CHAMELEON: AN EMBEDDED LANGUAGE PERSPECTIVE FOR MULTIPLE COMPUTER LANGUAGE PROGRAMMING

by

Terrence Monroe Brannon

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering and Computer Science

Lehigh University

1993

This thesis is accepted in partial fulfillment of the requirements for the degree of Master of Science.

(Date)

Samuel Gulden

Chairman of the EECS Department

Preface

I entered Lehigh University in August 1991 fresh from an intern at Caltech in parallel computing. After only a week at Lehigh, I chose Dr. Meghanad Wagh as my advisor as I was positive I wanted to pursue a thesis on parallel computing. I began my first summer intending to develop a mechanism for transparent remote procedure calls. As I developed the code to parse C header files in Python, I kept noticing I was writing functions that already existed in Emacs Lisp. After about two or three days of this repitition, I decided to remedy the solution. I wrote some code that would allow Emacs Lisp and Python to communicate using Emacs Lisp's built-in asynchronous interprocess communication primitives. The data conversion routines were very simple and only handled the types that I knew each language would produce.

Then I saw Guido von Rossom's STDWIN package which allowed the same C library calls to develop graphic user interfaces for a wide variety of windowing systems and architectures. The next step in my proselytization was running a Python demo program that allowed remote login and use of a language interpreter. I told Mukesh Sharma that it would be very interesting if individuals setup “language servers” for all of the popular interpreted languages so that a person could access the language if local use was impossible due to lack of disk space or required architecture.

From Python I moved to Tcl which had a powerful X-windows toolkit and an extension called Expect which allowed for a process to control both the stdin and stdout of a Unix process. I hacked an interface to the Unix `bc` calculator program and was enthralled at this extension of the Unix toolbox philosophy. I was beginning to see over and over that it is much better to link things together than to try to write everything.

About this time I began my informal paper *Survey of Quick Interpreted Languages* which gave a brief summary of the capabilities of the above-mentioned languages as well as Perl. Numerous people were very interested in the paper and thanked me for authoring it. However, this paper is constantly growing and probably never will stop growing. I have added one new language and several new programs to the survey.

There were two icings on the cake during my second year at Lehigh which made me certain that there was no perfect language. The first layer of icing was Sven Delmas' TkSteal, a piece of software that allowed Tcl to have Emacs and Postscript widgets. The second layer of icing was three programs I had to write for Dragana Brzakovic's pattern recognition class. These programs were quite rigorous. During the second program, I remember having written about the whole thing in Sather when I needed to calculate the inverse of a matrix. To my suprise, there was no inversion function in the Sather matrix class. I rewrote the entire program in Tcl and then hashed up an Expect interface to a Maple process and had Maple perform the inverse. I then parsed the results from Maple stdout and continued with the Tcl program.

The development of monolithic environments like Smalltalk with no functionality for interlanguage communication and the common occurrence of languages calling their C callout mechanisms “foreign interfaces” indicates the lack of foresight or cooperative attitude on the behalf of most language designers. My fight to crush language barriers does not stop in the computer realm. I have studied Chinese for two years and will attend the Princeton in Beijing program this summer to achieve fluency in Mandarin Chinese. I also plan to master Spanish and Esperanto before my death. The natural language problem exists in the non-spoken realm as well. For instance, Chinse sign language is different from English sign language resulting in a language barrier even amongst the deaf.

Acknowledgements

The Pacific Institute of Oriental Medicine's Qi Gong class resurrected a seemingly irreversible seven-month downward trend in physical illness and mental depression. Before Qi Gong, I was feeble, lazy, and scatterbrained — now I'm much more assertive and mentally collected. To Zheng Jiang and Mr. Zhao, I am eternally grateful. Also, to Bill and Deborah: I appreciate your nurturing of my undeveloped state and am sorry that Sahaj Marg was not the last stop on my spiritual journey.

I must thank Mom and Dad for spiritual, moral, academic, social, economic, and intellectual support. In addition, I am grateful for them giving birth to me. Thanks to my sister Cheryl for showing how good I could have done in school. Keep it up. I must thank Mother for making me tuck in my shirt and comb my hair. And of course I cannot forget my home away from home, Aunt Vernetta and Richard Shealy.

To my rap group, the Senegal Jazz Community: we must be wise and keep our eyes on the prize and steadily keep on the rise in the face of all demise. To Lai4 Xiu4 Zhen1, you showed me what I'm all about and were a true girlfriend. Too bad parental xenophobia forced you to marry Mr. Ling2. To all my other girlfriends, thank you for showing me what characteristics I should not look for in a woman. Ma always told me to leave you devils alone. Thanks to Shen Sheng for remaining a constant friend and of course my old undergraduate colleagues, Ato Arkaah and Richard Yancy, who are currently my colleagues at Lehigh University.

I am now completing my seventh consecutive year of education beyond high school. My parents were the primary factor in my success. However, my Chinese teacher Constance Cook was certainly a beacon of light for me as she has traveled several paths in Asian culture that I intend to tred before death. To the Esperanto League of North America, I bow down and praise your tireless efforts to promote communication between the peoples of the world. Professor Samuel Gulden has been more than an advisor. He put up with the comments I would write in Chinese about him on the board before he started class. He put up with my inconsistencies and temper. He is a strong supporter of functional [1] programming, my favorite type of programming. Finally, his observant Jewish faith provides evidence of the utilility of a moderate and regular life. Thanks to Mukesh Sharma for the tennis matches and Joel Elson for the intellectual conversations. Thanks to John Jaros for the Prolog book and his cooperative attitude. I must also thank Henry Odi for providing my fellowship. It is due to this fellowship that I was able to pursue the research I wanted instead of the research of a professor with money and a mission. Finally, I must thank Lehigh University — everyone from my custodian friends [2] to the telephone operator to my soccer-playing friends to the food service staff to the WLVR radio station to the beautiful little town of Bethelehem, I will certainly miss you.

[1] Applicative, declarative or logic programming if you wish.

[2] Who taught me much Spanish

Contents

List of Tables

List of Figures

Abstract

Chameleon is a C library which facilitates programming in multiple computer languages. It achieves this facility by providing routines to create encapsulated instances of a language, evaluate expressions in a language, and convert data between differing languages.

Numerous techniques for coordinating computation in multiple computer languages such as embedded languages, generic command languages, multi-paradigm languages, and "to-C" languages are examined and compared with the Chameleon approach to heterolingual computation. After critiquing these approaches, the design policy, implementation and programmer's interface of Chameleon Version 1.0 is described.

The thesis concludes with an evaluation of Chameleon Version 1.0 finding that although the design policy of Chameleon is excellent, numerous improvements can be made in the programmer's interface and implementation.

Chapter 1

Introduction

During the prototype and development phases of software development, two notions are often observed: the need to use source code or datatypes already developed in another language and the necessity to solve a complex problem using various paradigms offered by different computer languages.

Categorizing languages into either interpreted or compiled, this thesis focuses on methods for cleanly integrating the results of any number of instances of any number and type of interpreted language into the execution of a compiled or interpreted program [1] .

Interpreted languages have two categories: embeddable and non-embeddable. Embedded languages were developed with the intent of offering the programmer encapsulated computing power. They all export library routines for the interpreter creation, expression evaluation, and interpreter deletion and thus are nearly perfect vehicles for extending the computing power of the language accessing them.

[1]For the remainder of the thesis, the process of achieving a computational goal by the use of multiple computer languages will be referred to as *heterolingual computing.* The author has intentionally avoid the term *multilingual computing* as this term is popular among individuals attempting to develop software for use in multiple natural languages.

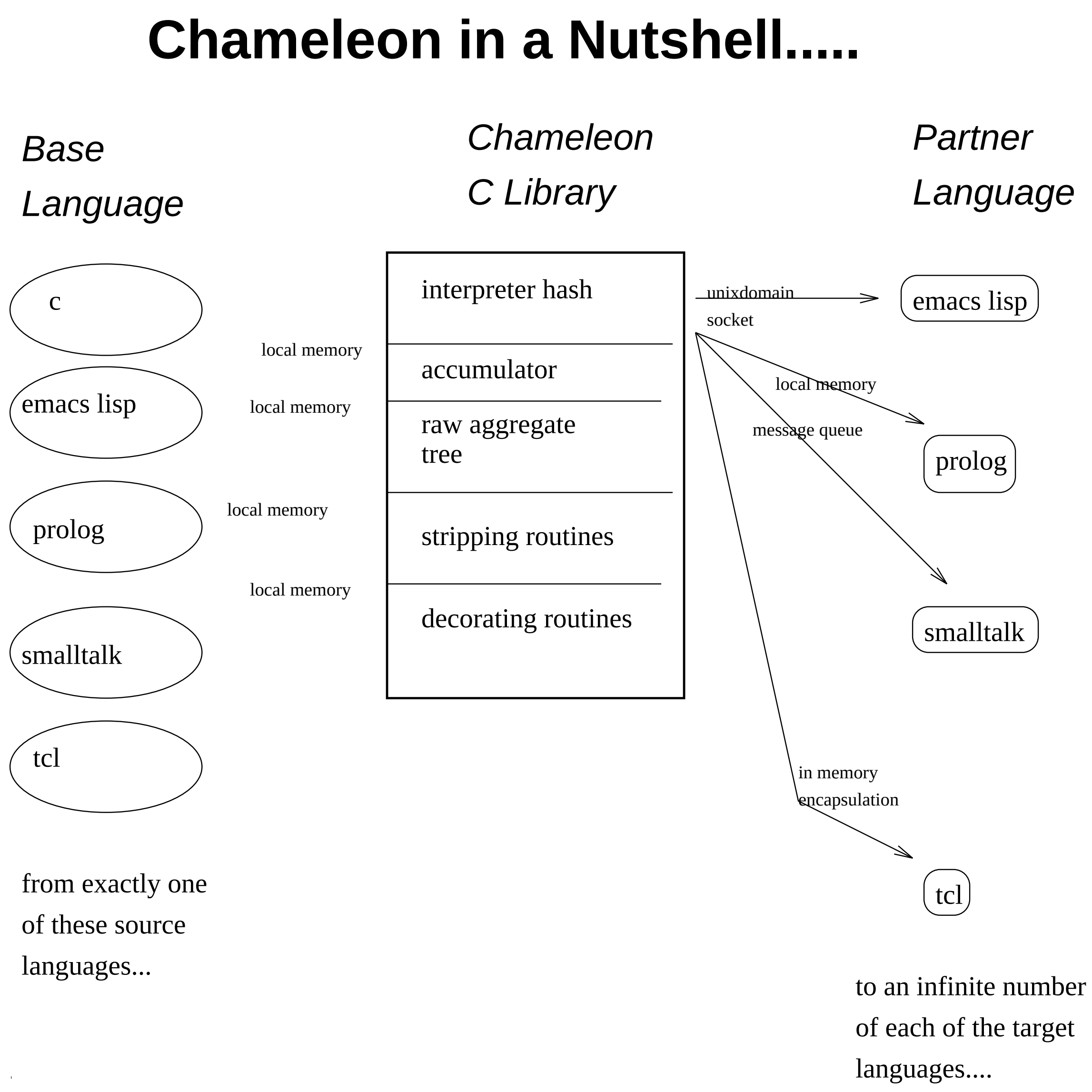

Figure 1.1: Chameleon allows the user to choose any language base his base language and then make use of arbitrarily many instances of partner languages when he desires datatypes or functionality in these partner languages.

However, no method exists of converting the results of one embedded language to another. Consider an Elk association list and a Tcl associative array. Each complex data structure has the same computational utility: to make a statement about how two or more objects relate. However, there is no well-defined method for allowing the relational data structure of Tcl/Elk to be used with the relational data structure of the other language. The obvious dilemma is that a set of relations in one language is fully oblivious and useless to another language.

Non-embeddable interpreted languages such as Smalltalk or Emacs Lisp offer little or no mechanism for communication with processes outside of their own. Though these languages offer a wealth of libraries, they may only be useful for short segments of program execution due to their high interpreted-language overhead or lack of paradigm or programming library. Prior to Chameleon, there was no simple uniform process for manipulating instances of these environments forcing the programmer to program either entirely with or without these computational fascists.

Chameleon exports and improves the embedded language paradigm. In addition to the typical routines for interpreter creation, expression evaluation, and interpreter deletion, Chameleon adds the following features to the embedded language paradigm to achieve heterolingual computing:

Apparent embedding of the non-embeddable Version 1.0 provides the programmer an embedded programming perspective on two former belligerent monoliths: Emacs Lisp and Smalltalk.

Embedded languages for embedded languages Embedded languages typically cannot access other embedded languages. Version 1.0 allows Tcl to embed arbitrarily many Emacs Lisp or Smalltalk interpreters.

Multiple instances of any interpreter with no namespace clashes

Intentionally semi-automatic conversion of data between languages In addition to allowing expression evaluation in the partner language, the programmer can request that the results of expression evaluation be coerced to a base language datatype.

1.1 Motivation

The intent of programming languages is to express human thought in a manner comprehendible by a computer [Tay89]. To facilitate this communication process, *programming paradigms* and *software libraries* are developed in chronological order. Popular programming paradigms are object-oriented, functional, declarative/logic, and imperative programming. Each paradigm is immensely useful for solving certain problems.

The object-oriented paradigm is useful when one wishes to treate the concrete entities (data) of a problem with importance equal to or greater than the functions applied to these entities. For example, a mechanical engineer may wish to model the stress and strain on a building in terms of the objects which will receive and issue stress. In an object-oriented perspective, the task is straighforward: endow a set of objects with their physical characteristics and their responses to stress and strain and then initiate the simulation by issuing the first stress or strain message. The same problem in a typical imperative language would require defining new instances of aggregate data types, manually endowing each with its characteristics and relations to similar objects, applying functions to each data object and then figuring out which data object should next have a function applied to it.

Declarative or logic programming shifts the emphasis from telling the computer how to do something and instead allows the programmer to state what he would like done. A declarative or logic program typically consists of a set of rules and facts from which the language's own inference engine produces solutions to problems. Declarative programming is very popular for artificial intelligence applications [CNV88] but is also a powerful counterpart for experimenting with meta-programming [HL92], simulation and modelling [CNV88] and compiler design [War80].

The original functional language, Lisp, was developed in order to have a language which would perform similarly to Alonzo Church's lambda calculus. Since then, numerous functional languages have been developed with the most recent ones having very efficient compilers and definitional and pattern-matching aspects similar to logic languages.

The perspective of imperative programming is very similar to the lower levels of computer programming such as assembly and micro code. Imperative languages are typically used when speed or directness is an issue.

Identifying the functional components of a complex software package (one composed of many simple pieces) will often reveal that a number of paradigms are required to solve all the problems efficiently. For example, a real-time semi-intelligent data monitor requires an expert system (logic programming), mathematical analysis (lambda calculus computing), and real-time response to sensory input (high-speed imperative computing).

1.1.1 Multiple programming paradigms become available

A software package with numerous requirements may be developed easiest by tackling subsets of the design requirements with the various programming paradigms discussed above. Heterolingual computing provides a well-defined interface between languages of varying paradigms and thus facilitates the solution of programming problems which are best solved in multiple paradigms. As a result, the programmer can use a language which is extremely well-suited to the each particular subtask instead of making compromises in just one language.

1.1.2 No more re-inventing the wheel

One of the golden rules of software engineering is to develop code in a modular fashion to promote reuse. Granted most individuals obey this policy within a language but the lack of inter-language communication facilities hinders achievement of this goal across languages.

Many languages have large, powerful, and debugged pieces of software already written. To re-code software with similar functionality is a complete waste of time unless the original language lacks functionality needed to extend the software. For example, Emacs Lisp has a program named ange-ftp which automates file-copying between any two internet sites. The program was initially coded in 1989 and has since undergone continual improvement to deal with internal bugs as well as the quirkiness of various ftp servers. Attempts to duplicate this work would result in first duplicating the immense and sometimes unique functionality of Emacs Lisp and then developing the well-tested ange-ftp software written in Emacs Lisp.

In summary, having surveyed the potential of several popular languages and their libraries (see [Bra93c] for more details), the author knows of language libraries to address important problems. However, coordinating the solution of the complex of these problems in a large software package demands heterolingual computing facilities such as the ones prototyped in this Master's thesis.

Chapter 2

Prior Work in Heterolingual Computing

2.1 Mother and Embedded Languages

Given a language A, if an evaluator for language B is written in language A, then language B is said to be **embedded** in language A. Language A is termed a *mother language.* An embedded language is developed for one of two reasons [1] First and most often, an embedded language is developed to extend the functionality of the mother language. On the other hand, other embedded languages require that most of the functionality for an embedded language be coded in the mother language and that the embedded language simply serves as a shell-level command line interface to mother language functionality. [DB93] The only eminent language in the second category is S-Lang. S-Lang is the embedded language for the JED Editor. Popular

[1] Aside from the reason irrelevant in this thesis which is to experiment with language design using the mother language as a virtual machine, thereby having access to the debugging and testing facilities of the mother language. [AS85]

embedded languages of the first type include Elk, Emacs Lisp, Perl, Python, REXX, and Tcl all of which export a wide variety of powerful routines to the programmer. In addition, the C callout routines for a countless parallel and sequential languages are developed with this hunger for the huge C codebase in mind. Chameleon is motivated and designed with this same hunger for codebase; but, instead of stopping with C intends to allow the programmer to access code in any language he desires.

The three most popular mother languages are C, Prolog, and Common Lisp. C is used to develop interpreters for numerous languages such as Smalltalk, Tcl, Perl, Python, Prolog, Scheme, and Common Lisp. Prolog is used as the mother language for newer generation logic of programming systems such as ECLIPSE and Goedel. Common Lisp has found extensive use as the mother language for numerous pseudo-functional offshoots such as ALLOY and SETL2.

2.1.1 Mother Languages

C

C has been widely touted as a portable assembly language. In conjunction with the wealth of operating system code written in C, C has ample facilities for executing the desired functional and data semantics of any language.

Variable argument lists A C function can have a variable number of arguments provided they are all of the same type [2]. This feature allows the variable argument lists of most interpreted languages to be implemented directly in C.

[2] In order to allow the elements of a variable argument list to differ in type, the programmer can use a union of the various types that he may wish in the list. The GNU Smalltalk source code contains examples of this programming technique.

Process snapshot A Unix process is defined by its executable disk file. A running unix process is capable of examining its entire process space and creating this executable disk file. Most Emacs processes that a user typically invokes are the result of this process. The Smalltalk snapshot mechanism also takes advantage of this ability.

Ability to save/restore stack A C program can record the entire stack into a complex data structure for later recall. Scheme continuations are implemented this way.

Pointers to data Each data item in a C program can have its address stored in an integer variable. Later, this integer can be used to change and access the information stored in the original data item.

Pointers to functions C also allows the programmer to store the address of each defined C subroutine in an integer variable. Although it is possible to change the instruction set of a function on the fly this way, the more typical use of a pointer to a function is to allow invocation of subroutines which were dynamically linked into the process after the initial creation of the process.

Aggregate data structures The C `struct` and `union` datatypes allow a programmer to collect primitive machine types into more higher-level data structures for use in the embedded language and access them in constant time[3].

For example, here is C source code defining the Emacs Lisp `subr` datatype. The `subr` datatype is used to maintain information about an Emacs Lisp subroutine.

[3]as opposed to using a Lisp list which takes $O(n)$ access time.

```
struct Lisp_Subr
  {
    Lisp_Object (*function) ();
    short min_args;
    short max_args;
    char *symbol_name;
    char *prompt;
    char *doc;
  };
```

The first field of the structure contains a pointer to a function. This is used at runtime to find the where to place the program counter when this subroutine is invoked. The next two fields specify the minimum and maximum number of arguments the function will accept. The `symbol_name` field is a pointer to a sequence of characters, or a *string.* This string contains the name of this function. The `prompt` field contains a string that that can appear in the Emacs minibuffer [4] if this function was designed for interactive use. The last field is a string containing documentation for the function.

Prolog

Prolog, short for PROgramming in LOGic was a strong step toward *declarative programming:* specifying what should be computed instead of how to compute something. The ability to allow the computer to figure out how to compute something instead of directly programming it is achieved by automatic depth-first search of a

[4] The Emacs minibuffer is the one-line region at the bottom of the Emacs frame where messages and prompts for input are printed

space of facts and rules. The definitional and logical nature of Prolog gives its procedures functionality not easily achieved in an imperative language. For example, unlike a LISP append procedure which can only take two lists and create a third, a PROLOG append procedure can do this plus generate every possible combination of two lists that could be appended to create a third list. The reader is referred to [Coh85] and [CM84] for more examples of Prolog's amazing automatic computing facilities.

Even though Prolog did much for declarative programming, it falls far short of cleanly coordinating the logic of a program with function and data control of a program [Mac87] [HL92]. As a result, numerous new generation logic languages such as Goedel are being embedded in Prolog.

Common Lisp

The development of Common Lisp was motivated by a desire to have a method for expressing Alonzo Church's lambda calculus on computers. Common Lisp's core syntax and semantics are extremely simple. However, Common Lisp has been the breeding ground for numerous dialects with various scoping policies and datatypes. The result is that a statement as simple as

```
(+ a b)
```

is very dependant on whether the scoping of the Lisp dialect is dynamic or lexical/static and whether this an object-oriented extension of Lisp with operator overloading.

2.1.2 Embedded Languages

Emacs Lisp

Emacs Lisp is a dialect of Lisp with dynamic scoping and non-standard types useful for editing such as buffer, window, frame, marker, process, stream, keymap and syntax table. It also contains a wealth of useful editing operations such as cutting and pasting strings between buffers, saving regions of buffers in registers for later recall and saving buffers to disk.

The wealth of extra datatypes made Emacs Lisp the defacto choice for significant free software packages such as:

ange-ftp Automates and facilitate ftp file transfer

calc Best known as the "poor-man's Mathematica," Calc is an arbitratry-precision calculator with numerous operations for all major mathematics areas. Its support includes rewrite rules, integration/differentiation of continuous functions, matrix and vector operations, and probability and statistics functions.

edb A full-featured fully programmable relational database.

origami Allows for hierarchical editing of an Emacs Lisp buffer.

tree-dired Allows traversal of a directory hierarchy through one and two keystroke commands.

Tcl

Tcl, short for Tool Command Language, is impressive for its simple syntax yet amazing power to emulate any number of language paradigms and for its simple but powerful X11 toolkit Tk.

Concerning emulation of language paradigms, three object systems have been developed by Tcl programmers. That each Tcl instruction returns the result of its execution makes emulation of the functional paradigm a given. As each Tcl instruction consists simply of command and arguments, the imperative paradigm is Tcl's very core. As logic programming is essentially the search of a set of relations to determine the validity of a query, one can use Tcl's associative array to maintain a set of relations between predicate and objects [5].

Tcl's user interface capabilities in the form of the Tk toolkit are as powerful as any GUI [6] toolkit the user has ever encountered. The famous `hello, world` program can be written in two (2) lines of Tk code. In addition, menu-driven user-interface development kits have been written in Tk.

Evaluation

At present, embedded languages cannot achieve satisfactory heterolingual computing for the following reasons:

1. No protocol exists for communicating between different embedded languages.

2. The multilingual extent of embedded languages is at best bilingual: itself and the embeddee language.

As a result, though the intent of many singular embedded languages is admirable, the multiple embedded languages is far more tedious than it should be.

[5]As a matter of fact, the author embedded a pure interpreter for Prolog within Tcl in 5 weeks time. The implementation was indescribably simple: no new datatypes needed to be created and the raw Tcl library functions were more than adequate for all aspects of the embedding.S

[6]GUI is short for Graphical User Interface

2.2 Information Modelling Environments

Large-scale software packages such as CAD and physical simulators transform a set of inputs (mouse-movements, circuit specifications) into output (data-file, graphic plot). The typically standalone nature of these products is "... one of the key problems impacting quality and productivity in design..." [LSK+93] The n-DIM project at Carnegie-Mellon exemplifies the leading-edge research in *tool integration.* N-DIM (n-dimensional information modelling) is a software package intended to support all phases of engineering product design. To meet these goals, n-DIM exports a distributed multimedia graphic database to the designer or group of designers.

2.2.1 n-DIM

One phase of work in the n-DIM project resulted in automatic conversion of application-specific n-DIM specifications to a format useable by ABLOOS, software designed to automate input to a standalone package and capture the package's output.[LSK+93]

The n-DIM group is also developing methods for incorporating multimedia information into the n-DIM environment.

Evaluation

Drawbacks in the n-DIM approach are:

1. n-DIM is a layered environment relying on numerous large pieces of commercial software such as the ISIS Distributed System Development Kit and numerous databases.

2. Designing a heterolingual program in n-DIM requires X-Windows.

2.3 User Interface Packages

2.3.1 Generic Syntax Packages

The following two packages have developed uniform language-independant command sets which allow for user interface development regardless of the native language. WAFE is a language which uses a generic user interface language to develop X-Windows interfaces. In STDWIN, user interface commands will create a user interface on a number of platforms.

WAFE (Widget Athena Front End)

Wafe (Widget Athena front end) is a package that implements a symbolic interface to the Athena *widgets* (X11R5). A typical Wafe application consists of two parts: First, a process (of any language) requiring the use of graphics which has the ability to open a UNIX socketpair or System V stream and second a wafe process which reads graphic instructions from stdin. Examples in the WAFE distribution show Perl, GNU awk, Prolog, TCL, C and Ada all talking to the same Wafe binary. In a sense, WAFE is the exact opposite of Chameleon. Chameleon effects one-to-many heterolingual programming while WAFE effects many to one heterolingual programming.

Evaluation

1. Only graphic information can be shared among process

2. Requires X-Windows

STDWIN

Guido von Rossum's STDWIN C library allows the same source code to do graphics on the following systems:

1. X-windows
2. Macintosh
3. Atari ST
4. DOS
5. Silicon Graphics SGI workstations

This approach allows flexibility but means that no one graphics system's potential is maximized. The set of generic widget and drawing commands available are powerful enough to handle most user interface and graphics chores.

Evaluation STDWIN is not acceptable for heterolingual programming as it does not address data conversion issues. Furthermore, any language wishing to use STDWIN must first cobble up its own interface to C.

2.3.2 TkSteal: Emacs and Postscript Widgets

TkSteal, written by Sven Delmas, allows for Emacs and Postscript widgets in a Tcl/Tk program. The running Tk has complete control over the running widget meaning that source code can be sent to the running widget for bilingual programming.

2.4 Multi-paradigm Languages

Although multi-paradigm languages have one syntax, the semantics of the syntax can vary depending on the paradigm/mode of operation the user has chosen. Since a language is defined as having one set of syntactic rules mapping to one semantic interpretation, the abilitiy of the following languages to change modes implies heterolingual programming.

2.4.1 ALLOY

ALLOY manages to directly support functional, object-oriented and logic programming styles as well as eager or lazy evaluation using only 29 primitives [Mit91]. ALLOY has a functional core. Object-oriented programming is supported through a number of primitives typical of object-oriented tasks such as class and instance creation. ALLOY's functional LISP-like core eliminates the need for message-passing primitives as an object is nothing more than a function. In fact objects can be accessed and manipulated using `eval` or `mapcar`. Such an approach allows for powerful synthesis of object-oriented and functional programming techniques.

Mitsolides' determination to make every function an object has interesting pros and cons. One outstanding aspect of his approach is that he can replicate an object (essentially an encapsulation of a Hoare precondition on variables along with a finite set of functions capable of achieving the postcondition) over numerous processors and processes and have the object process messages in parallel yet use communicating synchronizing variables to ensure that the interleaving of message reception does not result in an invalid postcondition/state. Such an approach resembles that of using a controlled collection of actors which splits apart to solve, process a precondition and then re-coalesce back to the desired postcondition. However, maintenance of an object system leads to the additional irremovable overhead of the object (a function) calling another function (the method).

ALLOY's generator primitive supports a form of *programming in the large,* meaning that single commands can perform reductions and transformations of aggregate data structures such as vectors, arrays, and lists. This feature allows direct specification of what would typically be a contrived and bottlenecked stream computation. The following example make use of pseudo-code over the cryptic ALLOY operators. Assume that you want to print the numbers from one through five. The typical solution would be to have a producer produce a stream of the numbers and a printing consumer print them. However this computation is achievable in one line in ALLOY:

```
((REPLICATE (print)) enough times to handle (GENERATE (make-int 1 5)))
```

Actual ALLOY code would be

```
(let () #(print w *(ints 1 5)))
```

The implementation of a generator allows parallel results to be sent directly into pre-indexed concurrently writeable array positions.

Evaluation

ALLOY has the problem that any new multi-paradigm language would have: no codebase and few users. One of the major purposes of heterolingual programming is to take advantage of existing software. ALLOY does not support this goal.

2.4.2 Prolog

Four aspects of Prolog allow it to be heterolingual very easily. First, Prolog can modify its syntax to semantics mapping at run-time by the addition or redefinition of an operator. [CNV88] Second, the declarative nature of Prolog allows it to acquire data types and programming paradigms by simply adding facts about these clauses to its knowledge base. [CNV88] [Mac87] Third, the extensive work on Prolog-based compilers implies the ability to digest the semantic intent of numerous languages at runtime. [War80] Finally, all data in Prolog is inherently abstract, meaning that the user only declares data while Prolog determines how it will be internally represented. [Mac87]

To exemplify the first characteristic consider the addition of a biconditional operator to Prolog:

```
:- op(950,xfx,'-:-').
```

The first argument gives the precedence of the operator. The second arguments indicates that it is an infix operator lacking both right and left associativity. The third argument is the character string representing the operator [CNV88].

Given that a Smalltalk dictionary is nothing but a list of associations, observe how Prolog achieves full functionality of a Smalltalk dictionary by the following clauses:

```
dictionary(nil).
dictionary([H|T]) :- association(H), associations(T).


associations(nil).
```

```
associations([H|T]) :- association(H), associations(T).

association([key,value|[]]).
```

A further advantage of using Prolog to add datatypes to a language is that the user only specifies the characteristics of the data while Prolog handles the internal representation of the data [Mac87].

The reader is referred to [War80] for work on the role of logic programming in compiler design.

Evaluation

As long as the Prolog implementation can be called from C and supports C callouts (and a number do), the author sees no problems with Prolog as the vehicle for implementing a heterolingual C library. In fact, the author intends for Prolog to play a major part in Chameleon version 2.0. See page 49 for further details.

2.4.3 DLP (Distributed Object-Oriented Prolog)

In addition to the benefits articulated under Prolog, DLP allows a programmer to create prototype objects whose methods are Prolog clauses. The author of DLP, Anton Eliens was motivated to develop DLP because he saw the emergence of three fields — parallel, logic, and object-oriented computing that had no robust integration. Prior work attempting this integration was found deficient to achieve the requirements of distributed artificial intelligence computation [Eli92].

Two steps are required for process creation in DLP:

Create an instance of the logical object prototype Given a logical object named `lib`, we can create an instance of the object by the statement: `OBJ_INST::new(lib)`

Request that the logical object evaluate a goal Evaluation of a goal within a logical object proceeds asynchronously. To receive the results of goal evaluation, we instantiate a pointer to the process in which the goal is being solved and query the pointer later. For example, to get the results of the query `member(a,cons(a,cons(b,nil)))` with the `OBJ_INST` instance of the `lib` prototype we execute the following statement:

```
PROC_PTR::OBJ_INST!member(a,cons(cons(b,nil))), PROC_PTR?
```

It is possible to statically assign resolution of a goal to process at process creation time. Otherwise resolution proceeds on the local processor.

Both asynchronous communication (mistakenly called a rendez-vous in his paper) and synchronous communication (via channels) is supported.

With DLP, both and-parallelism and or-parallelism can be freely intermixed to achieve high speed computation. The encapsulation of logical programs within objects allows for excellent partitioning of a program into functional subunits (objects) with inferencing capabilities. The ease and power of this method of partitioning is shown in a mock medical expert system.

Evaluation

DLP is an exciting language overcoming earlier approaches trying to integrate object and logic programming. However, the sole issue that Chameleon wishes to address is data conversion not process mapping, program decomposition into objects, etc. As a result, DLP offers too many features for the limited goals of Chameleon.

2.4.4 Emacs Lisp

The Emacs Lisp editing commands are `mode-dependant` which implies that the meaning of each character can be changed depending on the mode of the buffer. Commands such as `forward-word` and `forward-paragraph` perform differently on the same text depending on the chosen buffer mode. For example, a semicolon is a non-balanced comment character in the `lisp-mode` syntax table and it is considered part of a word in `fortran-mode` [LaL93]

In addition, Emacs is capable of spawning and controlling asynchronous and synchronous processes. This allows for the "programming" of these processes from Emacs Lisp.

Evaluation

Emacs Lisp's syntax-directed behavior is very different from the modes that the other multi-paradigm languages offer. Understandably, the Emacs Lisp modes allow for convenient decomposition of text fields while the other languages allow for convenient decomposition of more general computing problems.

Parsing and lexical analysis are essential for Chameleon's data conversion routines. As a result, Emacs Lisp syntax modes are attractive. However, Emacs' asynchronous interaction with proceses can only send 255 characters (in most Unix implementations) to a process at once as this is the usual limit placed on buffering of standard input. Finally, it would be burdensome to have to link in the entire Emacs distribution every time a program wanted heterolingual computing abilities!

2.5 "To-C" languages

A number of high-level languages have been developed which use C as their portable "assembly" language and are appropriately labeled as **to-C** languages. Reasons for such development vary.

- Some to-C languages, such as Common Lisp and Scheme are developed to obtain interpreted functionality. Sather also achieves interpreted functionality even though compiled through the use of a Sather interpreter and higher-order functions.

- Other to-C languages are developed to achieve object oriented capabilities such as parameterized classes lacking in languages such as C++ or Objective-C. Sather [Omo93] and the COOL library [Nor] are two such languages addressing shortcomings in other C-based object languages.

- A final category of to-C languages wish to provide error checking for legal yet undesired program results. Sather can be classified as such as language.

Regardless of these languages' individual motivation all benefit from retaining compatibility with the huge C language codebase and excellent execution speed due to years of compiler technology. [FOT92]

2.5.1 Sather

Sather is an object-oriented language recently designed and implemented at the International Computer Science Institute in Berkeley. It compiles into C and is intended to allow development of object-oriented, reusable software while retaining

C's efficiency and portability. Through a comparative performance study and analysis of Sather and C programs on a RISC machine, it was found that while Sather users still pay a moderate overhead for programming convenience (in both run time and memory usage) the overall CPU and memory usage profiles of Sather programs are virtually identical to those of comparable C programs. [LS91]

In addition to the typical C callout mechanism, Sather exports classes for lexical analysis and parsing, allowing for runtime translation of "language" into Sather objects. [Omo93]

Evaluation

The wealth of classes and closeness to C of Sather make it an appealing option for implementing Chameleon. If logic programming classes were ever added to Sather, it would be an excellent language in which to develop a portable C library.

It may be wise to simply code a logic programming class for Sather instead of using a full-fledged Prolog implementation so that the issue of converting numerous language's data to Prolog's types does not surface.

2.5.2 Common Lisp

With the advent of *CLiCC,* a Common Lisp to C compiler, Common Lisp is now a viable to-C language. CLiCC generates C-executables from Common Lisp application programs. [Goe]

Evaluation

Common Lisp offers no inherent functionality which would make it a viable language for developing heterolingual computing facilities. Furthermore, it lacks Prolog's inferencing capabilities.

2.6 Overall Evaluation

The embedded approach is attractive due to its intent to work with another language. The only shortcoming of this approach for heterolingual programming was stated in the Introduction: multiple embedded langauges only easily share data for simple types such as integers and strings and languish when trying to share more complex types.

Sather's parsing classes make it an interesting choice for implementing data conversion. Extensive work would be necessary to implement data convesion however.

Chapter 3

Design Policy

The primary intent of Chameleon is to provide an embedded language paradigm offering the goals stated in the Introduction in addition to the typical goals of an embedded language.

The Chameleon programmer's interface (API) was heavily influenced by the Tcl API. Tcl's API allows for easy creation, deletion and evaluation in several Tcl interpreters simultaneously with no namespace conflict. All of Chameleon's embedded language manipulation routines are routines analagous to Tcl's routines. [Ous94]

A uniform interface to each language consists of an interpreter creation primitive, an expression evaluation primitive, and a data conversion primitive.

3.1 Interpreter creation primitive

For each instance of a partner language, the user receives a pointer to a Chameleon interpreter which handles the creation of an instance of the remote language. Using C as the example native language, the user would create an instance of a partner language by the following C code:

```
ci = Cham_CreateInterp("c",partner_language,host.domain:0.0);
```

Where `partner_language` is one of emacs, prolog, smalltalk, or tcl and host and domain are replaced by the hostname and domainname of the current machine.

3.2 Expression evaluation primitive

After the user has used the native language to create a string expression evaluatable in the partner language, he would have chameleon automate the interaction with the remote language via:

```
Cham_Eval (ci, "(+ 40 20)") ;
```

If C is the base language, the result of expression evaluation is stored in the field `ci->result`. Interpreted base languages use the following technqiue:

```
Cham_Result [Cham_Eval $ci "(+ 40 20)"]
```

3.3 Semi-automatic data conversion primitive

The results of executing an expression in a partner language are automatically stored within the structure pointing to the partner language instance. However, the results are not automatically converted to a form useable by the native language unless the command `Cham_Coerce` is given. The reason for this is that the partner language may have only been called for side effect.

The steps the user must perform to convert data to a form comprehendible by the base language are stripping and decorating.

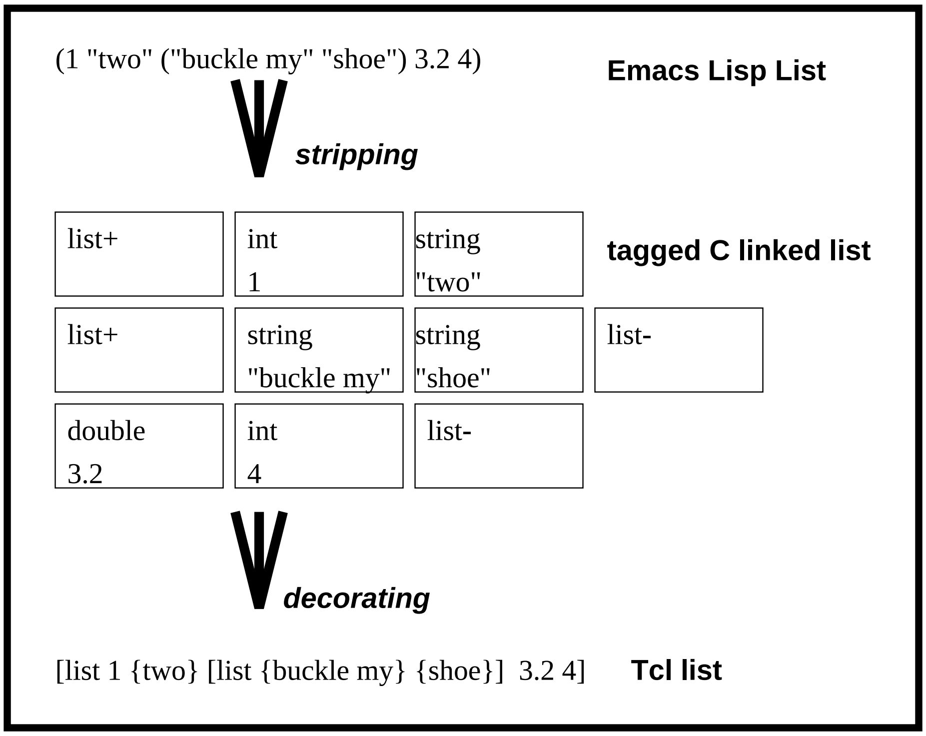

Figure 3.1: Data conversion always proceeds by stripping the language into a tagged C linked list and then decorating the tagged C linked list in a format comprehendible by the language needing the data.

3.3.1 Stripping of partner language's result

The result in the `ci->result` field is converted to a tagged C language linked list by the command:

```
Cham_Coerce (ci) ;
```

This process is called **stripping**. The result of is stored in the field:

```
ci->aggregate_data_stack
```

3.3.2 Decorating stripped data for native language

The tagged C language linked list is converted to data digestible by the native language using the command:

```
Cham_Decorate (ci) ;
```

This process is called **decorating**. For C language, the result of decoration is stored in the field `ci->result_of_tree_traversal`. In the case of an interpreted language, the result of decoration is returned from a call to the `cham_coerce` function. For example, for Tcl to get back converted data in a variable called L, the command would be

```
set L [cham_coerce $ci]
```

Chapter 4

Implementation

Because access to a partner language only requires a pointer to a `Cham_Interp` structure, each native language need only retain the pointer to this structure whenever it uses the Chameleon library routines to manipulate the structure.

All decoration is driven by the table in `cham-decorator.c` which generates the appropriate syntactic token based on a 2-input function consisting of the target language and the generic tagged datatype.

4.1 Interpreter Creation

Even though the interface itself is homogeneous, Chameleon's internal methods for creating encapsulated instances of each partner language are quite heterogeneous varying from the use of forking and message queues to simple memory allocation.

4.1.1 Emacs Lisp

To understand the communication link from the Chameleon to Emacs, one must first understand the ability of Emacs to spawn and communicate with an inferior subprocess. First the Emacs Lisp command `start-process` is invoked with the name of an executable process to be spawned as a subprocess of Emacs. Next, the programmer typically specifies a function to serve as a *filter function* for the subprocess. The filter function is invoked each time the subprocess generates output to the Unix standard output. Filter functions are used to interpret the output of the subprocess so that some action may be taken within Emacs. See the file `emacs-create-interp.c` for more information.

In the case of having an interface between the Chameleon C library and Emacs, an Emacs process is invoked with a `-l gnuserv.el` argument indicating that the source code in `gnuserv.el` should be loaded. Loading this source code creates a subprocess which serve as an Emacs Lisp command interpreter.

4.1.2 Smalltalk

Source code was added to GNU Smalltalk to force it into a server mode where it would create a message queue and read Smalltalk syntax from the queue. See the file `patch-pkg/smalltalk/mstmain.c` in the Chameleon distribution for more information.

4.1.3 Tcl

Creation of a Chameleon interpreter with Tcl as the partner language simply required calling `Tcl_CreateInterp` and storing the pointer to the allocated `Tcl_Interp` data structure in a `Cham_Interp` data structure.

4.2 Expression Evaluation

4.2.1 Emacs Lisp

The filter connected to the Emacs subprocess takes the text output from the Chameleon C library and evaluates it as an Emacs Lisp form. The printed representation of the resulting Emacs Lisp form is then redirected to a temporary file. See the file `emacs-eval.c` in the Chameleon distribution for more information.

4.2.2 Smalltalk

Upon reading Smalltalk syntax from the queue, the following sequence occurs:

The original process forks. The child of this process creates a two-way pipe. Another fork occurs. The grandchild of the original process creates another pipe, closes the standard output and then opens a file descriptor to which it writes the Smalltalk expression and then forks. The child of the grandchild of the original process creates a two-way pipe and then closes standard input and waits for the Smalltalk expression. After reading and evaluating the expression, the expression is immediately printed to standard out. However, since the standard out was closed above, the grandchild of the original process receives the resulting output. See the file `patch-pkg/smalltalk/mstlib.c.patched` in the Chameleon distribution for more information.

4.2.3 Tcl

Evaluating a Tcl expression simply required calling `Tcl_Eval` and storing the result of the evaluation within the `Chameleon_Interp` data structure.

4.3 Semi-Automatic Data Conversion

4.3.1 Stripping

Emacs Lisp

Stripping of Emacs Lisp forms is done by a lexical analyser written in `flex`. See the file `emacs-stripper.c` in the Chameleon distribution for more information.

Smalltalk

Stripping of Smalltalk forms is done by a lexical analyser written in `flex`. See the file `smalltalk-stripper.c` in the Chameleon distribution for more information.

Tcl

Stripping of Tcl is done by a lexical analyser written in `flex`. See the file `tcl-stripper.c` for more information.

4.3.2 Decorating

Regardless of which language requires decoration of data, decoration is driven by the table in cham-decorator.c which generates the appropriate syntactic token based on a 2-input function consisting of the target language and the generic tagged datatype.

Table 4.1: Status of Implementation

	C	Emacs Lisp	Prolog	Smalltalk	Tcl
C	–	full	limited	limited	full
Emacs Lisp	none	–	none	limited	full
Prolog	full	none	–	none	none
Smalltalk	full	limited	none	–	limited
Tcl	limited	full	none	limited	–

Conversion from the interpreted languages to C was problematic as C lacks built-in data structures similar to the ones in Emacs Lisp, Tcl and Smalltalk. The existing solution is to leave the data in the tagged linked list.

Chapter 5

Example Uses

5.1 C as the Base Language

5.1.1 Calling Emacs Lisp

```
#include "chamInt.h"


void
main (argc, argv, envp)
int argc ;
char ** argv, ** envp ;
{
  Cham_Interp *  ci ;
  int            result ;

  if (argc != 2)
    {
```

```
    printf ("usage: %s displayname:0.0 \n", argv[0]) ;
    exit (1) ;
  }

  ci = Cham_CreateInterp ("c","emacs",argv[1]) ;
  sleep (20) ;

  result = Cham_Eval (ci ,
       "(list (* 23423 123)
                    (make-vector 2 \\\"joe bloe\\\")
                    \\\"winston salem\\\"
                    (+ 23 23 2323 23 23 23 3) (+ 20 30 40))") ;

  Cham_Coerce (ci,"c") ;

  ci = Cham_CreateInterp ("c","emacs",argv[1]) ;
  sleep (20) ;
  result = Cham_Eval (ci ,
               "(list (* 23423 123)
                (make-vector 2 \\\"joe bloe\\\")
                \\\"winston salem\\\"
                (+ 23 23 2323 23 23 23 3) (+ 20 30 40))") ;

  Cham_Coerce (ci,"c") ;
}
```

5.1.2 Calling Smalltalk

```
#include "chamInt.h"


void
main (int argc, char * argv [])
{
  Cham_Interp *
    ci ;
  int
    result ;


  /* this says:
      Chameleon initialize an interpreter. I am calling
you from C language and wish to be able to
access smalltalk
     */
  ci = Cham_CreateInterp ("c","smalltalk") ;


  printf ("no sleeping...\n") ;
  sleep (20) ;


  result = Cham_Eval (ci ,
                "| x | x _ (Dictionary new) .
                 x at: 'ab' put: 27 .
                 x at: 'ba' put: 33.
```

```
               Cham _ x. Cham printNl !") ;
  printf ("result: %s \n", ci->result) ;

  Cham_Coerce   (ci,"c") ;
  Cham_Decorator ("init session",ci) ;
  Cham_Decorator ("print tree") ;

  result = Cham_Eval (ci ,
              "| x | x _ (Array new: 4).
               x at: 1 put: 27 .
               x at: 2 put: 33.
               Cham _ x. x printNl !") ;
  printf ("result: %s \n", ci->result) ;

  Cham_Coerce   (ci,"c") ;
  Cham_Decorator ("init session",ci) ;
  Cham_Decorator ("print tree") ;

}
```

5.1.3 Calling Tcl

```
#include "chamInt.h"

main ()
{
```

```
  Cham_Interp * ci ;

  ci = Cham_CreateInterp ("c","tcl",0) ;

  printf ("%ld \n", ci) ;

  Cham_Eval (ci, "expr { 2 + 5 * 200 + 80001 *20 }") ;

  printf ("result: %s \n" , ci->result) ;

}
```

5.2 Emacs Lisp as the Base Language

As stated on page 52, the best method for integrating execution of compiled code with interpreted code is not understood. As a result, no support currently exists in Chameleon for calling compiled languages.

5.2.1 Calling Smalltalk

```
;;; -*-lisp-interaction-*-
;;; Here is a simple test of emacs' ability to call tcl
;;;



(setq si (smalltalk-create-interp))
```

```
(setq result
     (smalltalk-eval si
                     "| tp | tp _ (Dictionary new).
                      tp at: 'address' put: '221 Baker Street';
                         at: 'job' put: 'private I';
                         at: 'friend' put: 'Lai Xiu Zhen'.
                      tp printNl !"))
(message result)
(cham-coerce si)
(cham-print-tree si)
(setq y (eval (car (read-from-string (cham-traverse si)))))

(assoc "friend" y)
(assoc "address" y)



;;;
;;; type C-j at the end of each the
;;; parentheses-enclosed lines below:
;;; (simpleton)
;;;

(goto-char (point-min))
(search-forward ";;; (sim")
(end-of-line)
```

5.2.2 Calling Tcl

```
;;; -*-lisp-interaction-*-
;;; Here is a simple test of emacs' ability to call tcl
;;;

(defun sum-of-lengths ()
  (interactive)
  (setq y (tcl-create-interp))
  (tcl-eval y "set a [list 1 2 3 4 5 65 6]")
  (tcl-eval y "set b [list 8 2 3 1 3433 9]")
  (message "sum of both lists is: %s"
           (tcl-eval y "expr [llength $a]+[llength $b]")))

;;;
;;; type C-j at the end of each the
;;; parentheses-enclosed lines below:
;;; (sum-of-lengths)
;;;
```

```
(goto-char (point-min))
(search-forward ";;; (sum")
(end-of-line)
```

5.3 Tcl as the Base Language

5.3.1 Calling Emacs Lisp

```
set ei [emacs_create_interp neptune:0.0]
emacs_eval $ei "(list 20 230 40 405)"
cham_coerce $ei
cham_print_tree $ei
set res [lindex [cham_traverse $ei] 0]
```

5.3.2 Calling Smalltalk

```
set si [smalltalk_create_interp]
set result [smalltalk_eval $si "| tp | tp _ (Dictionary new).
                                tp at: 'address' put: '221 Baker Street';
                                   at: 'job' put: 'private I';
                                   at: 'friend' put: 'Lai Xiu Zhen'.
                                tp printNl !"]

cham_coerce $si
```

```
cham_print_tree $si

set y [cham-traverse $si]


puts $y
```

Chapter 6

Improving the Design of Chameleon

6.1 Experiences with Version 1.0

The programmer interface offered in Chameleon is very general and flexible, allowing for easy use within any of the popular frameworks for multi-process (and hence multiple compter language) computing. The major shortcomings of the approach were:

6.1.1 Syntax Alone is Not Enough

In Chameleon 1.0, data is converted to neutral format after the printed representation of the data is received from the partner language. This means that the precise type *and semantic intent* of the data must be determined from lexical analysis of the data alone — an impossible task. To a lexical analyzer, a Lisp property list looks just like any other list. As a result, allowing a lexical analyzer to convert the results will result in a loss of semantics.

6.1.2 Syntax may not exist

No printed representation exists for a Tcl associative array or an Emacs Lisp buffer which would allow you to recover all possible information from these data structures.

However, it is imaginable that you would want to transfer this data with no printed representation between languages/processes.

6.1.3 C language has no idea what it got back from the interpreted language

In Chameleon 1.0, a C program which coerces results back from a partner language receives the results in a tagged linked list. The problem with this

is that languages such as Lisp only support sequential structures to create aggregate types As a result, the most efficient conversion is not made. For example, the Lisp pseudo-property list:

```
(age int-val name string-val)
```

Is best converted to this C structure:

```
typedef struct P_List {
   int  * age  ;
   char * name ;
}
```

but instead will be converted to a tagged linked list which has $O(n)$ instead of $O(1)$ access time.

6.1.4 Raw partner data cannot be communicated

Chameleon 1.0 assumed that data would need to be converted to neutral format before being sent to another partner language. However, when sending data between two instances of the same language or instances of languages with similar data structures (i.e., Scheme and Common Lisp), the raw partner data can be sent, thus saving conversion time.

6.1.5 Conversion routine is too low-level

A very serious problem is that the conversion routine makes use of a tagged linked list, pointers to functions, and other hard-to-follow C code. In order for this mechanism to be useful (and bug-free) for language authors who wish to add their language to the family of languages available for heterolingual computing, the interface must be much more high-level.

6.1.6 Converted results are not persistent

After a data value has been converted, there is no way to store it away for later use. The resulting shortcoming is that certain computations must be redundantly performed in order to regenerate data.

6.1.7 Spawning certain processes required X-Windows

In order to have Emacs run asynchronously with the base language and not overtake the terminal, Emacs is invoked with the `-d` flag thereby forcing it into it's own window. This approach implies that Emacs Lisp cannot be used from Chameleon without an X-terminal.

To overcome this serious hindrance, the next version of Chameleon will make use of the Free Software Foundation's iscreen [scr] package which provides you with an ANSI/vt100 terminal emulator capable of multiplexing up to 10 pseudo-terminals.

6.1.8 The conversion routine is narrow-minded

Front-to-end traversal of a list was appropriate enough for the languages used in [Bra93a], but is not general enough to handle datatypes such as Haskell recursive types. [Hud92]

6.1.9 Chameleon is Unix-dependant at architectural and operating-system level

The Chameleon source code makes heavy use of unix operating system calls to spawn interfaces to language interpreters. It also makes use of message queues, and Unix sockets to effect interprocess communication. The distributed computing languages to be layered on top of Chameleon are all Unix-oriented. As a result, Chameleon has no protocol for interfacing to languages on non-Unix architectures and operating systems.

6.1.10 Compiled languages aren't welcome in Chameleon

No support exists for invoking and deleting multiple instances of compiled code in Chameleon. In fact, compiled code is rarely used in this fashion. The solution for how to integrate compiled code with the Chameleon paradigm must be explored fully.

6.1.11 Development of evaluable expressions is difficult

Version 1.0 offered the user no help on developing the expression to be evaluated by the embedded language. In addition to the string processing acrobatics this forced the programmer through, it is difficult to think in more than one syntax at once.

The solution requires several parts:

1. Allow creation and saving of "pointers" to simple data to be sent to the embedded language

2. Provide aggregate builder functions which take lists of simple data (or aggregate data elements) and return "pointers" to the aggregate data element

3. Have Chameleon build the expression to be sent to the remote language. Having created numerous pointers to data for the embedded language, we simply specify the function we want and the data pointers we want as arguments to the function and have the Prolog-based expression builder create the proper syntax.

An example unifying this solution:

```
#include "Chameleon.h"

main () {

  Cham_Interp        * ci ;
  Cham_DataPointer   * data_pointer_one, data_pointer_two,
                       data_pointer_three ;
  Cham_Expression    * lisp_expr ;
```

```
  ci = Cham_CreateInterp ("emacs",
                          "pl122c.eecs.lehigh.edu:0.0") ;
/*
  because ci is an emacs-oriented, Chameleon interpreter,
Cham_Build creates an Emacs_Lisp object
whose value is the integer 7.
*/
  data_pointer_one = Cham_Build ("integer", ci, 7 ) ;
  data_pointer_two = Cham_Build ("string",  ci, "get busy baby" ) ;
  data_pointer_thr = Cham_Build ("list",    ci,
                                 data_pointer_one, data_pointer_two) ;
/*
    We want to create the Emacs Lisp expression:
(length '(7 "get busy baby"))
*/
  lisp_expr = Cham_BuildExpr ("length" , data_pointer_thr) ;

/* now send it off to the Emacs Lisp interp */
  Cham_Eval (ci, lisp_expr) ;

}
```

6.1.12 Interfaces to popular types of parallel/distributed computing languages are needed

In concept, any distributed programming language could coordinate computation among processes (virtual machines) of various languages. See [BST89] or [Bal90] for a comprehensive survey of distributed programming languages.

A process is a program in motion. Consider processes A and B written in sequential programming languages A and B respectively. If there is a method for these two process to communicate whether it be a file, hardware switching, or operating system message queueing, we have effected distributed computing.

This chapter examines how each major distributed computing paradigm [1] would use the Chameleon C library to specify the coordination of heterolingual processes in the solution of a trivial problem: have Emacs Lisp calculate the sum of 20 and 30 and have Tcl open up an X-Window displaying the solution.

Distributed data structure

A distributed data structure language is one in which multiple processes write to a shared data object for communication [Bal90]. Two popular distributed data structure languages are Orca and Linda. For more information on these languages see [Bra93b]. The current subsection will introduce Linda and compare it with Chameleon.

Linda perceives concurrent computing as having two tasks: computation within atomic process and coordination of communication among these processes, commonly known as the $C+C$ paradigm [CG89b]. After a brief description of Linda (the

[1]The imperative paradigm introduced in languages such as Occam is omitted as it is far too limited in terms of runtime process creation to support the dynamic nature of interpreted langauges.

reader is referred to [CG89a] for a detailed introduction to Linda and a comparision of it to object-oriented, logical, and functional approaches to concurrent computation), the author will continue to compare and contrast Linda with Chameleon.

Linda itself is not a programming language but simply a collection of routines which allow multithreaded manipulation of an object called the Tuple Space. A tuple is a sequence of scalar types such as integers, floats, and strings. Accordingly, a Tuple Space is a collection of such tuples. Processes may place keyed tuples into the Tuple Space and other processes make a blocking request for a tuple with a certain key. As long as a process can link into the Tuple Space (by adding the appropriate Linda libraries to its code) it can send and receive messages. This implies that processes of any language may send and receive tuples. Mitsolides shows how a Tuple Space is nothing more than a dictionary and queue in ALLOY [Mit91].

Major problems exist with the Tuple Space. For example, tuples may not be modified in place but instead must be removed from the Tuple Space, modified and replaced, resulting in high overhead for large tuples. Another problem is that mutually exclusive reading and writing of tuples is guaranteed only for single tuples, making it possible for conflicts to occur when the intent of two processes if two update a collection of tuples.

The Tuple Space is very similar to the Chameleon C library. The only difference is that the Tuple Space is capable of two-way communication whereas Chameleon only allows one way communication from the native language to the partner languages. In addition, since Linda only deals with Tuples, it does not require Chameleon's extensive data conversion capabilities — the burden of data conversion lies with the languages.

One nice feature of the Tuple Space is that tuples are *persistent,* meaning that delivering a tuple to another process does not necessarily destroy it. As a result, broadcasting is straightforward. The current version of Chameleon does not support saving a tagged linked list of data from a partner language. The unfortunate implication is that any data to be broadcast must be broadcast immediately.

Problem Solution (In C Linda)

```
#include "ChamInt.h"


/*
PROBLEM: Have Emacs Lisp calculate the sum of 20 and
30 and have Tcl open up an X-Window displaying the solution.
*/


main ()
{
  Cham_Interp * emacs_interp, tcl_interp ;


  emacs_interp =
        Cham_CreateInterp ("emacs", "pl122e.eecs.lehigh.edu:0.0") ;
  tcl_interp = Cham_CreateInterp ("tcl", (char *) NULL) ;


  Cham_Eval (emacs_interp, "(+ 20 30)") ;
  Cham_Coerce (emacs_interp) ;
  /*  "in" is a Linda command which places a tuple in the
```

```
Tuple Space. Note that the design of Chameleon does
not allow for maximal parallelism in Linda. Preferably,
we would like for the data conversion to take place in
a separate thread so that we could initiate the Tcl
process to wait on the results. */
  in  ("for_tcl", emac_interp->result) ;
  out ("for_tcl", result) ;
  sprintf ("button .output -text {%s} -command {exit}
            pack append . .output {top}", result) ;
}
```

Logic

Numerous logic languages have been developed. Perhaps the most recent and best-supported language in this spirit is Program Composition Notation (PCN), developed jointly between Caltech and Argonne National Laboratories. PCN is the third parallel computing language masterminded by Steven Taylor, the first two being Flat Concurrent Prolog and the other Strand88. It was developed in consultation with one of the world's foremost exports on theoretical parallel computing, Mani Chandy.

Program Composition Notation (PCN) is intended to serve as notation for the conditional, parallel, or sequential composition of atomic actions coded in PCN, FORTRAN, or C. The data conversion issue is sidestepped by requiring that all C/FORTRAN code be called by reference. As a result, arbitrarily complex data structures consist of a simple integer representing the memory address of the head of the structure. The semantics of the notation are extremely clear. PCN could serve as an excellent coordination mechanism for Chameleon. [CT92]

Problem Solution (In PCN)

```
/*
PROBLEM: Have Emacs Lisp calculate the sum of 20 and
30 and have Tcl open up an X-Window displaying the solution.
*/

{;
  {||
     emacs_interp =
        Cham_CreateInterp ("emacs", "pl122e.eecs.lehigh.edu:0.0"),
     tcl_interp = Cham_CreateInterp ("tcl", (char *) NULL),
     Cham_Eval (emacs_interp, "(+ 20 30)")
  }
  {||
     Cham_Coerce (emacs_interp),
     sprintf (string, "button .output -text {%s} -command {exit}
              pack append . .output {top}", emacs_interp->result),
     Cham_Eval (tcl_interp, string) ;
  }
}
```

Note how PCN is able to extract high parallelism from this code. In the first parallel block, both interpreters are created in parallel. This is hard to specify in Linda. Even though it would appear that the third statement would execute in parallel and before `emacs_interp` is bound, a property of PCN is to wait until all variables are bound in an expression before executing it. In the second parallel block, a process is created for data coercion while another is waiting to sprintf this evaluation into `string` and another process is waiting for `string` to be bound before having the string sent to the Tcl interpreter.

Functional

We use any nesting-parenthesis language as the functional language prototype.

Problem Solution

```
;;; PROBLEM: Have Emacs Lisp calculate the sum of 20 and
;;; 30 and have Tcl open up an X-Window displaying the solution.

(Cham_Eval
  (setq
    tcl_interp   (Cham_CreateInterp "tcl" NULL)
    emacs_interp (Cham_CreateInterp "emacs"
                                    "pl122e.eecs.lehigh.edu:0.0"))
  (+ 20 30))

(Cham_Eval tcl_interp
           (sprintf (string  "
                  button .output -text {%s} -command {exit}
                  pack append . .output {top}",
                 (result-field (Cham_Coerce emacs_interp)))))
```

The functional version is just as parallel as the PCN version assuming an intelligent compiler [2] In addition, prototyping and testing is very easy in functional languages thanks to the interpreted development environment.

Note how in the second parallel block the functions eval, sprintf, and coerce are in exactly the opposite order of the functions in PCN. The reason for this is that a functional language forces you to setup an expression so that it returns the value to the next function to operate on it. In contrast, PCN forces you to setup an expression so that the next function to operate on it shares a variable with the preceding expression. The low-level dataflow and function application is exactly the same but the high-level syntax is exactly the opposite.

Object-oriented

A very popular paradigm for distributed computing is the object-oriented paradigm [Bra93b]. The great interest in this field has generated a vast variety of languages with varying policies on inheritance, [3] object-mobility, [4] and orthogonality of process to object [5]. The arbitrary decision is to use a class-based Smalltalk-like language whose unit of parallism is the object.

Problem Solution (in AutoSmalltalk-80)

[2]The functional version may actually run slower because the `result-field` of the `emacs_interp` data structure would be accessed in constant time in PCN or C. The same operation in this functional language requires an additional function/data constructor call. Different functional compilers may not optimize this data access to the same constant time as PCN or C. If it doesn't then the function call overhead in addition to the function execution must be calculated into running time.

[3]prototype-based, class-based, or none

[4]The ability to move an object from one processor to another at runtime

[5]If orthogonality is low, then a process is created for each object, usually implicitly be the language. If orthogonality is high, then the user can assemble many objects into the same process space.

```
# PROBLEM: Have Emacs Lisp calculate the sum of 20 and
# 30 and have Tcl open up an X-Window displaying the solution.

tcl_interp   _ Cham_CreateInterp: "tcl"   onhost: NULL
emacs_interp _ Cham_CreateInterp: "emacs"
                           onhost: "pl122e.eecs.lehigh.edu:0.0"
emacs_interp eval: "(+ 20 30)"

string _ StringClass sprintf: "button .output -text {"
                      with:   (emacs_interp  result)
                      with:   "} -command {exit}"
                      with:   "pack append . .output {top}"
tcl_interp eval: string
```

This version is just as capable of concurrency as the others. However, this version has the overhead of object maintenance.

User-Level Software Bus

Glish is an event-oriented language which allows interpreted distributed computing. Both mapping (process and processor topology) and process creation are fully controllable at runtime.

[USE93]

Problem Solution (in Glish)

```
# PROBLEM: Have Emacs Lisp calculate the sum of 20 and
# 30 and have Tcl open up an X-Window displaying the solution.

tcl_interp   := client ("Cham_CreateTcl",   "remotehost.ca.us")
emacs_interp := client ("Cham_CreateEmacs", "remotehost.in")
$value[1] = 20
$value[2] = 30

emacs_interp->add ( $value )

whenever emacs_interp->operation_complete do
  {
tcl_interp->plot ( $value )
  }
```

Evaluation

A language for distributed computing addresses the issues of process creation, mapping, communication, and deletion. Chameleon address one issue in process communication: data conversion. Development of a new full-fledge distributed computing language for this one issue is ludicrous It would also be a shame to hardcode heterolingual functionality into a particular distributed computing language as no clear champion will ever exist.

As a result it is better that the issue of data conversion be fully orthogonal to

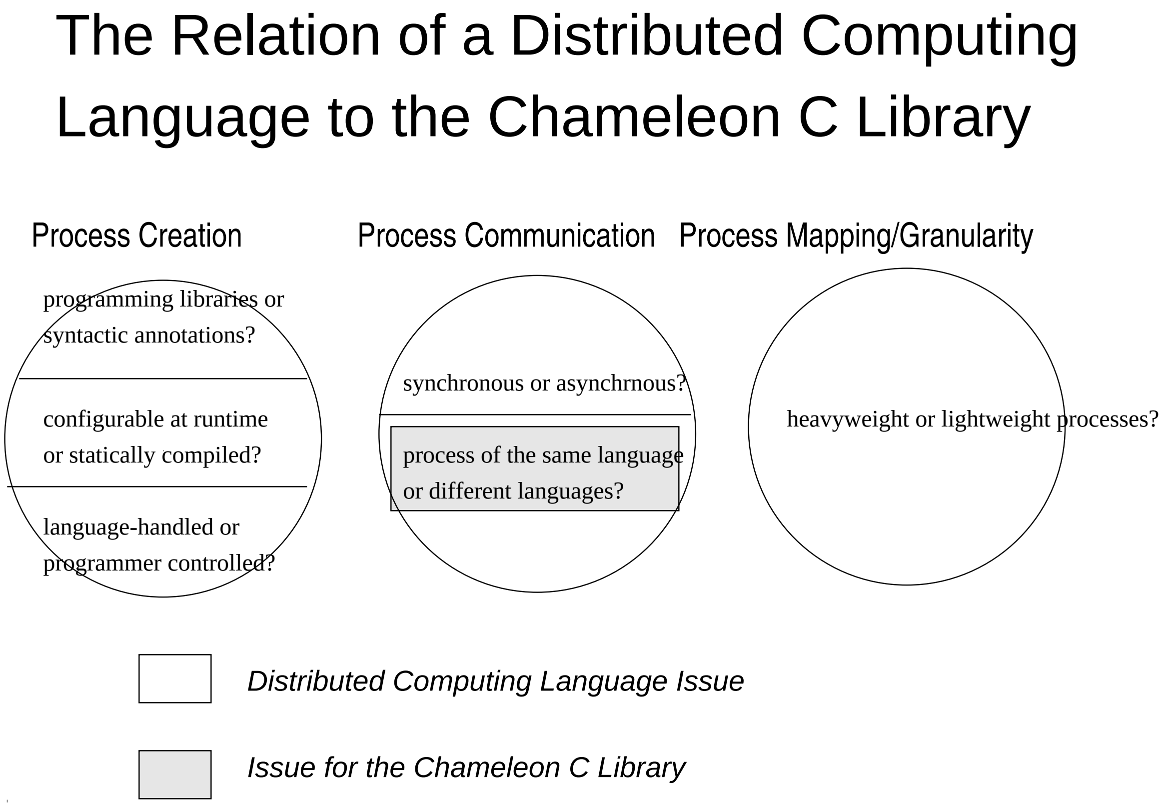

Figure 6.1: Chameleon is concerned with process creation in a very limited sense. Distributed computing languages have many more issues to deal with than Chameleon

distributed computing issues and create a library which will cleanly integrate with a number of popular distributed programming languages.

6.2 Prolog "Specification is Implementation" Solution

As stated in [War80], using Prolog to specify the existence of a sentence in a programming language is very nearly the implementation of translation of that sentence to a sentence in another programming language. For example, the following two Backaus-Naur form sentences:

```
<EXP> -> <TERM> '+' <EXP> [action: return $1 + $3]
<EXP> -> <TERM>           [noaction]
```

have succint translation into the following Prolog:

```
exp(Beg,End,R) :- term(Beg,T,Mida), input(Mida,'+',Midb),
                  exp(Midb,E,End), sum(T,E,R).
exp(Beg,End,R) :- term(Beg,End,T).
```

Another attraction of the Prolog approach is that it is a high-level declarative language, allowing for the easy addition of new conversion specifications and debugging of existing specifications.

The final attraction is that Prolog offers automatic data abstraction for any datatype immediately. For example, here is the Prolog representation for a Smalltalk dictionary: [6]

[6] A Smalltalk dictionary is a list of associations.

```
dictionary([]).
dictionary([association(Name,Value) | Tail]) :- dictionary(Tail).
```

An obvious shortcoming of using Prolog is that unless the Prolog implementation has structure-sharing optimizations [MW88], enormous amounts of data will be copied twice during conversion.

6.3 Compiler Solution

Given that compilers exist for many languages, it might be worthwhile to simply compile them all to C and link the resulting object code together. This approach has the following shortcomings

1. If the language doesn't support C callouts, then it would be tedious to allow Chameleon to convert data between the languages
2. Compilers would have to be written for each language

6.4 Hypertext/Multimedia Solution

A multimedia hypertext browser gives a single consistent user interface to a variety of media and protocols. [BL] It achieves this interface by parsing accessing various information-retrieval protocols [7] as well as the data formats of those protocols [8] Similarly, the Chameleon programmer API gives a single consistent interface to a variety of language interpreters which each have their own media (datatypes) and protocols (methods of returning data).

[7] FTP, Telnet, NNTP, WAIS, and gopher to name a few

[8] ASCII, GIF, Postscript, DVI, TeXinfo, and so forth.

Bibliography

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[Bal90] H.E. Bal. *Programming Distributed Systems.* Silicon Press, 1990.

[BL] Tim Berners-Lee. Www primer. This document is obtainable by use of the World Wide Web's Hypertext Transfer Protocol (http) at the following URL: http://www.vuw.ac.nz/non-local/gnat/www-primer.html. If you don't know what http and URL mean, then use the online introduction to the World Wide Web by telnetting to info.cern.ch and let a new world of information unfold in front of your eyes.

[Bra93a] Terrence Brannon. Chameleon: An embedded language perspective for programming in multiple computer languages. Master's thesis, Lehigh University, 304 Packard Lab No. 19, Decemeber 1993.

[Bra93b] Terrence Brannon. Survey of distributed object langauges. Partial fulfillment of Csc 403 Independant Study, December 1993.

[Bra93c] Terrence Brannon. Survey of quick interpreted languages. Posted irregularly to Usenet, 1993. Retrieve via anonymous ftp from archive.cis.ohio-state.edu in the directory pub/gnu/emacs/elisp-archive/packages/HyperActiveFTP/survey.tex.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):262–322, September 1989 1989.

[CG89a] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–58, April 1989.

[CG89b] Nicholos Carriero and David Gelernter. Author's response to criticism of linda in context. Communications of the ACM, October 1989.

[CM84] W.F. Clockin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.

[CNV88] Michael Covington, Donald Nute, and Andre Vellino. *Prolog Programming in Depth*. Scott, Foresman and Company, 1988.

[Coh85] Jacques Cohen. Describing prolog by its interpretation and compilation. *Communications of the ACM*, 28(12):1311–24, December 1985.

[CT92] K. Mani Chandy and Steve Taylor. *Introduction to Parallel Programming*. Addison-Wesley, Inc., 1992.

[DB93] John Davis and Terrence Brannon. personal communication. email to davis@amy.tch.harvard.edu, 1993.

[Eli92] Anton Eliens. *DLP: A Language for Distributed Logic Programming.* John Wiley and Sons, 1992.

[FOT92] Ian Foster, Robert Olson, and Steven Tuecke. Productive parallel programming: the pcn approach. Technical Report MCS-P295-0392, Argonne National Laboratory, Mathematics and Computer Science Division, April 1992. To appear in scientific programming.

[Goe] Wolfgang Goerigk. *CLiCC Manual Page.* Christian-Albrechts-Universitaet zu Kiel, Retrieve via anonymous ftp from ftp.informatik.uni-kiel.de in the directory pub/kiel/apply.

[HL92] P.M. Hill and J.W. Lloyd. *The Goedel Programming Language.* University of Bristol CSTR-92-27, University of Bristol, University Walk, Bristol BS8 1TR, 1992. Included with the Goedel source code.

[Hud92] Paul Hudak. A tutorial introduction to haskell. *ACM SIGPLAN Notices*, 27(5):1–58, September 1992.

[LaL93] Dan LaLiberte. *The GNU Emacs Lisp Reference Manual.* Free Software Foundation, 1993.

[LS91] Chu-Cheow Lim and Andreas Stolcke. Sather language design and performance evaluation. Technical Report TR-91-034, International Computer Science Institute, 1647 Center St., Suite 600, Berkeley, CA 94704, 1991.

[LSK+93] Sean Levy, Eswaran Subrahmanian, Suresh Konda, Robert Coyne, Arthur Westerberg, and Yoram Reich. An overview of the n-dim environment. Technical Report 05-65-93, Carnegie Mellon Engineering Design Research Center, snl@cmu.edu, 1993.

[Mac87] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation.* Holt, Rinehart and Winston, 1987.

[Mit91] Thanasis Mitsolides. *The Design and Implementation of ALLOY, a Higher Level Parallel Programming Language.* PhD thesis, New York University, Department of Computer Science, June 1991.

[MW88] David Maier and David S. Warren. *Computing with Logic: Logic Programming with Prolog.* Benjamin Cummings Publishing Company, 1988.

[Nor] Lance Norskog. *COOL (C Object-Oriented Library) Manual.* thinman@netcom.com.

[Omo93] Stephen Omohundro. The sather language: Efficient, interactive, object-oriented programming. *Dr. Dobb's Journal*, October 1993.

[Ous94] John K. Ousterhout. *The Tcl Book.* Addison-Wesley, 1994.

[scr] screen@uni-erlangen.de. Iscreen readme. /anonymous@prep.ai.mit.edu:pub/gnu/screen*.tar.gz.

[Tay89] Steven Taylor. *Parallel Logic Programming Techniques.* Prentice-Hall, 1989.

[USE93] USENIX. *Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems*, /anonymous@ftp.ee.lbl.gov:/glish, Winter 1993.

[War80] D.H.D. Warren. Logic programming and compiler writing. *Software - Practice and Experience*, 10:97–125, 1980.

Appendix A

Glossary

Tcl Tool Command Language developed by J.K. Ousterhout

Tk Tcl's greatly simplified yet powerful interface to the X11 toolkit.

alphabet A set of symbols

arguments Streams of zeros and ones which control the behavior of a command. Arguments may be *commands* or *data*.

command Streams of zeros and ones with action intent. A command is the half of the foundation of any language, data being the other half. Regardless of attempts to encapsulate it with data and call it an object or see it as a transformer of data in a functional paradigm or see it as black boxes on a datastream, or the facts, clauses and queries of a logic paradigm it remains syntax with the intent to achieve action. A command performs transformations on an arbitrary number of arguments which may themselves be commands or arguments depending on the power of the language. [Ous94]

data streams of zeros and ones with informational content.

language A language is a one-to-one mapping between syntactic rules and semantic interpretation.

language: **composition** The language which will be used to create and program instances of partner languages and to coordinate partner-to-partner and partner-to-native language communication.

language: **native** The language which wishes to access other, partner languages. Currently one of C, Emacs Lisp, Prolog, Smalltalk, or Tcl.

language: **partner** The language which is being accessed by the native language. Typically, the native language needs some functionality of the partner language which is lacking in the native language such as a math library of string processing routines. The current set of native languages is equivalent to the set of native languages.

semantic interpretation The "meaning" or intent of a sentence.

syntactic rules Valid sequences of vocabulary. An individual syntax rule is known as a *sentence.*

universal metalanguage programming The use of a syntactically simple, semantically clear, conceptual language whose conceptual semantics are mapped to the same syntax of one or more computer languages automatically. Take the example of getting a pointer to exactly enough memory to store an integer, and moving a five into that memory. This concept has numerous language syntax:

C `int * cptr = (int *) calloc (1,sizeof(int)) ;`

```
*cptr = 5 ;
```

Emacs Lisp `(setq cptr 5)`

Tcl `set cptr 5`

vocabulary The vocabulary of a language is the words it will accept.

widget A widget is a graphic window with certain capabilities. For example, a menubar widget is capable of displaying a menubar and returning the user's choice. A photo widget is capable of displaying digitized photographs. A text widget is capable of displaying and possibly editing text.

word Permutative collections of an alphabet

Appendix B

Obtaining the Packages Discussed

Ange FTP `/anonymous@spiff.gnu.ai.mit.edu:/ange-ftp`

CLiCC `/anonymous@ftp.informatik.uni-kiel.de:/pub/kiel/apply`

Emacs Lisp `/anonymous@prep.ai.mit.edu:/pub/gnu`

Goedel `/anonymous@137.222.102.102:/goedel`

PCN `/anonymous@info.mcs.anl.gov:/pub/pcn`

Prolog Numerous implementations of Prolog exist. The best public domain versions are available via anonymous ftp from swi.psy.uva.nl in pub/SWI-Prolog/pl-1.6.12.tar.Z and sbcs.sunysb.edu in pub/XSB/XSB.tar.Z

Sather `/anonymous@ftp.icsi.berkeley.edu:/pub/sather`

Smalltalk `/anonymous@prep.ai.mit.edu:/pub/gnu`

Tcl/Tk `/anonymous@harbor.ecn.purdue.edu:/pub/tcl`

Appendix C

Vita

Terrence Monroe Brannon was born in Columbia, SC on May 11, 1969. His interests include doing rap music and learning computer and natural languages. He has one younger sister, Cheryl Teresa Brannon. His parents, Sherley and Calvin Brannon are employees of Allied Corporation in Columbia, SC. Terrence's parents love him and do their very best to take care of him. After graduating from Irmo High School in May 1987, Terrence entered Morehouse College in August 1987. He completed his B.A. in Computer Science in May of 1991. He then entered Lehigh University in August 1991 and completed his M.S. in Computer Science in December of 1993. After a period of relaxation at home in Irmo, South Carolina from January 1994 to June 1994 and intensive immersion in Chinese language in Beijing, China from June 1994 to August 1994, he will attend Hahnemann University in August 1994 for the Ph.D in Neurophysiology.