [146] that multiparadigm languages could be used both as a tool to teach programming as well as creative problem solving. We now examine possible points in the curriculum where the introduction or use of such a language might be appropriate.

2.2.1. The Graduate Level

New ideas are usually studied first in the graduate curriculum, where students are predisposed to be at the cutting edge. Leda has been the subject of graduate courses and seminars at Oregon State University as well as in Europe for a couple of years now. Just as many years ago the object-oriented paradigm was examined only in graduate level courses, perhaps the best approach to introducing multiparadigm languages into computing curricula is to start at the graduate level and let the ideas percolate downwards.

2.2.2. The Compiler Construction Course

In a recent paper, Justice, Pandey, and Budd [95] describe the construction of a compiler for a subset of the C language using Leda and a multiparadigm approach. In this compiler, parsing is done via logical relations, the symbol table is constructed from objects, and optimizing transformations on the intermediate form are described in a functional manner. Control of the compilation process is provided by the imperative paradigm. This architecture is illustrated in Figure 2.5.

Since the compiler construction course normally follows a course on programming languages in the undergraduate curricula, students would be aware of the various programming paradigms. Hence the compiler construction course could employ a multiparadigm programming language, and reinforce student understand-

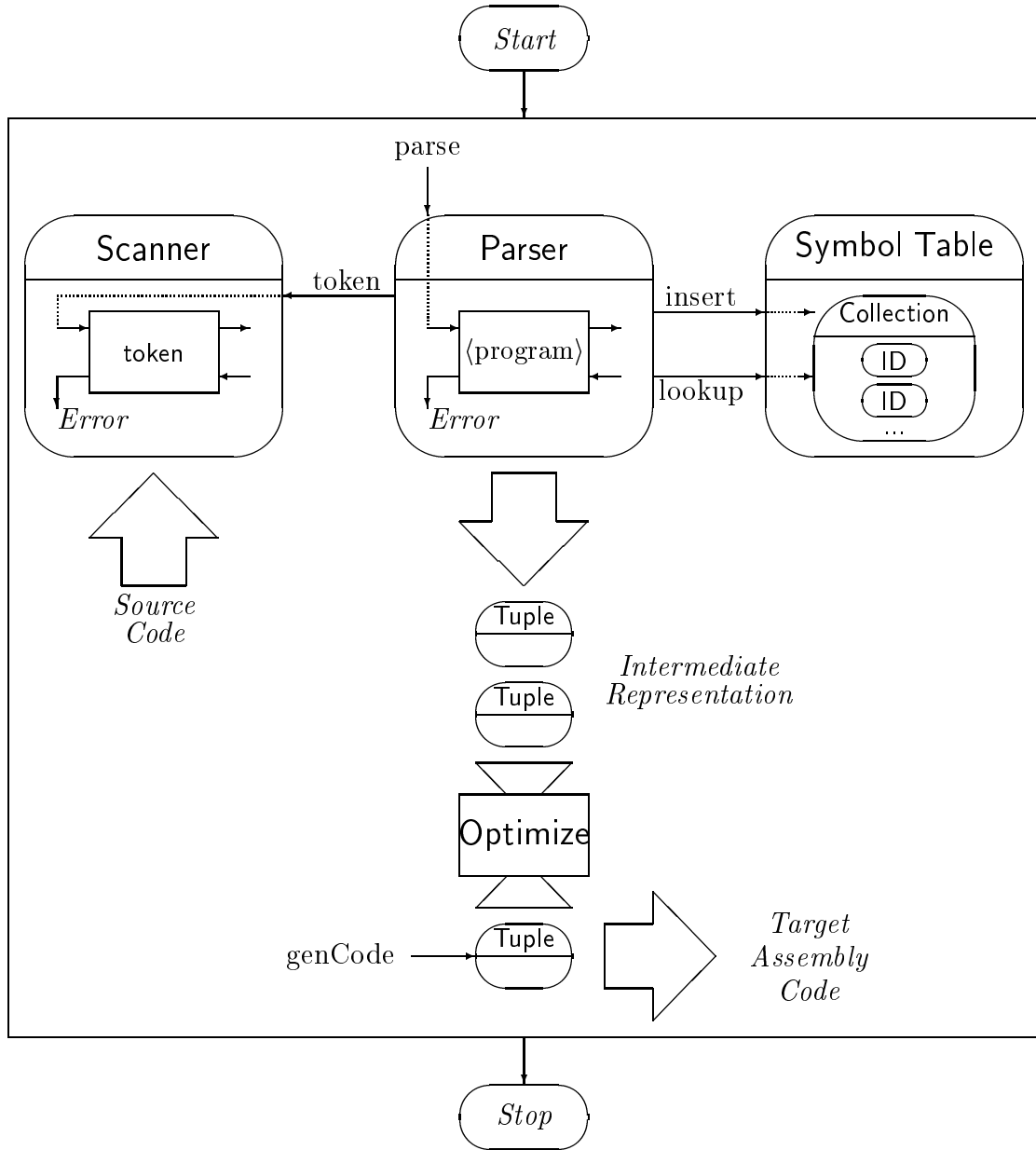


FIGURE 2.5. A Multiparadigm Compiler Model.

ing of the major programming paradigms while also illustrating the synergy provided by simultaneous access to several paradigms.

2.2.3. The Programming Languages Course

The shift in emphasis from the study of languages to the study of paradigms is most apparent in the programming language course. *Computing Curricula '91* [2] prescribes the study of programming paradigms (PL11) as part of the programming languages component. While examining the evolution of the programming languages course in the curriculum, King [101] observes the shift in emphasis from specific languages to the more general paradigms in programming language texts: “In recent years, a new breed of books has emerged—books that emphasize language paradigms instead of individual languages.”

Recent texts exhibiting this shift include Appleby [5] and Bal and Grune [10]. Comparative reviews of programming languages texts [4, 139] indicate that almost *all* texts suitable for use in the programming languages course include significant study of the four major paradigms (functional, imperative, logic, object-oriented).

Leda allows coverage of the imperative, functional, logic, and object-oriented paradigms within one linguistic framework. Two other paradigms that occasionally are included in programming languages textbooks are the database and the concurrent paradigm, which are usually covered in other classes (database and operating systems/parallel programming courses). The concurrent or parallel programming

paradigm is presently not a very “stable” one, with incredible conceptual diversity among constituent languages. The parallel language utilized is very dependent on the parallel hardware available, making it beneficial to examine this locale-specific paradigm in a separate course.

2.2.4. The CS1/CS2 Courses

Although changing the language used at the CS1/CS2 level is very difficult, the choice of language is perhaps most critical at this stage. In a recent paper, Stephen Jay Gould [72] mentions the “messy and personal side of science,” remarking that “ways of learning about the world are strongly influenced by the social preconceptions and biased modes of thinking that each scientist must apply to any problem.” While teachers will naturally bias students to some extent, programming languages play an extensive role as well. Alan Kay observed [98] this same “biased mode of thinking” caused by the first computer language students learn:

Our experience, and that of others who teach programming, is that a first computer language’s particular style and its main concepts not only have a strong influence on what a new programmer can accomplish but also leave an impression about programming and computers that can last for years. The process of learning to program a computer can impose such a particular point of view that alternative ways of perceiving and solving problems can become extremely frustrating for new programmers to learn.

If a multiparadigm programming language were taught first to students, their formation and curriculum might be more rounded. Later (junior or senior level) classes could utilize languages prevalent in industry. An advantage of this approach

is that the students' "better" foundation might reduce first-language bias as well as influence the evaluation and use of the industry languages.

2.3. Conclusions

Multiparadigm languages like Leda seem to have a future in both industry and academia. In industry, users have identified shortcomings in single paradigm languages that are motivated by the existence of other useful paradigms, leading to the hybrid languages. In academia, one of the main motivations of programming language selection is to provide students with firm foundations in the paradigms (and combinations of paradigms) that students may eventually be exposed to during their careers.

3. PROGRAMMING ENVIRONMENTS

An environment is the aggregate of surrounding things, conditions, and influences that affect the existence or development of someone or something. It is well understood that changes in the environment can have significant and potentially drastic effects. It is equally well understood that differences in environment can have major effects; individual development, for example, has been shown to have both genetic and environmental components [190].

In this chapter, we survey the area of programming and software engineering environments, providing a taxonomic and developmental context within which to place the LacEDAemon programming environment. The next section describes a taxonomy of environments. Short descriptions of exemplar environments in each of the four categories identified by the taxonomy are the subject of the next four sections. Next, we touch upon research focussed on incorporating Artificial Intelligence into environments. Lastly, we describe various approaches to constructing programming environments-delving into detail on the messaging approach utilized by LacEDAemon.

3.1. Classification of Environments

Dart et al. proposed a widely accepted taxonomy of software development environments based on the tool, user interface, and architectural trends of environments [53]. The four categories identified in this study were:

- *Language-centered environments* - a tightly integrated set of tools built around a single language. In many cases the tools and the environment together are identified as the language due to the tight integration.
- *Structure-oriented environments* - also called syntax-directed environments. Language structures are directly manipulated, and due to the language-independent nature of this approach, the notion of generators for these types of environments arose.
- *Toolkit environments* - collection of small tools, primarily intended to assist the user in the coding phase of the software development cycle. Emphasis is on extensibility of the environment, via a simple underlying data model.
- *Method-based environments* - also called process-oriented or process-based environments. These environments provide tools for a broad range of activities in the software development lifecycle. The tools define and enforce a particular development methodology.

Dart et al. note that this taxonomy was more for clarification of trends than for categorization of particular environments. Environments can in fact fit more

than one of the categories enumerated above. Other classification schemes have been proposed by ECMA/NIST [129], Perry and Kaiser [142], Pressman [148], and Sommerville [169]. We describe the ECMA/NIST model below, and use the Perry and Kaiser model in chapter 12 when evaluating LacEDAemon.

3.2. Language-Centered Environments

- *Interlisp*

Interlisp [186] is a programming environment based on the LISP [175] language. Interlisp provides a variety of user facilities, including syntax extension, error handling, error correction, an integrated structure-based editor, debugger, compiler, and file system.

- *Cedar*

Cedar [185, 13, 181] is the programming environment for the Cedar programming language. The language Cedar is a strongly-typed, compiler oriented Pascal-like extension of the Mesa language. The Cedar environment was designed to run on specialized hardware (the XEROX Dorado), and included the Tioga editor and document preparation system, Watch and Spy performance tools, electronic mail, resident debugger, compiler and interpreter.

- *Smalltalk*

The Smalltalk [69] programming environment consists of the object-oriented language Smalltalk and a collection of tools for interacting with language com-

ponents. Tools include a text editor, a tool to examine object instances (called inspector), project, protocol and class browsers, and a debugger.

3.3. Structure-Oriented Environments

- *ALOE*

The ALOE syntax-directed editor generator [123] produces general structure editors. The semantics of the programming language are described procedurally. ALOE is template-based in that it provides templates for all productions in the tree and does not automatically allow parsing.

- *CPS*

The Cornell Program Synthesizer [184, 158] is a syntax-directed programming environment that utilizes a structural perspective throughout program development. CPS includes a derivation-tree editor and syntax-directed interpreter that use predefined program fragments called *templates*. CPS utilizes either the PL/1 or Pascal programming languages. The Synthesizer Generator [159] is a tool for generating syntax-directed environments.

- *Emily*

Emily [78] is an early example of a structure editor. Program fragments are manipulated in a textual system, where the fragments are analagous to non-terminal symbols in Backus-Naur Form (BNF).

- *Gandalf*

Gandalf [76] is an incremental programming environment generator that has been tested on several algebraic languages. Gandalf includes the ALOE syntax directed editor generator (mentioned earlier), as well as the Incremental Programming Environment (IPE) [124]. Gandalf provides support for managing a project that involves the interaction of several programmers, and for the manipulation of system compositions and version control.

- *MENTOR*

MENTOR [57] is a programming environment for Pascal via the manipulation of structured data. This data is represented as operator-operand trees, better known as *abstract syntax trees*. MENTOR is driven by a tree manipulation language called MENTOL.

- *PECAN*

PECAN [152, 154] provides multiple views of the shared data structures. The internal program representation is abstract syntax trees. The user sees concrete representations of the abstract syntax tree through views such as the syntax-directed editor, Nassi-Shneiderman structured flowchart, and the module interconnection diagram.

3.4. Toolkit Environments

- *Apollo DSEE*

The Apollo Domain Software Engineering Environment [111] was a distributed, production quality software development environment for Apollo workstations. DSEE provided source code control, configuration management, release control, advice management, task management, and user-defined dependency tracking with automatic notification.

- *Arcadia*

Arcadia [183, 96] attempts to provide an extensible, incrementally improvable, flexible, fast and efficient software development environment as a collection of capabilities integrated to support developers and managers. Its architecture allows for interoperable components among multiple users and user classes. Components include tools for process definition and execution, object management, user interface development, measurement and evaluation, language processing, analysis and testing, and composition of components.

- *CAIS*

The Common APSE (Ada Program Support Environment) [127] Interface Set arose from the STONEMAN report [41] and the recognition of the need for a common interface for tools that would form the APSE. The implementation

is database centered, utilizing an Entity-Relationship model called the CAIS node model.

- *FIELD*

The Friendly Integrated Environment for Learning and Development (FIELD) [156, 150, 153, 151] pioneered the notion of broadcast messaging (loose integration) as a basis for tool integration. FIELD is built upon the Brown Workstation Environment (BWE) [157] toolkit. FIELD is the basis of many current commercial programming environments such as DEC FUSE, HP SoftBench and SUN SPARCWorks.

- *Multics*

The Multics [134] project was a cooperative effort between Project MAC at M.I.T., Bell Telephone Laboratories, and the General Electric Company. The goal of the Multics programming system environment was to allow users to build “complicated and sophisticated software subsystems.” To this end, Multics provided interprocedure communication, process and file system services, signals and exception handling, and interprocess communication. Multics also provided a dynamic linking mechanism as part of the operating system.

- *PCTE*

The Portable Common Tool Environment [16, 189] was developed by the ESPIRIT (European Strategic Program for Research and Development in Information Technology) research program in the European Community. PCTE

is the definition of a public tool interface for an open standard repository. It defines a set of operations that provide basic integration facilities (including object management, data, process execution and management) which can be used by tool and environment builder.

- *PWB/UNIX*

The Programmers Workbench UNIX [56, 99] system adds various tools to the UNIX operating system. These tools include a Remote Job Entry (RJE) subsystem, a Source Code Control System (SCCS), a configuration tool (make), tools for text processing and document preparation (PWB/MM), and various test drivers.

3.5. Method-Based Environments

Method-based environments are environments in which the processes used to produce and maintain software products are explicitly modeled in the environment. Method-based environments can be divided into two broad categories: Integrated Project Support Environments (IPSE), where all the tools relevant to a process are integrated and provided, and Computer-Aided Software Engineering (CASE) integration frameworks, where some tools as well as mechanisms for the integration

of other tools is provided¹. We give some examples of both types of environments below, as well as identifying some of the standards that are relevant to this area.

- *IPSEs*

Integrated Project Support Environments [122] arose from the difficulty of supporting large-scale software engineering projects with just the facilities provided by conventional operating systems and stand-alone tools. The purpose of an IPSE is to provide an infrastructure which stores all the information relevant to a particular project or projects, and which enables tools to be used in a coherent way. Examples of IPSEs include:

- *Aspect*

Aspect [18] is an IPSE centered around a relational database. Aspect is one of three IPSE projects sponsored by the Alvey Directorate in the U.K. (Eclipse and IPSE 2.5 are the others).

- *Eclipse*

Eclipse [3] was developed using PCTE, as the second generation IPSE in the Alvey program. Eclipse relies on a central database repository, and provides a number of tool sets to support different parts of the software development lifecycle.

¹The distinction between IPSEs and CASE environments made here is somewhat artificial, due to the lack of a prevailing definition of what constitutes an IPSE or a CASE.

– *IPSE 2.5*

The goals of IPSE 2.5 [168] were to raise the level of integration within IPSEs beyond the “store and tools” world through process modeling support, and to provide effective support for formal methods. IPSE 2.5 central unifying concept is a Process Modeling Language (PML) executed by a Process Control Engine (PCE).

– *Software Through Pictures*

Interactive Development Environment’s Software Through Pictures [192] is a collection of graphical editors and associated tools to support a variety of methods for software analysis and design. Methods supported include Structured System Analysis, Structured Design, Entity-Relationship modeling, and User Software Engineering (USE). The editors are supported by tools for tasks such as checking dataflow diagrams, RAPID/USE language generation, picture description generation, and storing graphical information in data dictionaries.

• *CASE Integration Frameworks*

Computer-Aided Software Engineering (CASE) environments [47] usually provide an integration mechanism that allows for the extension or customization of the environment via the addition (or modification) of tools. This feature also allows for the definition of the process that the environment will support. Some examples of CASE integration frameworks include:

– *DEC FUSE*

Digital Equipment Corporation’s FUSE system [80, 201] is a modified version of FIELD, which was licensed by DEC. FUSE provides a Motif-based interface, adds the concept of tool groups, and provides callbacks for handling message replies. Tools in FUSE include editors, a make utility builder, code manager, cross-referencer, debugger, call-graph and class browsers, and a profiling tool—all connected via its messaging server, EnCASE.

– *HP SoftBench*

Hewlett Packard’s SoftBench [8, 51, 44, 66, 45, 14, 114, 196, 197] is a CASE integration framework built around the Broadcast Message Server (BMS). SoftBench includes the Encapsulator task-and-process automation tool [63], a debugger, program builder, editor, static analyzer, and development manager.

– *Sun SPARCworks*

Sun SPARCWorks [62] is an integration framework based on Tooltalk [43]. The SPARCworks toolset consists of a debugger, source browser, FileMerge, make and an analyzer—all integrated via the SPARCworks Manager, a session manager.

– *SLCSE*

The Software Life Cycle Support Environment (SLCSE) [178] is a proto-

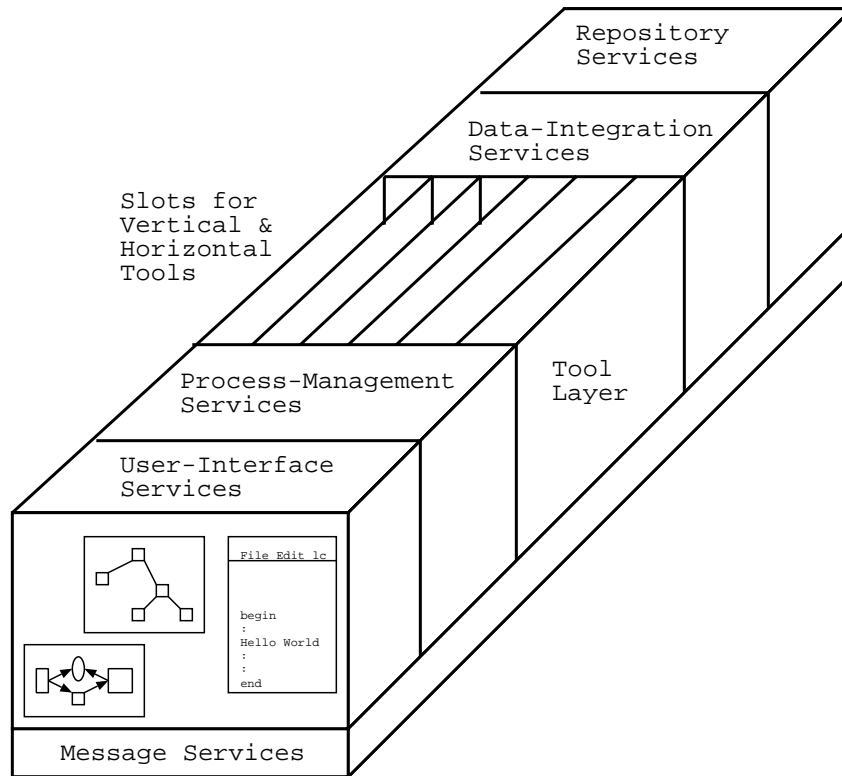


FIGURE 3.1. The ECMA/NIST "Toaster" Model.

type software engineering environment that provides an integration framework and an extensible lifecycle toolset. SLCSE was funded by Rome Air Development Center, and developed by General Research Corporation. SLCSE provides complete support for the DoD-STD-2167 life cycle process. ProSLCSE, a productized, open standards-based version (SLCSE was tightly integrated into VAX/VMS) of SLCSE, is currently under development.

- *Standards and Reference Models*

- *ANSI X3.138* - The ANSI committee on Data Representation has produced two standards, X3.138-1988 and X3.138A-1991, that specify a computer software system that provides facilities for recording, storing, and processing descriptions of an organization's data and data processing resources.
- *IEEE P1175* - is a draft standard for tool interconnection, within which there are four information sharing methods: direct (tool-to-tool), file-based, communication-based, and repository-based. This standard was created by the Task Force on Professional Computing Tools.
- *EIA CDIF* - Electronic Industries Association CASE Data Interchange Format provides a published set of vendor-independent and method-independent definitions for meta-data concepts in general, and for modelling data and related concepts in particular.
- ISO/IEC JTC1/SC7/WG11 - Software Engineering Data Definition and Representation (ISO/CDIF). This group is promoting a CDIF standard tailored to Software Engineering for ISO standardization.
- *NIST/ECMA* - the National Institute of Standards and Technology in conjunction with the European Computer Manufacturers Association have proposed a reference model for CASE integration [129], the structure of which is depicted in Figure 3.1. This model is a catalog of services available in environment frameworks, not a description of architecture

or a standard. Vertical tools are used in a specific phase of software development², while horizontal tools are used throughout.

- *OMG OMA/CORBA* - The Object Management Group (OMG) is a consortium that creates architectural standards for allowing interoperability and portability of distributed object-oriented applications. The Object Management Architecture (OMA) is a high level vision of a complete distributed environment, consisting of two major parts: system oriented components (Object Request Brokers and Object Services) and application oriented components (Application Objects and Common Facilities). The Common Object Request Broker Architecture (CORBA) specifies a system which provides interoperability between objects in a heterogeneous, distributed setting.
- *Microsoft ActiveX, COM, OLE* - Component Object Model (COM) and its variants are examples of Microsoft architecture standards for creating applications from binary software components, forming the basis of higher level services, such as Object Linking and Embedding (OLE). ActiveX is a language-neutral interface to Microsoft component technology.

²Vertical tools are sometimes partitioned further into “upper CASE” tools which support analysis and design activities, and “lower CASE” tools which support coding and testing.

3.6. AI Programming Environments

There has been some research devoted to supporting the software development process with artificial intelligence techniques [12, 82]. Some development environments have been created that add AI tools in an attempt to further expedite the software development process. ASPIS [149], The Programmer's Apprentice, and KBEmacs [160] are examples of such environments.

ASPIS includes four knowledge-based tools called assistants: Analysis, Design, Prototype and Reuse Assistants. The Analysis and Design assistants embody knowledge about both the method and application domain, while the Prototype assistant verifies system properties, and the Reuse assistant helps developers reuse specifications and designs.

The Programmer's Apprentice is a long-term ongoing project. The goal of this project is to provide a junior partner and critic for the software engineer, taking over the simple tasks completely, and assisting with the more complex tasks. Two principles have been identified (intelligent assistance and inspection methods), a knowledge representation format developed (the Plan Calculus), and a program implementation demonstration system created (Knowledge-Based Editor in Emacs, or KBEmacs).

3.7. Constructing Programming Environments

There are three basic strategies that have been employed to create programming environments:

- build as a single, monolithic system

Examples include Emily [78], Interlisp [186], Gandalf [76], Cedar [185], Poe (Editor Allen Poe) [60], PECAN [152], CPS [184], Magpie [55], and Mentor [57].

- build as a collection of independent tools

Examples include Apollo DSEE [111], Arcadia [183], CAIS (Common APSE Interface Set) [127], Multics [134], UNIX [99] and UNIX/PWB [56].

- build as a set of related tools with a communication mechanism

Examples include FIELD [151], HP SoftBench [44], DEC FUSE [80], and Sun SPARCworks [62].

3.8. Communication-Based Environments

Controlling and coordinating tool interactions in an environment requires an approach to tool integration that is flexible, adaptable, simple and efficient. One must be able to integrate new tools, but not at the cost of impairing existing tools. Integration ranges from *loose*, where the tools being integrated have little or no knowledge of one another, to *tight*, where tools possess much knowledge of the other

tools. The traditional approach in tool integration has been based on data sharing, where all the tools deposit their data in a common database. This approach, called data integration, requires a common data schema, and imposes high overhead and limitations on the extent on tool integration.

A more recent approach has been communication based, or control integration. In this approach, a programming environment is viewed as a collection of services provided by different tools. The tools send and receive control signals, and decide whether these signals require any action on their part. Variations on the control integration approach include point-to-point message passing, where tools make explicit requests and responses via direct messages from tool to tool, and broadcast messaging/message passing/messaging server approaches, where messages are transmitted to all tools (or a select subset decided upon by the message server), and the tools decide upon a response, if any. Three examples of environments that rely on control integration as the integrating mechanism of the environment are FIELD [151], SoftBench [44] and Tooltalk [43].

FIELD is the origin of most of the work in the realm of message passing. The FIELD message server, *Msg*, passes messages as text strings of arbitrary length. Thus there is no predefined protocol for messages, ensuring flexibility. This also allows for tools to form their own collaborations, and define their own message formats for closer cooperation. *Msg* can also be amended to allow for the addition of new messaging capabilities. FIELD defines two categories of messages:

- Commands - message sent to a particular tool or class of tool requesting some service be performed. The message includes the name of recipient, command name, system name, and arguments.
- Information Messages - message broadcast to all interested tools informing them of some event. The message includes the name of the sender, the event causing message, system name, and arguments.

Messages can be synchronous (tool sends a message and waits for a response) or asynchronous (tools sends a message and continues). As expected, command messages are synchronous, while information messages are asynchronous. The system name allows the message server to distinguish between different invocations of the same tool.

SoftBench was the first commercial message passing based-environment. The SoftBench message server, Broadcast Message Server (BMS), is similar in operation to FIELD's Msg. BMS does provide a more formal and structured interface to messages than does FIELD, and also offers some additional facilities over Msg. SoftBench standardizes messages through the notion of tool protocols and a standard message format. A tool protocol defines a standard set of operation and information messages for each grouping of tools. This allows for the substitution of new or different tools directly for existing tools provided they maintain the protocol for their tool group. The BMS standard message format consists of seven fields:

- Originator - tool that sent the message

- Request-Id - unique identifier for the message
- Message Type - indicate whether this message is a request (command) message, a success notification, or a failure notification
- Command Class - tool grouping protocol name
- Context - location of the data to be processed
- Command Name - name of command (request message) or type of event (notification message)
- Arguments - arguments accompanying the command or parameters of the event

Tooltalk, a message system from Sun Microsystems, is similar to FIELD and SoftBench in many ways. Messages in Tooltalk consist of fourteen fields:

- Arguments - arguments to a message or the reply value
- Class - indicates whether the message is a notice or a request
- File - the file for the message
- Operation - the name of the operation to be performed
- Object, Otype - the object and object type involved in the operation
- Scope, Address, Handler, Handler_Ptype - these fields identify to whom the message should be sent

- Disposition - what to do if the message can't be handled by any running process
- Sender_Ptype, Session - these identify the sender of the message
- Status - information about the state of the message

Tooltalk allows the sender much finer control over the recipient of a message than FIELD or SoftBench. Tooltalk embraces object-oriented notions, viewing the world as processes responsible for a given object or object type and messages being the methods or operations for the object. Note that the status of a message is maintained as part of the message. Tooltalk also allows for a way to handle messages that have no handlers. A process can be started to handle a message, or the message can be queued for later delivery to the appropriate process. Tooltalk notice messages are sent asynchronously, while request messages are sent synchronously, with the caller blocking until a reply is received.

Alan Brown of the Software Engineering Institute [17], while evaluating the messaging-based integrated toolkit approach of FIELD, SoftBench and Tooltalk declared:

While a number of open issues and shortcomings of the approach have been identified in this paper, there is clearly evidence to support further investigation of the message passing approach as the basis for providing an SDE architecture that is more open to the addition of new tools. In particular, the simplicity and flexibility that the approach provides appear to facilitate experimentation with different levels of integration between tools. As a result, it may well provide the ideal platform for experimenting in some of the most crucial aspects of SDE technology...

LacEDAemon provides an architecture that is more open to the addition of new tools, and allows for experimentation with different levels of tool integration.

4. AN OVERVIEW OF LACEDAEMON

4.1. Introduction

A main goal of LacEDAemon is to provide a tool integration framework. In order to describe the tool integration framework capabilities afforded by LacEDAemon, we must first examine integration and tool integration in general. We then examine the tasks and tools we wish to integrate in LacEDAemon. Next we describe the hypertool architecture¹ of LacEDAemon, followed by the message definitions that occur in the framework. Last, we describe how to extend LacEDAemon.

4.2. Integration

Tool integration is about the extent to which tools agree. The subject of these agreements may include data format, user-interface conventions, use of common functions, or other aspects of tool construction. While there is much research activity in this area, few quantifiable integration metrics exist [187].

¹Weiser [193] observed the term “landscaping” might be a better term than “architecture” in the context of environment creation. He felt that landscaping gave a better feel for the unknown realm of foundations for environments, and also better conveyed how an environment creator had to make do with the existing lay of the land (i.e. underlying system and software).

Generally speaking, there are five forms of integration that are relevant in a discussion regarding a tool integration framework:

- *data integration* - the sharing of information between tools in the environment i.e. the use of data by tools.
- *presentation integration* - the provision of a common user interface for the tools in the environment i.e. the details of the user interaction.
- *control integration* - the management of cooperation, interoperation, and communication between independent tools in the environment.
- *platform integration* - the platform or platforms on which the framework services are provided.
- *process integration* - the role of tools in the software process.

We will concern ourselves primarily with control, data, and presentation integration here. While platform integration is an interesting topic, our focus is on the relationships among tools, and we regard the platform as simply providing the basic elements on which the agreement policies and usage conventions for tools are built. As for process integration, a *programming* environment concerns itself with the coding aspects of any subsuming process. The field of multiparadigm programming has not yet originated and validated any process² that requires inclusion at the

²Some progress has been made recently via the study of multiparadigm design patterns by Knutson [106].

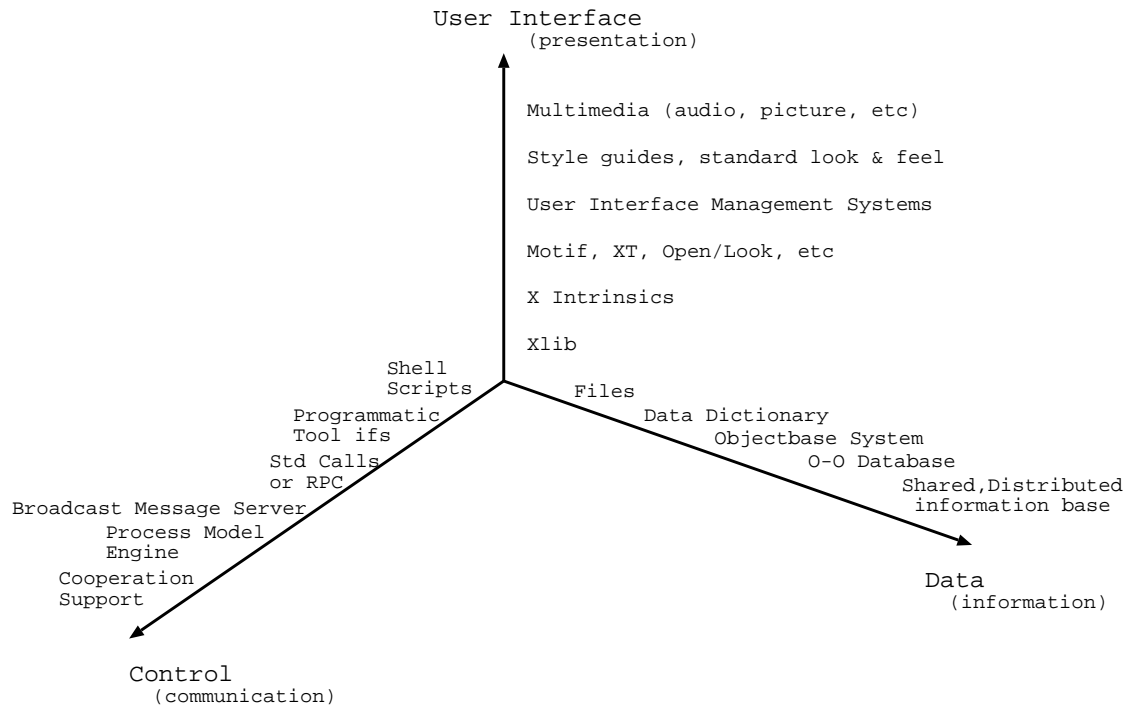


FIGURE 4.1. The Three Dimensions of Integration.

coding level (beyond the “edit-compile-debug” process that is implicitly supported). Process integration is of great interest in CASE and IPSE environments, and of little interest to a programming environment that is independent of any encompassing CASE or IPSE. The three areas of integration we will concentrate on are depicted in Figure 4.1.

4.3. Tool Integration Frameworks

Event-based programming is a common solution approach in computing. User interfaces such as the X-Window system [164] wait for and react to events such as a “key press” or “mouse button down”. Various artificial intelligence systems

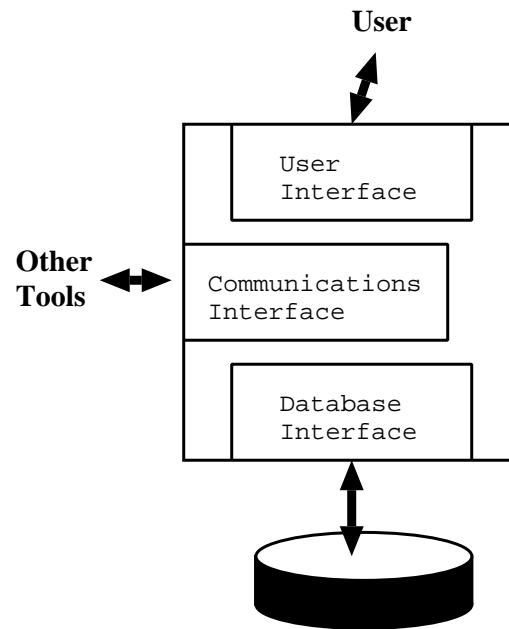


FIGURE 4.2. Idealized Integrated Tool.

(actors, blackboard systems, some rule-based systems) use an event-based approach as well. Some operating systems (such as the Commodore Amiga) provide event announcement primitives for application integration. Lastly, software integration systems consist of multiple software modules reacting to events announced by other modules. Tool integration frameworks utilizing event-based integration (also called implicit invocation or control integration) have become a popular approach to constructing integrated programming environments.

A tool integration framework consists of:

- a communication mechanism
- a facility for data storage and control

- a vehicle for constructing consistent user interfaces

An integrated programming environment consists of tools embedded in a tool integration framework (Figure 4.2). The toolset provided, along with the level of cooperation between tools, determines the exact nature of the environment. Flexibility in an environment is achieved via variations in toolset as well as variations in the interactions between the tools.

Tool integration is largely determined by the approaches taken to five issues:

1. method of event communication
2. expressiveness of the tool interaction descriptions
3. intrusiveness of the tool interaction descriptions
4. static vs. dynamic behavior definition
5. tool naming and awareness

Tool integration frameworks are either data-centered or tool-centered, depending on how the issues of tool control and data management are dealt with. Data-centered frameworks embed both data management and tool communication in a single, central repository. An example of data integration is the ECMA PCTE shared repository approach [189]. Tool-centered environments view data management and tool communication as separate issues.

Environments can also be considered tool-driven or user-driven. Tool-driven environments (such as FIELD) allow tools to react somewhat autonomously, mak-

ing assumptions about the support required by the user, and working in concert with other tools. User-driven environments (such as SoftBench), expect the user to drive inter-tool communication explicitly via user interface interactions. A similar environment-related distinction is in the form of whether tools in an integrated environment are loosely integrated or tightly integrated. Loosely integrated tools are not very aware of the presence of other tools, which tightly integrated tools are very involved in terms of interacting among other tools.

4.4. Tasks and Tools

The three primary programming tasks that the LacEDAemon programming environment will initially facilitate are:

- creation
- correction
- comprehension

These three tasks are intertwined, and the entire set of LacEDAemon tools will aid in all three tasks. Ideally, the environment should possess a task focus, not a tool focus, i.e., several tools provide some functionality that is part of a larger task. Creation of programs occurs within a text editor. The user might avail of preexisting Leda programs during this process (requiring comprehension of the existing code). The correction of programs requires editing as well, along with the

need to understand the workings of the programs. Along with a text editor, the initial toolset includes a program database, debugging “hooks”, program execution visualization, and class and program file browsers. The integral nature of the tasks must be reflected in the tools. The observation regarding tools that “amalgamation is not equivalent to integration” is certainly true when considering the tasks of creation, comprehension and correction.

4.5. The Hypertool Architecture Of LacEDAemon

A tool in LacEDAemon is a Tcl/Tk *hypertool*: a small, reuseable, stand-alone (usually existing) application that is pressed into service perhaps unforeseen by the tool originator. This use of the tool is made possible through the embedding of a Tcl interpreter within the tool (or, embedding of the tool within a Tcl interpreter). The result is that the tool gains the additional capability of executing Tcl commands, and from this ability a hypertool is born. As there are Tcl commands to provide inter-tool communication, hypertools provide a low-cost, versatile integration framework.

LacEDAemon provides framework wrappers for applications written in C, C++, Java, and Tcl/Tk. These wrappers embed the tool within a Tcl/Tk interpreter, and provide messaging stubs for all of the LacEDAemon messages. The tool integrator can now decide which event messages the tool is interested in responding to, which requests the tool will honor, and what will comprise the response messages.

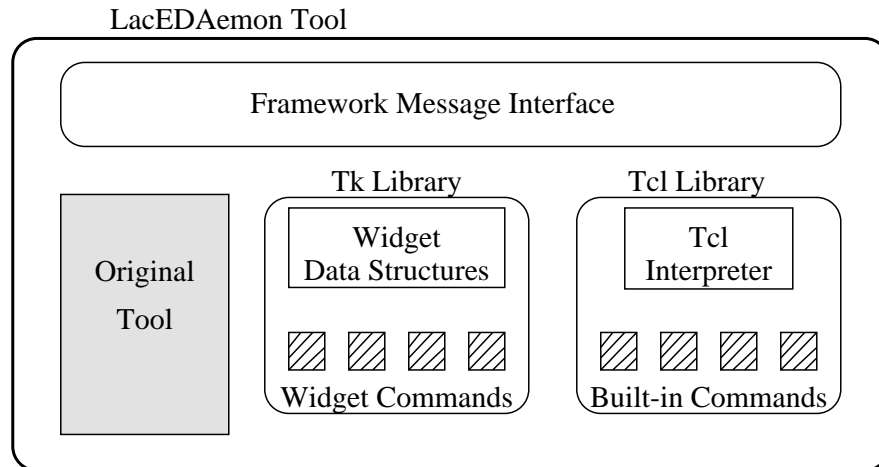


FIGURE 4.3. A Generalized LacEDAemon Hypertool.

These parameters are also registered with the LacEDAemon environment database in the participant information repository.

The LacEDAemon integration framework differs from prior approaches to integration (such as FIELD and SoftBench) in providing a much more flexible messaging capability, at lower cost. A LacEDAemon message can cause the execution of a Tcl command in any hypertool. Tcl commands can be standard commands as defined by the language Tcl, or ones created by the user and embedded in the hypertools. LacEDAemon messages can be synchronous or asynchronous, point-to-point or broadcast. Providing the full power of an embedded programming language for messaging provides more flexibility than a rigidly defined set of messages, and by having messages correspond to language commands and tool application programming interfaces, greatly reduces the programmer burden in creation of the environment.

4.5.1. Messaging In Tcl/Tk

Message passing in Tcl/Tk can be illustrated by two very simple Tcl scripts. The script `lc.tcl`, illustrated in Figure 4.4, provides two simple services via the two procedures that are defined in the script. `LedaVer` determines the version of Leda that is available to the system, and returns the result of determining that information from the Leda interpreter to the requestor. `ExecLeda` takes as an argument a file name, and invokes the Leda interpreter on that file.

Figure 4.5 provides two examples of messages requesting services from the `lc.tcl` script. Note the application name of the services defined by `lc.tcl` is `Leda_Interp`. The script `send_msg.tcl` requests the Leda interpreter version via the first `send` command. The script (with an application name of `Send_Msg`) blocks and awaits the results of the request, as the message is sent synchronously. The second request is to invoke the Leda interpreter on the file `chap3.led`. This request is asynchronous, and the `Send_Msg` application continues executing without waiting for the results of its request.

The messages as defined here must occur on the same X window display, however Tcl/Tk has the ability to work in a distributed manner over a TCP/IP network. This allows for the possibility of maintaining a single, network-based environment repository (we will examine this issue more in the next chapter).

```
#!/opt/hppd/bin/X11/wish

proc ExecLeda {file_to_exec} {
    set output [exec lc $file_to_exec]
    puts $output
}

proc LedaVer {requestor} {
    set leda_version "[exec lc -v]"
    send $requestor EchoInfo $leda_version
}

tk appname Leda_Interp
```

FIGURE 4.4. The lc.tcl Script.

```
#!/opt/hppd/bin/X11/wish

tk appname Send_Msg

proc EchoInfo {args} {
    puts $args
}

puts [winfo interps]
send Leda_Interp {LedaVer Send_Msg}
send -async Leda_Interp {ExecLeda chap3.led}
exit
```

FIGURE 4.5. The send_msg.tcl Script.

4.5.2. Embedding Interpreters in Tools

Figure 4.6 illustrates adding a new command to a Tcl interpreter via the C programming language. In this case, the command is the trivial equivalent of “hello world”. The Tcl command `hi` will now result in the output of the string `Hello, World!`. As a result of having the Tcl interpreter embedded within a tool, one can utilize Tcl’s strengths in messaging and Tk’s facilities for interface construction to expedite the additional requirements placed on a tool within an integrated toolkit environment.

Embedding Tcl interpreters within various tools yields other advantages beyond providing a messaging mechanism and the basis for a common user interface. Here we describe one such example extension that is easily afforded by the presence of the embedded Tcl/Tk interpreter.

The Leda language as defined has no graphics capabilities. Adding graphics to Leda is not trivial, and most such graphical extensions (X window, Motif, Windows MFC, etc.) greatly increase the complexity of the language. The access to Tk which is provided by embedding the Leda language within a Tcl/Tk interpreter allow for access to graphics to the Leda language in a very simple manner, in keeping with the language design goals.

Figure 4.7 shows a simple spinbox application implemented in Leda utilizing Tk graphics features. Figure 4.8 illustrates the Leda code that implements this


```
#include <stdio.h>
#include <stdlib.h>
#include <tcl.h>
#include <tk.h>

int HelloWorld (ClientData client_data, Tcl_Interp* interp,
               int argc, char *argv[])
{
    Tcl_ResetResult(interp); /* reset result data */
    Tcl_AppendResult (interp, "Hello, World!", (char*) NULL);
    return TCL_OK;
}

int Tcl_AppInit (Tcl_Interp* interp)
{
    int status;

    status = Tcl_Init(interp);
    if (status != TCL_OK) return TCL_ERROR;

    status = Tk_Init(interp);
    if (status != TCL_OK) return TCL_ERROR;

    Tcl_CreateCommand (interp,
                      "hi",      /* command name */
                      HelloWorld, /* C function */
                      NULL,      /* client data, unused */
                      NULL);     /* delete function, unused */

    return TCL_OK;
}

int main (int argc, char *argv[]) {

    Tk_Main(argc, argv, Tcl_AppInit);
    return (0);
}
```

FIGURE 4.6. Embedding a Tcl Interpreter.

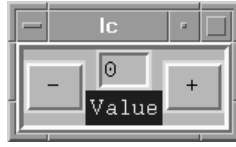


FIGURE 4.7. The Spinbox Application.

application. Figure 4.9 illustrates a portion of the `graphics.lcd` definition file showing how Leda can now “escape” to Tk.

In some sense, this is similar to the style of multiparadigm programming suggested by Koschmann and Evens [108] in describing their “boolean bridge” approach to integrating object-oriented and logic programming. Their premise was that the best solution for combining alternative programming styles was to allow each style-supporting subsystem to coexist in an application, with some interface between the various styles. Here we have created a “graphical bridge” that allows us to use Leda for what it does best, while leaving the graphical programming component to Tk. This technique to add graphics to textual programming languages has been exploited in the realm of functional languages such as Gofer [48] and ML [118], as well as the scripting language Perl [91]. Note that this subsystem aggregation technique gives the programmer direct access to the tools of LacEDAemon as well via the messaging facility. While we do not exploit the direct tool access capability now present in Leda in integrating tools into LacEDAemon, its presence allows for reflective tool integration experiments (changing the definitions of tool interactions from within the language).

```

include "std.led";
include "graphics.led";

var
  frm : frame;
  but1, but2 : button;
  ent : entry;
  lbl : label;

begin
  frm := frame();
  frm.set_name(".frm");
  frm.create ("ridge", "5");
  frm.pack();
  but1 := button();
  but1.set_name(".frm.but1");
  but1.create ("+");
  but1.side_pack("right");
  but1.configure("-command {set count [expr $count + 1]; update}");
  cfunction TkCommand ("set count 0");
  but2 := button();
  but2.set_name(".frm.but2");
  but2.create ("-");
  but2.side_pack("left");
  but2.configure("-command {set count [expr $count - 1]; update}");
  ent := entry();
  ent.set_name(".frm.ent");
  ent.create ("3", "sunken", "count");
  ent.pack();
  lbl := label();
  lbl.set_name(".frm.lbl");
  lbl.create ("Value");
  lbl.set_background("blue");
  lbl.set_foreground("yellow");
  lbl.side_pack("bottom");
end;

```

FIGURE 4.8. Leda Code for the Spinbox.

```
class widget;
var
  name : string;

  function set_name(widget_name : string);
  begin
    name := widget_name;
  end;

  function print_name();
  begin
    print (name);
  end;

  function create(text : string);
  begin end;

  function pack();
  begin
    cfunction TkCommand ("pack " + name);
  end;

  function configure(args : string);
  begin
    cfunction TkCommand (name + " configure " + args);
  end;
  :

end;

class label of widget;

  function create(text : string);
  begin
    cfunction TkCommand ("label " + name + " -text " + text);
  end;

end;
:
```

FIGURE 4.9. Leda Code for Graphics Library.

4.5.3. Combining Message and APIs

One subtle but interesting difference between the LacEDAemon Tcl/Tk-based messaging scheme and other integration frameworks is the merging of the Application Programming Interface (API) of tools and the messaging framework. The tool API is the set of functions and procedures that applications built upon a tool framework can call, i.e. the set of exported or public interfaces between a tool's programming library and the world. As a result of embedding the messaging facility within the tool itself, LacEDAemon gains access to the *entire* tool interface.

4.6. LacEDAemon Messages

Messages in LacEDAemon are in general Tcl commands. As part of the LacEDAemon messaging framework, a set of standard commands are provided for which the appropriate tools can provide responses. These commands fall into the following three categories:

- requests - requests from one tool to another for an action
- response - response to a request from a tool
- events - notification of some event that might require action by a tool

A request is an explicit request from one tool to another, i.e. a file browser asking an editor to display a file that the user has selected. A response provides information from the responding tool to the requestor i.e. the fact the editor was able

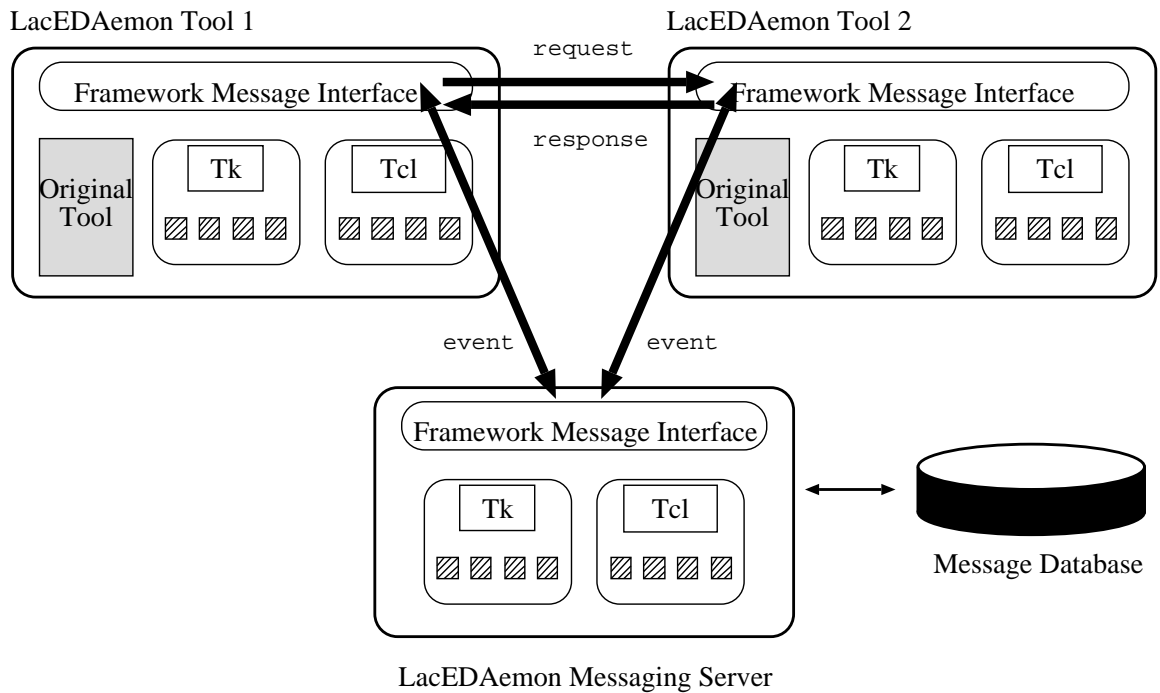


FIGURE 4.10. Messaging in the LacEDAemon Environment.

to display the file successfully. An event is simply notification to the LacEDAemon toolset of a happening that may be of interest to other tools, i.e. the Leda interpreter notifying tools that a specific line of a program has been executed, leading to the execution animator updating its display. Messages in the LacEDAemon environment are illustrated in Figure 4.10.

A message has the general form of:

```
send <originator> <destination> <message> <message arguments>
```

Note that TCP/IP messaging is also available and would allow for a programming environment distributed over multiple machines in a networked environment.

4.6.1. Request Messages

Some of the request message definitions present in the LacEDAemon message framework are listed in Table 4.1. Requests are messages from tools that require a specific action from one or more tools present in the LacEDAemon framework. Requests result in response messages. Brief descriptions of the actions that accompany a tool issuing a request are as follows:

- *register* - any tool can report its presence (and interest in various events) with the LacEDAemon Messaging Server with a *register* request.
- *open* - any tool with the capability of accessing a Leda program can request the user-selected item be displayed among the LacEDAemon tools.
- *close* - any tool with a user-selectable action for closing a Leda program presently being displayed by the tool can request that other members of the LacEDAemon framework do the same.
- *update* - any tool that allows some state of LacEDAemon to be altered can request other tools also update their state in response to the change.
- *parse* - the file browser can request that the Leda interpreter (or possibly a Leda compiler) parse the file provided as an argument.

message	originator	destination	purpose
register	any tool	Message Server	register presence of tool
open	any tool	file browser	open file for display
close	any tool	file browser	close file being displayed
update	any tool	file browser	update file being displayed
parse	file browser	Leda interpreter	parse a Leda file
compile	file browser	Leda interpreter	compile a Leda file
class_struct	class browser	LacEDAemon database	describe class structure
file_struct	file browser	LacEDAemon database	describe file structure
exit	any tool	Message Server	tool is exiting

TABLE 4.1. LacEDAemon Request Messages

- *compile* - the file browser can request that a Leda compiler³ compile the file provided as an argument.
- *class_struct* - the class structure browser of LacEDAemon needs current class information in order to display, which is provided by the program database.
- *file_struct* - the file structure browser of LacEDAemon needs current file structure information in order to display, which is provided by the program database.
- *exit* - the fact that a tool has been instructed to exit needs to be registered with the Messaging Server. This request may also cause other tools (or the whole environment) to exit.

³While the current Leda implementation is an interpreter, two prior implementations were compilers.

4.6.2. Response Messages

Some of the response message definitions present in the LacEDAemon message framework are listed in Table 4.2. Responses provide feedback to request messages issued by tools that expect a specific action from one or more tools present in the LacEDAemon framework. Requests result in response messages. Brief descriptions of the implications of the possible responses are as follows:

- *success* - the request message from a tool resulted in the desired action being successfully performed.
- *failure* - the requested action by the tool resulted in the desired action not being performed successfully.
- *error* - the requested action by the tool resulted in an error (as defined by the responding tool).
- *unavailable* - the service requested by the tool is not a service that the responding tool can provide.
- *data* - the requested action resulted in some data that is now being provided.

4.6.3. Notification Messages

Some of the notification message definitions present in the LacEDAemon message framework are listed in Table 4.3. Notifications provide status updates to

message	originator	destination	purpose
success	any tool	any tool	prior request successful
failure	any tool	any tool	prior request failed
error	any tool	any tool	prior request caused an error
unavailable	any tool	any tool	requested service unavailable
data	any tool	any tool	data generated by requested service

TABLE 4.2. LacEDAemon Response Messages

all tools that have registered an interest in being kept abreast of current status.

Brief descriptions of some of the notification messages are as follows:

- *LD_Init* - the user is requesting the initialization (or reinitialization) of a LacEDAemon session from a tool that allows this option.
- *LD_Exit* - the user is requesting the exit of a LacEDAemon session.
- *FileOpen* - a file is being opened by some tool.
- *FileClosed* - a file is being closed by some tool.
- *LineAdv* - the Leda interpreter/compiler has advanced a line in the current program being executed.
- *FuncCall* - the Leda interpreter/compiler is calling a function in the current program being executed.
- *OpCall* - the Leda interpreter/compiler is executing an operator in the current program being executed.

message	originator	destination	purpose
LD_init	messaging server	any tool	user initializing LacEDAemon
LD_exit	messaging server	any tool	user exiting LacEDAemon
FileOpen	file browser	any tool	a Leda file has been opened
FileClosed	file browser	any tool	a Leda file has been closed
LineAdv	Leda interpreter	any tool	interpreter has advanced a line
FuncCall	Leda interpreter	any tool	interpreter has executed a function
OpCall	Leda interpreter	any tool	interpreter has executed an operator
Breakpoint	Leda interpreter	any tool	interpreter has arrived at a breakpoint
Output	Leda interpreter	any tool	interpreter has emitted some output
success	any tool	any tool	a request was successful
failure	any tool	any tool	a request failed
error	any tool	any tool	a request caused an error
unavailable	any tool	any tool	requested service unavailable

TABLE 4.3. LacEDAemon Notification Messages

- *Breakpoint* - the Leda interpreter/compiler has arrived at a user-set breakpoint.
- *Output* - the Leda interpreter/compiler has arrived at a statement requiring output.
- *success* - a request message was successfully serviced.
- *failure* - a request message was unsuccessfully serviced.
- *error* - servicing a request message resulted in an error.
- *unavailable* - a request message could not be serviced.

4.7. Extending LacEDAemon

In order to facilitate the integration of tools beyond the default LacEDAemon toolset, a selective broadcast mechanism is also available. This allows tools to register their interest in events with the broadcast server, and for the broadcast server to appropriately propagate events generated by the tools as well as requests. Note that for full integration, the existing tools must be made aware of the addition of new tools for point-to-point transactions. Also note that all tools register their presence and the set of requests that they service.

Integrating a tool into LacEDAemon is not an extremely difficult task. However, just like any engineering task, the coding required for the integration of a tool should be preceded by analysis, specification and design. What functionality of the tool will the user wish to access? What other tool interactions will the tool be required to participate in? What new messages need to be originated? What is the impact of the presence of this tool on existing tools and tool interactions? All of these questions need to be examined prior to integrating a tool.

Provided the tool's source code is available, and the tool is written in C, C++, Java, or Tcl/Tk (or some other system scripting language), there are messaging frameworks available in LacEDAemon to provide basic messaging for the tool. If the tool does not have source code available, limited integration might still be possible via various Tcl/Tk applications [79] such as TkSteal. If the application

for which source code is not available is textual in nature (i.e. no graphical user interface), limited integration via Expect [113] is still possible.

5. THE LACEDAEMON PROGRAM DATABASE

In this chapter, we describe the PostgreSQL database system that comprises the heart of the LacEDAemon Program Database, as well as the LacEDAemon tool integration repository. We describe how tool interactions and Leda programs are represented in the database. The program information extractor, a tool for populating the program database is briefly touched upon. Lastly, other implications of the presence of a central database are examined: the possibility of creating a configuration management system and augmenting the Leda language to include persistent objects are explored.

5.1. Introduction: PostgreSQL

Postgres, Postgres95, and PostgreSQL are all releases of the Postgres project, an object-oriented database management system (OODBMS) research group at the University of California at Berkeley, led by Michael Stonebreaker [177].

Postgres extends the traditional database management system (DBMS), which comprises of a data model consisting of a collection of named relations, containing attributes of a specific type. Postgres extends the traditional DBMS model by adding classes, inheritance, types, functions, and a powerful production rule system. The query language in Postgres is called PostQuel.

PostgreSQL is an Object-Relational DBMS (ORDBMS), derived from the Berkeley Postgres database management system. While PostgreSQL retains the powerful object-relational data model, rich data types and easy extensibility of Postgres, it replaces the non-standard PostQuel query language with an extended subset of the standard SQL query language.

A study conducted by Dahanayke and Florijn [52] identified Postgres as one of the best object-oriented database management systems available with which to implement a repository for software engineering environments. Several databases, including Ontos, Obj/db, Itasca, Iris and Postgres were evaluated in the context of a generic model consisting of orthogonal services. The databases were rated based on their functionality in the areas of modeling, storage and manipulation, version, integrity and consistency, views, concurrency control, security, distribution, interface, and control integration. Postgres was the most highly rated public-domain database in their study.

5.2. Using Postgres as the LacEDAemon repository

Postgres is used as the environment database in LacEDAemon for two purposes:

- to store and supply information about LacEDAemon participant tools.
- to store and supply information about individual Leda programs.

LacEDAemon tools can register with the environment database—their presence, messages they are interested in, and messages to which the tool will respond. The LacEDAemon broadcast message server relies on the participant tool information to resolve the sending and receiving of all communication that is not point-to-point.

The environment database also stores a representation of the various Leda programs. Various tools rely on program structure information, and rather than gather that information multiple times, the Leda program information extraction tool stores that information in the environment database when a program is accessed for consumption by the tools.

5.2.1. Representing LacEDAemon Participant Information

The LacEDAemon participant relation is depicted in Table 5.1. The fields of the participant relation are utilized as follows:

- *tool name* - the particular name of the tool registering its interests.
- *tool path* - the path to the tool registering its interests.
- *interest events* - the events that this tool is interested in.
- *generate events* - the events that this tool might generate.
- *response events* - the events that this tool will respond to.

field	type	description
tool name	string	name of tool
tool path	string	file path to tool
interest events	array of Message	events the tool is interested in
generate events	array of Message	events the tool will generate
response events	array of Message	events the tool will respond to

TABLE 5.1. The LacEDAemon Participant Relation

field	type	description
Files	array of File	array of files involved in a project

TABLE 5.2. The Project Relation

5.2.2. Representing Leda Programs

Leda programs are represented by a number of relations, some of which are briefly described in Tables 5.2- 5.11. The relations map in a straightforward manner to the Leda grammar (described in Appendix A). We will examine how these relations store information about Leda programs in the next section.

field	type	description
File	string	name of the file
Path	string	full path to the file

TABLE 5.3. The File Relation

field	type	description
File	relation	inherits File relation
Line	integer	line number of the reference
LineTo	integer	ending line number of the reference

TABLE 5.4. The Location Relation

field	type	description
Location	relation	inherits Location relation
IncludeFile	relation	File included at Location

TABLE 5.5. The Includes Relation

field	type	description
Location	relation	inherits Location relation
Code	array of string	array of source lines

TABLE 5.6. The Source Relation

field	type	description
Location	relation	inherits Location relation
Name	string	name of class
Parent	string	name of parent class

TABLE 5.7. The Class Relation

field	type	description
Location	relation	inherits Location relation
Name	string	name of class
Function	string	function containing the declaration
Type	string	type assigned by the declaration
Category	relation	category of the declaration

TABLE 5.8. The Declaration Relation

field	type	description
Location	relation	inherits Location relation
Call	string	name of routine being called
From	string	name of routine call is being made from

TABLE 5.9. The Declaration Category Call

field	type	description
Location	relation	inherits Location relation
Name	string	name of function being defined
NumArg	integer	number of arguments to the function
Args	array of string	list of arguments to the function

TABLE 5.10. The Declaration Category Function

field	type	description
Location	relation	inherits Location relation
Member	string	name of class member being defined
Class	string	name of class in which member resides
IsData	boolean	data member (versus method)
Const	boolean	constant data member

TABLE 5.11. The Declaration Category Member

5.3. The Leda Program Information Extractor

One must have access to the program information (syntactic and semantic) in order to populate the database described in the previous sections. The Leda Program Information Extractor provides the service of extracting the relevant information from a Leda program in a form the LacEDAemon Program Database can consume. In this section, we describe the derivation of the Leda Program Information Extractor from the `lc` Leda interpreter.

Populating a database with information extracted via static analysis for the C and C++ languages was explored by Grass [75]. Using the tools *cia* (C Information Abtractor) and *cia++* (C++ Information Abtractor), Grass showed how creating a database of information allows for tools to graphically display various views of program structure, tools that answer queries about program symbols and relationships, and tools that can extract self-contained components from large systems. She later showed how this information could be used to recover design information (software archaeology) [74] and provide syntax-directed program differencing [73]. Duesing and Diamant [58] describe how this database of information also allows for creation of automated error detection tools in SoftBench.

The Leda Program Information Extractor is a static analysis tool that provides information regarding a Leda program, which the LacEDAemon program database then stores. Various tools rely on the information stored in the database to provide services such as a graphical depiction of the class hierarchy or file inclusion

structure, augmented pretty printing, and animated views of program execution. Since the program analysis tool is static, there are limitations on the knowledge contained in the program database.

The Program Information Extractor tool was derived from the language definition that is contained in the lexical analysis and grammar definition files of Leda. As a Leda program is parsed in exactly the same manner that it is prior to program execution, the LacEDAemon program database is updated with accurate information acquired via a modified parser. When syntactic structures of the Leda language are recognized by the parser, their details are supplied to the relevant program database relations.

A second version of the Leda Program Information Extractor was written in Java, using the Java CUP system. The Java Constructor of Useful Parsers (CUP) program [87] is a system for generating LALR parsers from simple specifications. CUP and its specifications are both closely modeled on yacc. CUP, however, is written in Java, has embedded Java statements in the specifications, and produced parsers implemented in Java. We utilize the Java-based version of the Leda Program Information Extractor to illustrate the fact that tools implemented in Java will soon have the capability of integration into the LacEDAemon integration framework. The mechanism for integrating Java-based tools is TclBlend [171], a mechanism from Sun Microsystems that allows Tcl programs to use all of the functionality of Java.

Regardless of language of implementation, the Program Information Extractor tool is easily derived from the language definition that is contained in the lexical analysis and grammar definition files of Leda. This process has relevance beyond Leda. Most research languages provide access to the implementation source code, or at least provide enough information that the language parser and grammar information can be reconstructed. Given the parser and grammar information, constructing a similar program information extraction system is not difficult.

5.4. An Example

Appendix B provides an entire Leda example program. The program in Appendix B simulates a Turing Machine, and provides Turing machine “programs” for the operations of testing for palindromes and binary addition. The file `turing.led` is the central file, creating the machine tapes and calling the Turing machine “programs”. All of the other program files are included into this file. The standard Leda library `std.led` is of course included, along with five other Leda source files.

The file `square.led` defines the tape Square class and provides class member functions or methods to access and print the contents of the tape. The file `rwhead.led` defines the Turing machine read/write head, with methods to read, write, and move the tape head, as well as print the contents of the current square. The file `tm.led` defines the Turing machine class and provides a method for executing the Turing machine program. The last two files, `tmpal.led` and `tmadd.led`

provide Turing machine “program” definitions for checking for palindromes and binary addition, respectively.

Once the Leda Program Information Extractor has processed the files in the project, the database reflects the information gathered. An illustration of a populated Class relation (Figure 5.7) for `std.led` is provided in Figure 8.4. Other relations in the Program Database are similar. Information extraction must also occur whenever a program update event (`extract_info`, `save`, `update`, `FileSave`, etc.) occurs.

5.5. A Revision Control System

Postgres supports the notion of historical data. Data in Postgres that is not part of the “current” state of the database is still maintained. Thus tuples that have been deleted or modified become part of a “past” state of the database, and these tuples are never physically overwritten or deleted. Both the old and new versions of the tuples are retained, with the old tuple invalidated. Postgres allows historical queries, or *time travel*, in which the user can specify timespans in which they are interested. A query that does not specify any time-related parameters will retrieve the current information. Queries can be made that result in past values of a class to be retrieved.

Ong has provided a generic description of managing versions within Postgres [132]. Here we briefly describe the aspects of version management pertinent

to creating a software configuration management system. There are two basic approaches to managing versions: *forward deltas* and *backward deltas* (or derivatives). Forward deltas start from an existing state of a program as a base, tracking additions to the tuples as well as the deleted/changed tuples in deltas. In order to create a current snapshot, the base tuples must have all of the modifications contained in the deltas performed upon them. Along with the possible performance penalty of having to reconstruct all the transactions to arrive at the current state, the forward delta approach also makes it hard to delete any tuples from the database.

Backward deltas perform changes to the program in place, while old values are recorded in the deltas. Since changes to the new version are made in place, no special procedure is required to access the “current” state of the program. Deriving old versions is somewhat more costly, but is a more infrequent operation. PostgreSQL utilizes the backward delta approach.

5.6. Persistent Objects

The presence of the LacEDAemon program database also makes it possible to create and store program values beyond the execution lifetime of a program, i.e. create *persistent objects* that can be shared between two or more program executions. We describe a conceptual framework for adding persistence to Leda here. Persistent programming languages are related to object-oriented database languages, but differ in that they neither allow all of the database services, nor do they necessarily allow

object access from multiple different languages. Persistence does allow systematic sharing of object values and services between program executions. In specifying persistent objects, three factors are of import: when persistent objects are specified, how they are specified, and which objects are eligible to be persistent.

Once persistent objects are present in a system, objects must be identified as either being persistent or transient. If a persistent object cannot become transient, then it is called permanently persistent, otherwise it is called temporarily persistent. Once a temporarily persistent object is deemed transient, its lifetime will not extend beyond the termination of the currently executing program. Also of interest is when objects are designated persistent: as part of the creation of the object, or at some time after its creation. The specification of a persistent object may be type-dependent or type-orthogonal, i.e., only specific types may be allowed to be persistent, or perhaps any object type can be persistent.

Another factor is the interaction between persistent and transient objects: the semantics of references in persistent objects to transient objects must be defined. The usual approach is to create a permanently persistent root for all persistent objects, and to perform reachability analysis from this root.

6. THE PRETTYPRINTER

6.1. Prettyprinting

Pretty printing is a form of static code program visualization [128]. A prettyprinter takes as input a stream of characters and prints them with aesthetically appropriate indentations and line breaks [7, 6]. Prettyprinters are integral components of any programming environment tool [133]. One of the most well-known prettyprinters is the SEE system, created by Baecker and Marcus [9].

Program formatting has been found to be a factor in program comprehension studies—Oman and Cook's [131] experiments with a book format indicated statistically significant improvements in programmer performance on comprehension and maintenance tasks. This improvement has led to some proposals to typographically augment programming languages themselves [1, 50].

Here, we describe a prettyprinter for Leda. This prettyprinter provides more aesthetically pleasing treatment of program text through the use of boldface and italic type, as well as uniform program indentation. Additionally, a facility for enhanced presentation of first-class functions is provided, called *function boxing*.

6.2. First Class Functions in Leda

First-class functions can be defined as functions that have the same status as any other values. Thus a first-class function can be the value of an expression, can be passed as an argument, and can be put in a data structure [166]. The functional programming paradigm, which treats functions as first-class citizens, permits the creation of powerful operations on collections of data. Some examples of functional programming languages include Haskell, ML, and Lisp.

Functions are also first-class in Leda, i.e. a function can be passed as an argument, assigned to a variable, or returned as the result of executing another function. Figure 6.1 illustrates currying in Leda. The unary function `triple` is the result of binding one argument of the nameless multiplication function to 3, using the `curry` function, which returns a function.

6.3. The Leda Prettyprinter

The Leda prettyprinter produces either ASCII or \LaTeX [110] output. The ASCII text output is simply indented in a standardized manner. This option is of limited use in the event the Leda programs were created with the language sensitive editor, as the programs will automatically be indented in a standard manner. However, as users may have different preferences with respect to the indentation level, the prettyprinter allows user specification of what a single indentation unit should be in number of spaces.

```
{ using functionals: the curry operation }

type
  binaryFunc : function(integer, integer)->integer;
  unaryFunc  : function(integer)->integer;

function curry (boundFun : binaryFunc,
                boundValue : integer)->unaryFunc;
begin
  return function (item : integer)->integer;
    begin
      return boundFun(item, boundValue);
    end;
end;

var
  triple : unaryFunc;

begin
  triple := curry(function(x: integer, y: integer)->integer;
                  begin return x*y;end, 3);
  print("triple of 7 " + triple(7) + "\n");
end;
```

FIGURE 6.1. Currying In Leda.

The `LATEX` option produces a much richer output, although using it requires the presence of the `LATEX` typesetting system and the availability of `LATEX` or PostScript viewers and/or printers. The output has italicized comments, bolded keywords, aesthetically improved program symbols, and graphically highlighted functions.

6.4. An Example

Figure 6.1 is an example of binding one argument of a higher-order function (or functional). The resulting function is commonly called a *curry*. There are two arguments to `curry`: the binary function and the argument that is to be bound to a constant. The resulting unary function, `triple`, triples the value of the passed argument, as a result of having one argument of a general (unnamed) multiplication function bound to the constant 3.

Figure 6.2 illustrates the result of passing the `curry` example from Figure 6.1 through the `prettyprinter`. Note comments are italicized, keywords are bold, some characters are aesthetically improved (such as \Rightarrow instead of `->`), and functions are enclosed in a box, allowing them to stand out.

Note the special typographic treatment of first-class Leda functions. When first-class functions are explained in a pedagogic setting, they are often circled or highlighted as to make clear the bounds of the function being manipulated. Func-

```

{ using functionals: the curry operation }

type
binaryFunc : function (integer, integer) ⇒ integer;
unaryFunc  : function (integer) ⇒ integer;

function curry (boundFun : binaryFunc, boundValue : integer) ⇒ unaryFunc;
begin
  return function (item : integer) ⇒ integer;
  begin
    return boundFun (item, boundValue);
  end
end;

var
  triple : unaryFunc;

begin
  triple := curry (
    function (x : integer, y : integer) ⇒ integer;
    begin
      return x * y;
    end
    , 3);
  print("triple of 7 " + triple(7) + "\n");
end;

```

FIGURE 6.2. Currying Example, Prettyprinted.

tion boxing provides an automated mechanism to highlight the presence of these functions.

6.5. Database Directed Prettyprinting

The presence of the LacEDAemon program database allows for the prettyprinter to take advantage of information gathered during static analysis of the Leda program. This information can be presented along with the enhanced program listing to provide the programmer with additional information that would require additional effort on the part of the programmer to acquire. This information includes the class hierarchy and file inclusion details. Program information in the form of statements such as “the class `boolean` inherits from class `equality`” or “the file `tm.led` is included in the file `turing.led`” collected in a readily-accessed location in the pretty-printer output may be of value to the programmer. The program database also facilitates indexing of information to allow for the creation of book-style program listings [131].

The book program listing can also be adapted to on-line access via Hypertext Markup Language (HTML) and an HTML browser. Simple queries to the program database combined with straightforward text processing provide the information required to create the necessary HTML version of the enhanced program listings.

7. THE PROGRAM ANIMATOR

Algorithm animation provides users with dynamic graphical depictions of the data and operations of an algorithm. The graphical depictions allow users to better understand programs, evaluate existing programs (for performance or reuse), and develop new programs. Polka is an algorithm animation system with an interactive animation interpreter and generator tool called Samba. Algorithm animation systems can also be used to provide a vehicle for other types of animation applications. Here we describe how we use Polka to provide animated views of Leda program execution in the LacEDAemon programming environment.

7.1. Leda does the Polka

A number of algorithm animation systems have been built over the last decade. These include systems like Balsa [21], Tango [173], and Zeus [22]. For a good summary of different algorithm animation and software visualization systems, see [PBS93].

The Polka algorithm animation system and environment [174] is a follow-on system to Tango and XTango [172]. Animations in Polka consist of C++ programs.

Samba provides an interpreted, interactive animation front end to Polka. The Samba interactive animation interpreter reads commands from a textual file to

acquire directions for creating an animation. Samba provides nearly 90% of Polka's base functionality. Stasko's description of students utilizing the Polka/Samba system to create animations provides a good summary of how we intended to integrate this tool into LacEDAemon [174]:

Essentially, a student must annotate the implementation of an algorithm with "print" statements to generate the commands to drive Samba. When the program executes, these print statements will be output in an order corresponding to the execution and will comprise a trace of the program's operations. The print statements correspond to the interesting events often used in algorithm animation systems. The output trace is then forwarded to Samba which generates the specified animation.

Animations developed with Samba are carried out in windows with a real-valued coordinate system that originally runs from 0.0 to 1.0 from left-to-right and from bottom-to-top (actually, the coordinate system is infinite, and Samba allows zooming and panning throughout the system). Graphical objects are created and placed within the coordinate system. These objects can be moved, change color, visibility, fill, etc. in order to depict the operations being depicted.

Polka and Samba were originally implemented for UNIX in C++, on top of the X11 Window system. A newer version of Polka and Samba are available for Windows 95/NT. There is also a Java version of Samba called JSamba.

```
1  include "std.led";
2
3  function square (val : integer) -> integer;
4  var
5      tmp : integer;
6  begin
7      tmp := val * val;
8      return tmp;
9  end;
10
11 var
12     i : integer;
13 begin
14     i := 3;
15     i := i + 1;
16     print (i);
17     i := square (i);
18     print (i);
19 end;
20
```

FIGURE 7.1. An Example Leda Program.

7.2. Using Polka/Samba to Present Leda Execution

Figure 7.1 depicts an example Leda program. The program simply includes the standard Leda library, `std.led`, has a function called `square` that squares the integer parameter passed to it, and some code to call `square` and print the results.

Figure 7.2 illustrates how the Leda interpreter `lc` provides textual clues as to its progress in program execution. Here, the program being executed is from Figure 7.1. Embedded in the output is the line number of the program being executed, along

with much other information that will not be of interest to novice (and possibly expert) programmers. By modifying the `lc` interpreter to broadcast a `LineAdv` notification, the programmer is provided with more comprehensible information via the Samba/Polka program animator. Figure 7.3 illustrates the Samba/Polka animation interpreter commands required to illuminate execution progress of the Leda interpreter. Note that the Samba *animates* the execution of the program, i.e., progress in the program is depicted via motion in the animation view. This animation provides additional reinforcement as to the progress of program execution.

7.3. Mediators: Addressing Impedance Mismatch in Tools

The difference between the output of the Leda interpreter `lc` and the input required by Samba are easily discerned. This is a case of *impedance mismatch* between tools being integrated. This is one of the key issues facing the tool integrator: providing the translation required between tools in order for them to interoperate.

Sullivan’s PhD thesis, “Mediators: Easing the Design and Evolution of Integrated Systems,” proposes the use of *mediators* (also called adaptors) to accomplish the task of tightly integrating the behaviors of separate software components [180]. Mediators provide data conversion for interoperability, allowing two or more software components to cooperate despite difference in language, interface, or execution platform. Here we use mediators to integrate the behavior of the Leda interpreter’s

```

$ lc -ds a.led
parse ok, starting execution
File a.led Line 14: expression statement
File a.led Line 15: expression statement
File std.led Line 347: return statement, yields
  1073918240
return from function unknown function(1074374680)
File a.led Line 16: expression statement
File std.led Line 505: conditional statement
File std.led Line 508: expression statement
do function (1074122600) call unknown function
  (1074369392), now do call
File std.led Line 300: return statement, yields
  1074122512
File std.led Line 456: expression statement
4return from function unknown function(1074387688)
File a.led Line 17: expression statement
File a.led Line 7: expression statement
  :
  :
  :
File a.led Line 18: expression statement
do function (1073918448) call ? (1074393760),
  now do call
File std.led Line 505: conditional statement
File std.led Line 508: expression statement
do function (1074122272) call unknown function
  (1074369392), now do call
File std.led Line 300: return statement, yields
  1074122184
return from function unknown function(1074369392)
File std.led Line 456: expression statement
16return from function unknown function(1074387688)
return from function ? (1074393760)
execution ended normally

```

FIGURE 7.2. Leda Statement Trace Output.

```

viewdef a.led 600 600
view a.led
bg black
text 1 0.05 0.95 0 white 1: include "std.led" ;
text 2 0.05 0.9 0 white 2:
text 3 0.05 0.85 0 white 3: function square ( val : integer ) -> int
text 4 0.05 0.8 0 white 4: var
text 5 0.05 0.75 0 white 5: tmp : integer ;
text 6 0.05 0.7 0 white 6: begin
text 7 0.05 0.65 0 white 7: tmp := val * val ;
text 8 0.05 0.6 0 white 8: return tmp ;
text 9 0.05 0.55 0 white 9: end ;
text 10 0.05 0.5 0 white 10:
text 11 0.05 0.45 0 white 11: var
text 12 0.05 0.4 0 white 12: i : integer ;
text 13 0.05 0.35 0 white 13: begin
text 14 0.05 0.3 0 white 14: i := 3 ;
text 15 0.05 0.25 0 white 15: i := i + 1 ;
text 16 0.05 0.2 0 white 16: print ( i ) ;
text 17 0.05 0.15 0 white 17: i := square ( i ) ;
text 18 0.05 0.1 0 white 18: print ( i ) ;
text 19 0.05 0.05 0 white 19: end ;
text 20 0.05 0 0 white 20:
triangle 0 0.01 0.98 0.04 0.965 0.01 0.95 red solid
view a.led
coords 0 -0.2 1 0.8
delay 500
color 14 green
delay 1500
color 14 white
viewdef std.led 600 600
:
:

```

FIGURE 7.3. Samba/Polka Animation Commands.

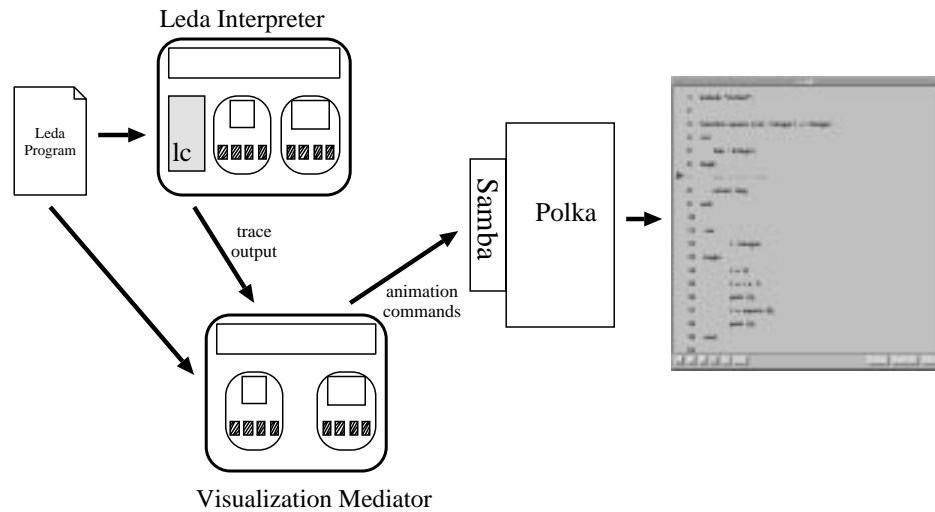


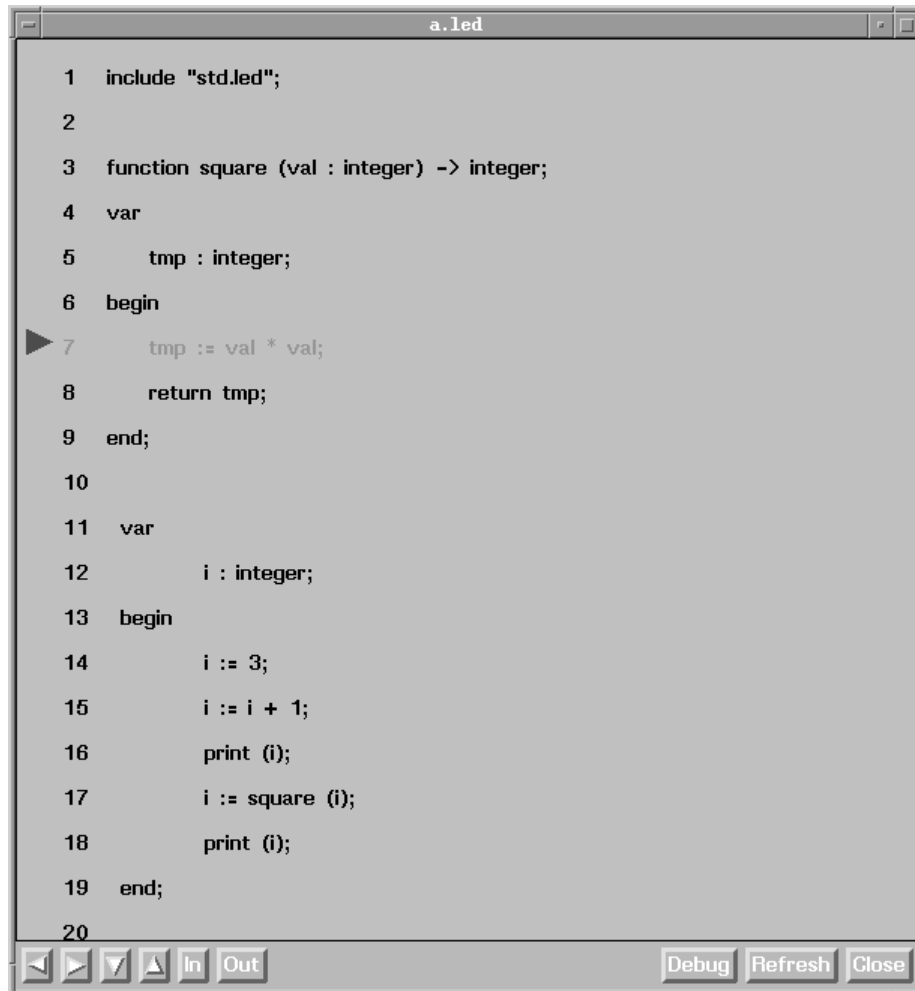
FIGURE 7.4. Execution Visualization in LacEDAemon.

textual progress clues and Samba capability to animate the same. This mediator-based design is illustrated in Figure 7.4.

An impedance mismatch can be addressed via the use of mediators, as the behaviors requiring modification are accessible to the tool integrator. A different problem is *architectural mismatch*, where tools make mismatched assumptions regarding the structure of the system it is to be a part of. Garlan et al. [65] describe the issues associated with architectural mismatch.

7.4. Algorithm Animation in LacEDAemon

While Polka provides an animated view of program execution, as depicted in Figure 7.5, its presence can be exploited for other program comprehension activities as well. Algorithm animations might expedite understanding of multiparadigm



```
1 include "std.led";
2
3 function square (val : integer) -> integer;
4 var
5     tmp : integer;
6 begin
7     tmp := val * val;
8     return tmp;
9 end;
10
11 var
12     i : integer;
13 begin
14     i := 3;
15     i := i + 1;
16     print (i);
17     i := square (i);
18     print (i);
19 end;
20
```

FIGURE 7.5. Execution Visualization using Samba.

algorithms, and with the Polka/Samba system integrated into LacEDAemon, such animations can be easily constructed.

8. THE PROGRAM EDITORS AND BROWSERS

An alternative to structure editing is the addition of language knowledge to standard text editors. *Language-knowlegable editors* are a compromise between straight text editors and structure editors. They provide such capabilities as automatic indentation, parenthesis checking, and even simple cross-referencing [155]. Here we describe the integration of a language-knowlegable editor into the LacEDAemon environment.

Browsers have been accepted as fundamental, powerful tools for exploratory program development. They also have the potential of being very effective during program maintenance. Maintainers are usually not the original developers of a program and often can depend only on the source code as the up-to-date documentation. Before making changes to a large, unfamiliar program, maintainers usually spend considerable time understanding the program structure and the interconnection of its components. Browsers help maintainers determine the scope of a change by allowing them to interactively examine the program structure and ask which components may be affected by a change [53]. Here we describe a variety of browsers available in the LacEDAemon environment.

8.1. A Customized Editor for Leda

Programmers have always browsed programs with text editors, such as vi [84] or emacs [170]. Emacs can be customized to great lengths in order to better support various programming-related tasks [68]. Emacs is also a very popular editor among programmers. The FIELD [151] project reports that “the primary lesson we learned from FIELD about editors is that users want to use the editor they’re used to, no matter what additional features a new editor provide.”

Emacs is the highest-function text editor available in the public domain. Emacs provides assorted functionality via *modes*, which allow for variable definitions based on the type of material being edited. Figure 8.1 illustrates XEmacs version 20.4 in the Leda major mode.

The Leda major mode allows for structure-oriented editor functionality such as:

- Understanding the structure of program constructs, and allowing for structure-based editing features.
- Defining and enforcing a specific indentation and code appearance standard.
- Providing specialized menus that are language-specific (Figure 8.2).

Customization is provided via Emacs Lisp definition files. These files can be compiled into byte code to create compact, faster running files. The Leda mode definition file consists of approximately 400 lines of Emacs Lisp code.

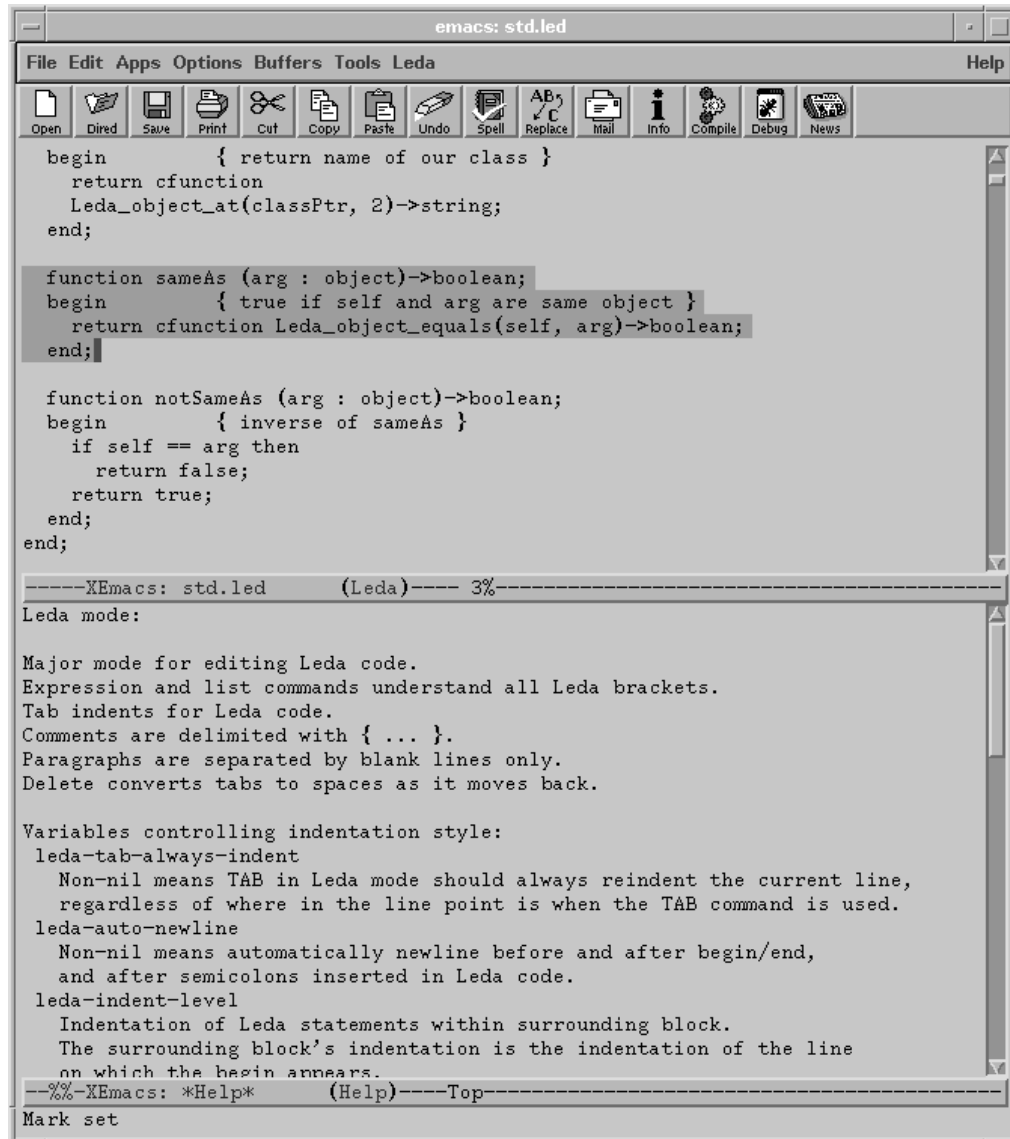


FIGURE 8.1. The Leda Major Mode in Emacs.

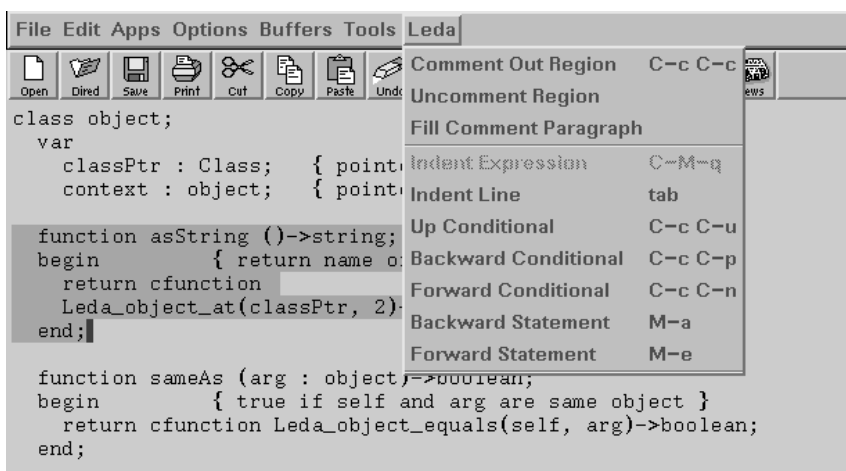


FIGURE 8.2. The Leda Menu in Emacs.

8.2. A Smalltalk-Like Browser

The Smalltalk language and associated environment [69] advanced programming and programming environments considerably: one such advancement was the Smalltalk browser. In a concise and straightforward manner, users could explore and examine program values. We attempted to create a similar browser for the Leda language.

The first attempt we made at creating such a browser was through the Common Open Software Environment (COSE) Common Desktop Environment (CDE) `dtbuilder` graphical user interface builder [42], which provided an graphical, interactive, direct manipulation environment within which to generate X-window/Motif [198] based user interfaces. Figure 8.3 depicts a prototype generated



FIGURE 8.3. Prototype of a Smalltalk-like Browser.

via this system. A second, more successful attempt was done by creating the Leda browser using Tcl/Tk, via the SpecTcl user interface builder.

8.3. Depicting Class Hierarchies

We again use the mediator concept, in conjunction with the program database, to depict class hierarchies. The program database is populated with information gathered via the Leda Program Information Extractor. Figure 8.4 illustrates the class information as stored in the LacEDAemon program database, as well as the definition of the class relation and an example of adding program information to the database. This is shown here in POSTQUEL, although in the LacEDAemon environment the tasks of definition, update and access of data are performed via the `libpgtcl` support for Tcl-based clients on the front-end, with `pgtclsh` adding Tcl commands for back-end interface.

A mediator is employed to transform the data acquired from the database into a form that can be consumed by the class hierarchy illustration tool. This transformation is straightforward—though the concept can be applied in situations of greater complexity. The integration framework of LacEDAemon coupled with the scripting power of Tcl allows for tools to be integrated into the environment with little effort: that of encapsulating the tool to allow messaging capabilities and the creation of appropriate mediators.

```

CREATE TABLE Location (
    File    text,
    Path    text,
    Line    int );

CREATE TABLE Class (
    Name      text,
    Parent    text
) INHERITS (Location);

INSERT INTO Class VALUES
('std.led', '/users/rpandey/', 44, 'equality', 'object');

SELECT * FROM Class;

```

File	Path	Line	Name	Parent
std.led	/users/rpandey/	12	object	
std.led	/users/rpandey/	44	equality	object
std.led	/users/rpandey/	67	boolean	equality
std.led	/users/rpandey/	100	True	boolean
std.led	/users/rpandey/	127	False	boolean
std.led	/users/rpandey/	157	ordered	equality
std.led	/users/rpandey/	191	Class	ordered
std.led	/users/rpandey/	236	real	ordered
std.led	/users/rpandey/	296	integer	ordered
std.led	/users/rpandey/	429	string	ordered
std.led	/users/rpandey/	704	array	equality

(11 rows)

FIGURE 8.4. Storing Class Hierarchy Information.

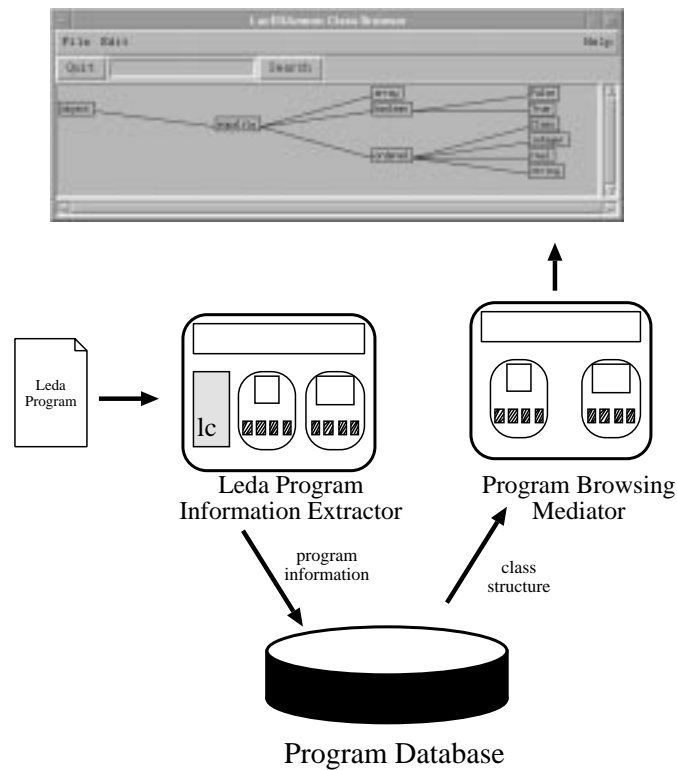


FIGURE 8.5. Graphical View Of Class Hierarchies.

Figure 8.5 depicts the architecture of the LacEDAemon class browser. Figure 8.6 provides a more detailed view of the class browser. Note that in order to keep the size of the window small, a search capability is provided. When a class name is entered by the user in this window, the hierarchy will highlight the class in the tree (if found), and the tree will be centered in the window with the desired class visible. Double clicking on any node of the tree opens a browser window with a listing of the Leda source code defining the class of interest.

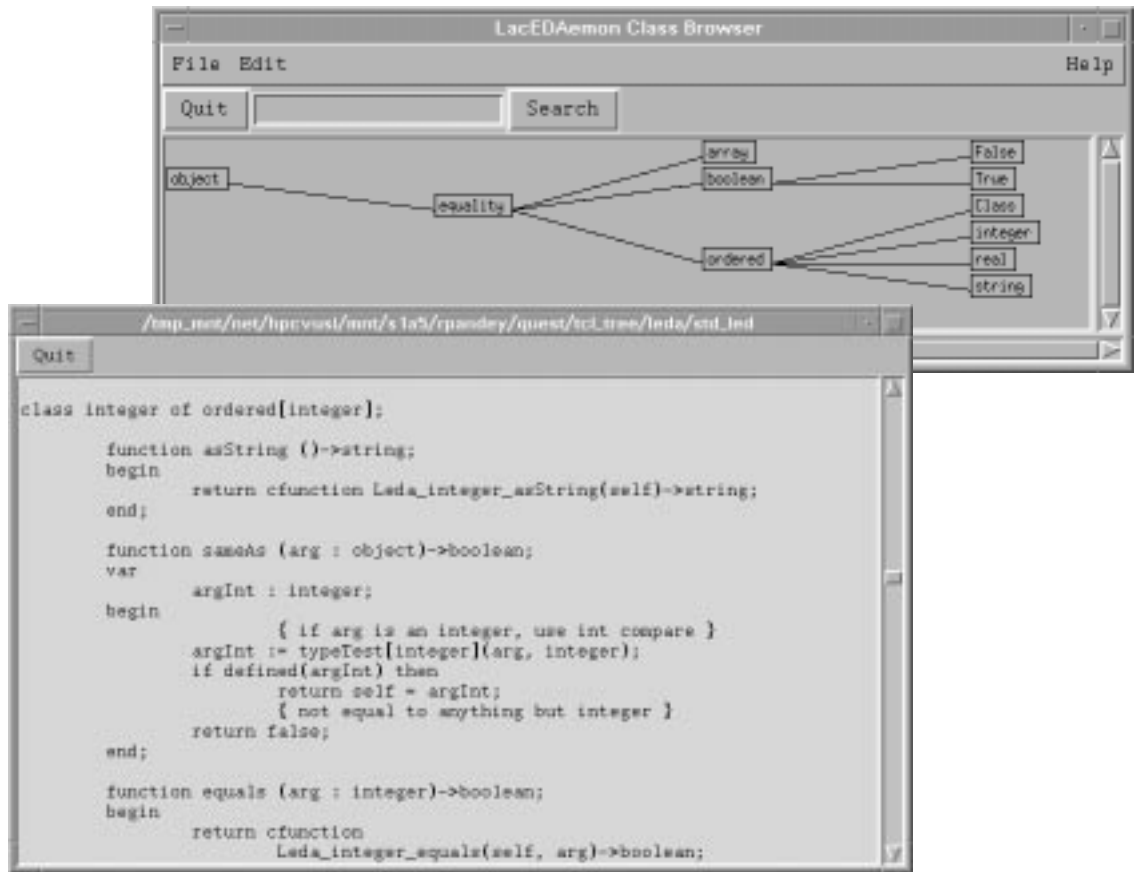


FIGURE 8.6. Class Hierarchy Browser.

9. LITERATE MULTIPARADIGM PROGRAMMING

Literate programming is a system of combining programming and internal documentation so that they may be co-developed with ease. One hope is that the very close proximity of code and documentation will help ensure their mutual correspondence. Literate programming was developed by D. E. Knuth in the late 1970s [103]. One of the best examples of usage of a literate programming system continues to be Knuth's typesetting system, \TeX [104].

LacEDAemon permits the creation of literate programming systems by allowing for relations expressing the connection of documentation to occur in the program database, while storing the documentation within any database class. The browsers and editors that populate LacEDAemon can be made sensitive to the documentation field of database classes, displaying the documentation when the particular code fragment is accessed.

In this chapter, we describe the database classes modified to support documentation in parallel to code fragments, the tools sensitized to this documentation, and examine adapting an existing literate programming system to LacEDAemon. We also show how non-textual documentation (i.e. graphical animations) can be added to the system due to the presence of animation tools.

field	type	description
Files	array of File	array of files involved in a project
Documents	array of Document	array of documents involved in a project

TABLE 9.1. The Documented Project Relation

field	type	description
Document	string	name of the document file
Path	string	full path to the document file
Accessor	string	full path to the document accessor

TABLE 9.2. The Document Relation

field	type	description
Location	relation	inherits Location relation
Code	array of string	array of source lines
Comment	array of string	array of comment lines

TABLE 9.3. The Commented Source Relation

9.1. CWEB

CWEB [105] is a version of Donald Knuth's WEB system, adapted to C and C++. CWEB consists of two programs, CWEAVE and CTANGLE. When a user writes a program, the code and documentation is kept in the same file. The CWEAVE program creates a T_EX file that is a prettyprinted version of the program and documentation, with correct handling of typographic details as well as the use of indentation, italics, boldface, and mathematical symbols. The output when typeset also includes extensive cross-index information which is gathered automatically. Similarly, the CTANGLE generates a program file that can be consumed by the language compiler to yield executable code.

9.2. Adapting Literate Programming to LacEDAemon

The presence of a program database, prettyprinter, a graphical user interface and the ability to create non-text documentation (e.g. animations) modifies the approach to literate programming we take in LacEDAemon. The documented project relation described in Table 9.1 allows us to provide documentation files at a project level. The files, recorded in the Document relation (Table 9.2), can be text accessed by a text editor, drawings accessed via a graphics application, or animations accessed via the Polka/Samba animation engine. This relation records the accessor to allow for non-text documentation.

Each source line can be accompanied by comments, which is illustrated in Table 9.3, the Commented Source Relation. This relation obviates the need of programs such as `CWEAVE` and `CTANGLE`. The editor and prettyprinter can present the comments along with source statements for programmer benefit, while presenting only source statements for the purposes of parsing, program information gathering, and compilation.

10. DEBUGGING MULTIPARADIGM PROGRAMS

The task of *correction*, altering a program's behavior to make it the desired behavior, requires tools for comprehension of the program's behavior. These tools are most often debuggers and browsers. While browsers provide an overview of the program structure, debuggers are able to provide fine-grain resolution of the details of program behavior. Here we describe debuggers in more detail, followed by our work in augmenting the behavior of the Leda interpreter to provide debugger-like facilities. Finally, we illustrate tool interactions when a programmer is engaged in correction (and hence comprehension) of a Leda program. The motivation in minimally augmenting the Leda interpreter to provide some debugger facilities is to show how the presence of other tools integrated into the LacEDAemon environment magnifies the impact of the debugger tool, and expedites the task.

10.1. Debuggers

Rosenberg [161] defines debuggers as “tools to illuminate the dynamic nature of a program—they are used to understand a program as well as find and fix defects.” Along with a source code editor, the debugger is considered a requisite tool for the development of software.

Major capabilities a debugger should provide include:

1. breakpointing
2. single-stepping
3. fault detection
4. watchpointing

Good examples of debuggers include dbx [115], GNU gdb/Dalek [130], and Hewlett-Packard's DDE [88].

Rosenberg goes on to describe four key debugger principles:

- Heisenberg Principle—the act of debugging an application should not change the behavior of the application.
- Truthful Debugging—the debugger must never mislead the user (see Zellweger's 1984 PhD thesis on debugging optimized code [203] for more on truthful debugging) .
- Context Information—the debugger must provide context information that illuminates program state and execution.
- Debugging Support Trails Systems Developments—debuggers and other programming tools always trail the development of new systems.

The multiparadigm programming language Leda is certainly in a state that supports the last point made by Rosenberg, namely that there is no debugging sup-

port in the `lc` Leda interpreter. Furthermore, Rosenberg's assertion that "debugging can often be an afterthought of systems design" is borne out by `lc` as well.

The possibility of adding debugging support to Leda was encouraging, however, since the `lc` system is an interpreter, and as Rosenberg identifies:

Interpretive programming environments, such as those available for Basic, Smalltalk, and Java as well as other high-level languages, provide very effective debugging solutions because the debugger is well integrated into the run-time interpreter and has very tight control over the running application.

Hence, the tantalizing possibility of adding effective debugging to the `lc` interpreter and the LacEDAemon environment was worth pursuing. No special interactions with the operating system or underlying hardware need take place, and the debugger can provide direct, immediate feedback on user changes, as well as providing a safe, protected environment in which both the target application and the debugger can run.

10.2. Augmenting The Leda Interpreter

The `lc` interpreter for Leda provides some debugging infrastructure through its ability to trace the execution of functions (the `-df` option), the execution of functions and program statements (the `-ds` option), and the execution of functions, statements and operators (the `-do` option). Here we describe adding some other critical abilities to the `lc` interpreter. We add a new option to the Leda interpreter, the `-db` option, to access this new functionality in stand-alone mode. Note that

access to this functionality from the LacEDAemon tool integration framework is trivial, as the Leda API is available to the messaging service, as described in section 4.5.3. Full debugger functionality could be provided in the case of a Leda compiler via the UNIX `ptrace()`, `ttrace()`, or `/proc` interfaces (and their equivalents in other operating system environments).

10.2.1. Adding Breakpointing

Adding breakpointing, the functionality of temporarily halting program execution at a user-designated location, is relatively easy in the context of having statement execution information available. The augmented `lc` interpreter simply makes a check during execution and determines whether the user has designated the current statement of execution as a breakpoint. If this is the case, program execution is halted at the juncture, and the user informed of the fact that a breakpoint has been reached.

10.2.2. Single-Stepping Through Programs

The ability to stop execution after the execution of every line of a Leda program is a trivial extension of breakpointing. Every line of the program is simply considered a breakpoint. The interpreter suspends execution of the program and reports this fact to the user.

Note that both in the case of breakpointing and single-stepping, there are lines of Leda programs that are not technically “executed”. We are not concerned with the difference between logical and physical program statements and their breakpoint candidacy here, as this functionality is dependent on the Leda compiler or interpreter implementation, and is beyond the scope of this work.

10.2.3. Examining Symbol Values

For the sake of simplicity, the symbols defined in the current scope are listed in debugger mode, with the user specifying the particular symbol in which they are interested. This results in the value of that symbol, if defined, being displayed. Most debuggers provide an elaborate command line interface for accessing and modifying symbol values. The Leda interpreter `lc` could be similarly extended, and a graphical interface within the LacEDAemon environment could also be provided.

10.2.4. Watchpointing

Setting a watchpoint, or instructing the debugger to inform a user that a specific program location has been executed or reached, is again a trivial case of breakpointing, with the sole difference that program execution is not suspended and user interaction solicited. The reaching of a watchpoint is duly reported to the user, with execution continuing beyond the watchpoint.

```
$ lc -db a.led

parse ok, starting execution
b - set breakpoint, c - continue, s - step, v - view
debug> s
File a.led Line 5: expression statement
    i := 3;
b - set breakpoint, c - continue, s - step, v - view
debug> s
File a.led Line 6: expression statement
    print (i);
b - set breakpoint, c - continue, s - step, v - view
debug> v
1: include "std.led";
2: var
3:     i : integer;
4: begin
5:     i := 3;
6:     print (i);
7: end;
8:
b - set breakpoint, c - continue, s - step, v - view
debug> c
File std.led Line 505: conditional statement
File std.led Line 508: expression statement
File std.led Line 300: return statement, yields 1074130176
File std.led Line 456: expression statement
3
execution ended normally
```

FIGURE 10.1. Single Stepping in Leda.

A data watchpoint, where the user instructs the debugger to inform them of changes in the value of a symbol, is similar to examining a symbol value. The difference is that a data watchpoint simply reports a change in value or access of a symbol, while the examination of symbol values occurs upon reaching a breakpoint.

10.2.5. Fault Detection

Fault detection support, or detection of events such as divide by zero, improper memory access, and improper I/O, is usually functionality provided by the underlying hardware or operating system.

Adding detection of divide by zero errors is easy to implement in the `lc` interpreter. The implementation of the division operator is accessible in the `std.led` library, which is one location for implementing this feature. We choose to implement it within the `lc` source code (in the `Leda_real_divide` function) for simplicity, however. Detection of other faults is more difficult, requiring operating system involvement, and we will not examine this issue further here.

10.3. Composing Interactions With LacEDAemon Tools

When debugging in the context of the LacEDAemon programming environment, the presence of the other tools allows for greater expression and elaboration of program state to the user. The augmented `lc` interpreter broadcasts messages announcing arrival at breakpoints, the Polka/Samba-based animation tool allows

for visualization of that specific line, the hypertextual browsing allows a user to examine implications of arriving at that line, the class browser allows for comprehension of class structure, and so on. The debugger by itself could not provide this information, but by simply providing the information it possesses to the other tools, it is able to allow the other tools to provide whatever additional information might be pertinent to the user.

11. EVALUATION

11.1. Evaluation of LacEDAemon

We will attempt to identify how LacEDAemon fits into various programming environment models, as well as how LacEDAemon compares in cases where the model is evaluative in nature. Direct comparisons are always difficult and prone to misinterpretation, particularly in some cases where we will be comparing the experimental prototype LacEDAemon to commercial systems. However, such comparisons can be useful, both in showing ways in which LacEDAemon could grow and by identifying alternative ways of achieving the same objectives.

11.1.1. The SMP Model and IFCS Taxonomy

Perry and Kaiser [142] have proposed the SMP model of software development environments consisting of three components: structures, mechanisms and policies. The structures are the underlying objects and object aggregates on which the mechanisms operate, while the mechanisms are the visible and underlying tools and tool fragments, with the policy being the rules, guidelines and strategies imposed on the programmer via the environment:

$$\text{General SDE Model} = (\{\text{Structures}\}, \{\text{Mechanisms}\}, \{\text{Policies}\})$$

Perry and Kaiser further outline the IFCS taxonomy, where an environment supports activity at the individual, family, city, or state class level. This sociological metaphor attempts to convey the problems of scale. Each category incorporates the prior classes, i.e. a family is a collection of individuals. The individual class emphasizes construction, and is dominated by mechanisms. The family class emphasizes coordination, and is dominated by structures. The city class emphasizes cooperation, and is dominated by policies, and the state class emphasizes commonality, and it dominated by higher-order policies. Perry and Kaiser find few city-class environments, and none in the state class.

The individual toolkit model is defined as follows:

TKE Model =
 (
 { file system / object management system¹ },
 { *assorted construction tools* },
 { *laissez faire, tool-induced policies* }
)

LacEDAemon for the most part qualifies as an individual, toolkit environment or individual, interpretive model in that it provides a collection of primarily language-specific tools, with few restrictions on tool usage (although LacEDAemon

¹*Italics* are used for general component descriptions, normal typeface for specific components.

imposes more restrictions than most toolkit environments due to its structure). The primary structure of LacEDAemon is the program database (which is closer to the integrated family model), but this database at present is designed to supply the needs of an individual. LacEDAemon also provides little policy enforcement with regard to tool usage (though, again, more than most toolkit environments).

Interesting observations include the fact that with little work, LacEDAemon could transition to the Family class of environments, as well as the fact that the SMP Model proposed by Perry and Kaiser is not really adequate for classifying LacEDAemon. This indicates some unique combinations of architectural attributes present in LacEDAemon that have not occurred in other programming environments.

11.1.2. Tool Integration in LacEDAemon

Wasserman [191] proposes viewing presentation, data, and control integration as defining a three-dimensional space, where different points along an axis define tool support for a particular integration mechanism in that dimension, as depicted in Figure 11.1. This notion can be formulated for an arbitrary tool T_i by the mathematical function:

$$T_i = f(D, P, C)$$

where D represents the data integration dimension, P represents presentation integration, and C control integration.

Thus a particular form of integration for tool T_1 could be

$$T_1 = f(\text{"UNIX file system"}, \text{"X11/Motif"}, 0)$$

indicating this tool uses the UNIX file system for data integration, uses X11/Motif for presentation, and provides no support for control integration. The average tool in LaCEDAemon, T_L could be described as

$$T_L = f(\text{"POSTGRESQL OODBMS/UNIX file system"}, \\ \text{"X11/Motif/Tk Toolkit"}, \text{LacEDAemon Messaging Server})$$

indicating that the tool relies on either the UNIX file system or the LaCEDAemon database (perhaps via a mediator) for data integration, the Tk toolkit (which on UNIX is based on X11) for presentation integration, and the LacEDAemon Messaging Server for control integration².

From this discussion, the issue of integration in a toolkit environment becomes apparent: minimal tool integration requires that tools agree on integration in at least one dimension, and effective tool integration requires tools to agree on all three axes. The obvious and easiest strategy for integration is to support the lesser integration mechanism along the three axes. The greater the degree of integration, the further out along the three dimensions a general tool lies.

LacEDAemon employs what Wasserman describes as the "more difficult approach" to integration: supporting the more advanced mechanism along each axis, via harness programs and changes to tools as needed. LacEDAemon tools present

²This is the case if tools are coupled using loose integration. Note that tight integration, or tool-to-tool messaging, is also available in LacEDAemon.

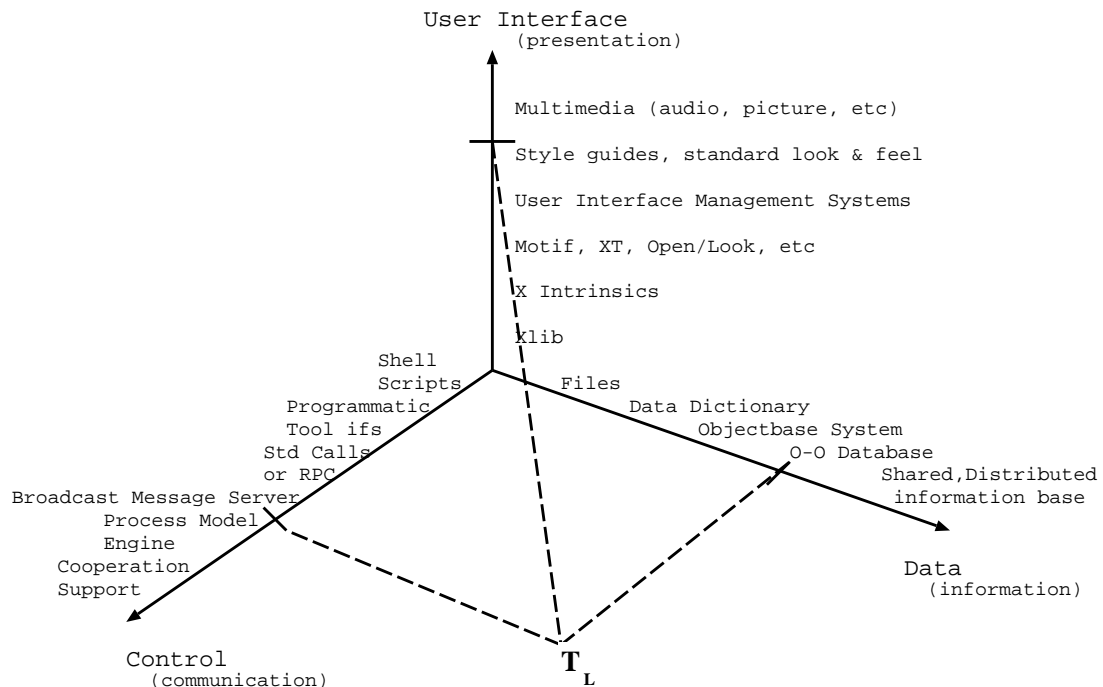


FIGURE 11.1. Mapping LacEDAemon Tools on the Three Dimensions of Integration.

a standard look and feel due to use of the same user interface toolkit, the PostgreSQL object-oriented database (which has the capability of being shared and distributed), and a broadcast message server approach to control integration. Unfortunately, Wasserman does not identify the level of integration provided by other toolkit environments.

11.1.3. The EBI Framework

Barrett, Clarke, Tarr and Wise proposed a generic framework for discussing event-based integration, called the EBI Framework [11]. This model provides a flexible, object-oriented model for discussing and comparing event-based integration

approaches. The EBI Framework model attempts to capture the following five aspects of an event-based integration framework:

- methods of communication
- expressiveness of module interaction descriptions
- intrusiveness of module interaction descriptions
- static versus dynamic behavior
- naming issues

While the EBI Framework model is designed mainly for loose integration frameworks, describing the LacEDAemon integration framework in terms of the EBI Framework may still provide some basis for comparison. Since the EBI Framework does not capture tightly integrated frameworks, however, we have not utilized this model to evaluate LacEDAemon.

11.1.4. Comparing Inter-Tool Communication

Harvey and Marlin proposed an operational model based on information structures to formally describe tool integration devices, facilitating their comparison [81]. The focus of this model is to provide a formal approach to describing the semantics of the intertool communication features (encompassing control integration, together with some aspects of data integration) of integration devices.

Harvey and Marlin propose formal description via layered operational semantics of the following eight communication events:

- Notification_publication
- Request_publication
- Notification_send
- Request_send
- Reply_send
- Notification_receive
- Request_receive
- Reply_receive

In [81], Harvey and Marlin examine the Request_send, Reply_send and Reply_receive communication events of FIELD and SoftBench. In Figures 11.2–11.4 we provide descriptions of the similar events for LacEDAemon.

11.1.5. The CEARM Model

Penedo [141] proposed the Conceptual Environment Architecture Reference Model (CEARM) as a conceptual and functional framework for the discussing, presenting and comparing of software engineering environment architectures. Figure 11.5 illustrates the reference model they propose for comparing environments.

```

Request_send →
1   A ← find item in thisTool.outputMsgs where {
2     msg == requestedService };
3   if A.msgBindings == NULL then
4     if START tool where {operation == requestedService} then
5       A ← find item in thisTool.outputMsgs where {
6         msg == requestedService };
7     end if;
8   end if;
9   if A.msgBindings ≠ NULL then
10    MSGID ← provide_msgID;
11    B ← insert item in thisTool.replyData where {
12      rID ← MSGID };
13    for all C in A.msgBindings do
14      B.rCount ← B.rCount + 1;
15      send message to C.toolID where {
16        messageID ← MSGID,
17        senderID ← thistool.toolID,
18        messageType ← "Request" };
19    end for all
20    /* synchronous event */
21    suspend operation except for {A.msgBindings};
22  end if.

```

FIGURE 11.2. The Request_send event in LacEDAemon.

```

Response_send →
1   [ either [ A ← “Success”;
2       | A ← “Fail” ]; /* error, unavailable */
3   send message to lastMsg.senderID where {
4       messageID ← lastMsg.messageID,
5       messageType ← “Response”,
6       messageData ← A,
7       senderID ← thistool.toolID }.

```

FIGURE 11.3. The Response_send event in LacEDAemon.

```

Response_receive →
1   A ← find item in thisTool.replyData where {
2       rID == lastMsg.messageID };
3   if A not NULL then
4       if lastMessage.messageData == “Fail” then
5           A.rCount ← A.rCount - 1;
6           if A.rCount == 0 then
7               theReply ← “Fail”;
8               remove item from thisTool.replyData where (A);
9               resume operation; /* synchronous */
10          end if;
11       else if lastMsg.messageData = “Success” then
12           theReply ← “Success”;
13           remove item from thisTool.replyData where (A);
14           resume operation; /* synchronous */
15       end if;
16   end if.

```

FIGURE 11.4. The Response_receive event in LacEDAemon.

The CEARM focusses on the functionality and properties of services provided by an environment. The CEARM consists of a set of layers: platform, framework, common services, tools/capabilities, environment adaptation and interaction. Each layer contains one or more grouping of services, as depicted in Figure 11.5. This graphical depiction can be thought of as providing a “may use” relation, in that each functional grouping in a layer may use the capabilities and services of a lower layer.

An environment mapped to CEARM results in a graphical depiction of the mapping, coupled with explicit mappings. While the graphical depictions can be misleading and subjective (as they provide little indication of how and whether the components are integrated and interoperate), the explicit mapping provides a better understanding of the system. In the graphical depiction, the guidelines for filling boxes is: no-fill means those services are not provided, 25% full means a few services are provided, 50% full means that a good approximation of services are provided, and 100% full means that almost all needed services are provided.

In [141], Penedo provides graphical depictions of six environments, along with explicit mappings for three of those systems. Here we provide the graphical depiction and explicit mapping for LacEDAemon.

CEARM Grouping	System Components
Platform	UNIX-based workstation, machine with GUI and C compiler
Virtual OS	POSIX
Object Management	POSTGRESQL Object-Oriented Database Management System
User Interface Management	X11/Motif runtime, Tk Toolkit
Environment Management	POSTGRESQL tool integration repository
Common Services	Request, response, notification reaction definition
Reuseable Components	Most tools implemented in C, C++, Java, Tcl
Integration Support Services	C, C++, Java, Tcl harnesses; Defined reaction to environment messages; LacEDAemon Messaging Server
Environment-building Services	C, C++, Java, Tcl harnesses
Life-cycle Functional capabilities	Program development (creation, correction, comprehension) and documentation, other UNIX tools
Adaptation Support	
Front-end/Desktop	graphical activity-driven interaction

TABLE 11.1. LacEDAemon Explicit CEARM Mapping

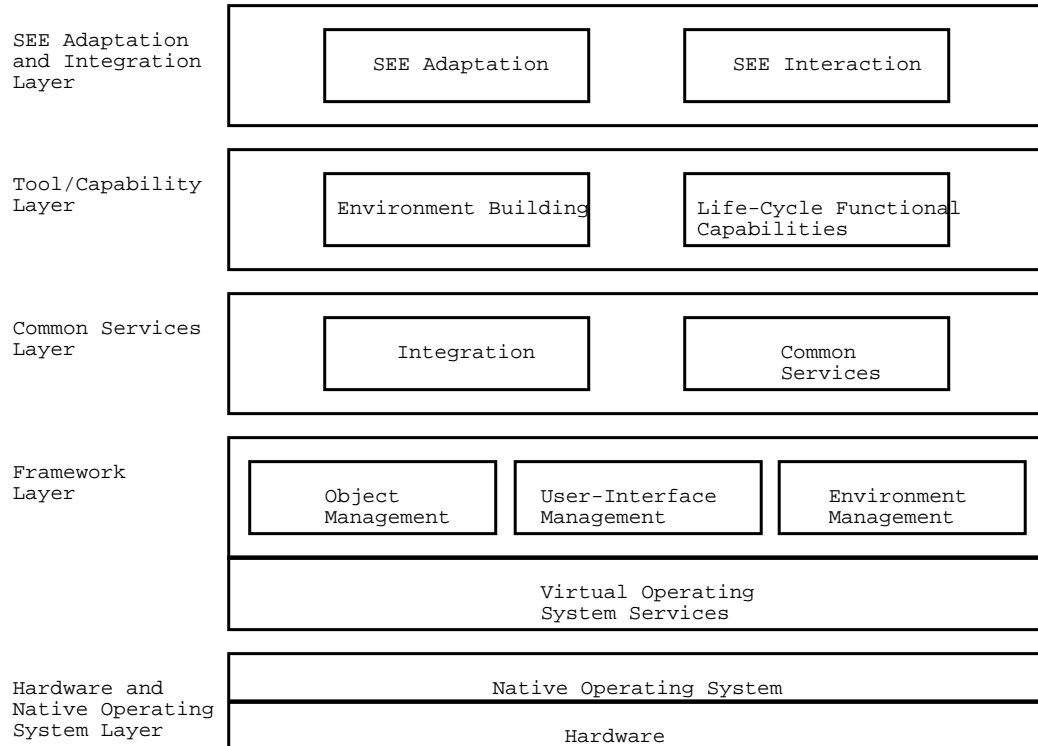


FIGURE 11.5. Conceptual Environment Architecture Reference Model.

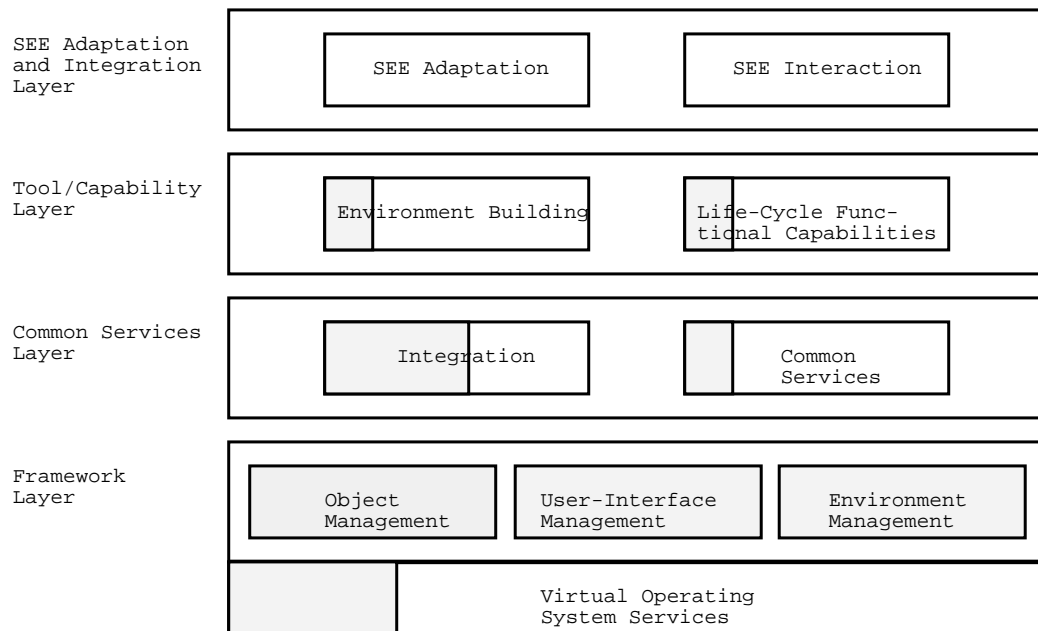


FIGURE 11.6. Mapping of LacEDAemon to CEARM.

11.2. Future Directions

While the assorted evaluation measures explored in this chapter indicate the relative uniqueness of LacEDAemon, there are a number of studies that can be conducted while utilizing the environment:

- Live testing via use in classroom - Effects of the presence of the environment can be determined in a controlled setting with a number of subjects with near-uniform backgrounds. Comprehension and maintenance exercises, such as those performed by Oman and Cook [131], would provide a good indication of the leverage provided by LacEDAemon. Student feedback from experience with the environment could also provide a subjective measure of toolset effectiveness.
- Nonintrusive usage/language studies via event recording and database monitoring - Since the environment is based on events, recording the events provides a good indication of the user activities. This event-journaling approach was used by Goldenson and Wang [71] in studying user behavior in GENIE. Another aspect of LacEDAemon is the presence of the program database—monitoring the database contents can also provide a detailed view of programmer activities. The combined timestamped transcript of environment events and program information changes allows for a variety of studies.

- Record-and-playback based studies - Tools like TkReplay [79] allow for explicitly recording the programmer activities for playback at a later time. While this facility is usually useful for demonstrations and help-related activities, record-and-playback can be used for programmer activity analysis as well.
- Study of effort required to add new tools - As a testbed framework, a major test of the utility of LacEDAemon will be in the effort required to add new tools. An industrial-strength Leda compiler project, Argo [93], will be the first measure of whether the structure of LacEDAemon meets the goal of allowing toolset experimentation.

12. CONCLUSIONS

12.1. Contribution of This Work

While the design and implementation of a new programming language is considered a labor-intensive activity, it is an active area of research, resulting in the creation of over 2350 languages. Constructing a programming environment to support the use of a language is an even more labor-intensive activity—one whose pursuit has thus lacked the same vigor as language creation. Knudsen et al. [102] note that “developments of environments are very large tasks, comparable more with developing operating systems than compilers...” Widespread use of new languages is usually envisioned in either industrial or educational settings (or both).

A programming language without tool support will not be given serious consideration by software engineers. A widely held belief is that large software systems are difficult to construct without a cohesive set of tools to facilitate the process. Brown et al. [19] observe that “the support environments which are used for large software development projects have a profound effect on the project’s end-product.” A programming language offered by itself is very uninviting, unless the language bears close conceptual resemblance to an existing language, allowing adaptation of existing tools, or addresses a need so pressing that the absence of tools must be overlooked or overcome.

A newly developed language without a programming environment will also be dismissed in academic circles, especially if the language is proposed as a pedagogic vehicle for novice programmers. Here the environment provides a means of selective occlusion: the tools put a productive, user-friendly “spin” on the quirks and limitations of language and underlying machine until users are more prepared to deal with them, at which point the tools scale to allow more detailed exploration. Kölling and Rosenberg [107] suggest that the programming environment “may well have more of an influence on the learning experience than the choice of a language.”

Almost all new programming languages are thus never applied pedagogically or in the construction of large-scale software systems, and many never escape the realm of scholarly discourse at all. The problems that newly-created, possibly innovative languages are hence applied to are mainly of the “toy” variety, with little substantiation regarding suitability of the language for larger problems. This lack of constructing large software systems to validate new programming languages would be reprehensible were it not for the rich set of research issues that are addressed simply in the creation of most new languages. Construction of small programs is usually sufficient for communicating the essential details of a language, verifying language implementation, resolving theoretical issues, and constructing arguments regarding language features.

Unfortunately there are also a whole set of research issues that cannot be answered without constructing larger programs, where “seat-of-the-pants” program-

ming must be replaced by software engineering discipline. The resolution of these research issues necessitates the construction of tools to support the language, and requires a much greater research investment. Peyton Jones et al. [92] observed the same need to engineer “substantial artefacts of software”:

Scaling prototypes up into large “real” systems appears to be less valued in the academic community than small systems that demonstrate concepts, being sometimes dismissed as “just development work”. Nevertheless, we believe that many research problems can only be exposed during the act of constructing large and complex systems.

Multiparadigm programming languages represent an area of research that is in such a state. After a flurry of research activity that answered questions regarding the implementation of multiparadigm languages and explored the various paradigm combinations, resolution of many of the research issues that remain outstanding will require the construction of larger software systems, and hence the support of programming environments. The application of multiparadigm programming languages in academic settings will similarly require the availability of a programming environment.

This work has described and demonstrated a new method of creating a tool integration framework, as well as integrating existing tools into a framework, for a programming environment for a multiparadigm programming language called Leda. The somewhat unique position of the Leda language in the realm of multiparadigm languages has also been described here.

The primary contribution of this dissertation to the field of software engineering is to describe and demonstrate a new approach to creating a programming environment based on tool integration frameworks. By embedding scripting language interpreters within existing tools (which are then called *hypertools*), we enable a powerful and low-cost approach for various forms of integration. The primary contribution of this dissertation to the field of programming languages is to demonstrate the hypertool integration framework approach in creating a programming environment for the multiparadigm programming language Leda, enabling more detailed investigation of the implications of multiparadigm programming than would otherwise be possible. LacEDAemon, the resulting hypertool integration framework-based programming environment for the multiparadigm programming language Leda, also provides a testbed for the design and deployment of new programming tools. The general hypertool approach also provides a mechanism for other experimental programming languages to more easily develop an associated programming environment than otherwise possible. The integration of a wide variety of tools to form a single application via a messaging framework and common graphical interface is also applicable to domains other than programming environments.

Three areas of improvement come to mind when considering how to advance the state of software development:

- better languages
- better tools

- better methods

This research touches upon two of the three topics. Better languages allow for better cognitive fit between problems and solutions. Better tools allow for these solutions to be formulated with less effort and in less time.

Also important to note here is that this work indicates two significant changes have occurred with respect to the creation of programming environments:

- the quality and complexity of tools available for reuse and integration has steadily increased, and
- the complexity of integration mechanisms has greatly decreased, while their flexibility has greatly increased.

These two factors combined have allowed a single tool integrator to create a non-trivial programming environment. Environments such as LacEDAemon created through the hypertool approach exhibit the ability to blend control and data integration. This ability to blend integration approaches has been identified as the next step in environment evolution [20]:

CASE repository standards emphasize data integration while broadcast models emphasize control integration. Both models are evolving towards each other through the addition of control-integration services in repository standards, and the implementation of message broadcast services on current-generation IPSE frameworks. The resulting frameworks promise tool integrators flexibility in choosing among integration mechanisms.

12.2. Assessing Reuse in LacEDAemon

Kaiser and Garlan [97] describe three prerequisites to achieving an order of magnitude improvement in software production through reuse:

1. language independence
2. component reuse through composition
3. reuse of components in ways not anticipated by the original programmer

The hypertool integration framework of LacEDAemon does not allow for language independence, however, in this thesis we have shown how components from many of the most popular current languages (i.e. the ones in which the majority of candidate components will be available) like C, C++, Java and Tcl/Tk can be integrated, or composed to create a larger environment. The philosophy of hypertools *is* to reuse components in ways not anticipated by the original programmer. Hence, the hypertool integration framework of LacEDAemon allows for significant reuse.

From a lines-of-code point of view. Table 12.1 lists the size in terms of non-comment source statements of some of the components of LacEDAemon. According to Voas [188], “even the best programmers can churn out only 10 lines of code per day.” Any strategy that advances this number dramatically requires heavy reliance upon reuse. As apparent from Table 12.1, the construction of LacEDAemon has required much reuse, as the environment represents many person-years of programming effort based on the 10 lines of code per day measure.

Component	Language	Lines of Code
X11 Library <code>libX11</code>	C	91,837
X11 Library <code>libXt</code>	C	31,006
Motif Library <code>libXm</code>	C	204,797
Leda Interpreter <code>lc</code>	C	5,627
Tcl	C	4,757
Tk	C	13,615
PostgreSQL	C	144,365
Samba/Polka	C++	9,907
Messaging Framework	Tcl	4,522
Class Hierarchy Illustrator	Tcl	945
Leda Program Information Extractor	Java	1,958
TclBlend	C	4,845
	Java	15,118
	Tcl	111
JavaCUP	Java	8,398

TABLE 12.1. Reuse in LacEDAemon: Lines of Code

12.3. Conclusions

We have shown the viability of integration mechanisms provided by scripting languages like Tcl. Tools and tool architectures must evolve to take advantage of the new generation of integration-support mechanisms. Instead of asking how the integration support mechanisms support tools, we should be asking how to design and build tools so that they can best take advantage of the mechanisms available.

Constructing programming environments is a difficult task—one that takes person-years of effort and years of time, and hence rarely gets done. Many interesting languages and problem solving approaches languish as a result of not having an environment to support serious programming effort-based investigations. We have described and demonstrated a new technique for constructing integrated programming environments in an effective and low-cost manner. In doing so, we have also constructed an environment for Leda, one of the most promising multiparadigm programming languages. We hope this environment will provide additional leverage for research in the area of multiparadigm programming, while this method of creating environments will further applicability-related research with other experimental languages.

BIBLIOGRAPHY

- [1] Paul W. Abrahams. Typographical Extensions for Programming Languages: Breaking out of the ASCII Straightjacket. *ACM SIGPlan Notices*, 28(2):61–68, 1993.
- [2] ACM/IEEE-CS Curriculum Task Force. *Computing Curricula 1991*. ACM Press, New York, 1991.
- [3] A. Alderson, M.F. Bott, and M.E. Falla. An overview of the ECLIPSE Project. In John McDermid, editor, *Integrated project support environments*, pages 100–113. Peter Peregrinus Ltd., 1985.
- [4] D. Appleby. Comparative review of books on programming languages. *Computing Reviews*, 28(11):569–574, November 1987.
- [5] Doris Appleby. *Programming Languages: Paradigm and Practice*. McGraw-Hill, New York, 1991.
- [6] Mouloud Arab. Enhancing Program Comprehension: FORMATTING and DOCUMENTING. *SIGPLAN Notices*, 27(2):37–46, February 1992.
- [7] Mouloud Arab. Tool for Making Programs More Readable. *SIGCSE Bulletin*, 23(3):31–35, September 1991.
- [8] Michael Armistead and John Burnham. HP C++/SoftBench: A Development Environment for C++. *Journal of Object-Oriented Programming*, 3(4):82–85, November/December 1990.
- [9] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. ACM Press/Addison Wesley, Reading, MA, 1990.
- [10] Henri E. Bal and Dick Grune. *Programming Language Essentials*. Addison-Wesley, Wokingham, England, 1994.
- [11] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise. A Framework for Event-Based Software Integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, October 1996.
- [12] David R. Barstow and Howard E. Shrobe. From Interactive to Intelligent Programming Environments. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 558–570. McGraw Hill Book Company, 1984.

- [13] Richard J. Beach. Experience with the Cedar Programming Environment for Computer Graphics Research. Technical Report CSL-84-6, XEROX PARC, Palo Alto, CA, July 1985.
- [14] Robert C. Bethke. The SoftBench Static Analysis Database. *Hewlett-Packard Journal*, 48(1):16-18, February 1997.
- [15] Daniel G. Bobrow. If PROLOG is the answer, what is the question? In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pages 138-148, Tokyo, Japan, November 1984.
- [16] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. An Overview of PCTE and PCTE++. In *Proceedings of the ACM SIG-SOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 248-257, Boston, MA, 1988.
- [17] Alan W. Brown. Control Integration Through Message Passing in a Software Development Environment. Technical Report CMU/SEI-92-TR-35, Software Engineering Institute, Carnegie-Mellon University, December 1992.
- [18] Alan W. Brown, editor. *Integrated Project Support Environments: The Aspect Project*. Academic Press, London, 1991.
- [19] Alan W. Brown, Anthony N. Earl, and John A. McDermid. *Software Engineering Environments: Automated Support for Software Engineering*. McGraw-Hill, London, 1992.
- [20] Alan W. Brown, Peter H. Feiler, and Kurt C. Wallnau. Past and Future Models of CASE Integration. In *Fifth International Workshop on Computer-Aided Software Engineering*, pages 36-45, 1992.
- [21] Marc H. Brown. *Algorithm Animation*. ACM Distinguished Dissertation. MIT Press, Cambridge, MA, 1988.
- [22] Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-view Editing. Technical Report 75, Digital Systems Research Center, February 1992.
- [23] Timothy A. Budd. Avoiding Backtracking by Capturing the Future. Working document, March 1991.
- [24] Timothy A. Budd. Blending Imperative and Relational Programming. *IEEE Software*, 8(1):58-65, 1991.
- [25] Timothy A. Budd. *Classic Data Structures in C++*. Addison-Wesley, Reading, Massachusetts, 1994.

- [26] Timothy A. Budd. Data Structures in LEDA. Technical Report 89-60-17, Oregon State University, 1989.
- [27] Timothy A. Budd. Functional Programming in an Object-Oriented Language. Technical Report 89-60-16, Oregon State University, 1989.
- [28] Timothy A. Budd. LEDA: A Blending of Imperative and Relational Programming. Technical Report 89-60-7, Oregon State University, 1989.
- [29] Timothy Budd. *A Little Smalltalk*. Addison-Wesley, Reading, Massachusetts, 1987.
- [30] Timothy A. Budd. Low Cost First Class Functions. Technical Report 89-60-12, Oregon State University, June 1989.
- [31] Timothy A. Budd. The Multi-Paradigm Programming Language LEDA. Working document, September 1989.
- [32] Timothy A. Budd. Multiparadigm Data Structures in Leda. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 165-173, Oakland, California, April 1992.
- [33] Timothy A. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley, Reading, MA, 1995.
- [34] Timothy A. Budd. Sharing and First Class Functions in Object-Oriented Languages. Working document, March 1991.
- [35] Timothy Budd. Teaching Multiple Paradigms. Working document, March 1994.
- [36] Timothy A. Budd, Timothy P. Justice, and Rajeev K. Pandey. General-Purpose Multiparadigm Programming Languages: An Enabling Technology for Constructing Complex Systems. Technical Report 95-60-04, Oregon State University, May 1995.
- [37] Timothy A. Budd, Timothy P. Justice, and Rajeev K. Pandey. General-Purpose Multiparadigm Programming Languages: An Enabling Technology for Constructing Complex Systems. In *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems*, pages 334-337, Fort Lauderdale, FL, November 1995.
- [38] Timothy A. Budd and Rajeev K. Pandey. Never Mind *the* Paradigm, What About Multiparadigm Languages? *SIGCSE Bulletin*, 27(2):25-30,40, June 1995.

- [39] Timothy Budd and Jim Shur. Foundations Toward a Multiparadigm Programming Methodology. November 1991.
- [40] Timothy A. Budd and Nabil M. Zamel. Integrating Constraints into a Multiparadigm Language. Technical Report 93-60-22, Oregon State University, December 1993.
- [41] John N. Buxton and Larry E. Druffel. Rationale for STONEMAN. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 535–545. McGraw Hill Book Company, 1984.
- [42] CDE Documentation Group. *Common Desktop Environment 1.0: Application Builder User's Guide*. Addison-Wesley, Reading, MA, 1995.
- [43] CDE Documentation Group. *Common Desktop Environment 1.0: Tooltalk Messaging Overview*. Addison-Wesley, Reading, MA, 1995.
- [44] Martin R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.
- [45] Cheryl Carmichael. COBOL SoftBench: An Open Integrated CASE Environment. *Hewlett-Packard Journal*, 46(3):82–88, June 1995.
- [46] Vinoo Cherian. Implementation of First Class Functions and Type Checking for a Multiparadigm Language. Master's thesis, Oregon State University, May 1991.
- [47] Elliot Chikofsky, editor. *Computer-Aided Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, second edition, 1993.
- [48] Koen Claessen, Ton Vullingsh, and Erik Meijer. Structuring Graphical Paradigms in TkGofer. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 251–262, Amsterdam, The Netherlands, June 1997.
- [49] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, third, revised and extended edition, 1987.
- [50] Michael Cohen. Blush and Zebrackets: Large- and Small-Scale Typographical Representation of Nested Associativity. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 264–266, Seattle, WA, September 1992.
- [51] Joseph J. Courant. SoftBench Message Connector: Customizing Software Development Tool Interactions. *Hewlett-Packard Journal*, 45(3):34–39, June 1994.

- [52] Ajantha Dahanayake and Gert Florijn. Evaluation of Object Oriented DataBase Support for Software Engineering Environments. In *Proceedings of the 7th Conference on Software Engineering Environments*, pages 11–20, Noordwijkerhout, The Netherlands, April 1995.
- [53] Susan A. Dart, Robert J. Ellison Peter H. Feiler, and A. Nico Habermann. Software Development Environments. *IEEE Computer*, 20(11):18–28, November 1987.
- [54] Antony J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, Cambridge, UK, 1992.
- [55] Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz. Viewing a Programming Environment as a Single Tool. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 49–56, Pittsburgh, PA, 1984.
- [56] T.A. Dolotta, R.C. Haight, and J.R. Mashey. UNIX Time-Sharing System: The Programmer’s Workbench. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 353–369. McGraw Hill Book Company, 1984.
- [57] Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang. Programming Environments Based on Structured Editors: The MENTOR Experience. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 128–140. McGraw Hill Book Company, 1984.
- [58] Timothy J. Duesing and John R. Diamant. CodeAdvisor: Rule Based C++ Defect Detection Using a Static Database. *Hewlett-Packard Journal*, 48(1):19–21, February 1997.
- [59] Carine Fédèle, Michael Gautero and Olivier Lecarme. Meta-compilation of the functional aspects of a multi-paradigm language. November 1992.
- [60] C.N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel L. Stock. The poe language-based editor project. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 21–29, Pittsburgh, PA, May 1984.
- [61] Robert W. Floyd. The Paradigms of Programming. *Communications of the ACM*, 22(8):455–460, 1979.
- [62] Mary Jo Foley. Can We Talk? *SunExpert Magazine*, pages 50–57, January 1992.

- [63] Brian D. Fromme. HP Encapsulator: Bridging the Generation Gap. *Hewlett-Packard Journal*, 41(3):59–68, June 1990.
- [64] Alfonso Fuggetta. A Classification of CASE Technology. *IEEE Computer*, 26(12):25–38, December 1993.
- [65] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26, November 1995.
- [66] Colin Gerety. A New Generation of Software Development Tools. *Hewlett-Packard Journal*, 41(3):48–58, June 1990.
- [67] Carlo Ghezzi. Modern non-conventional programming language concepts. In John A. McDermid, editor, *Software Engineer's Reference Book*, pages 44/1–44/16. CRC Press, Inc., Boca Raton, Florida, 1993.
- [68] Bob Glickstein. *Writing GNU Emacs Extensions*. O'Reilly & Associates, Cambridge, MA, 1997.
- [69] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.
- [70] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.
- [71] Dennis R. Goldenson and Bing Jyun Wang. Use of Structure Editing Tools by Novice Programmers. In *Fourth Workshop on Empirical Studies of Programmers*, pages 99–120, New Brunswick, NJ, 1991.
- [72] Stephen Jay Gould. In the Mind of the Beholder. *Natural History*, 103(2):14–23, February 1994.
- [73] Judith E. Grass. Cdiff: A Syntax Directed Differencer for C++ Programs. In *Proceedings of the 1992 USENIX C++ Technical Conference*, pages 181–193, 1992.
- [74] Judith E. Grass. Object-Oriented Design Archaeology with CIA++. *Computing Systems*, 5(1):5–68, 1992.
- [75] Judith E. Grass and Yih-Farn Chen. The C++ Information Abstractor. In *Proceedings of the 1990 USENIX C++ Conference*, pages 265–277, 1990.
- [76] A. Nico Habermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, 1986.

- [77] Brent Hailpern. Multiparadigm Languages and Environments. *IEEE Software*, 3(1):6–9, January 1986.
- [78] Wilfred J. Hansen. User Engineering Principles for Interactive Systems. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 217–231. McGraw Hill Book Company, 1984.
- [79] Mark Harrison. *Tcl/Tk Tools*. O'Reilly & Associates, Cambridge, MA, 1997.
- [80] Richard O. Hart and Glenn Lupton. DEC FUSE: Building a Graphical Software Development Environment from UNIX Tools. *Digital Technical Journal*, 7(2):5–19, 1995.
- [81] Jennifer G. Harvey and Chris D. Marlin. Comparing Inter-Tool Communication in Control-Centered Tool Integration Frameworks. In *Proceedings of the 8th Conference on Software Engineering Environments*, pages 67–81, Cottbus, Germany, April 1997.
- [82] Robert Hawley, editor. *Artificial Intelligence Programming Environments*. Ellis Horwood Ltd., West Sussex, England, 1987.
- [83] Jürgen Herrman and Markus Witthaut. LEDA - A Learning Apprentice System that Acquires Design Plans for High-Level Synthesis of Integrated Circuits. In *CompEuro 1992 Proceedings*, pages 430–435, May 1992.
- [84] Hewlett Packard. *The Ultimate Guide to the vi and ex Text Editors*. Benjamin/Cummings, Redwood City, CA, 1990.
- [85] Jurgen Heymann. A 100% Portable Inline Debugger. *ACM SIGPLAN Notices*, 28(9):39–46, September 1993.
- [86] Christopher John Hogger. *Introduction to Logic Programming*. Academic Press, Inc., Orlando, Florida, 1984.
- [87] Scott Hudson. CUP LALR Parser Generator for Java User's Manual. <http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>, March 1998.
- [88] Arun K. Iyengar, Thaddeus S. Grzesik, Valerie J. Ho-Gibson, Tracy A. Hoover, and John R. Vasta. An Event-Based, Retargetable Debugger. *Hewlett-Packard Journal*, 45(6):33–43, December 1994.
- [89] Michael A. Jenkins, Janice I. Glasgow, and Carl D. McCrosky. Programming styles in Nial. *IEEE Software*, 3(1):46–55, January 1986.

- [90] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, New York, third edition, 1985.
- [91] Eric F. Johnson. *Cross-Platform Perl*. M & T Books, New York, 1996.
- [92] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, 1993.
- [93] Timothy P. Justice. *Applicability of Multiparadigm Programming to Compiler Construction Tools*. PhD thesis, Oregon State University, May 1995. Accepted PhD Proposal.
- [94] Timothy P. Justice, Rajeev K. Pandey, and Timothy A. Budd. Compiler Implementation in the Multiparadigm Language Leda. Technical Report 93-60-20, Oregon State University, December 1993.
- [95] Timothy P. Justice, Rajeev K. Pandey, and Timothy A. Budd. A Multiparadigm Approach to Compiler Construction. *SIGPLAN Notices*, 29(9):29–37, September 1994.
- [96] R. Kadia. Issues Encountered in Building a Flexible Software Development Environment: Lessons From the Arcadia Project. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 169–180, Tyson’s Corner, VA, December 1992.
- [97] Gail E. Kaiser and David Garlan. Synthesizing Programming Environments From Reusable Features. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability Volume II: Applications and Experience*, pages 35–55. Addison-Wesley, Reading, MA, 1989.
- [98] Alan Kay. Microelectronics and the Personal Computer. *Scientific American*, 237(3):230–244, September 1977.
- [99] Brian W. Kernighan and John R. Mashey. The UNIX Programming Environment. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 175–197. McGraw Hill Book Company, 1984.
- [100] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.
- [101] K.N. King. The Evolution of the Programming Languages Course. *SIGCSE Bulletin*, 24(1):213–219, March 1992.

- [102] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson, editors. *Object-Oriented Environments: The Mjølnir Approach*. Prentice-Hall, New York, 1994.
- [103] Donald Knuth. *Literate Programming*. Center for the Study of Language and Information Lecture Notes Number 27, Palo Alto, CA, 1992.
- [104] Donald Knuth. *The T_EXbook: A Complete Guide to Computer Typesetting with T_EX*. Addison-Wesley, Reading, MA, 1984.
- [105] Donald Knuth and Silvio Levy. *The CWEB System of Structured Documentation Version 3.0*. Addison-Wesley, Reading, MA, 1994.
- [106] Charles D. Knutson. *Pattern Systems for Multiparadigm Analysis and Design*. PhD thesis, Oregon State University, 1998.
- [107] Michael Kölling and John Rosenberg. An object-oriented program development environment for the first programming course. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, pages 83–87, Anaheim, CA, March 1996.
- [108] Timothy Koschmann and Martha Walton Evens. Bridging the gap between object-oriented and logic programming. *IEEE Software*, 5(4):36–42, July 1988.
- [109] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. The University of Chicago Press, Chicago, IL, second edition edition, 1970.
- [110] Leslie Lamport. *L^AT_EX: A Document Preparation System*. Addison-Wesley, Reading, MA, second edition, 1994.
- [111] D. B. Leblang and Jr. Chase, R.P. Computer-aided software engineering in a distributed workstation environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104–112, Pittsburgh, PA, May 1984.
- [112] Peter Lempp and Rudolf Lauber. What Productivity to Expect from a CASE Environment: Results of a User Survey. In Elliot Chikofsky, editor, *Computer-Aided Software Engineering*, pages 147–153. IEEE Computer Society Press, Los Alamitos, CA, 1993.
- [113] Don Libes. *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, Cambridge, MA, 1995.
- [114] Deborah A. Lienhart. SoftBench 5.0: The Evolution of an Integrated Software Development Environment. *Hewlett-Packard Journal*, 48(1):6–11, February 1997.

- [115] Mark A. Linton. The Evolution of Dbx. In *Proceedings of the Summer 1990 USENIX Conference*, pages 211–220, Anaheim, CA, June 1990.
- [116] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.
- [117] Paul A. Luker. Never Mind the Language, What About the Paradigm? In *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, pages 252–256, Louisville, KY, February 1989.
- [118] C. Lüth and S. Westmeier and B. Wolff. sml.tk: Functional programming for graphical user interfaces. Technical Report 8/96, FB3, Universität Bremen, 1996.
- [119] Bruce J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, Reading, Massachusetts, 1990.
- [120] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation and Implementation*. Holt, Rinehart and Winston, New York, second edition, 1987.
- [121] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, and Karl Tombre. *Object-Oriented Languages*, volume 34 of *A.P.I.C. Series*. Academic Press Inc., San Diego, California, United States edition, 1991.
- [122] John McDermid, editor. *Integrated project support environments*. Peter Peregrinus Ltd., London, 1985. (Proceedings of the conference on Integrated Project Support Environments (IPSEs), University of York, April 10–12, 1985).
- [123] Raul Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments*. PhD thesis, Carnegie-Mellon University, 1982.
- [124] Raul Medina-Mora and Peter H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, SE-7(5):472–482, 1981.
- [125] Kurt Mehlhorn and Stefan Näher. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, January 1995.
- [126] Yoshiaki Mima. A Visual Programming Environment for Programming by Example Abstraction. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 132–137, Kobe, Japan, October 1991.
- [127] Robert Munck, Patricia Oberndorf, Erhard Ploedereder, and Richard Thall. An Overview of DOD-STD-1838A (proposed), A Common APSE Interface Set,

- Revision A. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 235–247, Boston, MA, 1988.
- [128] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.
- [129] NIST IEEE Working Group and the ECMA TC33 Task Group on the Reference Model. Reference Model for Frameworks of Software Engineering Environments. Technical Report ECMA TR/55 NIST Special Publication 500-211, European Computer Manufacturers Association (ECMA) and National Institute of Standards (NIST), August 1993.
- [130] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. Dalek: A GNU, Improved Programmable Debugger. In *Proceedings of the Summer 1990 USENIX Conference*, pages 221–231, Anaheim, CA, June 1990.
- [131] Paul W. Oman and Curtis R. Cook. Typographic Style is More than Cosmetic. *Communications of the ACM*, 33(5):506–520, May 1990.
- [132] Lay-Peng Ong. Version modeling using production rules in the postgres dbms. Technical Report UCB/ERL M91/51, University of California, Berkeley, CA, June 1991.
- [133] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.
- [134] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972.
- [135] John K. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer*, pages 23–30, March 1998.
- [136] John K. Ousterhout. Tcl: An Embeddable Command Language. In *Proceedings of the Winter 1990 USENIX Conference*, pages 133–146, Washington, D.C., January 1990.
- [137] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.
- [138] John K. Ousterhout. An X11 Toolkit Based on the Tcl Language. In *Proceedings of the Winter 1991 USENIX Conference*, pages 105–115, Dallas, TX, January 1991.
- [139] B.B. Owens. Comparative review of books on programming languages. *Computing Reviews*, 33(1):49–56, January 1992.

- [140] Rajeev Pandey, Wolfgang Pesch, Jim Shur, and Masami Takikawa. A Revised **Leda** Language Definition. Technical Report 93-60-02, Oregon State University, January 1993.
- [141] Maria H. Penedo. Towards understanding Software Engineering Environments. Technical Report IMPSEE-TRW-93-003, TRW, Redondo Beach, CA, August 1993.
- [142] Dewayne E. Perry and Gail E. Kaiser. Models of Software Development Environments. *IEEE Transactions on Software Engineering*, 17(3):283–295, March 1991.
- [143] Wolfgang Pesch. Implementing Logic In **Leda**. Technical Report 91-60-10, Oregon State University, May 1991.
- [144] Wolfgang Pesch and Jim Shur. A **Leda** Language Definition. Technical Report 91-60-09, Oregon State University, September 1991.
- [145] Marian Petre and R. Winder. On Languages, Models and Programming Styles. *The Computer Journal*, 33(2):173–180, April 1990.
- [146] John Placer. The Promise of Multiparadigm Languages as Pedagogical Tools. In *Proceedings of the 1993 ACM Computer Science Conference*, pages 81–86, Indianapolis, IN, February 1993.
- [147] Dick Pountain. Functional programming comes of age. *Byte*, 19(8):183–184, August 1994.
- [148] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, New York, third edition, 1992.
- [149] P. Paolo Puncello, Piero Torrigiani, Francesco Pietri, Riccardo Burlon, Bruno Cardile, and Mirella Conti. ASPIS: A Knowledge-Based CASE Environment. *IEEE Software*, 5(2):58–65, April 1988.
- [150] Steven P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57–66, 1990.
- [151] Steven P. Reiss. *THE FIELD PROGRAMMING ENVIRONMENT: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, Boston, MA, 1995.
- [152] Steven P. Reiss. Graphical Program Development with PECAN Program Development Systems. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 30–41, Pittsburgh, PA, 1984.

- [153] Steven P. Reiss. Interacting with the FIELD Environment. *Software-Practice and Experience*, 20(1):89–115, June 1990.
- [154] Steven P. Reiss. PECAN: Program Development Systems that Support Multiple Views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, March 1985.
- [155] Steven P. Reiss. Software tools and environments. In Jr. Alan B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2419–2439. CRC Press, 1997.
- [156] Steven P. Reiss and Scott Meyers. FIELD Support for C++. In *Proceedings of the 1990 USENIX C++ Conference*, pages 293–299, 1990.
- [157] Steven P. Reiss and John T. Stasko. The Brown Workstation Environment: A user interface design toolkit. Technical Report CS-89-34, Department of Computer Science, Brown University, June 1989.
- [158] Thomas W. Reps. *Generating Language-Based Environments*. ACM Doctoral Dissertation Series. The MIT Press, Cambridge, MA, 1984.
- [159] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, Pittsburgh, PA, May 1984.
- [160] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. Addison-Wesley/ACM Press, Cambridge, MA, 1990.
- [161] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. Wiley Computer Publishing, New York, 1996.
- [162] Martin Ruckert. Conservative Pretty Printing. *SIGPLAN Notices*, 32(2):39–44, February 1997.
- [163] D. Schefström. System Development Environments: Contemporary Concepts. In D. Schefström and G. van den Broek, editors, *Tool Integration: Environments and Frameworks*, pages 558–570. John Wiley & Sons, Chichester, U.K., 1993.
- [164] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.
- [165] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, Massachusetts, second edition, 1988.

- [166] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, Reading, MA, 1989.
- [167] Jim Shur. Implementing Leda: Objects And Classes. Technical Report 91-60-11, Oregon State University, August 1991.
- [168] R.A. Snowdon. An Introduction to the IPSE 2.5 Project. In Fred Long, editor, *Proceedings of the International Workshop on Environments*, Lecture Notes in Computer Science, pages 13–24, Chinon, France, September 1989. Springer-Verlag.
- [169] Ian Sommerville. *Software Engineering*. Addison-Wesley, Wokingham, England, third edition edition, 1989.
- [170] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, MA, 1987.
- [171] Scott Stanton. TclBlend: Blending Tcl and Java. *Dr. Dobb's Journal*, pages 50–54, 100–101, February 1998.
- [172] John Stasko. Animating Algorithms with XTANGO. *ACM SIGACT News*, 23(2):67–71, Spring 1992.
- [173] John Stasko. Tango: A Framework and System for Algorithm Animation. *IEEE Computer*, 23(9):27–39, September 1990.
- [174] John T. Stasko. Using Student-Built Algorithm Animations As Learning Aids. Technical Report GIT-GVU-96-19, Georgia Institute of Technology, 1996.
- [175] Guy L. Steele, Jr. *Common LISP: The Language*. Digital Press, Bedford, MA, 1984.
- [176] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. Integrating Access-Oriented Programming into a Multiparadigm Environment. *IEEE Software*, 3(1):10–18, January 1986.
- [177] Michael Stonebraker and Lawrence A. Rowe. The postgres papers. Technical Report UCB/ERL M86/85, University of California, Berkeley, CA, June 1987.
- [178] Tom Strelch. The Software Life Cycle Support Environment (SLCSE): A Computer Based Framework for Developing Software Systems. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 35–44, Boston, MA, 1988.
- [179] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1991.

- [180] Kevin J. Sullivan. *Mediators: Easing the Design and Evolution of Integrated Systems*. PhD thesis, University of Washington, 1994.
- [181] Daniel Swinehart, Polle Zellweger, Richard Beach, and Robert Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Language and Systems*, 8(4):419–490, October 1986.
- [182] Masami Takikawa. CLEDA – LEDA With Constraint Logic Programming. Technical Report 93-60-03, Oregon State University, March 1993.
- [183] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia Environment Architecture. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–13, Boston, MA, 1988.
- [184] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM*, 24(9):563–573, 1981.
- [185] Warren Teitelman. A Tour Through Cedar. *IEEE Software*, 1(2):44–73, April 1984.
- [186] Warren Teitelman and Larry Masinter. The Interlisp Programming Environment. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, pages 83–96. McGraw Hill Book Company, 1984.
- [187] Ian Thomas and Brian A. Nejme. Definitions of Tool Integration for Environments. *IEEE Software*, 9(2):29–35, 1992.
- [188] Jeffrey M. Voas. Certifying Off-the-Shelf Software Components. *IEEE Computer*, 31(6):53–59, June 1998.
- [189] Lois Wakeman and Jonathan Jowett. *PCTE: The Standard For Open Repositories*. Prentice-Hall, Hertfordshire, UK, 1993.
- [190] Anthony I. Wasserman. The Ecology of Software Development Environments. In Anthony I. Wasserman, editor, *Tutorial: Software Development Environments*, pages 47–52. IEEE Computer Society, 1981.
- [191] Anthony I. Wasserman. Tool Integration in Software Engineering Environments. In *Software Engineering Environments: Proceedings of the International Workshop on Environments*, pages 137–149, Chinon, France, September 1989.

- [192] Anthony I. Wasserman and Peter A. Pircher. The Open Architecture of the Software Through Pictures Environment. In Marvin V. Zelkowitz, editor, *Proceedings of the University of Maryland Workshop on Requirements for a Software Engineering Environment*, pages 143–157, College Park, MD, May 1986. Ablex.
- [193] Mark Weiser, David Notkin, Bertrand Meyer, Mark Green, Glenn Pearson, David Stotts, Tony Wasserman, Alexander Wolf, and Rick Furuta. Landscaping for Programming Environments. In Marvin V. Zelkowitz, editor, *Proceedings of the University of Maryland Workshop on Requirements for a Software Engineering Environment*, pages 27–57, College Park, MD, May 1986. Ablex.
- [194] Mark B. Wells and Barry L. Kurtz. Teaching Multiple Programming Paradigms: A Proposal for a Paradigm-General Pseudocode. In *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, pages 246–251, Louisville, KY, February 1989.
- [195] Åke Wikström. *Functional Programming using Standard ML*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.
- [196] Stephen A. Williams. Using SoftBench to Integrate Heterogeneous Software Development Environments. *Hewlett-Packard Journal*, 48(1):22–27, February 1997.
- [197] Julie B. Wilson. The C++ SoftBench Class Editor. *Hewlett-Packard Journal*, 48(1):12–15, February 1997.
- [198] Douglas A. Young. *The X Window System: Programming and Applications with Xt*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, second edition, 1994.
- [199] Nabil M. Zamel. *Electra: Integrating Constraints, Condition-Based Dispatching, and Feature Exclusion into the Multiparadigm Language Leda*. PhD thesis, Oregon State University, 1995.
- [200] Nabil M. Zamel and Timothy A. Budd. Integrating Constraints into a Multiparadigm Language. In *Proceedings of InfoScience '93*, pages 402–409, Seoul, Korea, October 1993.
- [201] Donald A. Zaremba. Adding a Data Visualization Tool to DEC FUSE. *Digital Technical Journal*, 7(2):20–33, 1995.
- [202] Pamela Zave. A Compositional Approach to Multiparadigm Programming. *IEEE Software*, 6(5):6–9, September 1989.

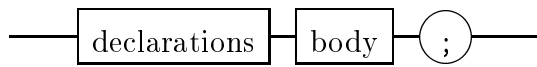
- [203] Polle Scott Zellweger. Interactive Source-Level Debugging of Optimized Programs. Technical Report CSL-84-5, XEROX PARC, Palo Alto, CA, May 1984.

APPENDICES

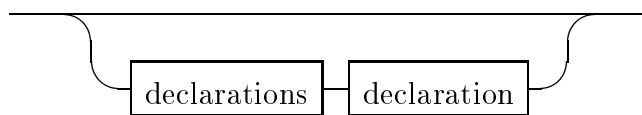
APPENDIX A. Leda Grammar

A.1. Overall Program Structure

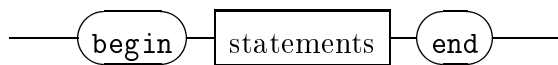
program



declarations

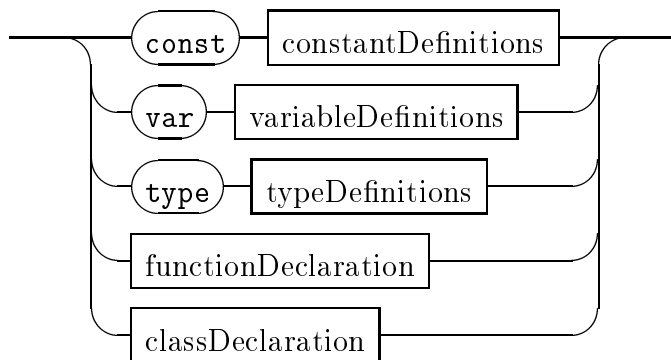


body

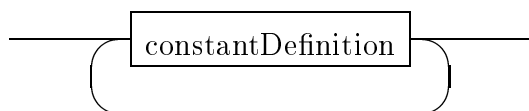


A.2. Declarations

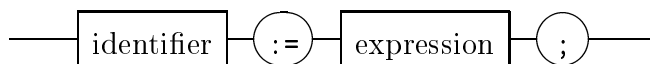
declaration



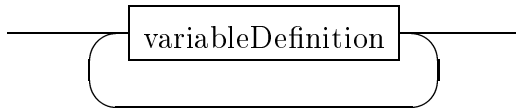
constantDefinitions



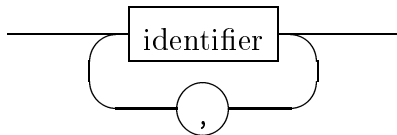
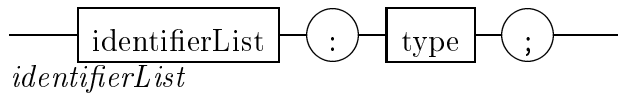
constantDefinition



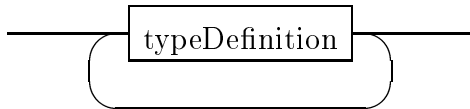
variableDefinitions



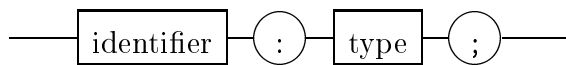
variableDefinition



typeDefinitions

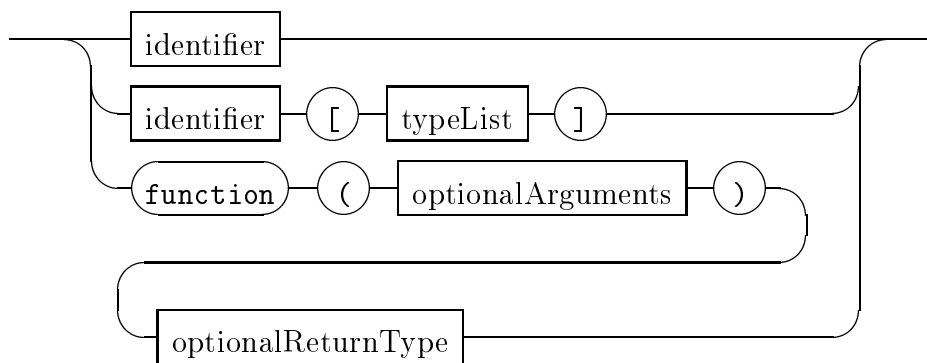


typeDefinition

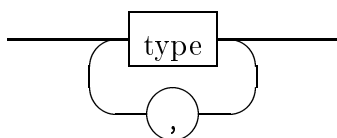


A.3. Types

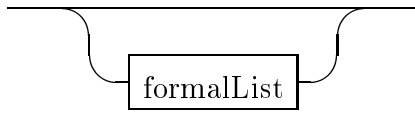
type



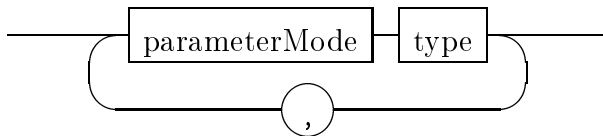
typeList



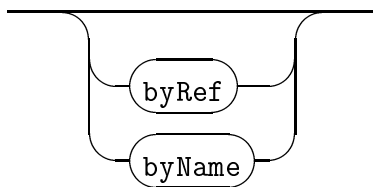
optionalArguments



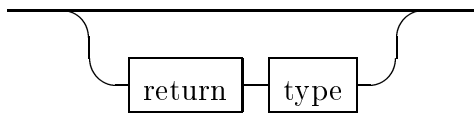
formalList



parameterMode

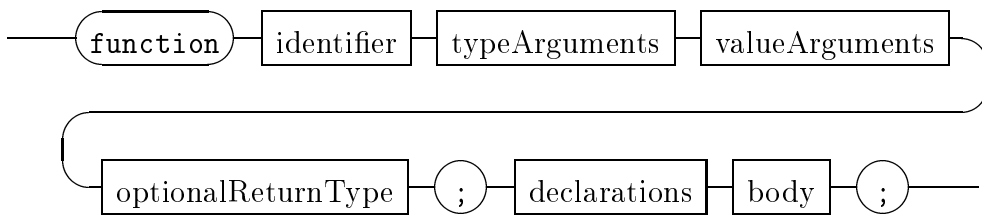


optionalReturnType

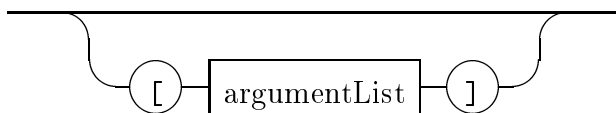


A.4. Function Declarations

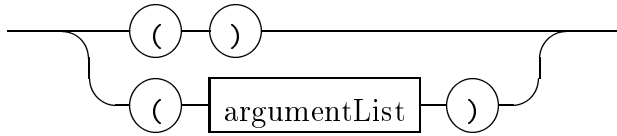
functionDeclaration



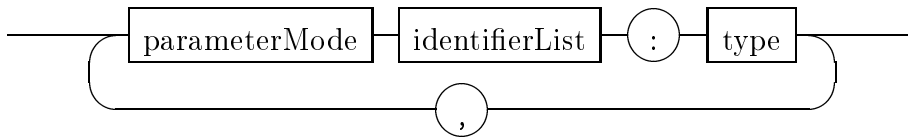
typeArguments



valueArguments

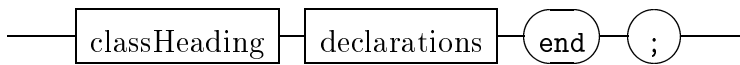


argumentList

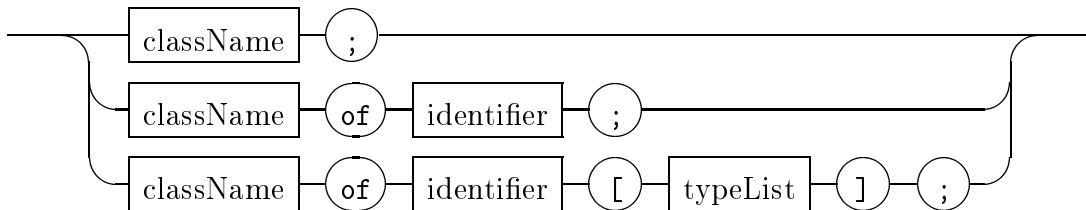


A.5. Class Declarations

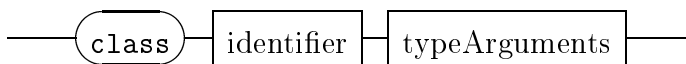
classDeclaration



classHeading

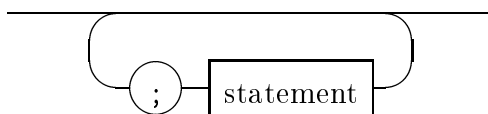


className

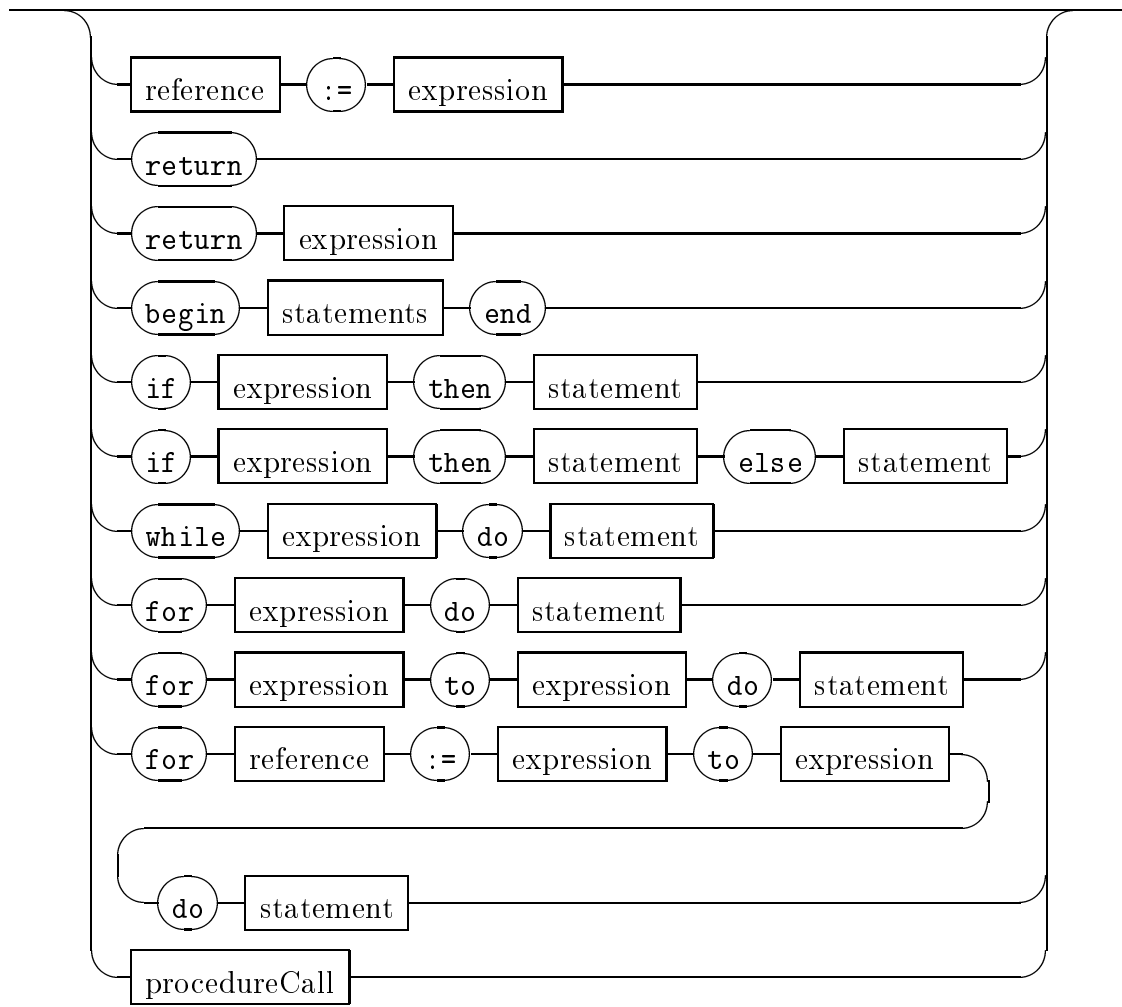


A.6. Statements

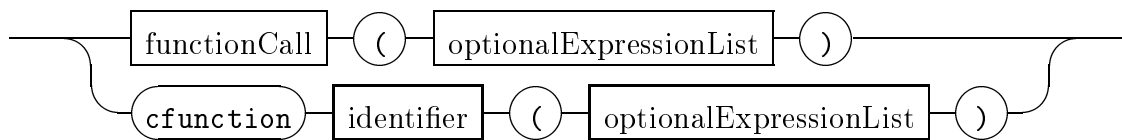
statements



statement



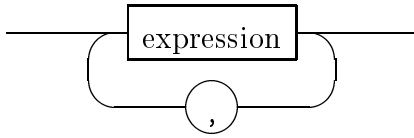
procedure Call



optionalExpressionList

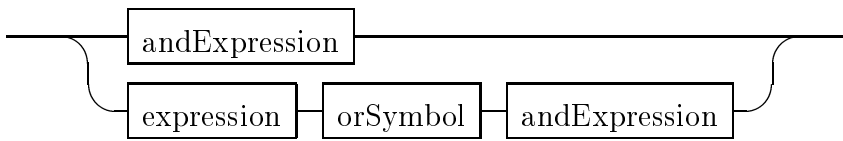


expressionList

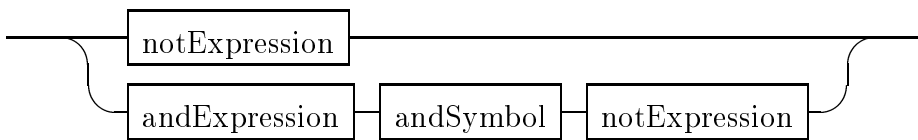


A.7. Expressions

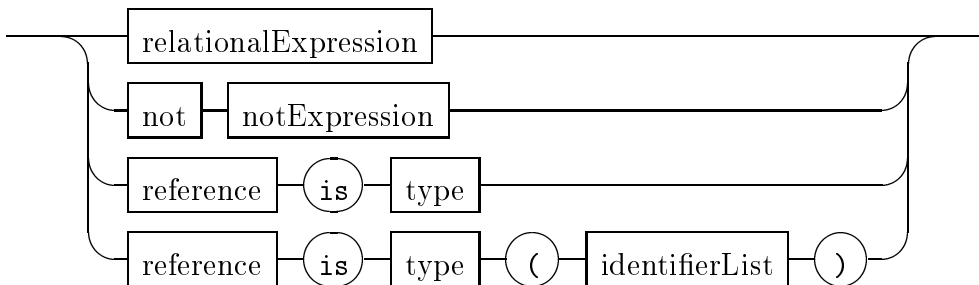
expression



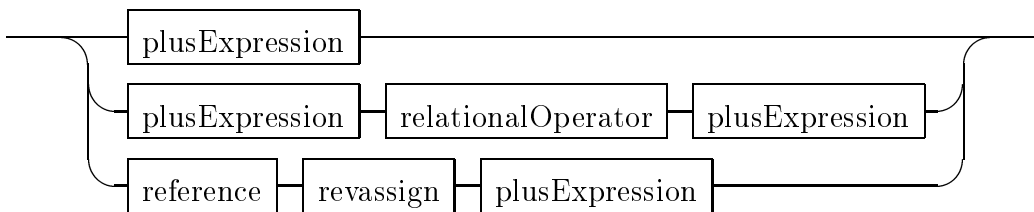
andExpression



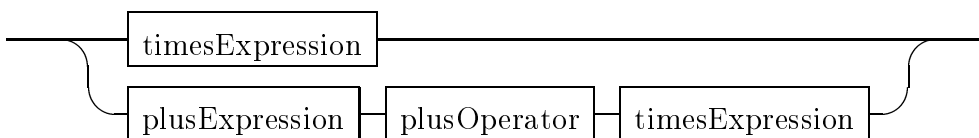
notExpression



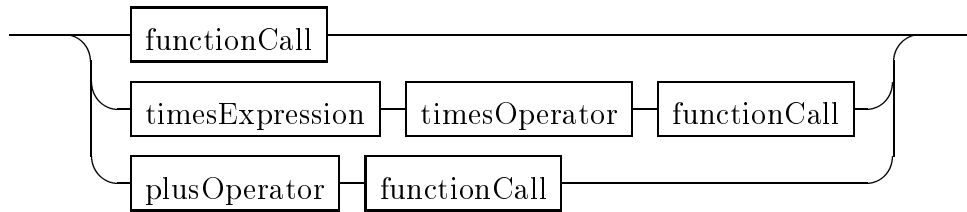
relationalExpression



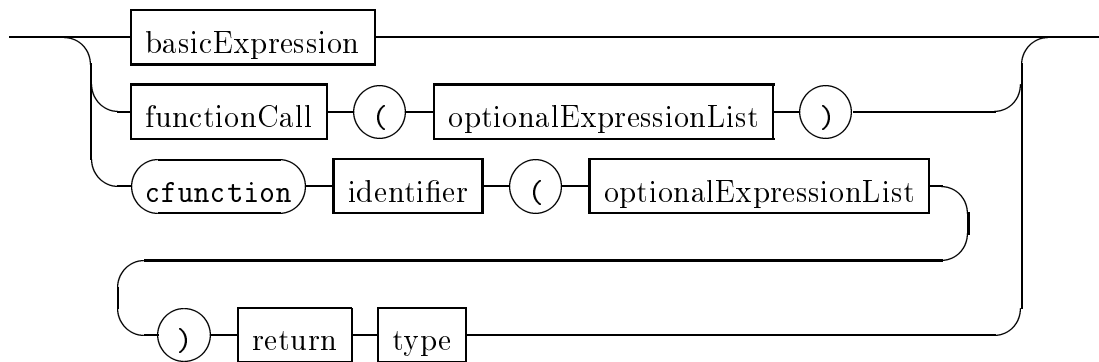
plusExpression



timesExpression



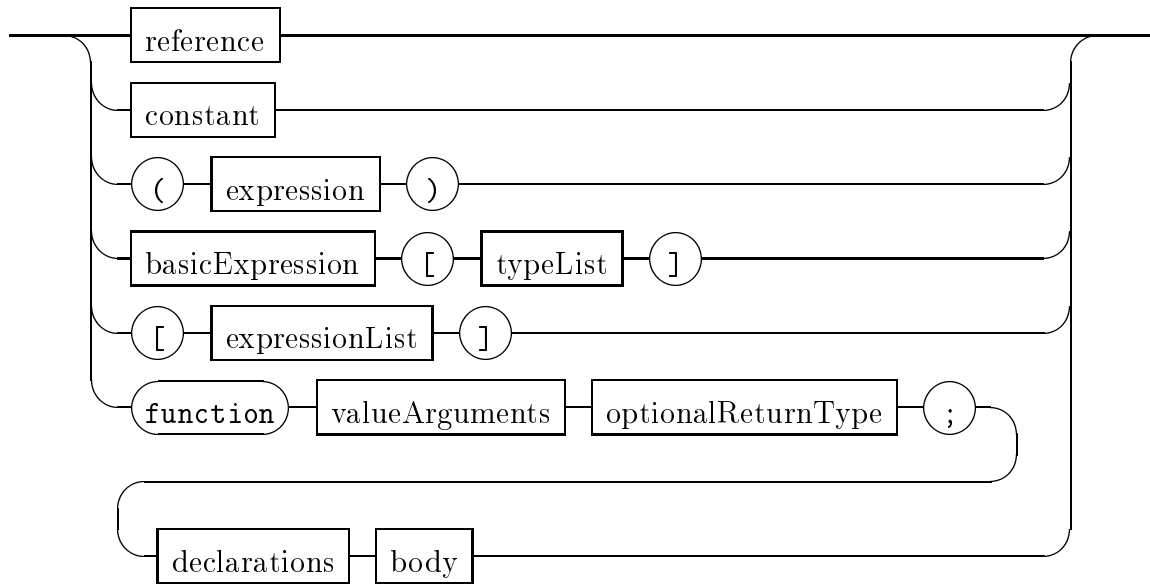
functionCall



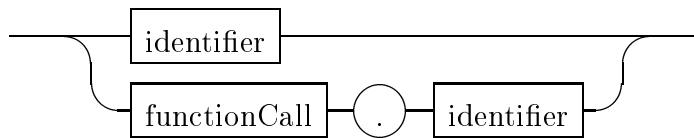
The *or* symbol is `|`. The *and* symbol is `&`. The six relational operators are `<`, `<=`, `=`, `<>`, `>=` and `>`, The two plus operators are `+` and `-`. The three times operators are `*`, `/` and `%`, the latter denoting remainder.

A.8. Basic Expression

basicExpression



reference



APPENDIX B. Leda Program to Simulate a Turing Machine

B.1. turing.led

```

{
  File: turing.led
  Description: example Turing machine simulator
  Source: "Foundations Toward a Multiparadigm Methodology"
    by Jim Shur
  Version: Leda-95 (converted from Leda-91 by Timothy Justice)
}

include "std.led";
include "square.led";
include "rwhead.led";
include "tm.led";
include "tmpal.led";
include "tmadd.led";

function NL();
begin
  print("\n");
end;

var
  t : TM;
  tape : Square;

begin
  "Test for palindrome:".print();
  NL();
  tape := (Square("0", NIL, NIL) + "0" + "1" + "0" + "1" + "0").getFirst();
  t := TM(0, RWHead(tape), palindrome);
  t.execute(true);
  NL();
  NL();
  NL();

```

```
    "Add two binary numbers:".print();  
    NL();  
    tape := (Square("1", NIL, NIL) + "0" + "1" + "1" +  
            "+" + "0" + "1" + "0" + "1").getFirst();  
    t := TM(0, RWHead(tape), add);  
    NL();  
end;
```

B.2. square.led

```
{
  File: square.led
  Description: Tape Square class for Turing machine simulator
  Source: "Foundations Toward a Multiparadigm Methodology"
    by Jim Shur
  Version: Leda-95 (converted from Leda-91 by Timothy Justice)
}
```

```
class Square;
  var
    value : string;
    left : Square;
    right : Square;

  function getValue()⇒string;
  begin
    return value;
  end;

  function putValue(c : string);
  begin
    value := c;
  end;

  function getLeft()⇒Square;
  begin
    if ~defined (left) then
      left := Square("-", NIL, self);
    return left;
  end;

  function getRight()⇒Square;
  begin
    if ~defined (right) then
      right := Square("-", NIL, self);
    return right;
  end;

  function getFirst()⇒Square;
  begin
```



```
    if defined (left) then
        return left.getFirst()
    else
        return self;
end;

function getPos()⇒integer;
begin
    if ~defined (left) then
        return 1
    else
        return 1 + left.getPos();
end;

function print();
begin
    value.print();
    if defined (right) then
        right.print();
end;

function plus(symbol : string)⇒Square;
var
    r : Square;
begin
    if defined (right) then
        return right.plus(symbol)
    else
        begin
            r := getRight();
            r.putValue(symbol);
            return r;
        end;
end;

end;
```

B.3. rwhead.led

```
{
  File: rwhead.led
  Description: Read/Write Head class for Turing machine simulator
  Source: "Foundations Toward a Multiparadigm Methodology" by Jim Shur
  Version: Leda-95 (converted from Leda-91 by Timothy Justice)
}
```

```
class RWHead;
  var
    curSquare : Square;

  function read()⇒string;
  begin
    return curSquare.getValue();
  end;

  function write(symbol : string);
  begin
    curSquare.putValue(symbol);
  end;

  function move(d : string);
  begin
    if d = "left" then
      curSquare := curSquare.getLeft()
    else
      if d = "right" then
        curSquare := curSquare.getRight();
      end;
    end;

  function print();
  var
    i : integer;
  begin
    curSquare.getFirst().print();
    "\n".print();
    for i := 1 to curSquare.getPos() - 1 do " ".print();
    "~".print();
  end;
end;
```

B.4. tm.led

```

{
  File: tm.led
  Description: Turing Machine class for Turing machine simulator
  Source: "Foundations Toward a Multiparadigm Methodology"
    by Jim Shur
  Version: Leda-95 (converted from Leda-91 by Timothy Justice)
}

class TM;
  var
    state : integer;
    head : RWHead;
    rules : function (byRef integer, byRef string, byRef integer,
                      byRef string, byRef string)⇒relation;

  function execute(trace : boolean);
    var
      newState : integer;
      symbol, newSymbol : string;
      direction : string;
    begin
      if trace then
        head.print();
        direction := "stay";
      while direction <> "HALT" do
        begin
          symbol := head.read();
          newState := NIL;
          newSymbol := NIL;
          direction := NIL;

          if rules(state, symbol, newState, newSymbol, direction) then
            ;
          head.write(newSymbol);
          head.move(direction);

          if trace then
            begin
              print("\t");
              print("(");

```

```
        print(state);
        print(", ");
        print(symbol);
        print(", ");
        print(newState);
        print(", ");
        print(newSymbol);
        print(", ");
        print(direction);
        print(")");
        print("\n");
        head.print();
    end;

    state := newState;
end;

end;
```

B.5. tmpal.led

```

{
  File: tmpal.led
  Description: Palindrome relation for Turing machine simulator
  Source: "Foundations Toward a Multiparadigm Methodology"
    by Jim Shur
  Version: Leda-95 (converted from Leda-91 by Timothy Justice)
}

function palindrome(byRef s1 : integer, byRef q1 : string,
                    byRef s2 : integer, byRef q2 : string, byRef d : string)⇒relation;

function rule(byRef x1 : integer, byRef y1 : string,
              byRef x2 : integer, byRef y2 : string, byRef z : string)⇒relation;

begin
  return unify[integer](s1, x1) & unify[string](q1, y1)
    & unify[integer](s2, x2) & unify[string](q2, y2)
    & unify[string](d, z);
end;

begin
  return rule(0, "0", 1, "-", "right")
    | rule(0, "1", 4, "-", "right")
    | rule(0, "-", -1, "-", "HALT")
    | rule(1, "0", 1, "0", "right")
    | rule(1, "1", 1, "1", "right")
    | rule(1, "-", 2, "-", "left")
    | rule(2, "0", 3, "-", "left")
    | rule(2, "1", -1, "1", "HALT")
    | rule(2, "-", -1, "-", "HALT")
    | rule(3, "0", 3, "0", "left")
    | rule(3, "1", 3, "1", "left")
    | rule(3, "-", 0, "-", "right")
    | rule(4, "0", 4, "0", "right")
    | rule(4, "1", 4, "1", "right")
    | rule(4, "-", 5, "-", "left")
    | rule(5, "0", -1, "0", "HALT")
    | rule(5, "1", 3, "-", "left")
    | rule(5, "-", -1, "-", "HALT");
end;

```

B.6. tmadd.led

```

{
  File: tmadd.led
  Description: Binary Addition relation for Turing machine simulator
  Source: "Foundations Toward a Multiparadigm Methodology"
         by Jim Shur
  Version: Leda-95 (converted from Leda-91 by Timothy Justice)
}

function add(byRef s1 : integer, byRef q1 : string,
            byRef s2 : integer, byRef q2 : string, byRef d : string)⇒relation;

function rule(byRef x1 : integer, byRef y1 : string,
             byRef x2 : integer, byRef y2 : string, byRef z : string)⇒relation;

begin
  return unify[integer](s1, x1) & unify[string](q1, y1)
         & unify[integer](s2, x2) & unify[string](q2, y2)
         & unify[string](d, z);
end;

begin
  return rule(0, "0", 0, "0", "right") { start, move right until }
         | rule(0, "1", 0, "1", "right") { "+" or mark }
         | rule(0, "+", 1, "+", "left")
         | rule(0, "i", 1, "i", "left")
         | rule(0, "o", 1, "o", "left")

         | rule(1, "0", 2, "o", "right") { test for 0 or 1 }
         | rule(1, "1", 3, "i", "right")

         | rule(2, "0", 2, "0", "right") { it was 0, right until }
         | rule(2, "1", 2, "1", "right") { blank or mark }
         | rule(2, "i", 2, "i", "right")
         | rule(2, "o", 2, "o", "right")
         | rule(2, "+", 20, "+", "right")

         | rule(20, "1", 20, "1", "right")
         | rule(20, "0", 20, "0", "right")
         | rule(20, "-", 4, "-", "left")
         | rule(20, "i", 4, "i", "left")

```

```

| rule(20, "o", 4, "o", "left")

| rule(3, "0", 3, "0", "right") { it was 1, right until }
| rule(3, "1", 3, "1", "right") { blank or mark }
| rule(3, "i", 3, "i", "right")
| rule(3, "o", 3, "o", "right")
| rule(3, "+", 30, "+", "right")

| rule(30, "1", 30, "1", "right")
| rule(30, "0", 30, "0", "right")
| rule(30, "-", 5, "-", "left")
| rule(30, "i", 5, "i", "left")
| rule(30, "o", 5, "o", "left")

| rule(4, "0", 6, "o", "left") { 0 + 0, no carry }
| rule(4, "1", 6, "i", "left") { 0 + 1, no carry }

| rule(5, "0", 6, "i", "left") { 1 + 0, no carry }
| rule(5, "1", 7, "o", "left") { 1 + 1, no carry }

| rule(6, "0", 6, "0", "left") { left past + }
| rule(6, "1", 6, "1", "left") { left past + }
| rule(6, "+", 8, "+", "left")

| rule(7, "0", 7, "0", "left")
| rule(7, "1", 7, "1", "left")
| rule(7, "+", 9, "+", "left")

| rule(8, "o", 8, "o", "left") { left past marks }
| rule(8, "i", 8, "i", "left") { left past marks }
| rule(8, "0", 2, "o", "right")
| rule(8, "1", 3, "i", "right")
| rule(8, "-", 11, "-", "right")

| rule(9, "i", 9, "i", "left")
| rule(9, "o", 9, "o", "left")
| rule(9, "0", 12, "o", "right")
| rule(9, "1", 13, "i", "right")
| rule(9, "-", 10, "-", "right")

| rule(10, "i", 10, "-", "right") { done except for last carry }

```

```

| rule(10, "o", 10, "-", "right") { clear left operand }
| rule(10, "+", - 1, "i", "HALT")

| rule(11, "i", 11, "-", "right") { done clear left operand }
| rule(11, "o", 11, "-", "right")
| rule(11, "+", - 1, "-", "HALT")

{ Carry is set }
| rule(12, "0", 12, "0", "right") { it was 0, right until }
| rule(12, "1", 12, "1", "right") { blank or mark }
| rule(12, "i", 12, "i", "right")
| rule(12, "o", 12, "o", "right")
| rule(12, "+", 120, "+", "right")

| rule(120, "0", 120, "0", "right")
| rule(120, "1", 120, "1", "right")
| rule(120, "-", 14, "-", "left")
| rule(120, "i", 14, "i", "left")
| rule(120, "o", 14, "o", "left")

| rule(13, "0", 13, "0", "right") { it was 1, right until }
| rule(13, "1", 13, "1", "right") { blank or mark }
| rule(13, "i", 13, "i", "right")
| rule(13, "o", 13, "o", "right")
| rule(13, "+", 130, "+", "right")

| rule(130, "0", 130, "0", "right")
| rule(130, "1", 130, "1", "right")
| rule(130, "-", 15, "-", "left")
| rule(130, "-", 15, "-", "left")
| rule(130, "i", 15, "i", "left")
| rule(130, "o", 15, "o", "left")

| rule(14, "0", 6, "i", "left") { 0 + 0, no carry }
| rule(14, "1", 7, "o", "left") { 0 + 1, no carry }

| rule(15, "0", 7, "o", "left") { 1 + 0, no carry }
| rule(15, "1", 7, "i", "left") { 1 + 1, no carry }

;
end;
```