AN ABSTRACT OF THE THESIS OF

John R. Placer for the degree of Doctor of Philosophy in Computer Science presented on November 4, 1988.

Title: G: A Language Based On Demand-Driven Stream Evalutions

Redacted for privacy

Abstract approved: ________________________________

Timothy Budd

A programming paradigm can be defined as a model or an approach employed in solving a problem. The results of the research described in this document demonstrate that it is possible to unite several different programming paradigms into a single linguistic framework. The imperative, procedural, applicative, lambda-free, relational, logic and object-oriented programming paradigms were combined in a language called *G* whose basic datatype is the stream. A stream is a data object whose values are produced as it is traversed.

In this dissertation we describe the methodology we developed to guide the design of *G*, we present the language *G* itself, we discuss a prototype implementation of *G* and we provide example programs that show how the paradigms included in *G* are expressed. We also present programs that demonstrate some ways in which paradigms can be combined to facilitate the solutions to problems.

G : A Language Based On Demand-Driven Stream Evaluations

by

John R. Placer

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed November 4, 1988
Commencement June 1989

APPROVED:

Redacted for privacy

Professor of Computer Science in Charge of Major

Redacted for privacy

Head of Department of Computer Science

Redacted for privacy

Dean of Gradutate School

Date thesis is presented November 4, 1988

Typed by John R. Placer for John R. Placer

Acknowledgements

I want to thank each of the members of my committee. Each of you made a unique contribution to my research efforts. Thank you Tim Budd for always displaying confidence in my work and for the special effort you made as my major advisor. Thank you Toshi Minoura for your friendship and for those long and challenging discussions. Thank you Bruce D'Ambrosio for knowing how to blend honest criticism with warmth and encouragement. Thank you Ted Lewis for your doubts; they pushed me deeper than I might otherwise have gone. Thank you Walter Rudd for the many ways you lent your support both as replacement committee member and as department chairman. I also want to thank Ralph Griswold who I had the good fortune to study under when I worked for my M.S. degree at the University of Arizona. Thank you Ralph for showing me how interesting and joyful the study of programming languages can be.

I want to thank each of the members of my family. Each of you supported the efforts that resulted in this dissertation in ways I can never fully express. Thank you Carol for your endless patience and encouragement. Thank you Carol, Jeff and Meredith for providing a background of love to whatever "work" I might be engaged in. Thank you mom for expressing support and encouragement in many different ways.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

A programming paradigm can be defined as a model or an approach employed in solving a problem, a definition suggested by Shriver [Shr86]. A programming paradigm represents a way of thinking about a problem, a way of modeling a problem domain. The ontologies represented by different paradigms are different. For example, the logic paradigm views the world as composed of predicates and relations while the functional paradigm models a world of functions, function composition and function application.

In general, the theoretical power of the languages that implement the various paradigms is the same. For example, if a problem can be solved in the logic paradigm in *Prolog*, then that same problem can be solved in the functional paradigm in *Lisp* or in the object-oriented paradigm in *Smalltalk.* But, as pointed out by MacLennan, "although it's possible to write any program in any programming language, it's not equally easy to do so" [Mac87]. For example, when solving problems that involve complex matrix arithmetic, *APL* and not *Prolog* is likely to be the language of choice. This is true not because *Prolog* is less powerful than *APL*, but because it is so much simpler to write solutions to matrix arithmetic problems in *APL.*

The recognition that paradigms can be applied to certain problems more easily than others has prompted new directions in the design of programming languages. The question is now being asked "Can we design languages that give programmers the freedom to choose among diverse paradigms?" Ghezzi and Jazayeri [GhJ87] have described the current directions in programming language foundations as consisting of the following three primary perspectives:

1. The various programming paradigms should be kept separate. The application should determine which approach is chosen.

2. One programming paradigm should be used in all applications.

3. An attempt should be made to integrate the various paradigms into one "uniform linguistic proposal".

Attempts to explore the feasibility of the third view expressed above form the basis of a relatively new area of current investigations called multiparadigm research. Hailpern [Hai86] has called this field of research "exciting and vital". He predicts that many cycles of experimental efforts followed by theoretical insights and consolidation will transpire before the research is mature and well understood. Shriver [Shr86] echos this enthusiasm when he predicts that "exciting times" are ahead of us as progress begins to be made in this research area.

The expression "multiparadigm language" has been used in the literature to refer to the languages that are being created as a result of this particular area of research [JGM86]. A multiparadigm language provides constructs that support more than one programming paradigm. According to Hailpern:

> "The ideal multiparadigm language would give the programmer the tools needed to solve each part of a programming task in the most natural and convenient way." [Hai87]

We agree with this assessment by Hailpern about what constitutes the ideal for multiparadigm languages. The challenge that naturally follows from this ideal is to create languages which integrate a maximum number of different programming paradigms in a consistent and natural framework. The research described in this document involves the creation of a multiparadigm language that combines a larger number of paradigms than have yet been combined into a single programming language.

Several paradigms which are referred to often in this document have been defined below for clarification. The general classification of these paradigms was taken from Jenkins, Glasgow and McCrosky [JGM86].

1. **functional:** The functional paradigm preserves referential transparency. There are two basic properties that define referential transparency. The first property is that all subroutines of a language must be true functions. That is, given a subroutine f and value x, the value of f applied to x must always be the same. The second property states that the value of a variable cannot be altered or modified in the middle of its scope (i.e. once a variable is assigned a value, it retains that value throughout its scope). Taken together these properties (i.e. referential transparency) imply that computation does not occur by the production of side effects but by defining functions which, for any given argument set, compute a unique value.

   Purely functional languages are also distinguished by their data structuring capabilities and by their ability to create higher order functions. The data structuring capabilities allow complex data structures to be passed as arguments or returned as results from expressions. A higher order function is either a function created by combining other functions or a function which returns another function when evaluated.

   The functional paradigm consists of the applicative and lambda-free paradigms.

   1a. **applicative:** The applicative paradigm uses function application and recursive function definitions. Each expression in this model can be broken up into components which are either operators or operands; the operators are applied to the operands. Pure *Lisp* [Wil84] is an example of a language based on this paradigm.

   1b. **lambda-free:** The lambda-free paradigm is a restriction of the applicative model to the use of two types of functional mechanisms. One type is used for applying a function to its argument, the other is used for creating and naming functional forms. Backus's *FP* [Bac78] is an example of a language reflecting this approach to programming.

2. **imperative:** The imperative paradigm is marked by fundamental use of commands such as assignment and flow control structures. The effects of the se-

quential list of commands that make up the imperative program combine to produce the desired computational effect. The language *Pascal* [SWP82] is an example of an imperative language.

3. **procedural:** This paradigm combines the imperative programming facilities described above with an abstraction mechanism to build procedures and functions. This mechanism allows the creation of generic commands. The language *C* expresses this paradigm. (Note that the imperative and procedural paradigms are often considered to be different aspects of one paradigm called the traditional or von Neumann paradigm.)

4. **relational:** The relational paradigm is based on a world of relations which in turn may be thought of as tables. Operators provided in this world of relations operate on old tables in order to create new tables. The relational model has become an important model for database systems. *SQL* [Dat86] is an example of a relational database programming language.

5. **logic:** Implicit in the logical paradigm is the existence of an underlying search mechanism. This paradigm emphasizes an incremental rule-based program structure and the logical variable. Logical variables allow the input and output specifications of relational expressions to be completely unspecified which in turn permits relations to be used in arbitrary modes. In addition to this, logical variables support the existence of partial data structures (i.e. data structures with unbound variables) and they support the binding of variables by the intersection of constraints. *Prolog* [StS86] is an example of a logic programming language.

6. **object-oriented programming:** Central to this paradigm is the notion of a world of objects organized into an inclusion hierarchy. The behavior of an object is determined by methods associated with the class of that object. Methods may be inherited from ancestor classes. *Smalltalk* [GoR83] is an example of an object-oriented language.

The approaches to programming outlined above represent well-established paradigms that have been targeted for integration in one combination or another by various researchers. The research described in this document has been directed toward the integration of all of these paradigms in a single programming language.

1.1 Motivations For Multiparadigm Research

There are a variety of interests and motivations that have provided the impetus for multiparadigm research. Shriver writes:

> "It is not clear that the paradigms in use today are either the right match to current and imminent technology or the right match to current and imminent user needs. We must develop new programming paradigms that lie beyond the ones currently studied and used in some degree of detail." [Shr86]

Jenkins, Glasgow and McCrosky [JGM86] created the multiparadigm language *Nial* mainly to use as a pedagogical tool for teaching different programming paradigms. Alan Kay, although he was writing in support of the "message-activity" model, actually offered a strong argument in favor of the development of multiparadigm languages for pedagogical purposes. In his now classic article that introduced Smalltalk and its object-oriented paradigm Kay wrote:

> "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." [Kay77]

Kay was actually making a case for a single general model that he felt was a superior paradigm, but we feel that his argument suggests that a linguistic framework that supports several powerful paradigms may well represent an excellent pedagogical tool. Novices instructed in the use of several programming paradigms, would be less likely to fall into the rigidity of thought that Kay described.

In addition to these pedagogical concerns, Jenkins, Glasgow and McCrosky also argue that another important practical reason exists for the creation of multiparadigm languages. They feel that combining several paradigms in one programming environment would result in an environment that directly supports the expression of multiple approaches to problem solving and would, therefore, represent a better environment in which to create large, complex software. The environment they envision would be made more feasible by an effective multiparadigm language. Hailpern supports this line of reasoning when he states that "multiparadigm systems are being created to give programmers the right tool at the right time [Hai86]".

Other investigators are trying to merge the capabilities of one paradigm with the special gains made in the understanding and utilization of another paradigm in order to extend the capabilities of a given language. In their paper in which they propose the extension of the functional language HOPE by the addition of unification, Darlington, Field and Pull write:

> "Techniques, such as graph reduction and data flow, have been evolved for the parallel evaluation of functional languages taking advantage of their simplicity of execution, and it would be advantageous if these techniques could also be used to support languages with the extra expressive capability of logic." [DFP86]

Lindstrom [Lin85], in his paper describing the extension of the functional language *FGL* with the logical variable, discusses how even the partial combination of programming paradigms can be beneficial. He asserts that even without the usual supporting features of logic programming (e.g. search and clausal programs), adding the logical variable to a functional language is worthwhile. The logical variable, he asserts, extends the range of efficient functional programming applications and provides a means by which functional programming can be utilized in a widened conceptual framework.

Yet another motivation found among multiparadigm researchers is the creation of a database programming language. *DSM* [Rum87] is a language that has been created by merging the object-oriented and relational models in order to create a database programming language. *DSM* is currently being used to write real applications programs. Korth [Kor86] has proposed the extension of relational (database) languages to include the functional and object-oriented paradigms. He argues that the resulting multiparadigm language would enable the relational model to be effectively applied to several areas outside of traditional "data-processing-style" applications. Korth lists computer-aided design databases, knowledge bases and user-interfaces as potential application areas for an extended relational model.

As indicated above, the benefits of multiparadigm research are already being reported as newly created multiparadigm languages begin to be used. Fukunaga and Hirose [FuH86] have reported on what they perceived as the significant advantage of unifying object-oriented programming and logic programming paradigms into a single language called *SPOOL.* They found that the knowledge representation capability of the object-oriented programming paradigm and the application independent inference mechanism of logic programming combined to yield a powerful "synergism". The benefits of the combined approaches was realized when *SPOOL* was used to write a non-trivial application program called *PROMPTER. PROMPTER* produces a higher-level description of the control program of an IBM operating system given its assembly language source code. Even though *SPOOL* is reported to need further linguistic support in order to exploit the full power of its combination of paradigms, Fukanaga and Hirose found it was useful for the development of *PROMPTER* in two major ways: the object-oriented framework of *SPOOL* greatly contributed to simplifying the architecture of *PROMPTER* and the logic capabilities of *SPOOL* helped clarify important ideas in the problem domain.

This discovery of the synergistic effect produced by a combination of paradigms is also reported by the creators of *CommonLoops* [BKK86]. They argue that the unification of object-oriented programming with the procedure-oriented design of *Lisp* resulted in something greater than the sum of the parts and that the mechanisms needed for integrating these two paradigms gave *CommonLoops* unexpected strength.

1.2 Previous Work

The field of multiparadigm research is quite young; there are a variety of efforts and approaches being attempted in order to learn how best to integrate diverse paradigms. Some researchers are working on interfaces to join languages of different paradigms, others are extending existing languages with the components of additional paradigms and a few are attempting the creation of wholly new languages. Regardless of the particular approach, however, a majority of the work done so far in the field of multiparadigm research has been focused on the integration of two or three well-established paradigms.

In particular, much effort has been directed toward the integration of the functional and logic paradigms. Examples of the results of this research include *Funlog* [SuY86,SuY84] and its extended unification algorithm, and the programming languages *TABLOG*[YoM86] and *LEAF*[BBL86]. These efforts have contributed insight into various characteristics of the logic paradigm such as non-directionality and the logical variable but they have not been attempts to produce broadly multiparadigm linguistic frameworks. In an analogous way, other limited attempts at paradigm integration have contributed to the understanding of characteristics of specific paradigms but have not been directed toward maximizing the number of paradigms woven into one linguistic fabric. Among the results of these attempts are, *Flavors* [Moo86] and *CommonLoops* [BKK86] which combine the applicative and the object-oriented paradigms, the language *SPOOL* and the Koschman and Evens [KoE88] language interface which combine the object-oriented and logic paradigms, and *DSM* [Rum87], mentioned above, which combines the object-oriented and relational paradigms.

The language Nial represents the only attempt known to us, beyond our own efforts, to approach the creation of the "ideal" multiparadigm language by combining several major paradigms in one language. Hailpern is also currently in the design stages of a project meant to integrate several paradigms, but this effort is being directed toward the creation of a multiparadigm "system" composed of several languages of different paradigms. *Nial* currently integrates the imperative, procedural, applicative and lambda-free approaches to programming[JGM86]. It does not support the logic and object-oriented paradigms, although research is currently being conducted that may allow a future version of *Nial* to offer extensions that support these other paradigms. These additional paradigms will not be directly supported by *Nial.*

1.3 Research Goals

As already noted earlier, Ghezzi and Jazayeri [GhJ87] described the current directions in programming language foundations as consisting of three primary perspectives. The third perspective, which embraces the field of multiparadigm research, was described in the following way:

> "An attempt should be made to integrate the various paradigms into one 'uniform linguistic proposal' ."

The questions raised by this third perspective are to what extent such an integration is possible and how can such an integration be realized. Harland has described this problem as a "challenge" to "devise a sufficiently expressive mechanism and then construct a fully integrated mixture of these paradigms [Har86]". This is the problem of central concern to the research described in this dissertation: "How can the major programming paradigms be combined within one 'unified linguistic proposal' ."

The structure of this dissertation generally conforms to the methodology we followed in order to answer the problem stated above. This methodology needed to produce both a language design and a demonstration that the design could result in a programming language capable of expressing the paradigms of interest. We have given below a brief outline of this methodology. Basically steps 1 through 4 correspond to the language design stage of our research and the remaining steps correspond to the implementation and demonstration of the language.

1. A single unifying datatype was chosen on which to base a new language design. The choice of this datatype was based upon the following criteria:

   (a) The datatype's ability to represent any arbitrarily complex data structure.

   (b) The datatype's compatibility with a dynamic environment and interpreted implementation.

   (c) The datatype's ability to support a simple "world view" with very few primitive functions.

   The basic datatype chosen was the stream. A stream is a data object whose values are produced as it is traversed.

2. The paradigms to be included in the language design were chosen. (The paradigms chosen were all of those defined earlier.)

3. The target paradigms were decomposed into their fundamental characteristics. These characteristics were analyzed with respect to the language semantics, structures and functionality that they implied and required. Furthermore, when necessary, the structures targeted for the language were "molded" into a form and interpretation compatible with and supportive of the "world view" implied by the fundamental datatype chosen for the language.

4. The remaining syntactic constructs necessary to directly support the essential characteristics of the paradigms were chosen and the language grammar and semantics were completed.

5. A prototype interpreter was implemented from the completed design.

6. Programs were written to demonstrate both how the individual paradigms are expressed and ways in which the paradigms may be integrated or mixed.

The execution of the steps outlined above resulted in the research presented in this dissertation. The details of each of these steps are described in the remaining chapters of this document. Steps 1 through 3 form the heart of our initial language design process whereby we combined an analysis of the paradigms of interest with a single unifying datatype that could embrace those paradigms. The details of steps 1 through 3 are discussed in Chapter 3 - *Developing the Underlying Structure of G.* Step 4 represents the final stage of the language design process where the language attributes and structures determined in the previous steps were united into a complete language specification. The results of the efforts made at this step are covered in chapter 4, *The Language G*, where details about the syntax and semantics of the final design of G are discussed.

The efforts made in the remaining steps of the outlined methodology resulted in a prototype implementation of the language G and in the creation of several programming examples. The details of the implementation of G are given in chapter 5 - *The Prototype Implementation of G.* The programs that were generated in step 6 to demonstrate the expression of the individual paradigms and some combinations of paradigms are presented and discussed in chapter 6 - *Expressing and Integrating the Paradigms of G.* Finally chapter 7 contains the conclusions we have reached concerning the research presented in this dissertation as well as a discussion of future work suggested by this research.

1.4 Research Results

The results of the research reported in this dissertation demonstrate that it is possible to unite several different paradigms into a single linguistic proposal. The chosen paradigms can be expressed both individually and in various combinations within the language G. Semantic issues were challenging during the language design period and yet the semantics of the completed language are surprisingly uncluttered; the syntax of G is simple and concise. The language G can be improved and expanded, it is not a "finished product." It does provide, however, a good framework on which to base further research and it does provide a good example of how the major paradigms can be united into one "unified linguistic proposal."

It was also our intention to create a small interpreter for G. This goal was also realized. The G interpreter consists of approximately 5000 lines of C code. Work is currently in progress on a second version of the interpreter which has already demonstrated significant reductions in memory utilization and response time. Recently we have been contacted by a research group designing VLSI based hardware support for streams; they have shown an interest in the language G. The possibility of implementing a version of G on specialized hardware opens up new potential for expanding our research.

All of this suggests that the design methodology developed for this research was appropriate to our stated goals and that it itself represents a contribution of the research described in this document.

Chapter 2

Related Work

This chapter discusses research efforts that in varying degrees are related to the work described in this dissertation. Some of the research efforts reported here have contributed ideas or techniques or both to the research and development that resulted in the language *G* while others have been related only in a peripheral way. Most of the sources cited in this chapter refer to projects with research goals different to one degree or another than our own, yet these projects often offered insight into programming language issues that were important to the research described in this dissertation.

2.1 Programming With Streams

There are a few languages based upon the stream, for example, *Lucid* [AsW77], *Seque* [GrO85,GrB85], *KRC* [Tur82] and *GRAAL* [BeR85]. None of these languages, however, are multiparadigm in the sense described in this document. Lucid has actually been called multiparadigm by Faustini and Lewis [FaL86] because it is a family of languages, each described by its own algebra. But this clearly is not meant in the same sense in which this document has used the term multiparadigm. Seque is an experimental language that resulted from ideas and notations developed by Ralph Griswold [Gri83]. *Icon* [GrG83], the language in which *Seque* is embedded, has had a considerable influence on the design of *G*. It was the power of *Icon* generators [GHK81,WaG81], integrated so completely into that language, which suggested that the stream could itself provide the fundamental data structure of a simple, compact multiparadigm language. Of the other stream-based languages mentioned above *GRAAL* is a language that implements the lambda-free (*FP* [Bac78] style) paradigm and *KRC* is a recursion equation language [Tur82].

2.2 The Functional Paradigm

Ideas developed by functional language implementors concerning the creation and manipulation of environments provided a major contribution to the implementation of *G*. These ideas derived mainly from work done in the *Lisp* community and are discussed in detail by several authors including Abelson and Sussman [ASS85] and Wise [Wis82]. Other helpful ideas offered by the *Lisp* community involved implementation of lazy evaluation. A good discussion of lazy evaluation can be found in Henderson [Hen80]. Brian Boutel's discussion of *ALICE*, a graph reduction computer, also offered some insight into how to efficiently handle primitive functions [Bou87].

The syntactic form of function definition used in *ISETL* [BDL87] was adopted for use in *G*. It can be used in a manner similar to the lambda form of *Lisp* [Wil84].

2.3 The Logic Paradigm

A number of researchers who are attempting to combine the functional and logic programming paradigms have written about problems associated with the logical variable. These researchers have discussed how the full range of power of the logical variable carries with it some serious performance difficulties.

Concerning the non-directionality or multimode character that logical variables provide logic programs, Uday Reddy has emphasized how different uses of the same relation can result in extreme differences in the amount of computational resources utilized by a language. In addition Reddy [Red86] writes that "... non-directionality also makes the operational behavior of logic programs hard to understand, and poses problems in developing parallel implementations of logic languages". He develops a notation for explicitly introducing directionality into logic programs in order to deal with this difficulty.

In an attempt to gain some use from this power but to limit its negative side effects Darlington, Field and Pull [DFP86] argue that because of the problems of implementation efficiency, some of the expressive power of logic programs should be reserved for specifications and not be supported at run-time. They present techniques for transforming these specifications into functions which can be executed more efficiently using only pattern matching and deterministic computation (i.e. by eliminating the logical variable).

In addition to these investigators, Abramson [Abr86] has warned about the problem of incorporating full unification into a functional language. He warns that bidirectional information flow during formal and actual parameter evaluation can require "delicate suspensions of function applications" until arguments have been sufficiently instantiated for evaluation.

Based on the results of these and other researchers, it seemed prudent not to initially incorporate the logical variable into *G*. Output variables are used in *G*, however, to provide some of the functionality of the logical variable. (Output variables are discussed in detail in chapters 3 and 4.)

2.4 The Object-Oriented and Relational Paradigms

Much work is being done today to unite the object-oriented paradigm with other paradigms. In particular, researchers are attempting to combine it with the relational model in order to work toward the creation of database programming languages.

Rumbaugh [Rum87] has made some helpful suggestions concerning the need for semantic and syntactic support for expressing relations directly in object-oriented languages. Several of the attributes needed to support the relation as a logical construct, as described by Rumbaugh, are provided in *G* as a natural consequence of its underlying stream semantics. For example, pattern-matching expressions and the basic stream semantics of *G*, which will be discussed in Chapter 4, provide simple query operation capabilities for membership testing and scanning a relation.

Rumbaugh has also suggested that relations should exist outside the type or class hierarchy of an object-oriented language. He argues that the relation should be a semantic construct equal to objects in an object-oriented structure. We have not found it necessary to move relations outside the type hierarchy of *G*. Relations are given special syntactic and semantic support and placed within the type hierarchy of G as a special case of the root type Stream.

The technique of using standard function invocation syntax for message passing is used in the language *Flavors* [Moo86] and in the language *Commonloops* [BKK86]. Each language interprets the function name of such an expression to be the method selector of a message-passing expression. This idea was used for built-in types in the language G as an aid in keeping the object-oriented structure of G invisible to the other paradigms included in G.

2.5 Incorporating Many Paradigms into a Multiparadigm System

Current work by Brent Hailpern to create a multiparadigm system offers some comparisons with our own work, although there are far more differences to be noted than similarities between the two projects. At a seminar he gave, Hailpern [Hai87] outlined some of the criteria that he has developed and is using for the development of a multiparadigm system meant to integrate the object-oriented, functional and imperative paradigms. Details are sparse in Hailpern's lecture notes, but he appears to be designing a "language" that is actually a system composed of other languages; each component language supports a paradigm to be included in his "multiparadigm language". G is, of course, a single language and is not composed of component languages. Our two projects are, therefore, starting from quite different ground.

It is possible, however, to find one significant similarity between our two "languages". Hailpern has chosen an object-oriented language, Emerald, to serve as the basis for his multiparadigm "system". The basic structure of G is also object-oriented. We feel that the fundamental structure of a multiparadigm language is well served by an object-oriented structure; it provides flexibility and easy extensibility to the language. Hailpern rejects, however, choosing a single powerful datatype as the foundation of his multiparadigm language. He feels that such a datatype would "optimize for one set of problems" and impede the flexibility of the language. We have found just the reverse. Our choice of streams as the fundamental datatype of G has supported flexibility and simplicity in our language design. All types within the G type hierarchy, including user-defined types, are descendents of the root type

Stream. This reflects a simple "world view" in which values of all types respond to the small set of stream primitives built into *G*. It should be noted here that we have been able to make user-defined types an integral part of the world of streams through an innovative interpretation of an instance of a user-defined type. The details of this interpretation are given in Chapter 3.

From this point on there seem to be few similarities between *G* and the language proposed by Hailpern. Where we have chosen to allow the basic design features of *G* support the use and creation of functions and functionals, Hailpern is experimenting with an extension to Backus's FP in order to manipulate functions which are maintained in a"well-defined function space". Again differences seem to arise between our two approaches in part because Hailpern's decisions are being made with respect to languages which are components of his proposed multiparadigm system whereas our research is directed toward the creation of a single linguistic framework.

2.6 Incorporating Many Paradigms Into One Language

Nial is a single linguistic framework that supports the imperative, procedural, applicative and lambda-free paradigms [JGM86]. *Nial* does not directly support the object-oriented, logic and relational paradigms. Jenkins, Glasgow and McCrosky do state, however, that current research is being done to develop "extensions" for logic and object-oriented programming [JGM86].

Another significant difference between the two languages stems from the fact that *Nial* is based on array theory which in turn is concerned with a universe of finite, nested, rectangular arrays. In this sense, *Nial* is a descendent of *APL* [GiR76]. Data values in *Nial* are, therefore, constrained to be finite whereas streams can accommodate infinite as well as finite data values.

G and *Nial* represent very different design methodologies. In the design methodology formulated to create the language *G*, we used the flexibility of a fundamental datatype, the stream, to allow us to mold the objects and entities of other paradigms into stream interpretations. In this way we were able to maintain uniformity with respect to how data values are treated and at the same time accommodate a diverse collection of paradigms. In addition, this methodology allowed us to restrict G to a small set of built-in primitives and it allowed us to choose a small set of concise syntactic structures with which to directly support the paradigms of interest. *Nial*, on the other hand, was strictly designed around array theory. "It uses the notations developed by More and colleagues for array theory as the basis for the syntactic constructs of the language." [JGM86] Since the array is general and flexible, several paradigms can be expressed within *Nial* by an appropriate selection of primitive functions. But the methodology of interpreting the elements of various paradigms in terms of a basic datatype and then choosing syntactic constructs to support those interpretations was clearly not used by the creators of *Nial*.

Chapter 3

Developing the Underlying Structure of G

This chapter describes the process by which the basic structures and attributes of the language G were chosen. There were several ideas that served as axioms on which we developed a methodology for determining the fundamental elements of G. We will discuss these guiding principles first and then discuss the details of the first three steps of our design methodology.

3.1 Guiding Principles

One of the basic axioms on which we based our language design strategy involved the choice of a fundamental structure for the language G. We felt strongly that we could achieve compactness, simplicity, and extensibility by utilizing an underlying object-oriented hierarchy for G. What was crucial to this idea, however, was the additional notion that an underlying object-oriented structure did not have to impose an object-oriented perspective on the expression of the other paradigms in the language. We avoided such an imposition by using a flexible fundamental datatype to provide an underlying semantics that helped unify the various paradigms. Furthermore, by making the fundamental datatype the root of G's type hierarchy, we made the object-oriented structure serve the basic datatype of the language. These ideas, of course, made it essential to chose a datatype with generality sufficient enough to provide the base for an "underlying semantics" that could embrace the paradigms of interest. The choice of this datatype was based on several criteria which are discussed later in this chapter.

Another important axiom of our design was the idea that we should not approach paradigms as atomic entities. Instead we decomposed the paradigms into their essential characteristics and then, whenever possible, treated those characteristics as though they were independent features. We made every effort to include as many as possible of the essential characteristics of the paradigms into the linguistic structure of *G*. Furthermore, the inability to integrate one characteristic of a given paradigm into *G* was not considered reason to eliminate that paradigm from inclusion in the language.

This idea of "unbundling" a paradigm was actually used, albeit in a somewhat more restricted context, by Lindstrom when he incorporated the logical variable into the language *FGL*. In his paper describing that extension, Lindstrom argued that the new ideas embodied in logic programming do not form a "monolithic semantic whole" but can be separated for individual consideration [Lin85].

The ideas discussed above suggested an approach to the design of a language with a single unifying semantic framework in which diverse paradigms could comfortably coexist. They suggested that a versatile underlying structure, a flexible fundamental datatype and supportive language attributes could provide the semantic "glue" necessary to bring diverse paradigms together in one programming language. These ideas were used to form the initial phase of a language design process. The steps of this process are summarized below.

1. Choose a single unifying datatype on which to base the new language. Base this choice upon the following criteria:
   (a) The datatype's ability to represent any arbitrarily complex data structure.
   (b) The datatype's compatibility with a dynamic environment and interpreted implementation.
   (c) The datatype's ability to support a simple "world view" with very few primitive functions.
2. Choose the paradigms to be included in the language.
3. Decompose the target paradigms into their fundamental characteristics. Analyze these characteristics with respect to the language semantics, structures and functionality that they imply and require. When necessary, "mold" the structures targeted for the language into a form and an interpretation compatible with and supportive of the "world view" implied by the fundamental

datatype chosen for the language.

These steps are discussed in detail in the remaining parts of this section.

3.2 The Choice of a Fundamental Underlying Datatype

The *stream* was chosen as the underlying fundamental datatype on which to base the design of *G*. A stream can be defined as a data object that is capable of producing values, where the values are produced on demand and constitute a sequence of values produced in time [GrO85]. Wise expressed this succinctly when he wrote "A stream can be pictured as a data structure which unfolds as it is traversed." [Wis82]. Stream programming can be used without assignment, it can be used to model systems that have state, and it can capture common patterns of data manipulation in concise abstractions. A good discussion of some of the intricacies of stream programming can be found in the book by Abelson and Sussman [ASS85]. A quote from their book gives some idea of the importance of this area of programming:

> "Perhaps the best that one can say at present is that time-varying objects and time-invariant streams both lead to powerful modeling disciplines. The choice between them is far from clear, and the search for a uniform approach that combines the benefits of both of these perspectives is a central concern of research in programming methodology."

The extreme simplicity and generality of the stream make it well suited to be the basic datatype of a language that integrates a large number of paradigms. Some of the main attributes that contributed to the decision to choose the stream as the fundamental datatype of *G* are discussed below.

1. *The stream can represent any arbitrarily complex data organization.*

   Streams can be used to organize data in any arbitrarily complex manner. Complex organizations of data with nested composite data values can easily be represented as streams of streams.

2. *The stream can support a simple "world view".*

   A datatype can imply the existence of elementary operations. For example, a stack datatype implies the existence of *push* and *pop* operations. In a language in which all objects are manifestations of a single underlying datatype, these elementary operators can be applied to all data values. If the operators are conceptually simple, few in number and computationally useful then an effective underlying semantics that links all values will have been established. In addition, the need to support only a small number of these basic operators helps keep the language design simple and compact. The stream is an excellent datatype with which to capture all of these qualities. The basic operators implied by the stream datatype are few in number; we defined only four. We need operators (1) to inquire what the current value of a stream is, (2) to move a stream on to its next value, (3) to inquire what is the index of the current value of the stream, (4) to concatenate streams together, (5) to inquire what particular type or manifestation of a stream a value is and (6) to implement a special enumeration protocol. Each of these basic operators is conceptually quite simple. A similar structure (the list) has already demonstrated how computationally useful the stream might be. Thus the stream embodies the criteria we listed above.

3. *The stream is compatible with a dynamic environment and an interpreted implementation.*

   The stream datatype allows dereferenced structures to be recycled with relative ease thus supporting dynamic memory management. This is much in line with attributes of functional languages based on the list datatype and is discussed by several others including Abelson and Sussman [ASS85] and Wise [Wis82].

3.3 Choosing the Paradigms

The paradigms that were chosen for inclusion in the language *G* were defined in chapter 1 and are listed below:

1. functional (applicative and lambda-free)

2. imperative

3. procedural

4. logic

5. relational

6. object-oriented

This is a list of programming paradigms that have been used extensively for several years and as such have shown themselves to be useful and powerful aids to problem solving on the computer.

3.4 Decomposition of the Paradigms

In the phase of the design process described in this section each of the paradigms was decomposed into its principal characteristics. Each characteristic was in turn matched to those language structures and attributes needed to support it. When necessary to maintain semantic unity, language structures were "molded" into forms compatible with and supportive of the general stream semantics of G.

3.4.1 The Functional Paradigm

Each of the fundamental characteristics of the functional paradigm is given below. Each characteristic is followed by a brief discussion of the language attributes and functionality that were chosen in order to support that characteristic.

1. *Expression-based semantics.*

   Expressions were chosen as the basic currency of the language G. This caused no conflict with the other paradigms.

2. *A hierarchical structure of expressions with components that may be decomposed into operators (possibly recursive) applied to operands.*

   Supporting such a structure necessitates both a mechanism for *applying* functions to their arguments and a function definition mechanism that permits recursive function definitions. Both of these mechanisms are provided in G.

3. *Data structuring capabilities and the ability to create higher order functions.*

   All values in G are guaranteed to be first-class values. This is all that is needed in order to assure that the language has data structuring capabilities and the ability to create higher order functions. Data structuring capabilities guarantee that structured or composite values can be used in the same way as any other value in a language. Structured values should be able to be passed as function arguments, returned as values by functions or used in assignment even when assigned as components to other structures. In an analogous manner, a language that can accommodate higher order functions will permit functions to be passed as function arguments, returned as values by other functions and assigned as values to variables. All of these capabilities are provided in a language that guarantees first-class status of all values.

4. *Referential transparency.*

   Referential transparency (defined in Chapter 1) can be supported in a number of ways. Two ways included in the basic structure of G are the parameter passing mechanism adopted for G and the assignment protocol enforced by G.

   Either a call-by-value or a call-by-need parameter passing mechanism would have been supportive of referential transparency. Since the stream was chosen as the fundamental datatype of G, however, the call-by-need protocol became the preferred mechanism. Call-by-need is really just a call-by-value mechanism realized in the context of lazy evaluation.

   In order to prevent aliasing as a consequence of assignment it was decided to enforce an assignment protocol that would preclude aliasing. Values are always copied before they are assigned. A copy of the right-hand-side value of an assignment expression is always assigned to the left-hand-side variable of the assignment. This allows single assignment to be used with no threat to referential transparency. It should also be noted that since it was decided to make G an expression-based language, the assignment operation is interpreted to be an expression that always returns the null value sequence. In this way an assignment statement does not contribute to the value sequence in which it is written but is used solely for its side effect of assigning a value to a variable. This interpretation of assignment is an example of "molding" a language feature into a form compatible with and supportive of the general stream semantics of *G*.

It should be noted that the parameter passing mechanism and the assignment protocol given above do not, by themselves, guarantee referential transparency; these mechanisms only serve to narrow the number of ways in which it can be violated. In order to guarantee referential transparency it also would be necessary to enforce the prohibition of destructive assignment. To build this prohibition into the linguistic framework, however, would also prohibit a prime feature of the imperative paradigm. Such prohibition was not, therefore, built into the structure of *G*. Expression of the "purely" functional paradigm in *G* does rely on the user choosing not to utilize destructive assignment in *G* programs.

3.4.2 The Logic Paradigm

Each of the fundamental characteristics of the logic paradigm that were identified are given below. Each characteristic is followed by a brief discussion of the language attributes and functionality that were chosen in order to support that characteristic.

1. *An underlying interpreter that supports search-based computation.*

   There was little question that our language would need to be interpreted. This characteristic did emphasize, however, that some search-based computational mechanism would need to be a part of the interpreter. The generate-and-test search mechanism is certainly an intuitive part of the semantics of any language based upon generators or streams. But far more efficient search strategies can be employed by the *G* interpreter when relations are used instead of general structured data types like the tuple. An important point to note here, however, is that the user can simply view the underlying search functionality in a simplistic way and know that every value is a stream and, therefore, every value can potentially participate in computations that involve search.

The actual structures that were chosen to support search-based computation were pattern-matching expressions (i.e. values of type Pattern) and conjunctions. Both of these structures are discussed in detail in chapter 4. In passing we do wish to mention, however, that the language *QBE* (see [Dat86] for an introduction to query-by-example) and the mapping operations of SETL [SDD86] readily suggested the syntactic form given to values of type Pattern. This decision was particularly influenced by consideration of the extreme ease with which relational queries can be expressed in *QBE.*

2. *Unspecified input and output specifications of relations.*

   What is needed to support this characteristic is some mechanism by which information can flow bi-directionally in and out of streams. What was chosen to support this need was a special type of variable called an *output* variable. Output variables are understood to be initially uninstantiated. The binding of output variables to values is coupled to the underlying search process built into the interpreter. The output variable is explained in detail in Chapter 4.

3. *The existence of data structures with unbound variables.*

   To fully realize this characteristic requires the functionality provided by unification. It was decided not to utilize unification in *G* for reasons discussed in Chapter 2. It is possible, however, to use output variables in conjunctions to allow expressions with unbound variables to exist in the form of *Prolog*-like rules. These structures are discussed in Chapter 4.

4. *The binding of variables by the intersection of constraints.*

   This characteristic also calls for a form of uninstantiated variable that can acquire bindings through the search mechanism built into the interpreter. Variables of this type must be able to be used more than once within an expression.

In addition, the semantics of such expressions must guarantee that the initial bindings of values to uninstantiated variables will be coupled to the underlying search mechanism and that these bindings will be retained for the remaining period of computation of that expression. The semantics of conjunction expressions in *G* was defined in exactly this way in order to support the binding of variables by the intersection of constraints.

5. *An optional incremental rule-oriented program structure.*

   Given the facilities already mentioned, *G* has the functionality necessary to realize incremental rule-oriented program structures. The conjunction expression of *G* can be used to create *Prolog*-like rules and the assignment mechanism can be used to "name" these rules. In this way rules can be created and named such that they form an arbitrarily complex hierarchy of rules built one upon the other. (The details of constructing such rules are given in chapter 4 and examples are included in chapter 6).

3.4.3 The Relational Paradigm

The relational paradigm models a world of relations (or tables) with operators that map relations to relations (or tables to tables). Relations are provided as first-class values in *G* and built-in functions that operate on values of type Relation are also provided. Stream values of types other than type Relation may also represent tables, therefore, in general the relational paradigm is easily expressed in *G*.

3.4.4 The Imperative Paradigm

The imperative paradigm is marked by fundamental use of commands such as assignment and flow control structures. As mentioned earlier an assignment mechanism was chosen for incorporation into *G*. The assignment mechanism does allow destructive assignment which is a prime feature of the imperative paradigm of programming.

Flow control operators have unique interpretations in *G*. The sequential operator (the comma) actually has a dual role in *G*; it is also the main constructor operator. This has resulted in there being no differentiation between a stream specification and a program in *G*. In a sense, therefore, there are no programs in *G*, just stream expressions.

Special constructs called code bodies were created in order to represent the additional control flow features of iteration and selection. Basically code bodies are interpreted to be value sequence specifications. They are syntactic entities used to specify repeated value sequences or alternative values sequences within a stream specification. Code bodies are discussed in chapter 4.

3.4.5 The Procedural Paradigm

The procedural paradigm utilizes an abstraction mechanism to build procedures and functions and allows the creation of generic commands. All of the facilities necessary to realize this paradigm have already been mentioned. A function definition mechanism which permits recursive definitions was incorporated into *G*. This can be combined with the assignment mechanism of *G* to permit named functions and procedures to be created in *G*.

3.4.6 The Object-Oriented Paradigm

Each of the fundamental characteristics of the object-oriented paradigm are given below. Each characteristic is followed by a brief discussion of the language structures and attributes that were chosen in order to support that characteristic. It is worth noting that, where necessary, elements of the object-oriented paradigm incorporated into *G* have been given novel forms or interpretations compatible with the underlying stream semantics of *G*.

1. *An inclusion hierarchy composed of types or classes.*

   G is provided with an inclusion hierarchy of 10 built-in types and with a facility for creating user-defined types. The root of the type hierarchy is the type *Stream*. All other types thus became a subtype of Stream and, therefore, can be interpreted to be special manifestations of streams. To help prevent an object-oriented perspective from being cast onto other paradigms expressed in *G*, the creation of instances of built-in types is assumed simply by typing the symbolic representation of that type. Instances of user-defined types, however, must be created with the built-in primitive *make.*

It should be emphasized that the built-in types in *G* are distinct from the underlying types (lower category objects) they mirror. For example, the type *Int* in *G* is considered to be populated by all single element streams whose sole element is an integer. Thus when the symbol `53` is encountered in *G* it represents a stream whose single element is the integer fifty-three. The reason for interpreting all types as actual streams, distinct from the more primitive types or objects implied by them, is to provide semantic consistency throughout the language. This interpretation allows any of the primitive stream operators to be applied to instances of any of the built-in or user-defined types. For example, it is as legitimate to ask for the current value of the stream `53` as it is to ask for the current value of the stream `[1,2,3]`. Thus any primitive stream operator can be applied to an identifier in a *G* program and a legal operation is always guaranteed to result regardless of the value of that identifier at the time of computation.

2. *The existence of a world of objects.*

   All *G* values can be viewed as objects which are instances of *G* types. Instances of built-in types, however, are "protected" from the outward protocol that is normally associated with the object-oriented paradigm. The two ways in which this is accomplished are listed below:

   (a) Each built-in type has a special syntax that allows its instances to be recognized by the interpreter without the use of a special creation primitive.

   (b) The message passing syntax associated with all built-in types is identical to standard function invocation syntax.

This "protects" other paradigms, which must use the built-in datatypes, from having the object-oriented approach to programming imposed on them.

Instances of user-defined types, however, are subject to some of the outward protocol of the object-oriented structure. They must be created with the primitive operator (*make*) and a special message-passing syntax must be used when they are passed messages.

User-defined values, however, have also been "molded" into a form that is very compatible with and supportive of the underlying stream semantics of *G*. The two major components of an instance of a user-defined type are:

(a) *An interface expression.*

The definition of a user-defined type includes an optional interface value; this can be any stream value, for example, a variable name, a *Prolog*-like rule or a general tuple expression. This interface value becomes the visible value of an instance of a user-defined type. Application of stream primitives to an instance of a user-defined type would result in those primitives being applied to the interface value of that instance. If no interface value has been defined then the empty stream is simply returned. We developed the idea of an interface value for user-defined types in order to make instances of user-defined types completely compatible with the underlying stream semantics of *G*. The application of any stream primitive to any instance of a user-defined type will always result in a legal operation. An example of how this interface value can be used is given in chapter 6 in the section *Integrating the Paradigms.*

(b) *Instance variables.*

Instance variables defined for a type are used to form the local environment of instances of that type. In other words, when an instance of a user-defined type is created, it contains a local environment which is populated with copies of the instance variables defined for that type. Instance variables may or may not have initial values associated with them. This local environment of the instance then forms the local environment in which methods are evaluated when that instance is passed messages.

3. *Message-passing and methods.* A special message-passing syntax was provided for user-defined types. Methods are function definitions that have been defined and associated with types. When an instance of a type is passed a message, the method associated with that message is evaluated within the local environment of the instance that received the message.

4. *Inheritance.*

   The property of inheritance is extended to both instance variables and methods defined in ancestor types. All instance variables inherited from ancestor types are combined into the local environment associated with an instance of a type.

Chapter 4

The Language *G*

This chapter discusses the specification of the language *G*. This specification resulted from uniting all of the language attributes, structures and functionality determined in the language design process discussed in chapter 3. But there was also much work done in deciding the final set of syntactic features with which to express the attributes that needed to be incorporated into *G*. It is not really enough to simply determine through analysis what needs to be combined into a language. A language must also "feel" right, which is to say there are human factors that are difficult to quantify but which are very important to language design. We engaged in a lengthy process of experimenting with a variety of syntactic devices in order to finally decide upon a set of structures that not only expressed the functionality needed in *G* but which also "worked well together" and gave the language a comfortable feeling.

4.1 Overview

The language *G* is an expression-based language whose basic data structure is the stream; there are no statements in *G*. Every expression returns a value and every value is a stream. All values are first-class objects able to be passed as function arguments, returned as elements of streams, assigned to variables and used as components of other streams. *G* is also an interactive language. A session with the language *G* consists of typing *G* expressions at a terminal. After an expression is entered the value sequence of that expression is enumerated and printed. *G* utilizes a demand-driven evaluation protocol; values are evaluated only when they are needed.

It is important to note that in *G* a stream is composed of both a sequence of values and a mechanism for enumerating those values. A stream is, therefore, a generator of values. In *G* the different types generally share the same enumeration protocol. This protocol enumerates the elements of a stream one-at-a-time, from left-to-right, as they are needed.

Operator Symbol	Operator Name
@	next operator
~	current operator
||	concatenation operator
and	and operator
type	type operator
index	current position operator

Table 1: The Primitive Stream Operators of *G*

4.2 Primitive Operators

There are six primitive stream operators in the language *G* that apply to every type. As noted earlier, every value in *G* is a stream. The primitive *G* operators reflect this fact and bring together the different types of *G* as separate cases of one general type, the stream. Every type must include definitions of the six primitive operators either in the methods associated with that type or in the methods of one of its super-types. The six primitive operators are shown in Table 1. Associativity and precedence information for these operators can be determined from the grammar given in Appendix C.

Streams of any type can be made to enumerate a value from their value sequence by application of either the at sign (@) or the tilde(~). The at sign is the "next" operator; it requests its operand to enumerate the current value in its value sequence and then to advance the stream to the next element. The tilde is the "current" operator; it requests its operand to enumerate its current value but not to advance the stream. For example, assume that the following expression was given to the *G* interpreter:

```
aList := [4,5,13,8]
```

In this expression the variable *aList* has been assigned the value [4,5,13,8]. Application of the "next" operator to *aList* (@*aList*) results in the value 4 being returned,

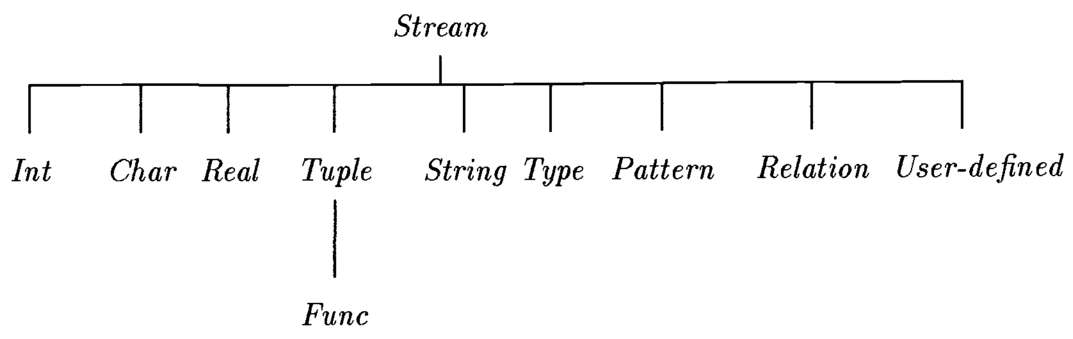

Figure 1: Type Hierarchy of the Language G

but if executed subsequent times in a loop, the values 5, 13 and 8 will be returned in that order.

The other primitive operators are *concatenation* ($||$), conjunction (*and*), *type* and *index*. Concatenation joins two streams into a single combined stream while conjunction joins streams and imposes a special enumeration protocol on the joined streams. The *type* operator returns the type of its argument stream and the *index* operator returns the position of the current element of the stream.

4.3 The Hierarchical Structure of G

G is organized around a tree structure. Each node in this tree represents a type in the language G. There are ten predefined types in the G hierarchy. The basic tree structure of G is shown in Figure 1.

A parent node in this figure is called the super-type of its children. A child node is called a sub-type of its parent. The domain referred to by each of the type names shown in Figure 1 is given in Table 2.

TYPE-NAME	DOMAIN
Int	integers
Char	characters
Real	floating point numbers
String	strings of characters
Type	type values
Tuple	any mixture of types
Func	any mixture of types
Pattern	any mixture of types
Relation	values with a fixed arity and types

Table 2: Types and Their Associated Domains

4.4 Types

Data types are divided into two categories : scalar and structured data types. Scalar types include types Int, Char, Real and Type. Structured data types are defined to be types which may contain more than one value in their value sequence. There is no restriction on the number of values that can make up a structured data type (i.e. infinite streams are permitted in the language *G*). In addition to this, the types Tuple, Func and Pattern may contain any mixture of types. Each of these stream types has one or more special attributes which will be discussed in subsequent sections of this chapter. It is important to note at the outset that every type in *G* is a stream. The different types of *G* merely represent a convenient classification of streams.

4.4.1 Scalar Types

The types Int, Real, Type and Char are called scalar types. When an integer, floating point number, type value or character is encountered in a *G* expression, it is interpreted to be a single element stream which responds to the primitive stream operators in the same way that any single element stream would respond. An example of each scalar and its notation is given below:

`345`	-	a stream whose single element has the integer value 345
`23.4`	-	a stream whose single element has the floating point value 23.4
`'e'`	-	a stream whose single element is the character *e*
`Int`	-	a stream whose single element is the type value representing integers

When considering the examples given above, it is important to note that each example represents a stream and not simply a single value. This means that each of the examples represents a generator of a single value and responds appropriately to the primitive operators of *G*. For example, application of the "current" operator to integer 345 (~345) would return the single element stream 345. The following example further illustrates this point. Consider the following sequence of expressions given to the *G* interpreter in the order shown. The comment to the right explains the effect of each expression.

`t := 5`	-	t is assigned the stream value 5
`~t`	-	the first value (5) of t is enumerated
`@t`	-	the first value (5) of t is enumerated and t is advanced
`~t`	-	stream t is exhausted; end-of-stream is returned

4.4.2 String

String is a special form of structured data type in *G*. Each of the elements of a value of type String is itself of type Char. Strings have a dual nature. A string may be used as an integral unit, for example in a lexicographic comparison with another string, or it may be regarded as a stream of individual characters which have no implicit association beyond their membership in the stream. To accommodate this dual nature of strings, they have been given their own unique type. In this way, functions that treat strings as integral units and functions that treat strings as less closely coupled elements can both be easily provided in the methods assigned to type

String or through inheritance in the methods provided in type Stream. Double quotes are used to delimit strings in the language *G*. For example "12" would represent a string whose first element is the character '1' and whose second element is the character '2'. Application of the primitive stream operators to any string works in the expected manner. For example the expression `~''hello''` returns the value 'h'.

4.4.3 Tuple

Streams of type Tuple may contain any number and any mixture of values. The value sequence of a given tuple is simply the enumeration from left-to-right of the values denoted within the tuple. The value sequence is always delimited with square brackets. The individual components of a tuple can be listed explicitly or indicated through the use of implicit sequence denotations called code bodies. Code bodies are not themselves streams but are sequence constructors; they have meaning only in tuple brackets. Regardless of the manner in which the values of a tuple are specified, implicitly, explicitly or both, a tuple is always delimited with square brackets. Shown below are two examples of tuples with explicitly denoted value sequences.

```
[1 , 'e' , 45.6 , ''hi'']
```

```
[ 'a' , [1, 'e'] , Int]
```

There are six basic categories of code bodies. These categories are:

1. Range Specification
2. Control Structures
3. Assignment
4. Local Declaration
5. Break Expressions
6. Recursive Function Calls

Regardless of the form, a code body is used to specify a sequence (possibly empty) of values. Range specifications implicitly specify a homogeneous sequence of values of type Int or type Char. The four notations for range specifications are given below.

1. `[n..m]` The values between 'n' and 'm' inclusive.

2. `[n..]` The values between 'n' and the end of the sequence defined for the type of 'n'.

3. `[n..m step k]` The values between 'n' and 'm' skipping k-1 values after each enumeration.

4. `[n.. step k]` The values between 'n' and the end of the sequence defined for the type of 'n' skipping k-1 values after each enumeration.

In each of the forms of range specification shown above, both 'n' and 'm' must be either of type Int or of type Char while 'k' must always be of type Int. Some examples of range specifications are given below.

`[1..10]`	-	is equivalent to `[1,2,3,4,5,6,7,8,9,10]`
`['a'..'e']`	-	is equivalent to `['a','b','c','d','e']`
`[1..]`	-	represents the stream of integers from 1 to infinity
`['A'..'G' step 2]`	-	is equivalent to `['A','C','E','G']`

There are four forms of code bodies that serve both as sequence constructors and as control structures. These forms are the *while*, *foreach*, *repeat* and *if* constructions. *While*, *foreach* and *repeat* constructions allow their tuple bodies to be repeatedly enumerated. They serve, therefore, as iterative control structures. *Repeat* constructions simply enumerate repeatedly the value sequence of their body. For example, the expression `repeat[1,2,3]` represents the infinite sequence `[1,2,3,1,2,3,...]`.

A *foreach* construction will enumerate the value sequence of its body once for each element of its argument that it is able to enumerate. Variables may be inherited by a *foreach* body from the global environment if they are not overridden by identifiers listed in a local declaration expression within the body. The values that the argument to a *foreach* construction will enumerate may be named by prepending the name to the argument value and separating the name and value by a colon. For example, in the expression

```
[foreach(nums:[1,2,3])[nums,'#']].
```

the variable *nums* will be assigned the value 1 on the first iteration of the loop, then the value 2 on the next iteration and finally the value 3 on the last iteration. The variable *nums* is local to the body of the *foreach* construction. The sequence described by the example above is `[1,'#',2,'#',3,'#']`.

A *while* construction will enumerate the value sequence of its body each time its argument expression returns a result other than the empty stream (i.e. each time it is successful). Variables may be inherited by a *while* body from the global environment if they are not overridden by identifiers listed in a local declaration expression within the body. The argument to a *while* construction may be named in the same manner described above for *foreach* constructions. In the following expression

```
[while(a < 5)[a:=a+1,1,2,3] ].
```

the sequence "1 2 3" will be enumerated each time the conditional within the *while* argument is successful.

If constructions enumerate the value sequence of one of an arbitrary number of alternative tuple bodies. An *if* construction will enumerate the value sequence of the tuple body associated with the first argument that is not at the end of its value sequence. If none of the arguments satisfy this condition and if an *else* clause exists, the body in that clause is enumerated otherwise the empty sequence is enumerated. For example, consider the following expression:

```
if(s1)[t1]
elif(s2)[t2]
else[t3]
```

The meaning of the above expression can be expressed this way:

> If an element can be enumerated from s1 then construct sequence t1 else if an element can be enumerated from s2 then construct sequence t2 else construct sequence t3.

Global variables may be inherited by the selected body of an *if* construction if they are not overridden by an identifier named in a local declaration expression within the body.

As a more concrete example, consider the following expression which constructs the repeating sequence `[1,2,3,1,2,3,...]` if 'a' is not at the end of its value sequence, otherwise it constructs the value sequence `[4,5,6]`:

```
[if(a)[repeat[1,2,3]] else[4,5,6]].
```

The above expression is another example of an infinite or potentially infinite stream. In order for such streams to be useful they need to be assigned to a variable. Since all computation is G is lazy, assigning an infinite structure to a variable causes no difficulty. After the assignment, values may be enumerated from that infinite structure one at a time as needed.

A *break* expression is interpreted to be a code body which returns the end-of-stream marker and so does not contribute to the value sequence of a stream in which it is found. It is used only for its side effect of terminating sequences. A *break* will immediately terminate the value sequence of any stream or code body in which it is encountered. It should also be noted that a *break* encountered at any arbitrary level of nesting of code bodies will terminate all code bodies up to and including the top level. If the *break* is encountered in a code body, then the value sequence denoted by that code body is immediately terminated. If the *break* is encountered as an explicit element of a tuple or function, then the value sequence of the stream represented by that tuple or function is immediately terminated. As an example of how *break* may be used, consider the following expression:

```
[foreach(a:[1,2,3])[if(a = 3)[break] else['a','b']]].
```

This expression represents the stream `['a','b','a','b']`. For the first two elements of the argument to the *foreach* code body, the argument to the *if* code body fails and so the sequence given as the *else* body is constructed. The last value of the *foreach* argument causes the *if* argument to succeed and the *break* to become part of the sequence. At this point the *if* and the *foreach* code bodies terminate their sequences.

Local declarations and assignments are also examples of code bodies that return only the empty sequence and which are used solely for their side effect. Local declarations enter variables and optionally their default values into the local environment. When a local declaration is encountered in a tuple, function or sequence constructor body, it has the side effect of declaring all of the identifiers listed within it to be local to that tuple, function or code body from the point at which the local declaration expression is found until the end of the tuple, function or code body. It should also be noted that a global identifier will be inherited if it is not preceded by a local declaration construction.

Assignments associate a copy of their right-hand-side value with the variable given on their left-hand-side. If the variable given does not exist in the visible environment, it is entered as a local variable and a copy of the right-hand-side value is associated with it as its default value. The following example shows both the assignment and local declaration constructions. It represents the value sequence `[3,2]`.

```
[local[x:1,y:2],x:=3,x,y].
```

The final code body to be discussed is the recursive call construction *self*. This code body is only permitted within function bodies and will also be discussed in the next section on functions. The *self* code body constructs a value sequence which depends on the function it refers to and on the arguments it is being passed. Examples are deferred until *self* is discussed in the following section on functions.

Below are given four more examples of values of type Tuple. Note that square brackets always delimit the values of a stream of type Tuple.

`['w' , 34 , "hello" , 23.5]`	-	a simple heterogeneous stream
`[ [23 , 'r'] , "green"]`	-	note here the nesting of streams
`[ 1..3,'a'..'d']`	-	or `[1,2,3,'a','b','c','d']`
`[foreach("ab")[1..3]]`	-	or `[1,2,3,1,2,3]`

4.4.4 Func

The type Tuple has a single sub-type, the type Func. We will call streams of type Func functions. A function is a structured data type which contains a tuple body but that has additional functionality beyond that of a simple tuple. Functions are really just parameterized stream expressions. The value sequence of a function is defined by the value sequence of its tuple body. Functions have the following special properties:

1. Functions allow parameters to be passed to a stream "by need".
2. Functions allow default values to be associated with parameters.
3. Functions allow recursive stream definitions. This is another way in which streams with infinite value sequences can be defined.

A function definition consists of a function header followed by a tuple. A function header consists of the reserved word *func* followed immediately by an arbitrary number of parameters separated by commas and enclosed in parentheses. Arguments may be referred to in the body of a function by the same name they were given in the formal parameter list as in most programming languages. Functions with an arbitrary number of arguments may be defined by specifying a function definition with no arguments and then referring in the body of that function to the special variable *args* which will refer to the entire stream of parameters passed to the function. If the list of formal parameters of a function definition is left empty, therefore, the only access to the arguments passed to an instance of that function would be through the special variable *args*. As a simple example of a function, consider the following expression of a function definition.

```
double := func(a)[2*a].
```

Note that the function is named in this example with an assignment operation. The assignment inserts the function definition into the visible environment. The built-in function *addop* could have been used to insert the function definition directly into the

G hierarchy. This last point will be discussed in a subsequent section. The function assigned to *double* simply doubles the value of its single argument. The function call `double(4)` simply returns the value `[8]`.

Consider now the following function that makes use of a code body:

```
doubleall := func(s)[foreach(s)[s*2]].
```

The function on the right-hand-side of the assignment returns its argument stream with each of its values doubled. For example, the function call

```
doubleall([1,2,3])
```

would return the value `[2,4,6]`. Now let us look at an example of a function that accepts an arbitrary number of arguments.

```
dbl := func()[foreach(args)[args * 2]].
```

The function in this example will accept an arbitrary number of arguments (not enclosed in brackets) and return each argument value doubled. Thus the function call `dbl(1,2,3)` will return the value `[2,4,6]`.

Parameters are local to a function, they are passed "by need". Local identifiers may also be declared by including a local declaration construction in the tuple body of the function as explained earlier. Global identifiers will be inherited from the environment in which a function call occurs if they are not overridden by identical names in the parameter list or by names listed in a local declaration construction within the function body.

A function definition can be made recursive by using the special variable *self* which indicates a call to the same function being defined. As discussed earlier, *self* is a code body which constructs a sequence determined by the function it refers to and the arguments it is passed. The following recursive function definition for factorial illustrates the use of *self*:

```
factorial := func(n)[if(n<=1)[1] else[n * [self(n-1)]]].
```

Notice in the example above how the recursive call is delimited with brackets. This is necessary since a recursive call is defined as a code body (i.e. a value sequence specification) and not a stream value. This presents no difficulty with the multiplication expression in which the recursive call is embedded since the arithmetic operators are vector functions.

Nameless functions can be created in a manner similar to that of lambda functions. The example below shows an example of a nameless function being called with the arguments 2 and 3. This function call returns the value 5.

```
func(a,b)[a+b](2,3).
```

In the example above, `func(a,b)[a+b]` is the nameless function and `(2,3)` is the argument to the function.

4.4.5 Relation

Another special form of structured data type is the relation. Relations are streams whose order of enumeration is determined by the implementation. This allows storage techniques to be utilized that permit efficient search strategies (e.g. hashing) to be used with relations. Such efficiency can become important when relations are used in the formation of *Prolog*-like rules. The creation of rules as a *G* programming technique will be illustrated later. Relations can be declared in *G* with the following special syntactic form:

```
#type1,type2,...,typeN#
```

Each element of a declared relation is constrained to contain the same number of components of the types designated in the declaration. The only types permitted for field values of relations at this time are scalar types and the type String. The above expression would normally be used in an assignment in order to assign a value of type Relation to a variable. For example, in the following expression, a relation of pairs of values each of type String is assigned to the variable *father*.

```
father := #String,String#.
```

The following special built-in functions have also been provided to support values of type Relation.

```
insert(var,avalue)
delete(var,avalue)
```

If the value of the variable *var* is not of type Relation in either case above then the empty stream is returned. The *insert* function will only insert *avalue* into a given relation if it conforms to the arity and type constraints that were given when the relation was created. In any other case it simply returns the the empty stream. If *avalue* does not exist in the given relation, then *delete* returns the empty stream otherwise it removes *avalue* from that relation and returns the value of *avalue*.

4.4.6 Pattern

Values of type Pattern allow a user to specify a stream to be enumerated and a pattern to be matched against values within that stream. These expressions are composed of a variable followed immediately by a pattern delimited with square brackets. Any type can be used as the base stream of a value of type Pattern. Values of the named stream are enumerated if they match the entire pattern specified. For example, the expression `foo[>10]` represents the stream of all integer elements of stream *foo* that are greater than the value 10. Values of type Pattern can be thought of as filters that delete all values of the underlying base stream that do not conform to the given pattern.

Output variables are special variables that may be used within values of type Pattern and that act as wild cards. These variables greatly enhance the power of the pattern-matching capability of such expressions. Although logical variables are not supported in *G*, output variables offer a helpful subset of their functionality. Output variables permit a restricted form of bi-directional information flow to and from streams and they allow variables to bind by the intersection of constraints. Furthermore, although the output variable does not provide the full utility of partial data structures (since unfettered non-directionality is not permitted) output variables do permit a limited type of partial data structure in the form of conjunction expressions which contain output variables. A simple example of a pattern-matching expression that uses an output variable would be the following expression:

```
fee[?word,>''Sam'']
```

This expression represents all values contained in stream *fee* that have two elements the second of which is lexicographically greater than the string "Sam". Each time one of these elements from stream *fee* is generated, the output variable *?word* will be assigned the value of the first component of that element. Note that output variables always begin with the '?' character. Pattern-matching expressions are goal-directed expressions and may be used to transform streams into expressions that can be used like *Prolog*(StS87) if-then rules in logic programming. This capability allows a form of programming with constraints and will be discussed in the section *And Expressions.*

The following example gives a quicksort using a recursive function definition and values of type Pattern as arguments to the recursive function calls.

```
qs := func(s)[local[x],x:=@s,if(x)[self(s[<x]),x,self(s[>=x])]]
```

Notice how the recursive calls specify two implicit value sequences that are to be constructed on each side of the value of x. The first sequence only involves a sort of values less than x while the second sequence involves a sort of values greater than or equal to x.

4.4.7 User-defined Types

The last form of structured data type available in *G* is the user-defined type. The following syntax declares a user-defined type in *G*.

```
addtype{name,supertype,expression,local[...]}
```

The built-in operation *addtype* installs a new type named *name* into the *G* hierarchy as a subtype of *supertype.* The expression given as the third argument to *newop* is "visible" to users and represents the stream that will be used for pattern-matching and other stream operations applied to an instance of the user-defined type. The list of locals given within the brackets preceded by the keyword *local* represents the instance variables associated with each instance of the user-defined type. These variables are "hidden" from the world in which an instance of a type exists (i.e. they can only be directly accessed by functions associated with the type itself).

Methods (functions) to be associated with a user-defined type may be introduced with the following built-in function:

```
addop{aType,''fname'',funcdef}
```

addop associates with type *aType* the given function name *fname* and function definition *funcdef*.

In order to create a new instance of a newly defined type the *make* function is used. The following example shows the syntax of an expression that returns a new instance (named *me* by assignment) of the user-defined type *Utype*.

```
me := make{Utype}
```

Finally the syntax for passing a message *fname* to the instance *me* is given below. We have assumed here that the method associated with message *fname* takes no arguments.

```
me::fname()
```

We see here that the double colon is a special lexical token that represents message passing to user-defined types.

The following example demonstrates how to define a new type named *Stack*, how to associate a method with that new type, how to create an instance of the new type and how to make function calls associated with that type. Consider this first expression:

```
addtype{Stack,Stream,stack,local[stack:0]}
```

In this expression a user-defined type name *Stack* has been defined and inserted as a sub-type of the built-in type Stream. An interface *stack* has been defined. This means that the value of an instance of type *Stack* will be determined by whatever the current value of variable *stack* is for that instance. Note also that the variable *stack* has been declared as a local variable of each instance of type *Stack*. This means that its value can be "seen" by users of an instance of type *Stack* and its value can be modified by methods associated with type *Stack* if such methods have been defined. Notice also that every new instance of type *Stack* will have its instance variable *stack* initialized to the value 0. Now consider the following method declaration.

```
addop{Stack,''push'',func(val)[stack:=val||stack]}
```

In the above expression a method named *push* has been defined and associated with the user-defined type *Stack*. Note that *push* when passed as a message to an instance of type *Stack* will have the side-effect of assigning a new value to the local (instance) variable *stack* of that object.

Below we have created an instance of the user-defined type *Stack* and named it *myStack* through an assignment expression.

```
myStack := make{Stack}
```

Now we will add the value 25 to the instance variable associated with the value of *myStack* by executing the following message passing expression:

```
myStack::push(25)
```

If the instance *myStack* is now handed to the *G* interpreter by simply typing the variable name *myStack*, the value `[25,0]` will be printed.

4.5 Libraries

Libraries can be maintained as files of simple *G* expressions. Any *G* expression is allowed in a library file although assignments or installation of user-defined types and methods into the *G* type hierarchy would normally be all that is included in a library file. These library files can be integrated into the *G* environment from the *G* interpreter by using the built-in function *include{filename}*. For example, assume that we have a library of function definitions for string analysis in the file named *stringlib*. When these functions are needed, the user can simply type the expression `include{stringlib}` and the library would be read into the current environment. There are several libraries provided in the appendices to this dissertation.

4.6 Function Call Semantics and Inheritance

The hierarchical type structure shown earlier is the basis of function call semantics in *G*. Each type in the *G* hierarchy is represented by a global variable of the same name that is maintained by the *G* interpreter. Each of these global variables has associated with it the name-definition bindings of functions associated with the type named by that global variable.

Function call semantics in *G* are similar to that used by object-oriented languages such as *Smalltalk* [GoR83]. The type of the first argument of a function determines where the *G* interpreter will begin to search for the meaning of that function. When a function name is encountered within a *G* expression, the methods associated with the type of the first argument of the function are searched for the function's name. If the function's name is found, then the associated function definition is applied to the arguments of the function. If the function's name is not found, then the search for the function's name resumes beginning with the parent type (super-type) of the type just searched. This process continues until either the function's name is found and the function definition is applied to the given arguments or there are no more super-types to search. At this point, however, *G* departs from *Smalltalk's* search protocol. If the appropriate function was not yet been found, a search is made of the environment for a variable of the given name whose associated value is a function definition. If this last search is successful, the defined function is applied to the arguments of the function call. If it was not successful, an error message is printed by the interpreter. This search protocol allows sub-types to "inherit" function definitions from super-types and it allows function definitions to be added to the environment by assignment of a function definition value to a variable.

For example, consider the *G* expression `foo(1,2)`. The *G* interpreter will begin its search for the name *foo* in the functions associated with type Int. If the name *foo* is not found, then the search will resume with the functions associated with type Stream. If this search is unsuccessful then the environment is searched for a variable *foo* whose value is a function definition. An appropriate error message would be printed if none of these searches is successful. In this way, both the type hierarchy of *G* and the environment of a variable can be used to define the meaning of a function in a function call. In the example given above, if either of the following two *G* expressions had been executed before the *foo* function call, the call would have been successful.

```
foo := func(a,b)[a+b].
```

```
addop(Int,''foo'',func(a,b)[a+b]).
```

4.7 Scope

Dynamic scoping is used in the language *G*. Understanding the scope of identifiers in *G* is easy when it is understood what the possible sources of identifiers are in *G* expressions and what the scope of identifiers from those different sources are. The sources from which identifiers may enter an expression are :

1. The global environment
2. Local declaration expressions
3. Function arguments
4. Pattern-matching expressions

The scope of identifiers from each of these sources is discussed in the following sections.

4.7.1 The Global Environment

A stream or sequence constructor may inherit identifiers that exist within its global environment. The name of a global identifier is always hidden from a stream or sequence constructor, however, by an identifier of the same name in a local declaration expression in the stream or sequence constructor. The masking of the global variable would only occur from the point of occurrence of the local declaration expression to the end of the stream or constructor body. Arguments to functions also mask global variables of the same name.

4.7.2 Local Declaration Expressions

Tuples, functions, and sequence constructors may include local declaration expressions as elements. These expressions introduce identifiers as locals to the stream in which they occur and may even denote the initial values of these new local variables. When a local declaration expression is encountered in a tuple, function, or code body, it has the side effect of declaring all of the identifiers listed within it to be local to the body that contains it from the point at which the local declaration expression is found to the end of that body. Local declaration expressions return the empty sequence and, therefore, do not contribute to the value sequence of which they are a part. Given below is a typical local declaration expression in a value of type Tuple.

```
[local[x:1,y:''hi''],x,y].
```

In the expression above two variables, x and y, have been declared as local to the tuple body. The variable x has been initialized to the value `1` and the variable y to the value "hi". The value of the entire tuple is `[1,''hi'']`.

4.7.3 Function Arguments

When an identifier is an argument to a function, that identifier becomes local to that function. In other words, the parameter passing mechanism of functions is *by need.* The value of the identifier passed as a function argument is not altered by that function. The only exception to this rule is when values of type Relation are passed as arguments. Relations are considered to be persistent objects whose values should not be copied when passed as function arguments. A new "view" into the relation is created when a relation is passed to a function so that it may be enumerated and searched without disturbing other uses of that relation, however, the values themselves within the relation are not copied. If a value is inserted into or deleted from that relation, computations involving other variables whose value is that relation can potentially be effected.

4.8 Built-in Functions

This section discusses more of the built-in functions provided by the G interpreter. The concatenation and *and* operations are considered to be primitive stream operations that may be applied to any G values.

4.8.1 Conditional and Arithmetic Operators

The standard set of arithmetic and conditional operators are provided by the *G* interpreter. Conditional expressions produce the rightmost term of the conditional expression if the conditional is successful otherwise they produce the empty stream. As an example consider the expression `10 < 20`. This simple *G* expression would return the value `20`. The expression `20 < 10` would produce the empty stream. Arithmetic expressions perform as expected. If a scalar is matched with a non-scalar, however, the operation is applied to the first value of the structured data type. For example, the value of the expression `2 + [1,2,3]` is `3`.

4.8.2 Concatenation Expressions

The concatenate operator (`||`) allows two values to be concatenated together. Since every value in *G* is a stream, this operator can be used legally with any *G* type. Consider the example below where a tuple, a function call and a string are all concatenated together.

```
[1,'z'] || func(a,b)[a+b](1,3) || ''no''.
```

The result of the above expression is the stream `[1,'z',4,'n','o']`.

4.8.3 And Expressions

The *and* operator has a special enumeration protocol that can be used to form complex pattern-matching expressions. These expressions allow the user to construct *Prolog*-like rules using conjuncts of type Pattern. The enumeration protocol of an *and* expression can be expressed in the following way:

> For each member of the Cartesian product of the conjuncts of an *and* expression, the value of the last conjunct is returned.

As a simple initial example consider the following *and* expression:

```
[1,2] and "hi".
```

The result of this expression is `['h','i','h','i']`. For each value of the first conjunct each of the values of the second conjunct are enumerated. Now consider a slightly more complicated example. Assume that a relation named *father* has already been created in the current *G* environment. The following expression would then assign to the variable *gf* a tuple that represents the grandfather relation:

```
gf := father[?gdad,?dad] and
      father[?dad,?gson] and
      [[?gdad,?gson]].
```

Notice how the binding of output variables in the above expression subjects that expression to certain constraints. Use of the same output variable in several subexpressions constrains each location of that variable to bind the same value. This then adds further constraints upon each pattern-matching expression within which the variables are embedded. Also notice that the last conjunct is double bracketed. The result of an *and* expression is the concatenation of all values it returns. It is necessary, therefore, to delimit any values that must remain associated within their own stream. *And* expressions are goal directed. Each time a value is requested from an *and* expression, the next successful element of the stream represented by that expression is produced.

4.8.4 Miscellaneous Built-in Functions

There are two built-in functions not mentioned previously which are used in examples later in this dissertation. These are the functions *write* and the function *random*. The function *write* is a simple output primitive that writes its argument values to the standard output. The function *random* takes a single integer argument and returns an integer in the range of 1 up to and including the value of the argument.

4.9 Summary

G is a language of streams and is based on expressions. *Streams* are values and all values are first-class objects in the language *G*. A stream is composed of both a sequence of values and a method for enumerating those values. The basic enumeration protocol of all streams (except *and* expressions) is enumeration of the values of a stream in left-to-right order, one element at a time *upon demand.* This demand-driven character of all *G* expressions allows streams with infinite value sequences to be represented in *G*. The language *G* is organized around a tree of types. Each type has associated with it a set of name-definition bindings. This hierarchy of types allows function call semantics similar to the object-oriented language *Smalltalk* and supports inheritance of function definitions and instance variables. Values of type Pattern and *and* expressions are *goal-directed.* They allow streams to be searched for values that match a given pattern. Pattern-matching expressions can be used to create expressions that act like if-then rules in Prolog.

Chapter 5

The Prototype Implementation of *G*

This chapter describes the prototype implementation created from the language design discussed in earlier chapters of this dissertation. In order to implement any language based on streams, it is necessary to use some form of lazy evaluation technique. The functional programming language community has written about and experimented with such techniques [Wis82,ASS85,Hen80]. The implementation of *G* was aided by these earlier efforts, however, *G* directly supports many paradigms and has some unique features. In general, therefore, the unique computational model of *G* and its large combination of paradigms demanded an implementation strategy with some unique elements.

The prototype implementation of *G* is composed of approximately 5000 lines of *C* code. The Unix tools *yacc* and *lex* were used to create the parser and lexical analyzer for the *G* interpreter. In the implementation, a *G* value is always represented by a value descriptor, the general form of which is shown in Figure 2. The *type* field contains a pointer into the *G* type hierarchy, the *value* field is dependent on the specific type and value represented by the value descriptor and the *next* field is used to link values together to form streams of multiple values. A special internal value descriptor is used to represent the end-of-stream (*EOS*) marker. The last value descriptor in a stream of values will always have its *next* field set to the *EOS* descriptor.

5.1 The Type Hierarchy

A small representation of the data structures used to implement the *G* type hierarchy is shown in Figure 3. The *type table* is an array of pointers, one pointer for each *G* type. Each pointer in the type table points to a data structure that records the string name of a type, the methods associated with that type and the parent or super type of that type. In addition to this information, an optional interface value

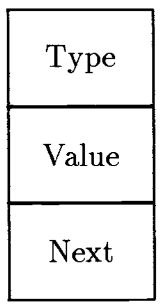

Figure 2: A *G* Value Descriptor

and/or an optional local environment is recorded where appropriate for user-defined types. Notice in Figure 3 how the methods associated with a type are recorded in a list of nodes. Each node contains the name of a method, a pointer to the function definition associated with that name and a pointer to the next method node in the list. Also note that the data structure associated with type *Stream* is the only structure whose *parent* field is nil.

5.2 Environments

Environments are represented in the implementation by a chain of nodes (which may be empty), each node of which contains the name of a variable, the value associated with that name and a pointer to the next environment node in the chain. The form of environment nodes is identical to the nodes shown in Figure 3 that record the function name and definition associations in the methods lists for types in the *G* hierarchy. When a change is made to a local environment, a new environment node is created and placed at the start of the list of nodes that represents that local environment. Complex data structures that contain local environments (e.g. tuples and functions) always maintain two pointers to the list of nodes that represent their local environment. One pointer is called the *static* pointer; this pointer always points to the initial set of environment nodes with which a *G* value was created.

The other environment pointer maintained for structures that have local envi-

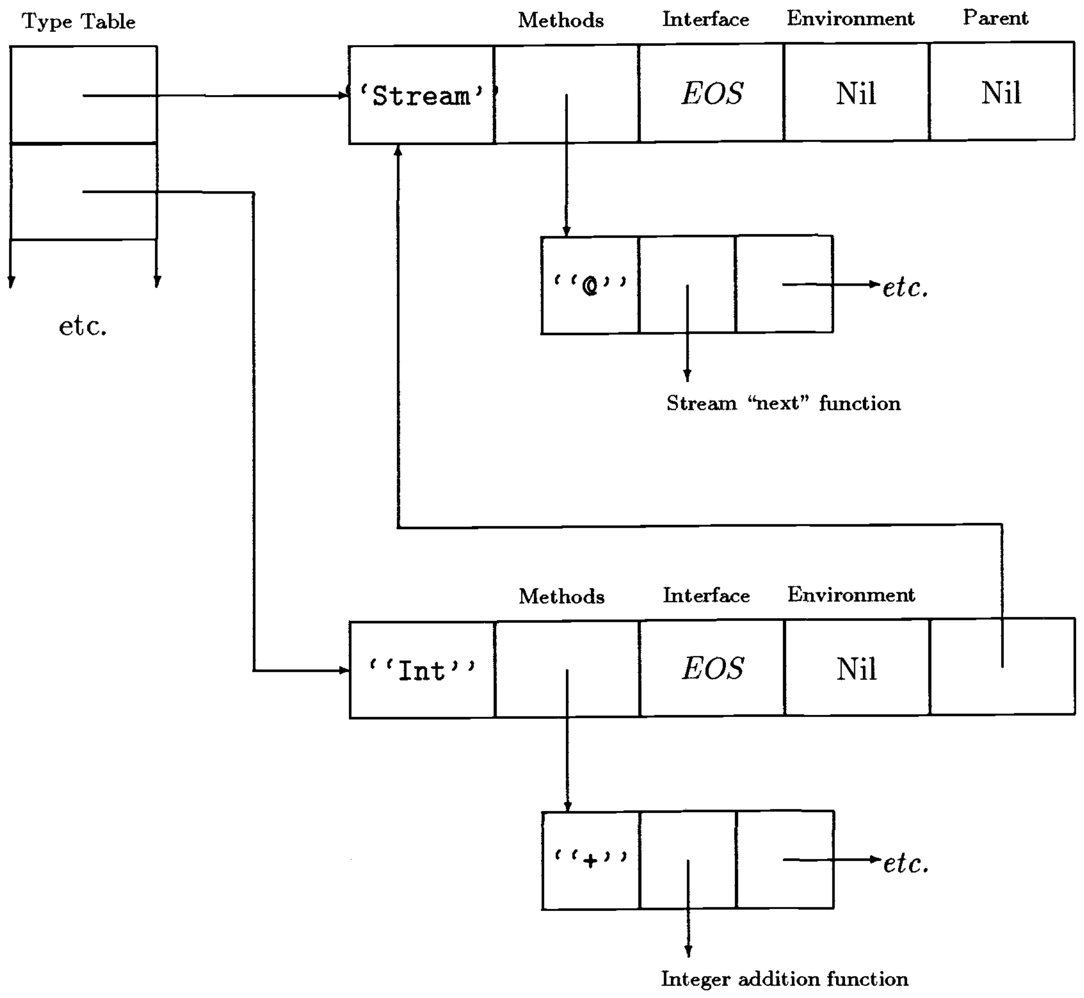

Figure 3: Data Structures Used in the *G* Type Hierarchy

ronment is called the *dynamic* pointer. When a change occurs in a local environment, the dynamic pointer is pointed at a newly created node that represents the change to the environment, and the new node is set to point at the head of the list of nodes that represented the local environment before the change was made. Thus the dynamic pointer is changed each time a node is added to an environment, but the static pointer is never changed and always points at the original head of the list of nodes that represent a local environment. Internally it is sometimes necessary to refresh stream values. All that needs to be done to refresh a stream value is to set the dynamic pointer back to the node pointed to by the static pointer.

A novel technique is used in *G* to create closures. A closure in *G* refers to a data value whose environment may only be altered locally. For example, when a complex structure is assigned as a value to a variable a copy of that value is made and is marked "closed". This means that the value has been taken out of whatever environment in which it was originally embedded and it may no longer alter environments global to that original environment. Arguments to functions are also copied and marked as closed values. An argument value is also a value that is to be used in an environment other than that in which it originated. It may not be allowed, therefore, to alter environments associated with its original context. Structured data values that are closed maintain a list of nodes which point to the states of the outer environments that existed at the time of closure.

5.3 Copying Values

The copy operation is frequently used in *G* computations since parameter passing is *by need* and since the assignment operation always assigns a copy of its right-hand-side value to its left-hand-side variable. Copying of scalars is quite simple and only involves creation of a new value descriptor plus a copy of the values of the *type* and *value* fields of the original value descriptor.

Copying complex data structures is slightly more complicated. Every complex data value consists of a value descriptor whose *value* field points to a header structure which contains local information for that data value. For example, the header structure maintains a record of the current index of the data value and it points to the first and current elements of the value sequence of the data value. One general rule that is observed for all the complex data structures is that the value sequence of a complex data structure is never altered. This allows complex structures to be copied by simply creating a new value descriptor and header structure and then copying the appropriate local information for the newly copied value into its new header structure. A new header structure will usually contain pointers into the original expression's value sequence or some segment of the original value sequence; there is generally no need to copy any of the values in the original value sequence. This results in efficient quick copy operations. More details about the copying process may be found in subsequent sections that discuss the representations of the complex data types of *G*.

5.4 Lazy Evaluation

Lazy evaluation is utilized in *G*. This means that computations are only performed when they are needed. Thus, when a *G* expression is parsed by the interpreter, computation of the value of the expression is delayed. The interpreter must create an internal data structure that records enough information about the expression to allow later computation of the value it represents. The *G* evaluation function, called *print*, recursively computes and then prints each value in its argument stream. As *print* moves deeper into a nested stream structure, the local environment of the nested structure is stacked onto the environment that has accumulated to that point and this composite environment is then used when needed to evaluate subsequent values encountered.

5.5 Representing the Values of *G*

This section describes the data structures used to represent each of the *G* types as well as the various forms of built-in expressions and primitives that may be specified in *G*.

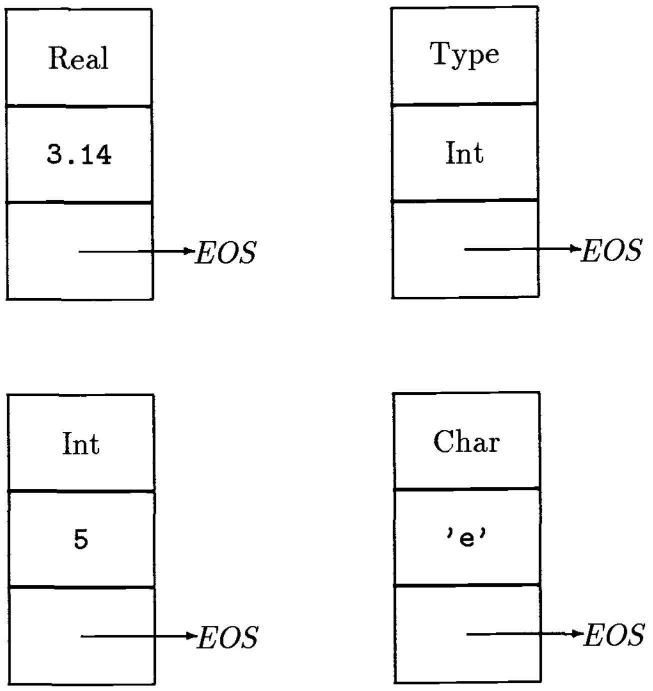

Figure 4: Examples of *G* Scalar Values

5.5.1 Representing Scalars

Scalar values are the simplest values represented in *G*. Figure 4 shows value descriptors representing the floating point value `3.14`, the integer value `5`, the type value `Int` and the character value `'e'`. Here we see clearly that scalar values are indeed stream values. Each scalar shown is a single element stream whose single value is followed immediately by the end-of-stream (*EOS*) marker.

5.5.2 Representing Strings

Figure 5 shows a data structure that represents the *G* value `''hello''` and which has the *G* type String. The *value* field of the value descriptor points to a simple header structure that contains an integer and a pointer to the character string associated with the *G* value. The integer recorded in the header structure is the current index of the stream. Thus, in Figure 5, the current position of the stream

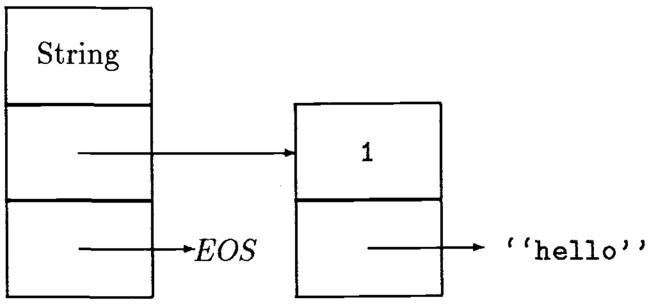

Figure 5: Data Structure that Represents the String "hello"

value displayed is 1 which means that none of the characters of this stream have yet been enumerated.

5.5.3 Representing Tuples

A data structure diagram for the simple tuple [15,4.5] is shown in Figure 6. As is typical of all the complex data types of *G*, the value descriptor of a tuple points to a header structure which contains all the information necessary for the *G* evaluation function to evaluate the tuple. The value of a tuple's type is the special value *Tuple.* Since the tuple shown in Figure 6 is not embedded in another stream, the *next* field of its value descriptor is set to the end-of-stream marker.

In Figure 6 we see that the tuple header contains a pointer to the local environment associated with that tuple, a pointer to the first value in the value sequence of the tuple and a pointer to the current active value of the tuple. For simplicity both the static and dynamic environment pointers are represented by a single arrow in Figure 6. We will use this simplification in all subsequent diagrams. Also note that the environment pointer points to the symbol λ which is being used here simply to denote the empty list of environment nodes. Thus in the case of Figure 6 there is no local environment associated with the tuple. The current value pointer and the first value pointer point to the same value in Figure 6, the *G* value 15. This means that the *next* operator has not yet been applied to the tuple value represented in the

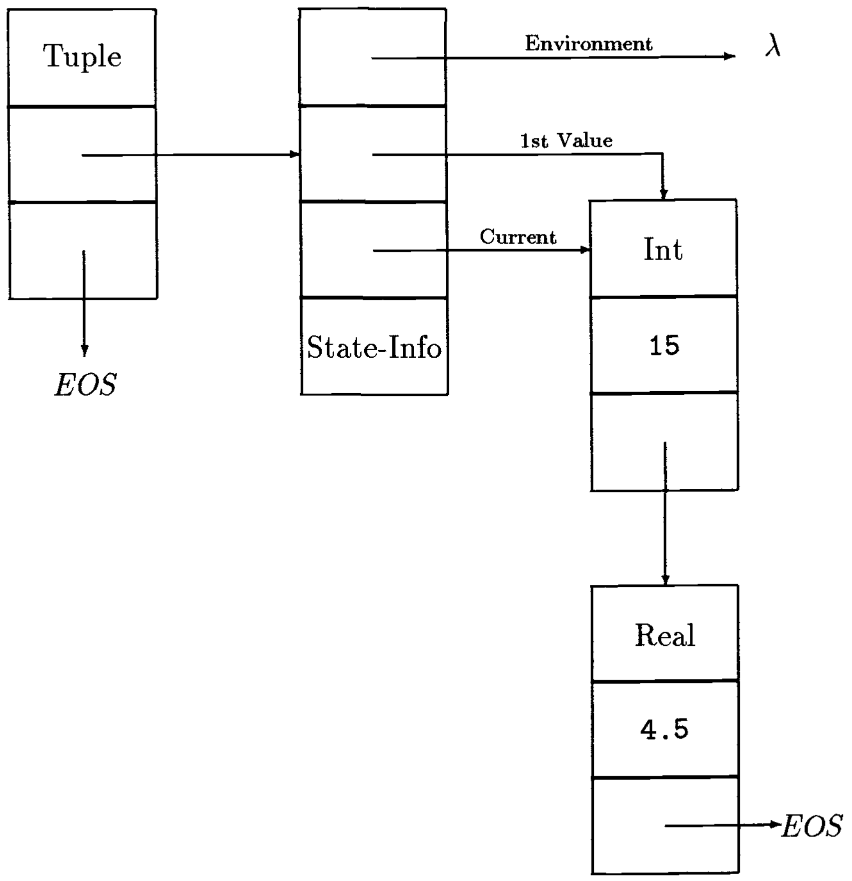

Figure 6: The Tuple [15, 4.5]

figure.

The tuple header also contains a field called *State-Info*. This field contains the current index of the tuple and a single bit of information that is used when the current value of the tuple is a code body. To understand how this bit field is used, recall first that no value in the value sequence of a tuple may be altered. This assures that a copy operation of a tuple need not make copies of the values that make up the value sequence of that tuple. Since the value sequence is never altered, all tuples that rely on the original value sequence (or some part of it) may exist independent of each other. Normally there is no difficulty in implementing this rule of non-modifiability of value sequences. However, when a code body is encountered in a value sequence a difficulty arises. A code body represents an implicit value sequence of the structure in which it is embedded. Because of this it is necessary to modify the header of a code body in order to maintain information concerning the current location into the implicit sequence represented by the code body. But modifying the code body data structure would mean modifying an element of the value sequence of the structure in which the code body is embedded and this is not allowed.

The solution used by the *G* implementation is the following. When a code body is encountered as the current value of a tuple or function and when the first value of that code body is requested, a copy of the code body is made and the current value pointer of the tuple header structure is pointed at the copy. In this way, the original value sequence of the tuple is not altered which is critical since a number of other tuples may be dependent on it. The copy of the code body points to the next value after the original code body and so no information has been lost concerning what values follow the implicit sequence represented by the code body. When a copy of a code body has been made, a bit field (mentioned earlier) in the tuple header is set to record that a copy has been made. Thus if and when another value is requested from the tuple it will be known that a copy of the code body has already been made and whatever changes are necessary may be made to the header of the code body copy in order to determine and enumerate the next value in its value sequence. When the code body has been exhausted and the next value after it in the tuple's value sequence is requested, the copied code body may be freed and the current value pointer of the tuple moved on to the next value in its value sequence.

Copying tuples requires creating a new value descriptor and header structure, and then copying values from the old to the new structures. As mentioned above, the value sequence of the original tuple is not copied; pointers into the original value sequence are merely assigned to the new header structure. The static and dynamic environment pointers of the new header structure are both set to the value of the original tuple's dynamic environment pointer. Figure 7 shows a simple tuple with

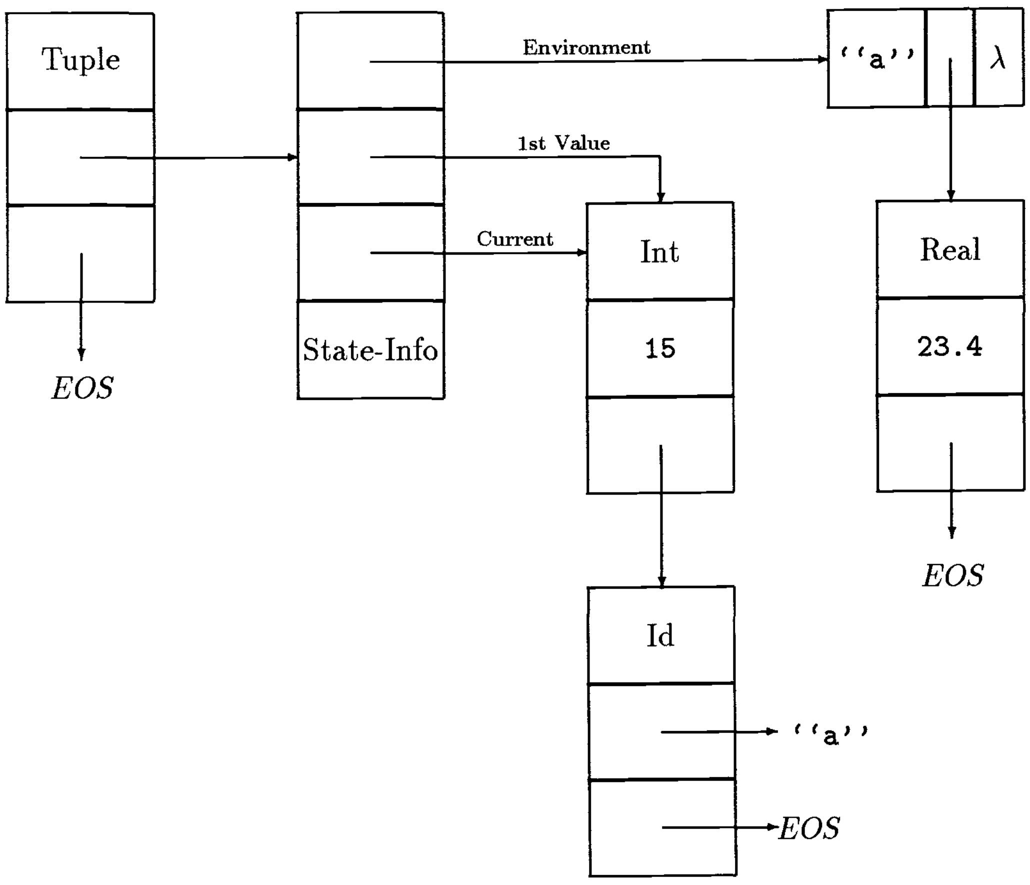

Figure 7: The Tuple $[local[a : 23.4], 15, a]$

a single local environment node. Making a copy of the value depicted in Figure 7 would involve making copies of all the structures shown except those in the value sequence which includes the *G* value `15` and the *G* identifier *a*. From this example we can see that the cost of copying complex data values is generally independent of the length of the value sequence of the value being copied. Before moving on to the next section note in Figure 7 that *G* identifiers are represented by value descriptors with the internal type value *Id.* The *value* field of the value descriptor for an identifier contains a pointer to the character string that represents the name of the identifier.

5.5.4 Representing Code Bodies

Code bodies that serve as control structures (i.e. *foreach*, *while*, *repeat* and *if* code bodies) have underlying representations very similar to the tuple. The main difference is that the header structure of these code bodies also contains a pointer to the value of the argument of the code body. The *repeat* code body is given a default argument that is guaranteed to always be true. In Figure 8 the data structures needed to represent the code body `foreach(a)[15,'e']` are presented. Note that the main difference between this diagram and a diagram for the representation of the tuple `[15,'e']` would be the added value descriptor for the *G* identifier ``a'' shown in Figure 8; this value descriptor represents the argument to the code body. In a tuple there is no pointer to an argument. Notice in Figure 8 that the type value for the entire code body is *Foreach.* Each different kind of code body has its own internal identifying type value that allows it to be recognized and evaluated appropriately.

Figure 9 shows the general form of the representation of an *if* code body. The information recorded in the header structure is somewhat different than that recorded by the code body types discussed above. An *if* code body needs to record all possible tuple clauses that it may manifest and it needs to record the information necessary to select and activate the appropriate clause. When an *if* code body is activated (i.e. when a value is first requested from an *if* code body) a clause is chosen from the list of potential clauses and then pointers for the first and current values of the

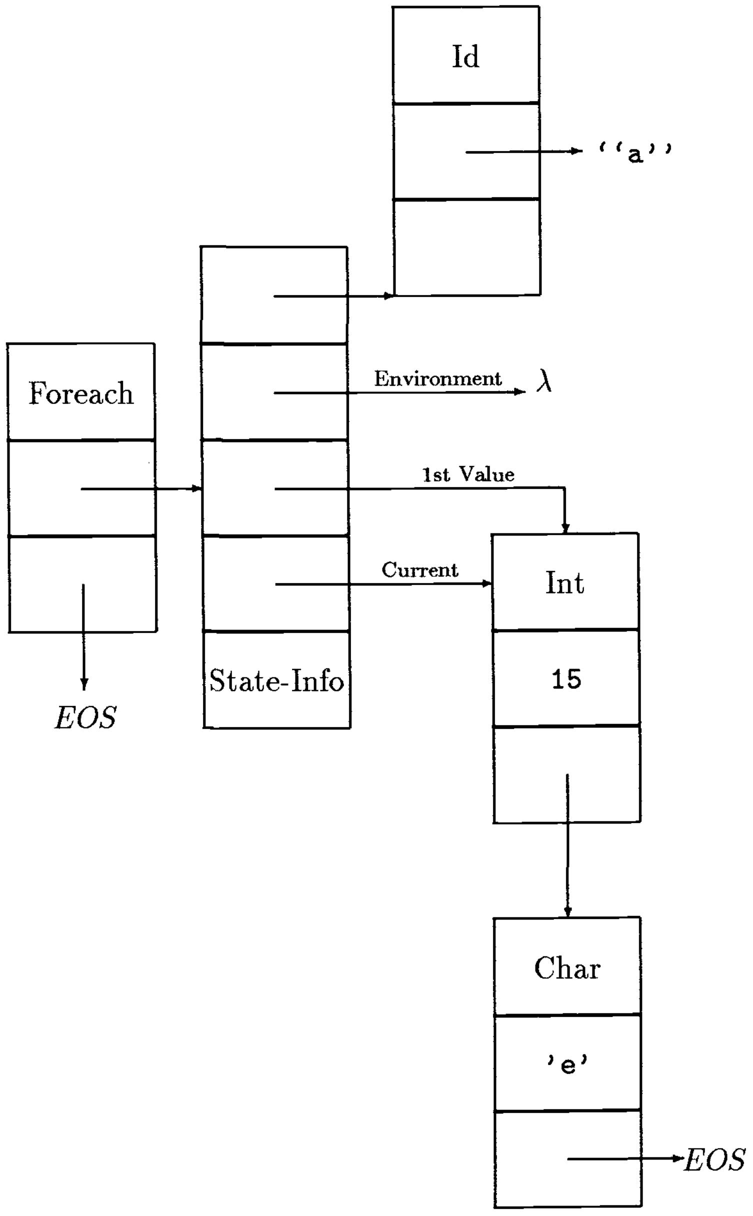

Figure 8: The *Foreach* Code Body `foreach(a)[15,'e']`

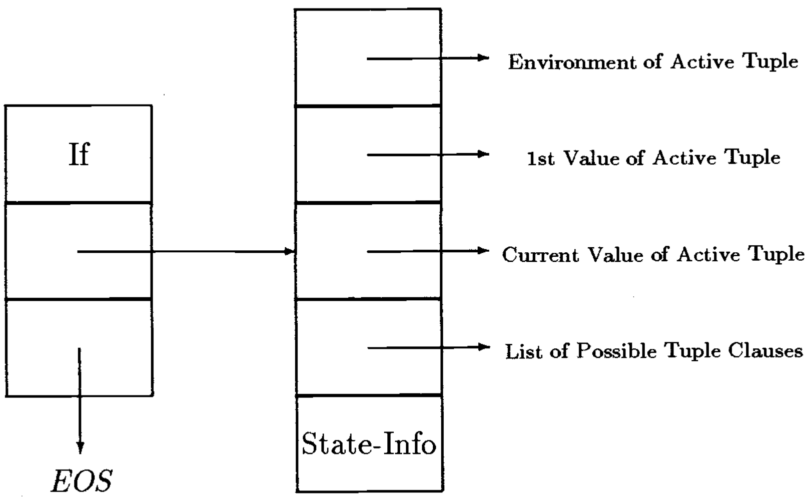
If
EOS
State-Info
Environment of Active Tuple
1st Value of Active Tuple
Current Value of Active Tuple
List of Possible Tuple Clauses

Figure 9: Form of the *If* Code Body Data Structure

value sequence of the chosen clause are set in the header structure. In addition to this, the environment pointers of the header structure are set. In this way the header structure is "transformed" into a header for whatever clause is appropriate when the *if* code body is activated.

Range code bodies are represented by two simple data structures, a value descriptor and a header. Figure 10 shows these data structures for a typical range code body. The header of a range code body records the first, last and current value of the implicit sequence that the code body represents. It also records the step or distance between successive values in that sequence. The value *Range* seen in the value descriptor of Figure 10 is the internal type value for all range code bodies.

5.5.5 Representing Functions

Functions are a sub-type of type Tuple and this is clearly reflected upon examination of their underlying representations. Figure 11 shows the data structures used to represent the function `func(a:23.4)[15,a]`. Note that in *G* a default value may be assigned to an argument of a function. If Figure 7, which shows the rep-

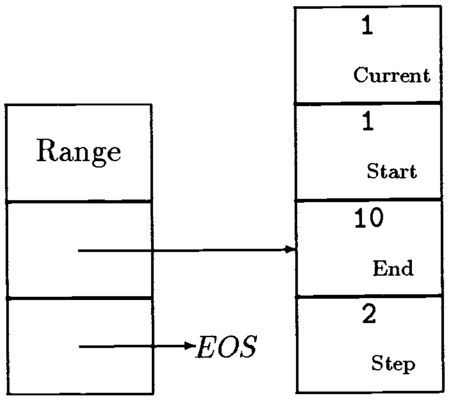

Figure 10: The *Range* Code Body `1..10 step 2`

resentation of the tuple `[local[a:23.4],15,a]`, is compared with Figure 11, it is seen that there is no difference in the data structures displayed in both figures. In fact, the only difference between the two figures is in the values of the *type* fields of the value descriptors for the function and the tuple. In processing functions, the *G* interpreter relies on the fact that the original arguments of a function are guaranteed to be located at the start of the static environment list of the function. This allows function calls to be properly handled. For more details about handling function calls refer to the section Representing Function Calls.

5.5.6 Representing *And* and *Concatenation* Expressions

Both *and* and *concatenation* expressions are represented with the same basic underlying data structures. The general form of this representation is shown in Figure 12. Both of these expressions are created with the type value Stream. A bit field in the *state-info* section of the header is used by the *G* interpreter to distinguish an *and* from a *concatenation* expression.

As shown in Figure 12, pointers in the header structure provide access to both an inactive *template* set of copies of the original conjuncts as well as a working or active set of conjuncts being used by the *and* expression. This same general scheme

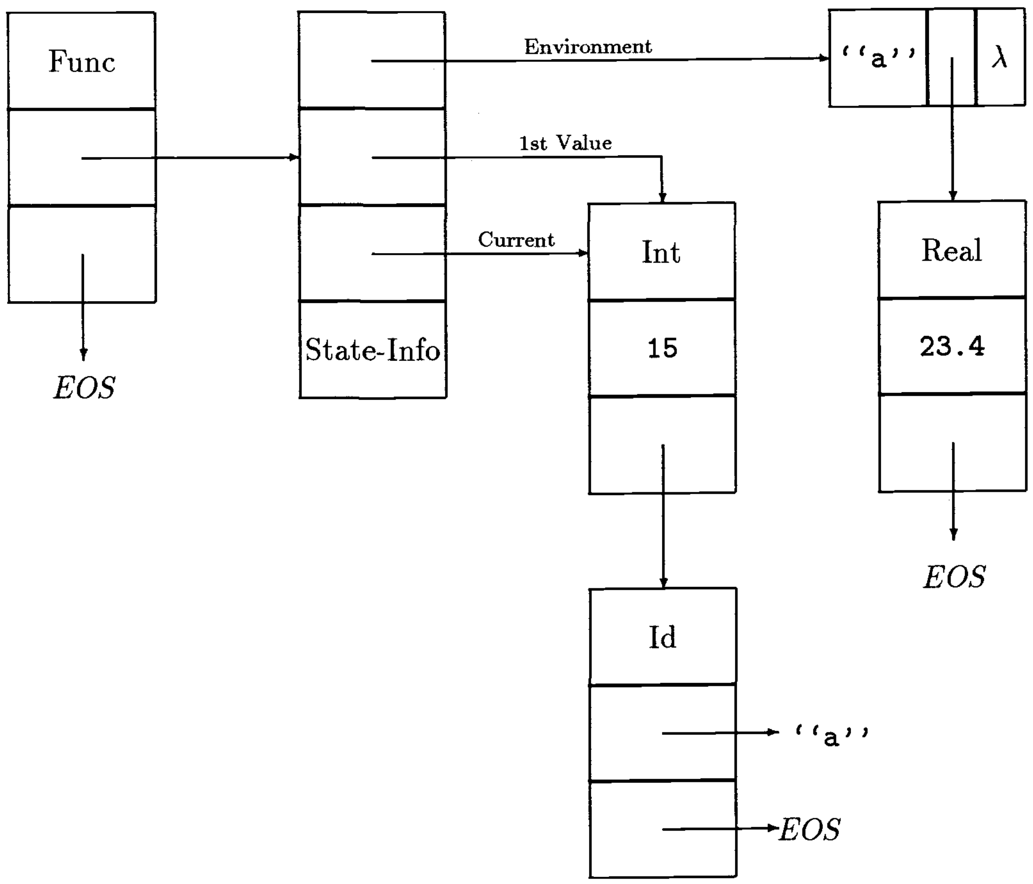

Figure 11: The Function $func(a : 23.4)[15, a]$

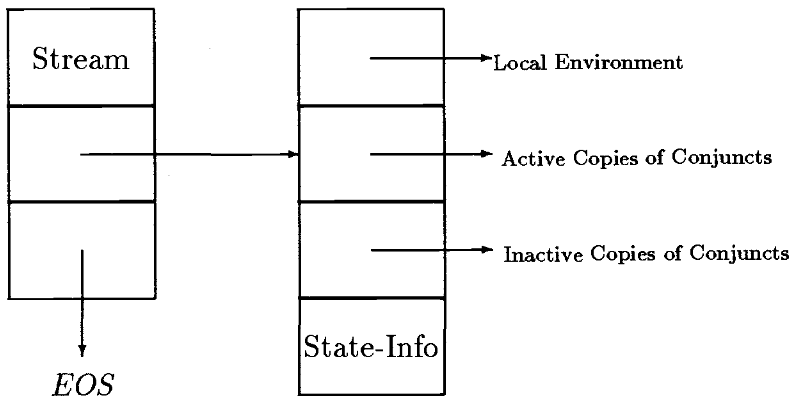

Figure 12: General Form of an *And* Expression

is used for the components of a *concatenation* expression. Maintaining two copies of the components of these types of expressions is used to facilitate more than one operation. In copy operations, the exact state of a set of working conjuncts can be captured by copying these active values to the inactive set of the newly copied *and* expression. It is also important to note that all references to identifiers in the components of these types of expressions can only have their referent value *captured* or recorded when the expression is actually activated (i.e. when the first value is requested from the expression). If such an activated expression is re-encountered later, for example, after a refresh of the stream in which the expression is embedded, its referents will need to be re-evaluated with respect to the environment when the second activation occurs. This process could occur repeatedly.

5.5.7 Representing Pattern-Matching Values

The data structures used to represent a pattern-matching expression all generally follow the same form. The value descriptor points to a header which maintains the following information.

1. Local state information. For example, the current index value of the expression.

2. A pointer to the current value of the base stream that the pattern is being applied to. If the pattern-matching expression has not been enumerated, then this pointer will be nil.

3. A pointer to a value descriptor representing the base stream identifier. This descriptor is followed by a list of nodes which represent the pattern to be applied to the base stream values.

Figure 13 shows the underlying data structures used to represent the *G* value `s[?x,<=99]`. Notice that output variables use the same value descriptor as regular *G* variables but have the special internal type value *Output*. The first occurrence of each output variable in an *and* expression is actually given the special type value *FOutput*. This allows the implementation to properly process output variables by either assigning them values or by using values they have already been assigned in the pattern-matching process. Also notice that a comparison component of a pattern has a special data structure that records the operation name and the *G* value to be compared against.

5.5.8 Representing Relation Values

The data structures used to represent a relation are quite simple. The standard value descriptor is used in the usual way with composite data values to record a header data structure. The header structure records the current index of the relation and points to a hash table where the values that make up the relation may be accessed. Each non-empty bucket of the hash table is made up of *G* values chained together by their next fields in the same way they would be linked if they formed a single stream value. There is no need for the header structure to record a local environment for a relation since values entered into a relation are always evaluated upon insertion into the hash table. Figure 14 shows the general structure of a relation.

Every relation is composed of either tuple values or scalars. In the case of scalars, the scalar value itself is used to generate the hash table index. For tuple

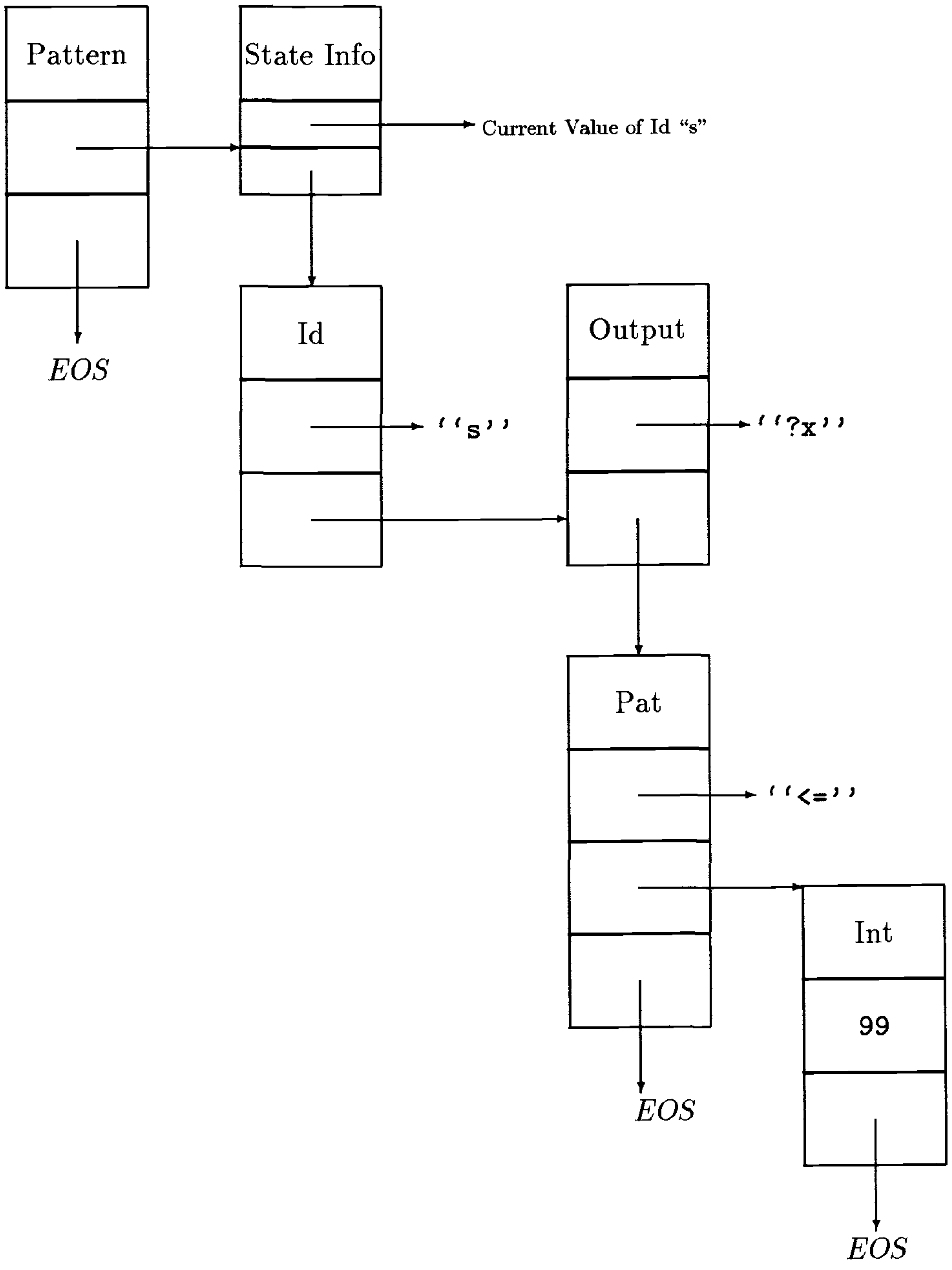

Figure 13: Data Structures for the Value s[?x,<=99]

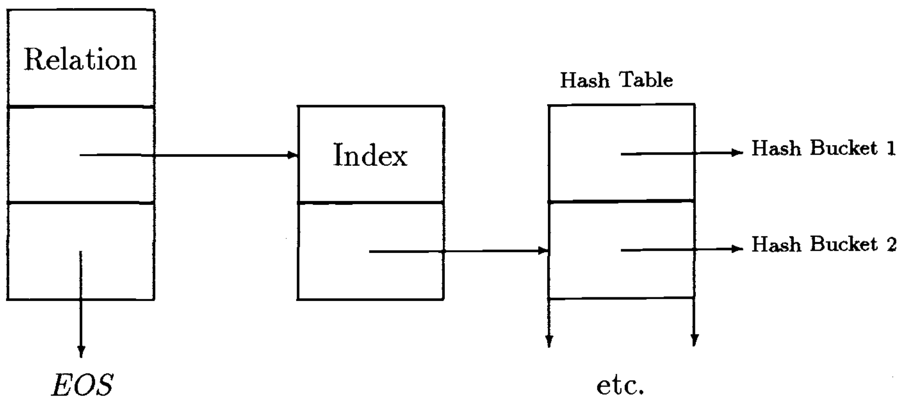

Figure 14: General Form of a Relation

values, the first value of the tuple is used to generate the hash table index. Hashing is used only for values of type Pattern whose base stream is a relation. For all other non-hashed access of a relation, the order of enumeration of a relation is implementation dependent. Values are simply enumerated one-at-a-time starting with the first hash bucket values and proceeding down the hash table until all values in all hash buckets have been enumerated.

5.5.9 Representing Primitive Expressions

Functions calls that invoke primitive (built-in) functions are called primitive expressions. Primitive expressions are treated uniformly by the *G* interpreter. A primitive expression is represented by a value descriptor whose *type* field contains the value Pfunc. The *value* field of the descriptor points to a header structure which contains two pointers. One pointer points to the primitive function that needs to be invoked in order to compute the value represented by the primitive expression. The other pointer points to a list of *G* values that are to be used as arguments to the primitive function that is invoked. The first *G* value of every list of arguments in

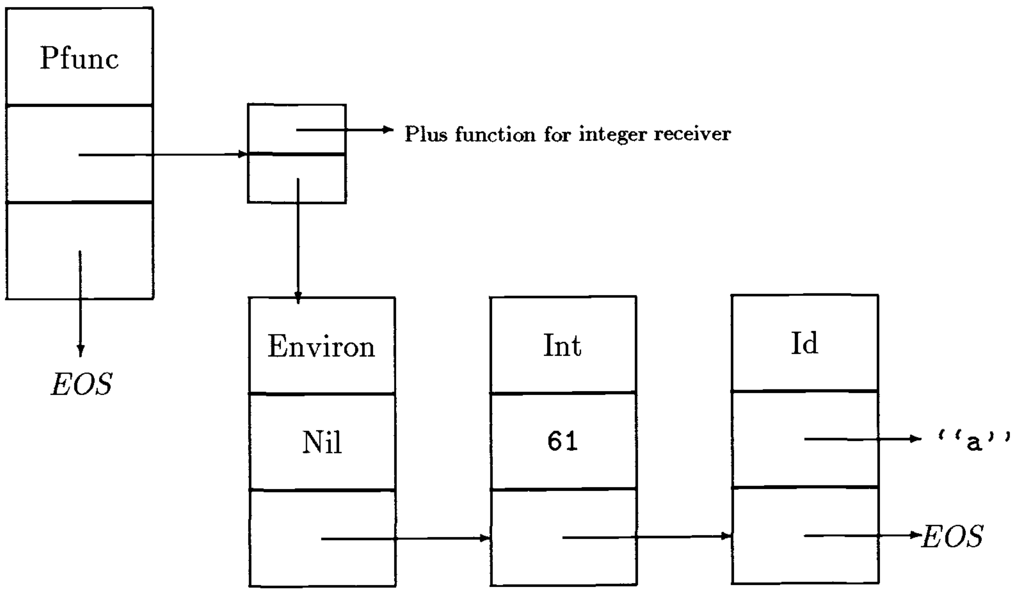

Figure 15: Data Structure for the Primitive Expression `61+a`

one of these representations is always of an internal type called Environ. The *value* field of this internal value is used by the *G* interpreter to pass an environment into whatever primitive function is being invoked. Figure 15 shows the data structures that are used to represent the primitive expression `61 + a`. Before closing this section it is important to note that, as with all *G* values, representations of primitives values do not result in any computation unless the value they represent is needed. All evaluation in *G* is lazy. The parser merely creates these underlying representations in case the values they represent are later requested.

5.5.10 Representing Function Calls

A function call has the same underlying form as the representation of any primitive or built-in function. It follows the same general form explained above in the section Representing Primitive Expressions. In the case of a function call, however, the primitive function pointed to in the header structure is a general function call routine that expects the name of the function being invoked to appear in an argument list. The argument list must be headed by this function name and followed by the

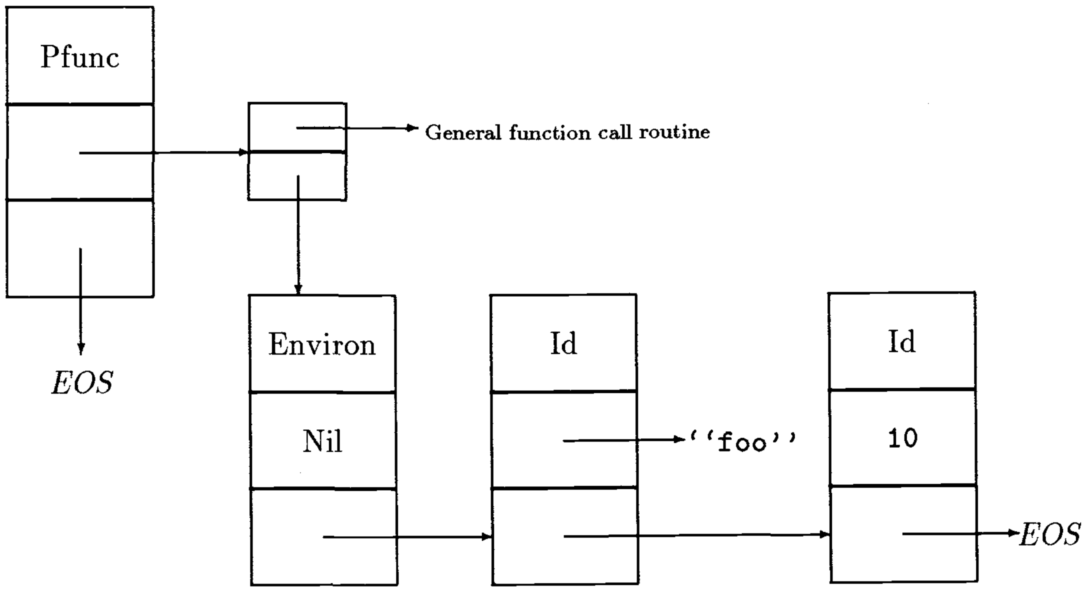

Figure 16: Data Structure for the Function Call `foo(10)`

actual argument values that are being passed to the named function.

The general function call primitive implements the object-oriented look-up protocol described in an earlier section that explained the function call semantics of *G*. Once the function definition is located, a copy of the definition is made and the formal parameters are instantiated with copies of the actual values of the arguments that were passed. The *G* interpreter then evaluates the newly created function. Figure 16 shows the data structures used to represent the simple function call `foo(10)`.

Figure 17 shows a representation of a nameless function call. Note that in this representation a value descriptor with a special internal type Funcdef is expected. The *value* field of this descriptor contains a pointer to the definition of the nameless function that is to be invoked. The actual arguments to the nameless function always follow the value descriptor that points to the function.

5.5.11 Representing User-defined Values

Instances of user-defined types are all represented in a uniform manner in the *G* implementation. An instance of a user-defined value is represented by a value

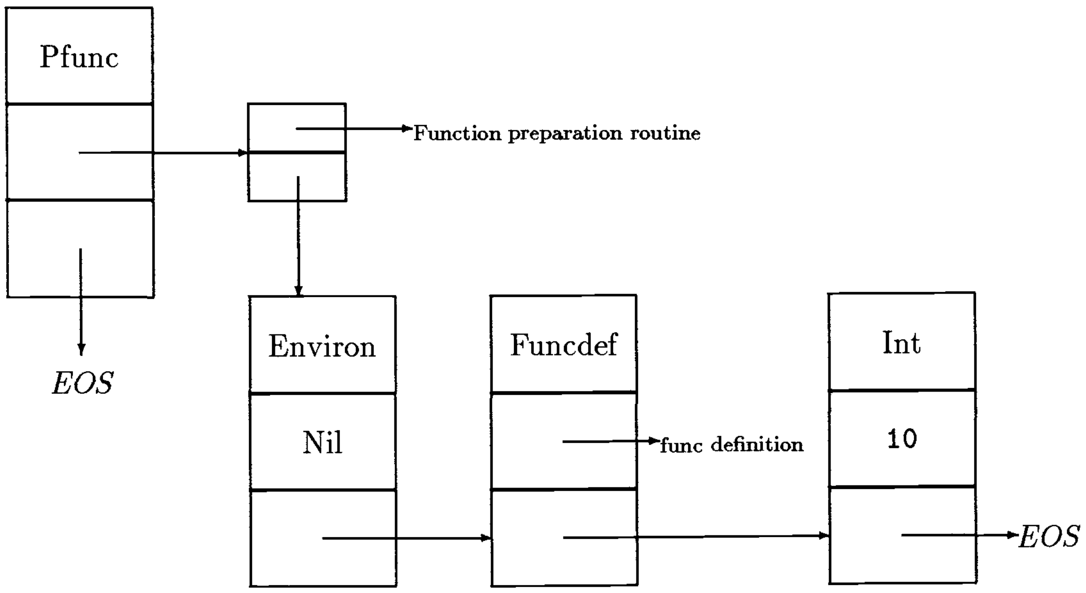

Figure 17: Data Structure for the Function Call `func(a)[a](10)`

descriptor whose *type* field contains a pointer to the data structure in the *G* type hierarchy that represents the user-defined type. The *value* field of the descriptor points to a header structure which contains two pointers. One pointer points to the local environment of the user-defined value if instance variables (local variables) have been declared for that user-defined type. The other pointer points to an interface value if one has been defined for that user-defined type. This general representation scheme for instances of user-defined types is shown in Figure 18.

A message passing expression is considered a primitive expression in *G* and is represented in a fashion analogous to any primitive expression. Figure 19 shows a representation of the message passing expression `mystack::push(10)`. This expression was used as an example in a previous section describing user-defined types in *G*. Notice how Figure 19 shows the same basic form of any primitive expression. The primitive function pointed to in the header structure is a generic primitive that does the following:

1. Locates the instance of the user-defined type named by the expression. This would be the value of `mystack` in Figure 19.

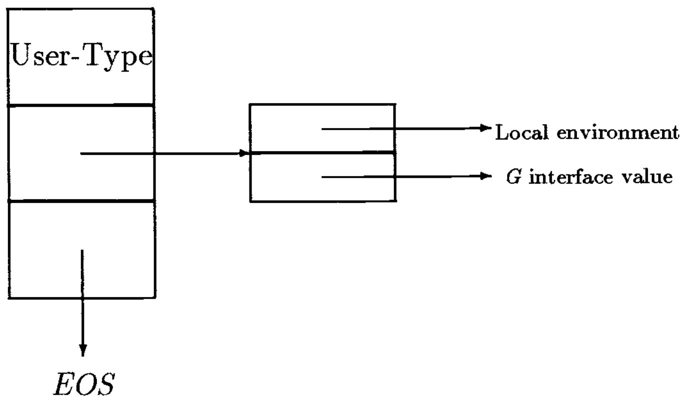

Figure 18: General Representation of an Instance of a User-defined Type

2. Locates the function definition associated with the given selector for the given user-defined type. In Figure 19 the selector is **push**.

3. Creates a function from the named function definition. The parameters are instantiated with the actual values passed as arguments. In Figure 19 the single argument `10` is being passed to the function associated with selector **push**.

As with all values in *G*, none of the computation just described occurs unless the value of the given expression is needed.

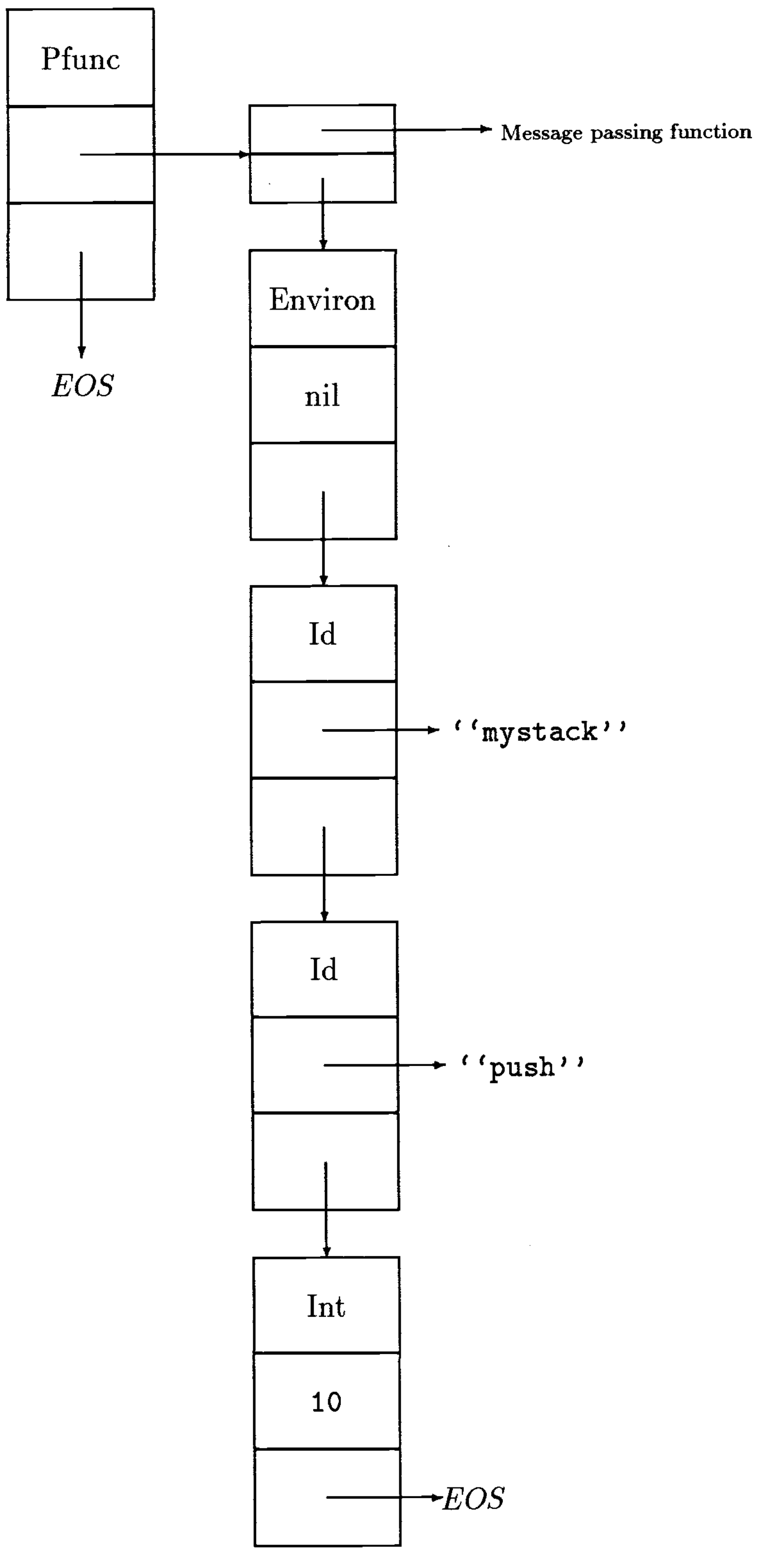

Figure 19: Representation for the Expression `mystack::push(10)`

Chapter 6

Expressing and Integrating the Paradigms of G

This chapter presents programs and various stream expressions that demonstrate how each of the paradigms included in *G* are expressed. In addition to this, the last section of the chapter presents programs that illustrate how different paradigms can be used together to form problem solutions. All of the programs and expressions presented in this chapter have been run on the *G* interpreter.

6.1 The Imperative and Procedural Paradigms

In order to demonstrate the traditional "von Neumann" model of procedural and imperative programming, a small Monte Carlo simulation is presented below. This simulation was taken from a problem presented by Walker [Wal86]. The *G* solution of the simulation follows the standard form of the traditional paradigm; it is composed of several function definitions, makes use of standard flow control structures including the while loop, the selection (*if*) operator and the sequential (the comma) operator, and it relies heavily on destructive assignment and its "accumulated effect" in order to complete a computation .

The simulation may itself be outlined in the following way:

```
1. Perform Initialization
    a. Number of wins for each team is initially 0.
    b. Initialize the number of rolls to be completed.
    c. Print introductory information.

 2. Repeat the roll of the dice the desired number of times.
    a. Roll the dice.
    b. If the sum of the dice is 7, 8, 9, 10, or 12 then record
       a win for player A else record a win for player B.
 3. Print the results of the simulation.
```

The *G* program that implements this simulation is given below.

```
monteCarlo := func()[
   local[winA:0,winB:0,sum,numberRolls:100,total:100],
   printIntro(),
   while(numberRolls > 0)[
      sum := random(6) + random(6),
      if(sum>=7 and sum<=10 || sum=12)[winA := winA+1]
      else[winB := winB+1],
      numberRolls := numberRolls-1
      ],
   printResults(winA,winB,total)].

printIntro := func()[
   write["This program simulates a game of dice.\n"],
   write["Side A wins on a roll of 7, 8, 9, 10, or 12.\n"],
   write["Side B wins otherwise.\n"]
   ].

printResults := func(awins,bwins,rolls)[
   write["When the dice were rolled ",rolls," times:\n"],
   write["Side A ",awins," wins or ",awins*100.0/rolls,"%.\n"],
   write["Side B ",bwins," wins or ",bwins*100.0/rolls,"%.\n"]
   ].
```

The main function, *monteCarlo*, initializes variables through local variable specifications and prints out introductory information by making a function call to *printIntro*. The main work of function *Monte Carlo* is carried out in a while loop which simulates dice rolling with a primitive function call and then updates the values of local variables to record the effect of the rolled dice. Notice the argument of the *if* sequence constructor embedded in the *while* loop. It can be interpreted as either a disjunction or a stream concatenation. For this application it is most easily interpreted as a disjunction. Finally the results are printed out by function *monteCarlo* through a function call to *printResults.*

An important aspect of the above code is that it is written exactly in the style of a traditional imperative and procedural language. The "statements" are written one after the other each one separated from the next by a comma which is the sequential operator. There is no need in the above *G* solution to consider stream semantics or even be aware that one is programming in an environment of streams. The imperative and procedural paradigms are effortlessly captured.

6.2 The Lambda-free Paradigm

John Backus, in his now classic Turing Award Lecture [Bac78], described a class of programming systems called functional programming (FP) systems. The structure of these FP systems provides the basis for describing a large class of languages with diverse styles and capabilities. As described by Backus, any given FP system is made up of the following parts:

1. A set of data objects. These are the permitted members of the domain and range sets of functions.
2. A set of primitive functions that map objects into objects.
3. A set of functional forms used to combine existing functions or objects to make new functional forms.
4. The *application* operation that allows a function to be applied to its arguments and produce a value.
5. A mechanism for naming new functions.

Backus presented one example of a set of primitive functions and a set of functional forms that could form the basis of an FP system. *G* code that implements these functions and functional forms is contained in Appendix A. A glance at Appendix A will illustrate how concisely each of the primitive functions and the functional forms is expressed as a *G* expression. This collection of FP primitives can be kept as a small library of *G* functions and simply included into the environment if it is anticipated that the lambda-free approach to programming will be used.

Data objects, the application operation and a naming mechanism are also required in order to create an FP system; these components already exist in *G*. Objects of the various types supported by *G* become the potential objects of our FP style system. Furthermore, in *G*, since a function is a first-class object, new functional forms may readily be defined; this is an extension of the capabilities of FP systems. The application operation is provided in *G* as a built-in facility and assignment can be used to bind names to function definitions. Assignment is no threat to referential transparency if destructive assignment is not utilized.

In order to insure that the lambda-free paradigm preserves referential transparency, it is also necessary that the primitive operator *next* (@) not be used in code meant to be restricted to the lambda-free paradigm. The *next* operator represents a composite operation which implicitly involves destructive assignment when applied to an identifier. It is simple to see this when one considers that after application of the *next* function to an identifier, the value of that identifier has been modified. Since moving a stream ahead to its next value without side-effects is a needed capability in the functional paradigm, this functionality has been provided by defining the function *tail* which is one of the primitive functions suggested by Backus and which is defined in the library given in Appendix A.

As a final protection of referential transparency the data type Relation should not be used. The primitives associated with relations (*insert* and *delete*) both directly alter the relation they are applied to and so do not preserve referential transparency. In addition to this relations are the sole exception to the *by-need* parameter passing protocol of *G* as explained in a preceding section that discusses Relations.

In order to give at least a feeling for the ease with which the primitives and functionals of Backus were coded in *G*, we will now discuss the *G* implementations of two primitive functions and one functional form that are presented in Appendix A. After these selected functions are discussed, we will explore how the inner product and matrix multiplication examples, also given by Backus, can be expressed in the G lambda-free paradigm.

6.2.1 Primitive Functions

We begin by presenting the function that distributes its second argument among each of the elements of its first argument. This is the function *distr* found in Appendix A. It was called *distribute from right* by Backus and defined by him in the following way:

$$distr : x \equiv x =< \phi, y > \rightarrow \phi$$

$$x =<< y_1, ..., y_n >, z > \rightarrow << y_1, z >, ..., < y_n, z >>$$

The G code that implements this primitive is:

```
distr := func(y,z)[foreach(y)[[y,z]]].
```

In the expression above a code body is used to create a tuple for each element of the first argument y. Each tuple so created contains the current value of y followed by the value of the second argument z to the function.

In our next example we consider a slightly more complicated function that transposes a matrix (i.e. the ith column of the original matrix becomes the ith row of the resulting matrix). Backus called this function *trans*. The G code that implements this primitive is:

```
func(s,n:1)[if(n<=len(~s))[[foreach(s)[~sel(s,n)]],self(s,n+1)]].
```

The *foreach* code body is itself enclosed in brackets within the function body since the function returns the resultant matrix as a list of lists. The *foreach* code body calls the function *sel(s,n)* which selects the nth element of stream *s*. Also notice that the second argument of *trans* is given a default value of `1`. The function is meant to be called with a single argument, which is the matrix to be transposed. Since the second argument will be missing from the original function call, it will be given the default value of `1`. Subsequent recursive calls will each increment the second argument. Each recursive call enumerates the *n*th row of the transposed matrix by selecting the *n*th element from each row of the original matrix. The *if* argument provides the exit condition for the recursion; it tests to make sure that *n* remains less than or equal to the length of the rows of the original matrix.

6.2.2 Functional Forms

As an example implementation of a functional form, we will consider the function *apply* since it is quite simple and yet it represents the same general technique used to create all the functional forms. The *G* code that implements this primitive is:

```
func(f)[func(x)[foreach(x)[f(x)]]].
```

Like all of the functionals, *apply* is implemented as a function that returns a function. In the case of *apply*, the "outer" function takes a single argument which has a function value. This function argument is then applied to each element of the argument of the inner function. In order to demonstrate the way in which a functional is used, assume that a simple function *dbl* has been defined which doubles the value of its single argument. The function *dbl* may then be applied to each element of the stream `[1,2,3]` by forming the following *G* expression:

```
apply(dbl)([1,2,3]).
```

The result of this function call would be the stream `[2,4,6]`.

6.2.3 Inner Product and Matrix Multiplication

In order to demonstrate how to use his hypothetical FP system, Backus gave FP solutions for inner product and matrix multiplication. In each case we present Backus's solution to the problem and then we present two *G* solutions for each problem. The first *G* solution presented for each case renders, as close as possible, a direct translation of the solution given by Backus. In addition to the "direct" translation solutions, however, we also offer simplified mixed paradigm solutions which use functions suggested by Backus within the context of *G* stream semantics.

The solution given by Backus for inner product was:

$IP \equiv (/+) \circ (\alpha *) \circ trans$

It should be explained here that "/" represents the *insert* functional, "+" the *plus* function, "*" the *multiplication* function, "o" the *composition* functional, "α" the *apply* functional, and *trans* the transpose function. The "direct" translation of this into G would be:

```
ip := func(s)[insert(plus)(apply(mult)(trans(s)))].
```

The *composition* functionals are a built-in feature of G and, therefore, are not explicitly shown in the G expression. Note that in G only the names of types may begin with an uppercase letter, therefore, we used the variable *ip* as the name of the G inner product function. This G definition can be simplified by recognizing that the multiplication operator in G (defined in the G standard library (Appendix B)) is a vector operation. Thus it is not necessary to transpose the argument matrix. The new simplified G expression for inner product is:

```
gip := func(s)[insert(plus)(~mult(s))].
```

The solution given by Backus for matrix multiplication is given by the following expression:

$MM \equiv (\alpha\alpha IP) \circ (\alpha distl) \circ distr \circ [1, trans \circ 2]$

In this representation the square brackets represent the *construction* functional, *distr* is the *distribute from right* operator and *distl* is the *distribute from left* operator. In addition to these symbols the numbers 1 and 2 stand for the select operations. For example, 1 means select the first element. The most direct translation of this into G would be:

```
mm := func(s)[apply(~apply(ip))
          (apply(distl)(distr([ sel(s,1),trans( sel(s,2))])))].
```

In the above expression *sel* represents the select operator. If the function names for the G expression were translated into their equivalent Backus symbols and if composition were shown explicitly, then the G expression would be:

$mm := func(s)[(\alpha\tilde{\alpha} IP) \circ (\alpha distl) \circ distr \circ [1 \circ s, trans \circ 2 \circ s]]$

Note that a near literal translation of the Backus formula is enclosed in a G function definition. This is the general form for translation of the FP expressions given by

Backus. A glance at the inner product example above shows this same form.

This is a very involved rendering of matrix multiplication. Again a mixing of the paradigms lends itself to a simplification of this expression, only this time the simplification is even more marked than was seen in the inner product example. Using only the primitives defined by Backus but recognizing that the multiplication operator is a vector operation in *G*, we can write a more concise and efficient representation of matrix multiplication in *G* code:

```
gmm := func(s)[
foreach(s1:@s)[[foreach(s2:trans(@s))[~insert(plus)(s1 * s2)]]]].
```

This expression is conceptually simpler than the Backus solution. Notice that each row of the first matrix will be assigned to `s1` in the first *foreach* loop and that each column of the second matrix will be assigned to `s2` in the second *foreach* loop. Furthermore, the expression `s1 * s2` is a vector multiplication in which each of the corresponding elements of `s1` and `s2` are multiplied together and the resulting vector is returned. Finally note that `insert(plus)(s1 * s2)` simply sums the elements of its argument stream `s1 * s2`.

Now we can see that the above expression forms element *j* of row *i* of the product of two matrices by multiplying row *i* of the first matrix by column *j* of the second matrix and summing over the product obtained. An explanation of Backus's solution is considerably more complicated.

6.3 The Applicative Paradigm

There are a number of requirements that must be met in order to program within the applicative paradigm. Most of these requirements are already met by the built-in facilities of *G*. Since all data objects in *G* are first class, including functions, *G* supports the creation of higher order functions and it allows entire data structures to be treated as single values. Both of these capabilities are requirements of the applicative paradigm. Another essential requirement of this paradigm is the preservation of referential transparency. Although *G* does provide support for this requirement, it does not guarantee its preservation and so some specific actions must be taken.

There are two important ways in which G supports maintenance of referential transparency. The first way is by assuring that parameter passing is strictly *by need*; the value of an argument is always copied when a function call is made. Another way in which G supports referential transparency is by prohibiting aliasing via assignment. When an assignment is executed, a copy of the value of the right-hand side is always assigned to the left-hand side variable. When combined with *by need* parameter passing this assures that two different G variables can never reference identical entities.

Yet these facilities of G, by themselves, are not enough to preserve referential transparency. In addition to these built-in supports it is necessary to prohibit each of the following:

1. Use of destructive assignment.
2. Use of the primitive operator *next* (@).
3. Use of the data type Relation.

One easy way to assure that destructive assignment is not used is to use assignment only to define new functions and then never to use the same function name twice when defining functions. The examples given below follow this protocol.

Prohibiting the use of the *next* primitive is necessary because it implements an implicit destructive assignment. Since side effects are not allowed in the applicative paradigm this primitive may not be used within code meant to express solely that paradigm. The ability of the *next* operator to move a stream to its next element is an important capability, however, and needs to be preserved without side effects. In Lisp, this capability resides in the *cdr* primitive. In G it is a simple matter to program a function to produce the results of a *cdr* primitive. This function already is provided in the standard stream library (Appendix B) and has been given the name *tail*. *Tail* returns a copy of its argument stream minus the first element.

Data values of type Relation may violate referential transparency in two basic ways. Primitives associated with relations in G directly alter the data structure of the relation they are applied to and values of type Relation are exempted from the strict *by-need* parameter passing protocol imposed on all other value types. This last point is discussed in detail in a preceding section that describes relations. The attendant potential for creating side-effects, however, precludes the use of Relations if it is intended that coding remain solely in the applicative paradigm in *G*.

Given the above criteria, we will now give examples that illustrate how to express the purely applicative paradigm in *G*. First two simple examples of functions, one non-recursive and one recursive, will be given. Then a solution to a non-trivial problem will be solved in the applicative paradigm.

6.3.1 Simple Function Definition Examples

Our first two examples are simple yet illustrative of function creation in the applicative paradigm. Initially we will consider a function called *putatendof*. This function was given as an example of a function in the applicative paradigm by Henderson [Hen80]. The function *putatendof* takes two arguments and makes the second argument the last member of the stream given as the first argument. Henderson presented the following *Lisp*-like function definition as an applicative implementation of *putatendof*:

```
if eq(x,NIL) then cons(y,NIL)
else cons(car(x),putatendof(cdr(x),y))
```

This is certainly within the bounds of good applicative programming form. Now consider the following *G* expression that represents the same function:

```
func(x,y)[foreach(x)[x],y].
```

This expression is simpler and yet still remains completely within the bounds of the applicative paradigm. The *foreach* code body simply constructs the value sequence of x and then the value of y is merely added on as a final value in that sequence.

Recursive function definitions also frequently occur in applicative programming. We turn to Henderson again for an example of how to program the recursive function *reverse* in the applicative paradigm. The function *reverse* simply reverses the order of the elements in its argument stream. Consider Henderson's solution:

```
if eq(x,NIL) then NIL
else append(reverse(cdr(x)),cons(car(x),Nil))
```

The G definition of this function looks similar but is somewhat less involved:

```
func(s)[if(s)[self(tail(s)),~s]].
```

Basically the G solution first calls itself recursively on the tail of the input stream s if s is not empty. It then enumerates what was the first element of stream s making it the last element of the returned value.

6.3.2 The Hamming Problem

As a final example of the applicative approach to programming in G we consider a solution to the non-trivial Hamming problem. The Hamming problem may be stated in the following way [Con88]:

Given as input a finite increasing sequence of primes `<A,B,C,...>` and integer n, output in order of increasing magnitude and without duplication all integers less than or equal to n of the form:

```
(A**i) * (B**j) * (C**k) * ..., i, j, k, ... >=0
```

Notice that if m is in the output sequence then so are:

```
A*m, B*m, C*m, ... <=n.
```

We will present a G applicative solution to this problem which is composed of three function definitions. *Filter* is a function which eliminates duplicates and adds the next set of output values to the output sequence. The G code for *filter* is given below:

```
filter := func(min,s,max,prim)
        [foreach(s)[min<s],foreach(z:gennext(min,max,prim))[z]].
```

The first *foreach* code body eliminates all occurrences of the minimum from the output stream and the second *foreach* sequence adds the new elements generated from the minimum to the output stream.

The function *gennext* is used to generate the new values that should be added to the output sequence by multiplying the target value (argument *seed*) by each of the given primes and eliminating any product that is greater than the allowed maximum. The *G* code for *gennext* is:

```
gennext := func(seed,max,prims)
        [local[t],foreach(prims)[if(seed*prims<=max)[seed*prims]]].
```

The final and main function which completes our solution to the Hamming problem is the recursive function *ham*. The function *ham* selects the minimum value in the output sequence generated so far, enumerates that value and then calls itself with its output sequence modified by the function *filter*. The *G* code for function *ham* follows:

```
ham := func(max,s,prims)[local[low],low:=~min(s),
        if(low<max)[low,self(max,filter(low,s,max,prims),prims)]].
```

The function *min* used within function *ham* is the minimum function and is defined in the standard library (Appendix B).

6.4 The Relational Paradigm

The relational database paradigm is easily expressed in G given the functionality of pattern-matching expressions and output variables. Each of the five primitive operators of relational algebra can be concisely expressed in *G* code as either a function or a pattern-matching expression. This will be demonstrated in the section *G and the Relational Algebra* in order to demonstrate that *G* is relationally complete. Also in that section, a few examples of relational algebra queries will be compared with equivalent *G* expressions. *G* expressions, however, tend to be expressed more naturally in a form similar to query-by-example. This will be demonstrated in the section *Query-By-Example*. All of the example queries given in these two sections have been taken from Date [Dat86].

The queries given in the following two sections assume that the streams *s*, *sp* and *p* exist and that they name streams whose values possess the following logical fields:

```
s(sn,city,sname,status) , p(pn,color,city) , sp(sn,pn,qty)
```

In this sample database, relation *s* is a relation of suppliers where field *sn* is a unique supplier number and *sname* is the name of the supplier. Relation *p* is the parts relation where *pn* is the part number field. Relation *sp* is the shipment relation where field *qty* is the quantity field referring to the quantity of a part with the part number *pn* and the supplier number *sn*. These streams could have been introduced as relations with the following three *G* expressions:

```
s := #Int,String,String,Int#.
p := #Int,String,String#.
sp := #Int,Int,Int#.
```

6.4.1 *G* and the Relational Algebra

The five primitive operations essential to relational completeness are selection, projection, union, product and difference. Union, product and difference are easily expressed as *G* functions. The code for each is shown below along with the auxiliary function *in*:

```
union := func(s1,s2)
         [foreach(s1)[if(not(in(s1,s2)))[s1]],foreach(s2)[s2]].
difference := func(s1,s2)[foreach(s1)[if(not(in(s1,s2)))[s1]]].
product := func(s1,s2)[foreach(s1)[foreach(s2)[s1||s2]]].
in := func(x,s)[foreach(s)[if(x=s)[x,break]]].
```

The auxiliary function *in* is used by the first two functions given above and so is included here for completeness. The function *in* returns its first argument if it is a member of the second argument passed to *in*, otherwise *in* returns the empty stream.

Projection is most easily expressed in *G* using pattern-matching and *and* expressions. For example, to project the single field representing *quantity* from the relation *sp* one could write:

```
sp[?sn,?pn,?qty] and ?qty
```

This would produce a stream of values corresponding to the third field of each value of relation *sp*.

The basic selection operation simply restricts the number of tuples selected from a given relation. This restriction is based on conditions which must be met by attributes of the relation values. The conditions could be based on comparisons among the attributes themselves or on comparisons of attribute values with values obtained external to the relation. An example of the former case, where two attributes are compared would be the following *G* expression which assumes a binary relation *intRel* of integer pairs:

```
intRel[?a,?b] and ?a<?b and [[?a,?b]]
```

The above expression would select all tuples from *intRel* whose first component is smaller than its second component. For an example of the second type of select we will consider our sample database . If we wanted to display information for all suppliers that supplied more than 20 units of part "p2" we could write the G expression:

```
sp[?sn,"p2",>20]
```

Below are given several examples of relational queries followed first by their relational algebra expressions offered by Date and then by an equivalent expression in *G*.

1. Get supplier names for suppliers who supply part "p2".

   Relational Algebra:

   ```
   ((s join sp) where pn = 'p2')[sname]
   ```

   G Expression:

   ```
   s[?sn,?x,?sname,?y] and sp[?sn,"p2",?z] and ?sname
   ```

2. Get supplier names for suppliers who supply at least one red part.

   Relational Algebra:

   ```
   (((p where color = 'red')[pn] join sp)[sn] join s)[sname]
   ```

   G Expression:

   ```
   p[?pn,"red",?c1] and sp[?sn,?pn,?q] and s[?sn,?c2,?sname,?st]
   ```

```
and ?sname.
```

3. Get all pairs of suppliers such that the two suppliers are located in the same city.
   Relational Algebra:
   ```
   define alias first for s; define alias second for s
   ((first times second) where first.city = second.city
   and first.sn < second.sn)[first.sn,second.sn]
   ```
   G Expression:
   ```
   s[?sn1,?c,?a1,?b1] and s[?sn2,?c,?a2,?b2] and ?sn1<?sn2
   and [[?sn1,?sn2]]
   ```

It should be noted that each of the *G* solutions shown above is a general solution that can be used for any type of base stream (i.e. *s*, *p*, and *sp* could be any type of value). Notice also how each of the *G* solutions above rely on constraints imposed by the patterns. For example, in the first query the same output variable *?sn* is used in the patterns for both the *s* and *sp* streams. This means that whatever value is bound to *?sn* in the first pattern for a given value of *s* must also be used as the value of *?sn* in the second pattern associated with stream *sp*.

6.4.2 Query-By-Example

The above examples of queries expressed in *G* demonstrate that the form or style of *G* queries is similar to query-by-example. The following additional examples illustrate more of this style. These examples were also taken from Date [Dat86]. For these examples we are assuming the existence of the same database used above except that the parts relation (*p*) has been extended to include a field for a part's weight.

1. Get supplier numbers for suppliers in Paris with status > 20.
   ```
   s[?sn,"Paris",?sname,>20] and ?sn.
   ```
2. Get supplier numbers and status for suppliers who either are located

in Paris or have status > 20.

```
s[?sn,?city,?sname,?status] and (?status>20||?city="Paris") and
[[?sn,?status]]
```

Notice in the second example above how the *or* operation is realized through the use of concatenation. If either or both of the conditions given in the concatenation are true then the concatenation will produce a non-empty stream. If both of the conditions are false then the concatenation will simply return the empty stream which is equivalent to failure.

3. Get parts whose weight is in the range 16 to 19 inclusive.
   ```
   p[?pn,?clr,?wt,?cty] and ?wt>=16 and ?wt<=19 and ?pn.
   ```
4. For all parts, get the part number and the weight of that part in grams. (Part weights are expressed in the relation in pounds.)
   ```
   p[?pn,?x,?wt,?y] and [[?pn,"weight =",454*?wt]]
   ```
5. Get all supplier-number/part-number combinations such that the supplier and part in question are colocated.
   ```
   s[?sn,?city,?x,?y] and p[?pn,?z,?wt,?city] and [[?sn,?pn]].
   ```

6.5 The Logic Paradigm

As mentioned earlier, through the use of output variables, pattern-matching and basic stream semantics, some of the important characteristics of the logical variable are able to be realized in *G* even though *G* does not actually employ the logical variable. In particular, *G* programs may be written in an incremental rule-oriented structure which allows some degree of bi-directional flow of information to and from *G* data values. Furthermore, output variables allow pattern-matching expressions to produce values based on the intersection of constraints.

In order to illustrate how to express in *G* the characteristics of the logic approach mentioned above we will first consider a *Prolog* program taken from Clocksin and Mellish [ClM81]. We will then consider an equivalent program written in *G*. Given below is the *Prolog* program.

```
aveTaxpayer(X) :-
        not(foreigner(X)),
        not(spouse(X,Y), grossInc(Y,Inc), Inc > 3000),
        grossInc(X,Inc), 2000 < Inc, 20000 > Inc.
grossInc(X,Y) :-
        not(recPension(X,P), p < 5000),
        grossSalary(X,Z),
        investInc(X,W),
        Y is Z+W.
```

The above program consists of two rules or Horn clauses. The first rule makes use of the second; both rules can be interpreted to be logical statements. The rule *aveTaxpayer* may be read, "X is an average taxpayer if X is not a foreigner and the spouse of X does not make a gross income of over 3000 and X makes a gross income of between 2000 and 20000." The body of the *aveTaxpayer* rule may be interpreted, therefore, as a compound conjunction whose component conjuncts are separated by commas. Note that not only does *aveTaxpayer* uses the rule for gross income (also given above) but it assumes the existence of additional rules or relations representing the entities *foreigner*, *spouse*, *recPension* and *investInc*.

An equivalent G program for deciding if a person is an average taxpayer is given below.

```
aveTaxpayer :=
        grossInc[?x,?Inc] and ?Inc>2000 and ?Inc<20000 and
        not(foreigner[?x]) and
        not(spouse[?x,?p] and grossInc[?p,>3000]) and
        [?x].
grossInc :=
        grossSalary[?x,?y] and
        investInc[?x,?z] and
        not(recPension[?x,<5000]) and
        [[?x,?y+?z]].
```

Like the *Prolog* program, the *G* program is incrementally constructed from program rules. The taxpayer rule and the gross income rule also depend upon the entities *foreigner*, *spouse*, *recPension* and *investInc*. These latter entities could also be rules or they could simply be "data" known to the system. In the case of *Prolog*, data would be represented by simple assertions or facts in the knowledge base. In the case of *G*, these facts would simply be stream values, quite possibly of type Relation.

Both the *Prolog* program and the *G* program are using uninstantiated variables within their rules to allow the binding of certain values to be based on the intersection of constraints. For example, in the final line of the *Prolog* rule for average taxpayer, the variable *Inc* is initially instantiated by the call to the *grossInc* rule and then subsequent conjuncts are used to assure that the value newly bound to *Inc* is between the limits of 2000 and 20000. This same technique is used in the first line of the *G* rule for average taxpayer.

Another similarity between the two programs above is that both programs may use the rules defined in more than one input/output mode. The determination of which variables are used for input and which are used for output may be varied for different applications of the rules allowing a bi-directional flow of information to and from the rules. For example, in *Prolog* and in *G* it is possible to use the *aveTaxpayer* rule to ask for the names of all the average taxpayers or to ask if a particular person, say John Smith, is an average taxpayer. Given below are examples of how to ask both of these questions in *Prolog* and in *G*.

Prolog Queries:

```
aveTaxpayer(X).
aveTaxpayer(johnSmith).
```

G Expressions:

```
aveTaxpayer[?x].
aveTaxpayer["JohnSmith"].
```

Both *Prolog* rules and *G* rules may be interpreted in more than one way. *Prolog* rules have both a logical and a procedural interpretation. Consider the following

Prolog rule:

```
son(A,B) :- father(B,A),male(A).
```

This rule may be viewed as a logical axiom and may be translated "A is the son of B if B is the father of A and A is a male." But this rule can also be interpreted procedurally by translating it "To answer the query *Is A the son of B?* answer first *Is B the father of A?* and then answer *Is A a male?* "

G rules may also be interpreted as logical implications similar to *Prolog.* The procedural interpretation of *G* rules differs from that of *Prolog*, however, since *G* is based on an underlying stream semantics. Consider the following *G* rule:

```
son := father[?B,?A] and male[?A] and [?A,?B].
```

This rule may also be viewed as a logical axiom and may be translated "A is the son of B if B is the father of A and A is a male". But the procedural interpretation of this *G* rule varies from that given for its *Prolog* counterpart. The *G* rule could be translated procedurally as "To generate the son and father pairs [A,B] first generate the father and son pairs B and A and then filter out any children that are not males." For a further illustration of a *G* procedural interpretation consider the average taxpayer rule. Seen from the viewpoint of the world of streams, this rule can also be viewed as a generator. The expression `grossInc[?x,?Inc]` serves as the base generator which generates values (*?x* and *?Inc*) that are subsequently run through several filters.

The programs given above demonstrate some of the similarities in style that can be achieved between logic programming in *Prolog* and programming in the logic paradigm in *G*. Although the logic approach to problem solving can be captured by *G*, there are differences between the two programs above that can be noted. First, note that *aveTaxpayer* and *grossInc* are being declared as the formal heads of Horn clauses in the *Prolog* program and as such are each declared with a specific number of argument variables. In *G*, however, *aveTaxpayer* and *grossInc* are simply variables being assigned *G* data values. No "argument" is associated with these variables. One consequence of this is that in *Prolog* it is possible to declare more than one *aveTaxpayer* rule each with a unique arity, whereas in *G* at a given scoping level, only one variable named *aveTaxpayer* may exist. No claim is being made for which situation is more desirable, it is merely being pointed out that this difference exists.

A second difference between *Prolog* and *G* rules rests in the radical difference between the underlying mechanisms of *Prolog* and *G*. Rules in *Prolog* are used to instantiate logical variables and the rules do not in themselves "return" a value. Rules in *G* are streams that return a stream value. Therefore, *G* rules may be used wherever any other value may be used in *G*. This allows rules to be used within any of the other paradigms that are able to be expressed in *G* thus achieving the integration of the logic paradigm with these other paradigms.

6.6 The Object-Oriented Paradigm

In the object-oriented paradigm data values exist as independent objects able to communicate by sending messages to each other. These types of data objects are easily created and manipulated in *G*. Simple primitives are supplied that allow the creation of user-defined types and the methods associated with those types. In order to demonstrate how to express the object-oriented paradigm in *G* a simple simulation of the operation of an ice cream store is developed below. Budd first developed this discrete event-driven simulation in Smalltalk [Bud87]. The Smalltalk code will not be presented.

Our simulation will involve the processing and recording of a number of actions. Each action will be marked with a time at which it should occur. The simulation will maintain its own relative "clock" which keeps track of the current time within the simulation. As time progresses, an event will occur when its marked time matches the current time on the simulation clock; the occurrence of that event will in turn control the sequence of future actions to occur in the simulation.

Our simulation will assume that there is never more than one event that is scheduled but which has not yet taken place. When this *pending* event does actually occur, it will cause the scheduling of the next pending event and so on. We will initially create a type called *Simulation* that will contain the actions of our simulation that are common to all similar discrete event-driven simulations and that are independent of our particular application. The user-defined type Simulation will then become the superclass of the application we will develop. The code that creates this new type is given below:

```
addtype{Simulation,Stream,
   local[currentTime:0,nextEvent,nextEventTime]}.
```

Notice that type Simulation has been defined as a subtype of Stream.

Three instance variables *currentTime*, *nextEvent* and *nextEventTime* have been defined. Here we are using the term *instance variable* in the same sense that it is used in the language *Smalltalk*. When an instance variable is defined for a given type, each instance of that type is created with a local variable of the same name. The variable *currentTime* will represent the simulation "clock" and will maintain the current time of the simulation. The variable *nextEvent* will record the next scheduled event and the variable *nextEventTime* will represent the time at which the next event is scheduled to occur. These three quantities would be common to all similar types of simulations and so are appropriately captured in the general class Simulation.

The code that creates the methods associated with the type Simulation is given below:

```
addop{Simulation,"time",func()[currentTime]}.
addop{Simulation,"addEvent",func(event,eventTime)
  [nextEvent:=event,nextEventTime:=eventTime]}.
addop{Simulation,"proceed",func()
  [currentTime:=nextEventTime,me::processEvent(nextEvent)]}.
```

The message *time* simply returns the current time of the simulation. In order to record the next event and its scheduled time the message *addEvent* has been provided. The message *proceed* will be sent when the next action is to take place. Since the interpretation of an event depends on the specific simulation being run, the method *proceed* leaves the interpretation and processing of the actual event to the subclass represented by the special variable *me*. This special variable refers to the original recipient of the message. At execution time it will refer to the actual type created for the specific application being run. One consequence of this is that the message *processEvent* must be recognized by each future subtype of type Simulation.

The type *IceCreamStore* will model our specific application of an ice cream store. The code to create this type is given next.

```
addtype{IceCreamStore,Simulation,local[profit:0]}.
```

Note that the type IceCreamStore is declared here as a subtype of type Simulation. Therefore, each instance of the type IceCreamStore will inherit all of the instance variables declared in type Simulation. Type IceCreamStore itself only has one declared instance variable, *profit*, that will be used to keep track of the profits accumulated during the simulation. In our simple simulation model, customers will arrive one at a time. A customer will order some number of ice cream scoops and then leave. The number of scoops ordered by a customer determines the profit received by the ice cream store. Our simulation has used *G*'s built-in function *random* to determine exactly how many scoops of ice cream each customer orders and to determine when the next customer will arrive.

The methods associated with type IceCreamStore that implement the above actions are given below.

```
addop{IceCreamStore,"init",func()[me::scheduleArrival()]}.
addop{IceCreamStore,"scheduleArrival",
   func()[me::addEvent(make{Customer},~me::time()+random(5))]}.
addop{IceCreamStore,"processEvent",func(event)[
   write["customer received at ",~me::time()],
   profit:=profit+event::numberOfScoops()*(0.17),
   me::scheduleArrival()]}.
addop{IceCreamStore,"reportProfits",
   func()[write["profits are ",profit,"\n"]]}.
```

The message *init* must be sent to initialize an IceCreamStore object. The method associated with *init* in turn sends the message *scheduleArrival* to the same receiver.

This message causes a new customer to be scheduled for a future time in the simulation. The message *processEvent* first reports the time at which the current customer arrived. It then records the profits made from that customer and finally it schedules the next customer and the time at which that customer will arrive. The final message *reportProfits* simply writes the profits that have been accumulated by the ice cream store during the simulation.

The last object needed by our simulation application is the customer. We must represent in some way the actions performed by the customers of our ice cream store. The *G* code that creates this new type named Customer is given below.

```
addtype{Customer,Stream}.
```

Note that the type Customer has been declared as a subtype of Stream, the root type of *G*. No instance variables have been associated with this new type. The message *numberOfScoops* is the only message associated with type Customer. This message results in the object "deciding" how many scoops of ice cream to order, printing out that information and then returning the number chosen to the sender of the message. The code for this method is given below.

```
addop{Customer,"numberOfScoops",func()[local[number],
   number:=random(3),
   write["customer has ",number,"scoops\n"],
   number]}.
```

All that is left now is to actually run our simulation. The expressions given below will do just that. It is assumed below that all of the code described above is contained in a file named *sim*.

```
include{sim}.
s:=make{IceCreamStore}.
s::init().
[while(s::time()<15)[s::proceed()]].
s::reportProfits().
```

The output from the above expressions is shown below in order to give some feel for the actual simulation.

```
customer received at 1
customer has 1 scoops
customer received at 6
customer has 2 scoops
customer received at 11
customer has 1 scoops
customer received at 14
customer has 1 scoops
customer received at 16
customer has 2 scoops
profits are 1.19
```

6.7 Integrating the Paradigms

In a preceeding section, which discussed the expression of the lambda-free paradigm in *G*, a matrix multiplication example was presented which integrated the stream semantics of *G* with the FP primitives suggested by Backus. In this section we present two programs that demonstrate the integration of the paradigms of *G* in a more extensive manner.

6.7.1 A Database With a View

For our first example, we will create a database to represent information about the employees and managers of a department store. To represent the employee information we will create a base relation which we will call *workRel*. Any given record of *workRel* contains the name of an employee and the name of the department within which the employee works. To represent manager information we will create a second base relation which we will call *managesRel*. Any given record of *managesRel* contains the name of a manager and the name of the department which the manager is in charge of. The two relations *workRel* and *managesRel* will be the only actual relations that exist in our database.

We will also create a view into our database. This view will represent information about who manages each employee in the department store. Any given record of this view will contain the name of an employee and the name of the manager of that employee. We will call this view *WorksForView.* Furthermore we will define one operation on this view called *add.* This operation will take two arguments: an employee's name and a manager's name. The *add* operation must implement the following rule:

```
If employee named is new but manager named is not then
   record new employee as working in manager's department
else if manager named is new but employee named is not then
   record new manager as managing employee's department
else if both manager and employee named are not new
   delete employee's old department information and then
   record employee as working in manager's department
```

We will now present the *G* solution to the database problem just presented. To begin, we will simply create the two relations named *workRel* and *managesRel.* This can be done directly in *G* and the code to do so is given below.

```
workRel := #String,String#.      & workRel[employee,department]
managesRel := #String,String#.  & managesRel[manager,department]
```

The comments given after each expression record the logical meaning of each field value for the relations.

Although we have created the basic relations of a "relational database" we will shift paradigms at this point and employ the object oriented paradigm in order to implement the view *WorksForView.* There are several ways in which representing a view as a type can be useful.

Defining a view as a user-defined type allows easy addition of new operations on the view by simply defining new messages and their associated methods for the view. In addition to this, and somewhat unique to object-oriented programming, we can define a number of different views and at the same time use many of the same messages for each view. For example, we could define several views as objects, each one of which utilizes the message *add.* Although the actual manipulations of the underlying database might be different for each view when it is sent the message *add*, the same logical operation of addition to a view would be occurring each time the message is sent, regardless of the view that receives it.

Making views types also allows the views to enforce limits on the manner in which the underlying database may be changed. A set of views and their messages can be defined which represent the only ways a user may legally interact with the database; the names of the actual underlying relations need not be known by the user. A system of views can, therefore, be used to enhance the security of a database and control the method of access to the relations that compose that database.

Directly below is given the *G* code that creates the view *WorksForView.*

```
addtype{WorksForView,Stream,
        workRel[?emp,?dept] and
        managesRel[?mgr,?dept] and
        [[?emp,?mgr]]}.
```

Notice that the view is installed as a subtype of Stream, the root of the *G* hierarchy. No instance variables have been defined on this view but an "interface" or visible aspect of the view has been defined. This interface represents the value returned by an instance of type WorksForView. If no interface is defined, then an instance of a type simply returns the empty stream. Note also that the interface defined has the form of a logical rule or a QBE query. This represents a virtual relation (i.e. view) into the actual underlying database.

The action *add* is defined for type WorksForView through the use of the *G* primitive *addop.* Using this primitive we simply define the message *add* and its associated method (function) for type WorksForView. The code to do this is given below.

```
addop{WorksForView,"add",func(emp,mgr)[
   not(workRel[emp,?x]) and managesRel[mgr,?dept] and
     insert(workRel,[emp,?dept]) ||
   not(managesRel[mgr,?x]) and workRel[emp,?dept] and
     insert(managesRel,[mgr,?dept]) ||
   managesRel[mgr,?new] and workRel[emp,?old] and
     (delete(workRel,[emp,?old])||insert(workRel,[emp,?new]))]}.
```

The body of the method or function associated with the message *add* is a conjunction composed of three major conjuncts. Each of these conjuncts implements the three conditions that message *add* was meant to embody and that were listed earlier. In the first conjunct, if the given employee's name is not found in the database but the manager's name is, then the employee's name is associated with the manager's department and is inserted into the underlying relation *workRel.* In the second major conjunct, if the manager's name given in not found in the database but the employee's name is, then the manager's name is associated with the name of the department within which the employee works and is inserted into the relation *managesRel.* Finally the third major conjunct takes a known employee and manager and removes the employee's old database record replacing it with a new record in which the employee is associated with the department of the given manager. We have thus provided a *Prolog* type rule as the body of our method for the message *add* which is itself associated with instances of objects of type WorksForView.

As an example of how to use our view, assume that information has been inserted into our base relations through the following *insert* expressions.

```
insert(workRel,["ben","cs"]).
insert(workRel,["bill","ee"]).
insert(workRel,["sally","ee"]).
insert(managesRel,["walt","cs"]).
insert(managesRel,["sue","ee"]).
```

Now we can simply create an instance of type WorksForView and assign the instance to a variable of our choice. For example, consider the following expression:

```
worksfor := make{WorksForView}.
```

After the above expression is executed, the value of variable *worksfor* is an object of type WorksForView. This variable then allows us to "see" into the database simply by typing the name of the variable. Thus if we were to type the name of the variable *worksfor* into the *G* interpreter the following value would be printed:

```
[[ ''ben'' ''walt'' ][ ''bill'' ''sue'' ][ ''sally'' ''sue'' ]].
```

If the expression `worksfor::add("ben","sue").` is then executed, the entry `["ben","cs"]` would be deleted from the underlying relation *workRel* and the value `["ben","ee"]` would be inserted into the same relation. Whatever actions are actually carried out on the underlying database, however, they are invisible and not of concern to the user of this view. What is important to the user is that the values "ben" and "sue" have been added to the value of *worksfor* in an appropriate way (i.e. that the view has been appropriately modified). Thus if we were again to type the variable name *worksfor* into the *G* interpreter the following value would now be printed:

```
[[ ''ben'' ''sue'' ][ ''bill'' ''sue'' ][ ''sally'' ''sue'' ]].
```

This abbreviated example of a database with a view demonstrates the manner in which the object-oriented paradigm can be integrated with the rule-based or logical approach to programming in order to solve a relational database problem. These are distinctly different paradigms and yet, since they are expressed within a single linguistic framework, they are intermixed and integrated in a manner appropriate to solving the problem at hand. Views are expressed as objects in order to benefit from the characteristics inherent in the object-oriented paradigm (see discussion above) whereas the use of rules as object methods allows the semantics of the methods to be emphasized promoting clarity and brevity which are qualities inherent in the logic programming paradigm. It is also important to note that the transition from one paradigm to another can be "seamless", that is, there need not be an awkward transition in moving from one paradigm to the next. This natural blending of paradigms supports writability in software; programs can be expressed in a manner that is natural for the problem or subproblem being solved.

6.7.2 The Buckets and Well Problem

For our next example of the integration of paradigms in *G* we present a solution to the buckets and well problem presented by Schwartz et. al in their discussion of the language SETL [SDD86]. In this particular statement of the problem, we are given two buckets, one of volume 3 quarts and the other of volume 5 quarts, and we are given a well full of water. We are asked to use the two buckets and the well to measure out exactly four quarts of water. Since *exactly* four quarts of water must be measured, the solution to this problem is limited to the execution of operations in which we can make exact measurements. The only operations that meet this requirement are operations of the following kind:

1. Any bucket can be filled completely full from the well.
2. Any bucket can be emptied completely.
3. Any bucket can be poured into the other until either the first bucket becomes completely empty or the second bucket becomes completely full.

Our solution to this problem will mainly make use of two paradigms, the applicative and the logic paradigms. We will use the applicative paradigm to construct a simple type of general problem solver where states are generated and then tested to see if they are target states. We will use the logic paradigm of programming to construct our generator of states so that the legal operations defined above can be encoded as simple rules which themselves generate the next set of states.

The *G* code that implements our "general problem solver" is given below:

```
wellprob := func(init)[local[rec],rec := #Int,Int#,
    if(istarget(init))[init] else[main([insert(rec,init)])]].
```

```
main:=func(states)[foreach(s:~genstates(states))[
     if(istarget(s))[reverse([s]||states)]
     else[self([s]||states)]
     ]].
```

The function *wellprob* simply tests the initial state given as its argument to see if it is a goal state. If it is, then the problem is trivially satisfied, otherwise the function *main* is called with that initial state. Notice that when *main* is called, the initial state is inserted into *rec* which is a relation that will maintain a record of all states generated in the course of the computations that solve the problem. The function *insert* simply returns the value it has inserted.

The function *main*, also given above, is a recursive function that tests each new state generated from the current state and then takes one of two possible actions for each state tested. If the state tested is a goal state *main* prints out the entire path of states that is associated with that goal state. This path represents a solution to the problem. If the state tested is not a goal state, *main* simply calls itself again adding that new state onto an accumulating path of states in which the new state becomes the current state. Notice that when no new states can be generated for a given current state, the function *main* returns the end of stream which effectively ends that line of inquiry.

The function *main* has the form of a general problem solver based on the generate-and-test search methodology. It requires that two functions *istarget* and *genstates* be defined. The function *istarget* simply tests a state for membership in the set of goal states. The function *genstates* generates a (possibly empty) set of new states from the current state so that the search for a solution may proceed. To use our general problem solver to find solutions for different problems with this same generate-and-test methodology, all that is needed is to redefine *istarget* and *genstates* for the problem domain of interest.

For our solution, the function *istarget* is very simple. If the sum of the two integers that make up the state passed to *istarget* is 4, then the state is a goal state.

The *G* code that implements this function is:

```
istarget := func(state)[if(plus(state)=4)[1]].
```

The function *plus* was borrowed from the FP style library (Appendix A). It simply adds together the first two elements of its argument stream.

The function *genstates* utilizes *Prolog*-like rules to encode the conditions that lead to the production of new states. The code for *genstates* is:

```
genstates := func(s)[local[st,b1,b2,b1b2,b2b1],
  st := ~s, b1 := @st, b2 := @st,
  b1b2 := lowest(5-b2,b1),    & amount to pour from b1 to b2
  b2b1 := lowest(3-b1,b2),    & amount to pour from b2 to b1
  not(rec[3,b2]) and insert(rec,[3,b2]) ||  & fill bucket 1
  not(rec[b1,5]) and insert(rec,[b1,5]) ||  & fill bucket 2
  not(rec[0,b2]) and insert(rec,[0,b2]) ||  & empty bucket 1
  not(rec[b1,0]) and insert(rec,[b1,0]) ||  & empty bucket 2
  not(rec[=b1-b1b2,=b2+b1b2]) and insert(rec,[b1-b1b2,b2+b1b2])||
  not(rec[=b1+b2b1,=b2-b2b1]) and insert(rec,[b1+b2b1,b2-b2b1])
  ].
```

The argument of *genstates* is a stream of states. It represents a partial path solution to the problem in reverse order. The first state of the argument stream represents the current state of that partial solution. The function *genstates* initially takes the current state and assigns the values of the bucket volumes within that state to local variables *b1* (for the 3 quart bucket) and *b2* (for the 5 quart bucket). It then computes the amount to be poured from bucket one into bucket 2 (variable *b1b2*) and from bucket 2 into bucket 1 (variable *b2b1*). Assignments always return the empty value sequence so none of the work done so far has contributed a value to the value sequence of *genstates*. The next term, however, is a disjunction whose disjuncts each encode a "rule" that, if satisfied, will generate a new state from the current state. These rules implement the three legal operations discussed earlier. The first two rules embody the operation of filling a bucket full. For example, consider the first disjunct, which is itself a conjunction. The code is:

```
not(rec[3,b2]) and insert(rec,[3,b2])          & fill bucket 1
```

The first conjunct of this expression is successful (i.e. returns a result) if the state that results from filling the first bucket full has not been previously generated (i.e. is not recorded in relation *rec*). If the state has not been generated previously then the second conjunct inserts it into relation *rec* and returns it, making it a member of the enclosing disjunction.

The third and fourth rules embody the operation of emptying a bucket completely. For example, consider the third disjunct, which is itself a conjunction. The code is:

```
not(rec[0,b2]) and insert(rec,[0,b2])          & empty bucket 1
```

In exactly the same manner explained above, if the state that results from emptying the first bucket has not been generated previously, then it is recorded and returned as one member of the larger disjunction within which the third conjunction exists. In an analogous way the last two conjunctions embody the operation of emptying one bucket into another. They also result in states being returned if they have not previously been generated.

For completeness it should be mentioned that the function *lowest* simply returns the lowest value of its two arguments. That function is defined below:

```
lowest := func(a,b)[if(a<b)[a] else[b]].
```

If the expression `wellprob([0,0]).` is typed into the interpreter and the code described above has been either included as a library or typed into the interpreter, the following two solutions are returned:

```
[[0 0] [3 0] [3 5] [0 5] [3 2] [0 2] [2 0] [2 5] [3 4] [0 4]]
[[0 0] [3 0] [0 3] [3 3] [1 5] [1 0] [0 1] [3 1]]
```

The solution presented for the buckets and well problem integrates two major paradigms in a way that takes advantage of the strengths of each one. The applicative paradigm was used to create an expression of a general problem solver which forms the basic framework of the solution. The simple procedural nature of the problem solver could be clearly and concisely stated using the applicative programming approach. The logic paradigm was used to allow the generation of states to be stated as rules. This allowed those operations that can legally generate new states to be encoded with clarity and brevity.

Chapter 7

Conclusions and Future Work

The exact nature of the relationship between thought and language is a deep and open question. The early twentieth century linguist Benjamin Lee Whorf saw the relationship as one of such intimacy that he wrote "thinking is a matter of different tongues" [Who79]. On the process of "understanding" itself Whorf has written:

> "The *WHY* of understanding may remain for a long time mysterious; but the *HOW* or logic of understanding - its background of laws or regulations - is discoverable. It is the grammatical background of our mother tongue, which includes not only our way of constructing propositions but the way we dissect nature and break up the flux of experience into objects and entities to construct propositions about." [Who79]

Whorf was referring to natural languages, but in an analogous fashion programming languages limit the ways in which we are able to think about problem domains of interest. Programming languages can impose upon our view a world of forms and processes inappropriate to the problem domain they are being applied to. A principal goal of multiparadigm research is the development of languages with which it is possible to express and integrate multiple and diverse world views. Such research investigates our potential to create flexible linguistic systems in which problem domains of interest may be "dissected" in a variety ways. The ideal is to be able to choose an approach to a problem that depends on the characteristics of the domain of our interest and not on the confining prejudice of the linguistic system we are utilizing.

The field of multiparadigm research is young; the foundations of the research are just now being explored. What effect, if any, the research will have upon programming languages of the future cannot be predicted at this time. What is needed are languages and language design efforts that help us to understand to what extent programming paradigms can be unified in one "uniform linguistic proposal". Once we have such languages, we need to earnestly explore what benefits such unifications provide. The language *G* takes us one step closer toward the development of a comprehensive multiparadigm language. It sheds light on some of the techniques that can be used to design such languages and it gives us a view of some of the success that can be expected by efforts to combine paradigms. Furthermore, it provides the core of a system that can be extended and experimented with in future research on multiparadigm languages and systems.

7.1 Conclusions

There are several ideas and conclusions that have emerged from this work. These conclusions are summarized below:

1. It is possible to integrate several paradigms into one "uniform linguistic proposal". The full extent to which this can be done has not yet been determined. The research presented in this paper has demonstrated, however, that it is possible to combine the main characteristics of the procedural, imperative, lambda-free, applicative, object-oriented and relational paradigms into one linguistic structure and that it is possible to integrate some of the attributes of the logic paradigm into this same linguistic framework.

2. Streams are a good datatype on which to build a multiparadigm language. They provide a unifying underlying semantics that helps to unite the diverse characteristics of different paradigms and they are very compatible with the requirements of a flexible, interpreted language.

3. The objects and structures of diverse paradigms can be given a form and interpretation compatible with and supportive of a deeper underlying semantics of a language. This can provide a unifying influence on the language that aids in the paradigm unification process. An example of this was the structure and interpretation of instances of user-defined types in *G*.

4. The object-oriented structure serves well as the basic structure of a multiparadigm language. It can provide a linguistic framework with extensibility, compactness and simplicity.

5. It is possible to make the object-oriented structure of a language serve a deeper underlying semantics based on the fundamental datatype of the language. This prevents the object-oriented structure from being imposed on the other paradigms integrated into the language.

7.2 Suggested Future Research

The research presented in this document has laid a foundation for the further exploration of multiparadigm language design and development; it has not produced a finished product. There are many improvements and extensions that could now be explored with respect to both the design of *G* and to the implementation of *G*. This section summarizes several of these possibilities for future work on the language design and implementation of *G*.

1. Develop a supportive multiparadigm environment around the *G* interpreter.

2. Add "paradigm assertions" to *G* which allow the programmer to declare his or her intention to remain completely within a given paradigm and to have certain policies enforced to assure that the paradigm is not violated. For example `assert(Functional)` could enforce a policy of no destructive assignment as well as dissallow the use of relations as function argument values. It could also be used to restrict function definitions from producing side effects.

3. Extend the pattern-matching facilities of *G* to include "sequence directed" pattern-matching. Sequence-directed pattern-matching expressions would allow the user to create and use stream patterns in the style of SNOBOL4 [GPP71] pattern-matching. A much enhanced capability for describing patterns would need to be developed, possibly based on SNOBOL patterns or regular expressions.

4. Add to *G* the ability to create concurrent processes that may be used to activate daemons and serve other special programming needs. Korth's [Kor86] "printer queue relation" could be implemented with this addition.

5. Extend *G* to include access-oriented programming [SBK86]. Add a new primitive that would allow functions to be evaluated each time a given variable is accessed.

6. Extend *G* to include trigger and constraint capabilities, better query optimization and the ability to handle large amounts of data. All of this would be directed toward making *G* a full database programming language. Bloom and Zdonik [BlZ87] provide an outline of the conflicting issues that have inhibited the creation of such languages. Andrews and Harris present an overview of the language *VBASE* which combines language and database advances in an object-oriented framework.

7. Implement *G* on the new VLSI stream hardware being developed at Simon Fraser University.

8. Develop a parallel implementation of *G*.

9. Add the logical variable to *G*.

Bibliography

[AnH87] Andrews, Timothy and Craig Harris. *Combining Language and Database Advances in an Object-Oriented Development Environment.* OOPSLA '87: Special Issue of SIGPLAN Notices, 22,12, December 1987, pp. 430-440.

[Abr86] Abramson, Harvey. *A Prological Definition of HASL: A Purely Functional Language With Unification-Based Conditional Binding Expressions.* Logic Programming: Functions, Relations and Equations, D. DeRoot and G. Lindstrom (Editors), Prentice-Hall, 1986.

[ASS85] Abelson, Harold and Gerald. J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[AsW77] Ashcroft, A. and W.W. Wadge. *Lucid, a Nonprocedural Language With Iteration.* Communications of the ACM, 20,7, July 1977, pp. 519-526.

[Bac78] Backus, John. *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.* Communications of the ACM 21,8, August 1978, pp. 613-640.

[Bai87] Bailey, Roger. *An Introduction to Hope.* Functional Programming: languages, tools and architectures. Susan Eisenbach (Editor), Ellis Horwood Limited, 1987.

[Bou87] Boutel, Brian. *Combinators as Machine Code for Implementing Functional Languages.* Functional Programming: languages, tools and architectures, Susan Eisenbach (Editor), Ellis Horwood Limited, 1987.

[BDL87] Baxter, Nancy, E. Dubinsky and G. Levin. *Learning Discrete Mathematics With ISETL.* Clarkston University, June 1987.

[BBL86] Barbuti, R., M. Bellia and G. Levi. *Leaf: A Language Which Integrates Logic, Equations and Functions.* Logic Programming: Functions, Relations and Equations, D. DeRoot and G. Lindstrom (Editors), Prentice-Hall, 1986.

[BeR85] Bellot, P. and B. Robinet. *Streams Are Not Dreams.* Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP Proceedings, G. Goos and J. Hartmanis (Editors), Springer-Verlag, May 1985.

[BlZ87] Bloom, Toby and S. B. Zdonik. *Issues in the Design of Object-Oriented Database Programming Languages.* OOPSLA '87: Special Issue of SIGPLAN Notices, 22,12, December 1987, pp. 441-451.

[BKK86] Bobrow, D.G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik and F. Zdybel. *CommonLoops: Merging Lisp and Object-Oriented Programming.* OOPSLA '86: Special Issue of SIGPLAN Notices, 21,11, November 1986, pp. 17-29.

[Bud87] Budd, Timothy. *A Little Smalltalk.* Addison-Wesley, 1987.

[ClM81] Clocksin, W.F. and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[Con88] The Conference on High-Speed Computing, Language Session Problems, Salishan Lodge, Gleneden Beach, Oregon, March 1988.

[Dat86] Date, C.J. *An Introduction to Database Systems: Volume I.* Fourth Edition, Addison-Wesley, 1986.

[DFP86] Darlington, J., A.J. Field and H. Pull. *The Unification of Functional and Logic Languages.* Logic Programming: Functions, Relations and Equations, D. DeRoot and G. Lindstrom (Editors), Prentice-Hall, 1986.

[FaL86] Faustini, Antony A. and E. B. Lewis. *Toward a Real-Time Dataflow Language.* IEEE Software 3,1, January 1986, pp. 29-35.

[FuH86] Fukunaga, Koichi and Shin-ichi Hirose. *An Experience with a Prolog-based Object-Oriented Language.* OOPSLA '86:Special Issue of SIGPLAN Notices, 21,11, November 1986, pp. 224-231.

[GhJ87] Ghezzi, C. and M. Jazayeri. *Programming Language Concepts.* John Wiley and Sons, 1987.

[GiR76] Gilman, Leonard and Allen J. Rose. *APL: An Interactive Approach.* John Wiley and Sons, 1976.

[GoR83] Goldberg, Adele and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[GHK81] Griswold, R. E. and D. R. Hansen and J. T. Kolb. *Generators in Icon.* ACM TOPLAS, 3,2 April 1981, pp. 144-161.

[Gri83] Griswold, Ralph E. *The Description and Manipulation of Sequences.* Tech. Rep TR 83-15, Dept. of Comp. Sci., University of Arizona, October 1983.

[GrO85] Griswold, R. E. and J O'Bagy. *Seque: A Language for Programming with Streams.* Tech. Rep. TR 85-2, Dept. of Comp. Sci., University of Arizona, January 1985.

[GrB85] Griswold, R. E. and J O'Bagy. *Reference Manual for the Seque Programming Language.* Tech. Rep. TR 85-4, Dept. of Comp. Sci., University of Arizona, March 1985.

[GrG83] Griswold, R. E. and M. T. Griswold. *The Icon Programming Language.* Prentice-Hall, 1983.

[GPP71] Griswold, R.E., J.F. Poage and I.P. Polonsky. *The SNOBOL4 Programming Language.* second edition, Prentice-Hall, 1971.

[Hai86] Hailpern, Brent. *Multiparadigm Languages and Environments.* IEEE Software 3,1, January 1986, pp. 6-9.

[Hai87] Hailpern, Brent. *Design of a Multiparadigm Language.* Notes from a session given by Dr. Hailpern at IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1987.

[Hen80] Henderson, Peter. *Functional Programming: Application and Implementation.* Prentice-Hall, 1980.

[Har84] Harland, D. M. *Polymorphic Programming Languages Design and Implementation.* Ellis Horwood Ltd., 1984.

[Har86] Harland, D. M. *Concurrency and Programming Languages.* Halsted Press, 1986.

[JGM86] Jenkins, Michael A., J. I. Glasgow, and C. D. McCrosky. *Programming Styles in Nial.* IEEE Software 3,1, January 1986, pp. 46-55.

[Kay77] Kay, Alan C. *Microelectronics and the Personal Computer.* Scientific American, 237,3, September 1977, pp. 230-244.

[KeR78] Kernighan, Brian W. and Dennis M. Richie. *The C Programming Language.* Prentice-Hall, 1978.

[Kor86] Korth, Henry F. *Extending the Scope of Relational Languages.* IEEE Software 3,1, January 1986, pp. 19-28.

[KoE88] Koschmann, Timothy and Martha Walton Evens. *Bridging the Gap between Object-Oriented and Logic Programming.* IEEE Software 5,4, January 1988, pp. 36-42.

[Lin85] Lindstrom G. *Functional Programming and the Logical Variable.* Twelfth Annual ACM Symposium on Principles of Programming Languages, January 1985, pp. 266-279.

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

[Moo86] Moon, David. *Object-Oriented Programmming with Flavors.* OOPSLA '86: Special Issue of SIGPLAN Notices, 21,11, November 1986, pp. 1-8.

[Red86] Reddy, Uday S. *On the Relationship Between Logic and Functional Languages.* Logic Programming: Functions, Relations and Equations, D. DeRoot and G. Lindstrom (Editors), Prentice-Hall 1986.

[Rum87] Rumbaugh, Jim. *Relations as Semantic Constructs in an Object-Oriented Language.* OOPSLA '87: Special Issue of SIGPLAN Notices, 22,12, December 1987, pp. 466-481.

[SBK86] Stefik, Mark J., D. G. Bobrow, and K. M. Kahn. *Integrating Access-Oriented Programming into a Multiparadigm Environment.* IEEE Software 3,1, January 1986, pp. 10-18.

[SDD86] Schwartz, J.T., R.B.K. Deware, E. Dubinsky, E. Schonberg. *Programming With Sets: An Introduction to Setl.* Springer-Verlag, 1986.

[Shr86] Shriver, Bruce D. *From the Editor-in-Chief.* IEEE Software 3,1, January 1986, pp. 2.

[StS86] Sterling, L. and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[SWP82] Schneider, G. Michael, Steven W. Weingart, and David M. Perlman. *An Introduction to Programming and Problem Solving With Pascal.* John Wiley and Sons, 1982.

[SuY86] Subrahmanyam, P. A. and J. You. *Funlog: A Computational Model Integrating Logic Programming and Functional Programming.* Logic Programming: Functions, Relations and Equations, D. DeRoot and G. Lindstrom (Editors), Prentice-Hall 1986.

[SuY84] Subrahmanyam, P. A. and J. You. *Pattern Driven Lazy Reduction: A Unifying Evaluation Mechanism for Functional and Logic Programs.* Eleventh Annual ACM Symposium on Principles of Programming Languages, 1984, pp. 228-234.

[Tur82] Turner, D. A. *Recursion Equations As A Programming Language.* Functional Programming and its Applications, J. Darlington, P. Henderson and D. A. Turner (Editors), Cambridge University Press, 1982.

[Wal86] Walker, Henry M. *Introduction to Computing and Computer Science With Pascal.* Little Brown and Company, 1986.

[WaG81] Wampler S. B. and R. E. Griswold. *The Implementation of Generators and Goal-Directed Evaluation in Icon.* Software—Practice and Experience, Vol 13, October 1983, pp. 495-518.

[Who79] Whorf, Benjamin Lee. *Language, Thought and Reality.* The M.I.T. Press, 1979.

[Wil84] Wilensky, Robert. *LISPcraft.* W. W. Norton and Company, 1984.

[Wis82] Wise, D. S. *Interpreters For Functional Programming.* Functional Programming and its Applications: An Advanced Course. J. Darlington, P. Henderson and D. A. Turner (Editors), Cambridge University Press, 1982.

[YoM86] Yonathan, Malachi and Zohar Manna. *TABLOG: A New Approach To Logic Programming.* Logic Programming: Functions, Relations and Equations, D. DeRoot and G. Lindstrom (Editors), Prentice-Hall, 1986.

APPENDICES

Appendix A

Functional Paradigm Library

```
&
& BACKUS'S PRIMITIVE FUNCTIONS
&
sel := func(s,n)[if(n=1)[~s] elif(n>1)[self(tail(s),n-1)]].
tail:= func(s)[local[x,z],z:=s,x:=@z,foreach(z)[z]].
id := func(s)[foreach(s)[s]].
atom:=func(s)[
   if(type(s)=Int||type(s)=Char||type(s)=Real||type(s)=Type)
      [1]].
reverse := func(s)[local[x],x:=~s,if(x)[self(tail(s)),x]].
null := func(s)[if(s)[break]else[1]].
not := func(s)[if(s)[break]else[1]].
distl := func(s)[local[y],y:=~s,
 foreach(z:~select(s,2))[[y,z]]].
distr := func(s)[local[z],z:=~select(s,2),
 foreach(y:~s)[[y,z]]].
len := func(s)[local[x:0],foreach(s)[x:=x+1],x].
or := func(s)[local[x],foreach(s)[x:=x||s],if(x)[1]].
appendL := func(y,z)[y,foreach(z)[z]].
appendR := func(y,z)[foreach(y)[y],z].
rightselect := func(s,n)[~select(reverse(s),n)].
tailR := func(s)[if(tail(s))[~s,self(tail(s))]].
rotateL := func(s)[local[x],x:=~s,foreach(z:tail(s))[z],x].
rotateR := func(s)[~rightselect(s,1),foreach(z:tailR(s))[z]].
trans := func(s,n:1)
[if(n<=length(~s))[[foreach(s)[~select(s,n)]],self(s,n+1)]].
&
& BACKUS'S FUNCTIONAL FORMS
&
composition := func(f,g)[~func(x)[f(g(x))]].
construction := func()
   [func(x)[foreach(args)[local[z],z:=args,z(x)]]]
condition := func(p,f,g)[func(x)[if(p(x))[f(x)] else[g(x)]]].
apply := func(f)[func(x)[foreach(x)[f(x)]]].
insert := func(f)[func(s)[local[x,y,z],
     s:=reverse(s),x:=@s,y:=@s,
     if(x)[if(y)[z:=~f([y,x]),
                 foreach(s)[z:=~f([s,z])],z]else[x]]]].
&
& LISP_LIKE ARITHMETIC OPERATORS USED BY BACKUS
&
```

Appendix A continued

```
plus := func(s)[~s + ~tail(s)].
sub := func(s)[~s - ~tail(s)].
mult := func(s)[~s * ~tail(s)].
div := func(s)[~s / ~tail(s)].
```

Appendix B

The Standard Library

```
&
& GENERAL ROUTINES
&
reverse := func(s)[local[x],x:=@s,if(x)[self(s),x]].
not := func(s)[if(s)[break]else[1]].
length := func(s)[local[x:0],foreach(s)[x:=x+1],x].
min := func(s)[local[low,x],low:=@s,
       foreach(s)[if(s<low)[low:=s]],low].
```

Appendix C

Grammar for the Language G

```
program:
          PERIOD
        | expression  PERIOD
        ;

expression:
          catterm
        | ID ASSIGN catterm
        ;

catterm:
          andterm
        | catterms CATOP andterm
        ;

catterms:
           andterm
        | catterms CATOP andterm
        ;

andterm:
          relational_term
        | andterms ANDOP relational_term
        ;

andterms:
          relational_term
        | andterms ANDOP relational_term
        ;

relational_term:
          term
        | relational_term RELOP term
        ;

term:
          factor
        | term ADDOP factor
        ;

factor:
```

```
            exponential
          | factor MULOP exponential
          ;

exponential:
            prim_term
          | prim_term EXPOP exponential
          ;

prim_term:
            base_term
          | PRIMITIVEOP prim_term
          ;

base_term:
            OUTPUTVAR
          | basic
           ;

basic:
            ID
          | GSTRING
          | CHARACTER
          | function_call
          | functional
          | intvalue
          | floatvalue
          | userobject
          | tuple
          | LPAREN expression RPAREN
          | relation
          | function_definition
          | pattern_match
          | TYPE
          | UTYPE
          | RANDOMkw LPAREN INTNUMBER RPAREN
          | INSERTkw LPAREN ID COMMA expression RPAREN
          | DELETEkw LPAREN ID COMMA expression RPAREN
          | INCLUDEkw LCURLY ID RCURLY
          | MAKEkw LCURLY UTYPE RCURLY
          | ADDTYPEkw LCURLY addtype RCURLY
          | ADDOPkw LCURLY addop RCURLY
          ;

intvalue:
```

```
          INTNUMBER
        | ADDOP INTNUMBER
        ;

floatvalue:
          FLOATNUMBER
        | ADDOP FLOATNUMBER
        ;

function_call:
          ID LPAREN expression_list RPAREN
        | function_definition LPAREN expression_list RPAREN
        ;

functional:
          function_call LPAREN expression_list RPAREN
        ;

function_definition:
          FUNCkw pexp ftuple
        ;


tuple:
          LBRACE local_setup texpression_list RBRACE
        ;

ftuple:
          LBRACE local_setup hdr_setup texpression_list RBRACE
        ;

local_setup
        : /* Action only - prepare a new local environment */
               {newlocal();}
        ;

expression_list
        : /* NIL */
        | expressions
        ;

expressions:
          expression
        | expression COMMA expressions
        ;
```

```
texpression_list
        : /* NIL */
        | texpressions
        ;

texpressions:
          texpression
        | texpression COMMA texpressions
        ;

texpression:
          expression
        | codebody
        | LOCALkw LBRACE local_stream RBRACE
        | SELFkw LPAREN expression_list RPAREN
        | WRITEkw LBRACE expression_list RBRACE
        | BREAKkw
        ;

relation:
          POUND relation_list POUND
        ;

relation_list
        : /* NIL */
        | relation_spec
        | relation_list COMMA relation_spec
        ;

relation_spec:
          TYPE
        | UTYPE
        ;

codebody:
          FOREACHkw cparam tuple
        | WHILEkw wparam tuple
        | IFkw ifparam tuple else_clause
        | REPEATkw tuple
        | range_specification
        ;

range_specification:
          range_item TOkw range_item
        | range_item TOkw
        | range_item TOkw range_item STEPkw stepvalue
        | range_item TOkw STEPkw stepvalue
        ;

stepvalue:
        INTNUMBER
        | ADDOP INTNUMBER
        ;

range_item:
          INTNUMBER
        | CHARACTER
        ;

else_clause
        : /* NIL */
        | ELSEkw tuple
        | ELIFkw ifparam tuple else_clause
        ;

hdr_setup
        : /* Action only - prepare a header for a function */
              {newhead();}
        ;

pexp:
          LPAREN parameter_stream RPAREN
        ;

parameter_stream
        : /* NIL */
        | parameter
        | parameter COMMA parameters
        ;

parameters:
          parameter
        | parameter COMMA parameters
        ;

parameter:
          ID
        | ID COLON expression
        ;
```

```
ifparam:
          LPAREN texpression RPAREN
        ;

cparam:
          ifparam
        | LPAREN ID COLON expression RPAREN
        ;

wparam:
          LPAREN relational_term RELOP term RPAREN
        ;

pattern_match:
          ID LBRACE pattern_stream RBRACE
        ;

pattern_stream:
          pattern
        | pattern_stream COMMA pattern
        ;

pattern:
         OUTPUTVAR
        | RELOP expression
        | basic
        ;

local_stream
        : /* NIL */
        | local
        | local COMMA locals
        ;

locals:
          local
        | local COMMA locals
        ;

local:
          ID
        | ID COLON expression
        ;
```

```
local_decl:
        LOCALkw LBRACE local_stream RBRACE
      ;

addtype:
        UTYPE COMMA atype
      | UTYPE COMMA atype COMMA expression COMMA local_decl
      | UTYPE COMMA atype COMMA expression
      | UTYPE COMMA atype COMMA local_decl COMMA expression
      | UTYPE COMMA atype COMMA local_decl
      ;

atype:
        TYPE
      | UTYPE
      ;

addop:
        atype COMMA GSTRING COMMA function_definition
      ;

userobject:
        ID COLON COLON ID LPAREN expression_list RPAREN
      ;
```