The Design and Implementation of High-Level Programming Language Features for Pattern Matching in Real Time*

Kelvin Nilsen

TR 88-30

July 21. 1988

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants DCR-8401831. DCR-8502015. and DCR-8713690.

This technical report has been submitted as a dissertation
to the faculty of the Department of Computer Science in
partial fulfillment of the requirements for the degree
of Doctor of Philosophy in the graduate college of
the University of Arizona.

TABLE OF CONTENTS

TABLE OF CONTENTS – Continued

ABSTRACT

High-level programming language features simplify software development by eliminating many low-level programming concerns and by providing programmers with useful abstractions to simplify description and analysis of their programs. This dissertation discusses briefly some of the special needs of structural pattern-matching programs that must execute in real time and suggests language features to support these needs. These language features are implemented in an experimental version of the Icon programming language and examples of how these language features can be used are presented. This dissertation also presents and discusses the implementation of these language mechanisms, including the implementation of a new algorithm for garbage collecting linked data structures and strings in real time.

One of the new language features is a stream data type, which allows programmers to perform pattern matching directly on sequences of data values produced by external sources, without requiring explicit read operations to bring the data into memory before analyzing it. Other new language features provide the ability to create and manipulate concurrent Icon processes, between which the stream data type serves as the principal mechanism for interprocess communication. Stream and concurrent process manipulation mechanisms integrate naturally with each other and with the existing mechanisms of the Icon programming language. Sequential Icon programs are, for the most part, unaffected by the new language capabilities.

Chapter 1: Introduction

Real-time programs are programs that interact with their environment in such a way that program correctness depends not only on correctly processing incoming data, but also on processing this data in conformance with strict timing constraints. Generally, real-time programs must process each quantum of information within some small fixed time of when that data quantum becomes available. Real-time constraints usually consist of an upper bound on system response time, the time required to respond to a particular stimulus. For real-time systems, delivering correctly computed results too late is often as wrong as delivering incorrectly computed results. For example, a robot working on an automobile assembly line is not functioning correctly if it carries out its actions after the car it was to assemble has left its assembly station.

Many of the problems scientists and engineers need to solve in real time are pattern-matching problems. Traditional data communication, for example, requires that certain sequences of bits received from a remote computer be recognized as data packets as they arrive. Voice recognition attempts to recognize particular analog signals as syllables, sequences of syllables as words, and sequences of words as sentences. Interactive voice recognition systems must perform this processing as words are spoken by human users. Sophisticated hospital equipment designed to monitor a patient's activity watches for patterns representing irregular heart beats, heart attack, respiratory distress, sleep cycles, and more. Even the automated monitoring of a busy airport's radar signals for potential collisions might be formulated as a pattern-matching problem.

Patterns classify or categorize all possible data values according to their compliance with certain rules or specifications. Regular expressions and context-free grammars are, for example, two popular notations describing patterns for string data. A pattern to recognize valid data packets might depend on calculated CRC checksums. Most patterns consist of groups of rules, all of which must be satisfied in order for a pattern to be matched. A pattern describing the potential for an aircraft collision might consist, for example, of one rule requiring that two airplanes pass over the same ground location at the same time, and another rule specifying that the two planes are flying at the same altitude. Many patterns are hierarchical. For example, a pattern representing a subject of an English sentence might be comprised of two subpatterns representing individually an article and a noun. Some patterns are recursive. For example, the definition of an English noun phrase includes the possibility of a noun phrase being composed of an adjective followed by another noun phrase.

The design of efficient pattern-matching software is a difficult problem by itself. This problem is even harder if tight bounds on the time required for execution are imposed on the solution. For example, hospital equipment designed to synchronize a mechanical device with a patient's heart rate may have to recognize the phases of each heartbeat and notify the controller of each phase change within a small fraction of a second.

1.1 Pattern Matching

A variety of different approaches are commonly used in computerized pattern matching. Most of the pattern-matching approaches in general use today and most research in the field of pattern matching are based on statistical pattern-matching techniques [1]. Statistical pattern-matching systems generally consist of two phases. The first phase takes certain measurements on the input data. This is called feature extraction. The second phase, called classification, places the data in some class or category according to the measurements produced by the feature extractor. Template matching, used by many optical character readers, is an example of statistical pattern matching. In this application, the feature extractor produces an array of measurements, each value denoting whether pencil marks are found in a particular location on a specially printed form. The output of the feature extractor is processed by the classifier, which measures the similarities between the array of measurements and a set of pattern templates, classifying the input as whichever character it most closely resembles, or as an unknown character if a certain threshold of differences is exceeded for all of the pattern templates.

This dissertation focuses on a different kind of pattern matching known as structural pattern matching. This pattern-matching strategy does not use probabilistic inference or statistical sampling. Instead, structural patterns describe relationships of rules that are hierarchical in that complicated patterns are constructed out of simple patterns. Syntactic pattern matching is the name given to structural pattern matching in which grammars are used to describe the relationships between simple patterns and the more complicated patterns constructed from them. For example, a context-free grammar production $A \rightarrow BC$ represents the idea that the pattern A is matched if the patterns B and C are matched. The simplest of syntactic subpatterns are called pattern primitives. Pattern primitives constitute the alphabet of a grammar and are recognized by methods that are distinct from those used to determine whether a string of pattern primitives fulfills the requirements of a pattern grammar. For example, the pattern primitives of a programming language compiler are lexical tokens that are typically recognized by lexical analysis carried out separately from parsing. It is common, in complicated pattern-matching applications, for pattern primitives to be recognized by statistical pattern-matching techniques. Both recognition of spoken syllables [2] and grouping of an asynchronous serial stream of bits into individual characters [3, p. 4] generally use statistical pattern-matching techniques. One of the first steps in designing a syntactic pattern-matching system is to select the pattern primitives. A variety of considerations contribute to this design decision. For example, the availability of commercial hardware or existence of efficient algorithms to convert sensor output into discrete values might influence the choice of pattern primitives. These are some of the reasons that individual syllables have been proposed in place of voice phonemes as the pattern primitives of a voice recognition system [2, p. 48].

As an example of syntactic pattern matching, consider the formulation of a pattern to recognize high risk configurations of airplanes on a radar screen. Assume that a square radar screen is divided evenly into n rows and n columns. Suppose that each of the n^2 divisions of the radar screen maintains a count of the number of airplanes in that division. These integer counts could be stored in a one-dimensional array of length n^2 within which the first n items represent the top row, the next n items represent the second row, and so on. This array is the standard input to the pattern-matching system. In order to simplify the detection of potential collisions, three pattern primitives are used. In this example, each pattern primitive corresponds to a single position in the incoming array. The three pattern primitives, represented in the following grammar by the terminal symbols 0, 1, and @, denote the idea that 0, 1, or more aircraft respectively are located in a particular division of the radar screen. Using these definitions, the grammar shown below, for which <danger> is the start symbol, describes any string of pattern primitives with more than one plane is flying in a single square[1]. In this grammar, ε represents a string of length zero.

[1] This solution is overly conservative in that it ignores aircraft altitudes. A more precise solution would use a 3-dimensional coordinate system.

```
<danger>      → <arb> @ <arb>

<arb>         → <any> <arb> | ε

<any>         → 0 | 1 | @
```

The risk of collision is high not only when multiple planes are flying within the same square, but also when planes are flying in adjacent or diagonally connected squares. This notion is approximated by the following grammar:

```
<danger>      → <arb> <trouble> <arb>

<trouble>     → @
                | 1 <plane>                    # horizontally adjacent
                | 1 <row> <plane>              # vertically adjacent
                | 1 <short-row> <plane>        # left-diagonally adjacent
                | 1 <long-row> <plane>         # right-diagonally adjacent

<plane>       → @ | 1

<short-row>   → <any>_1 <any>_2 ... <any>_{n-2}

<row>         → <short-row> <any>

<long-row>    → <row> <any>
```

In the grammar above, <short-row> represents any string of length $n - 2$. The subscripts in the production for <short-row> number each of these characters. All diagonally adjacent squares on the radar screen are separated in the linear representation of the two-dimensional array by strings either of length $n - 2$ or n. Adjacent squares are separated by strings either of length 0 or $n - 1$. Note that this grammar is only an approximation of the desired notion in that it ignores subscripting problems that occur with the first and last members of each row (for which certain diagonally adjacent squares do not exist).

Pattern matching is an important part of many commonly used applications. Most user interfaces, for example, employ some sort of pattern matching to reject illegal user requests. Likewise, an important responsibility of a programming language compiler is to verify that a source file represents a legal program. And in computer networks, low-level software recognizes thousands of valid data packets each second. Many of the patterns in common use today benefit from the fact that the data processed and the patterns themselves were designed by people for relatively easy matching.

However, many real-world patterns and data do not benefit from these simplifications. Grammars for natural languages are, for example, usually ambiguous and require non-deterministic parsing algorithms. Even though many of the languages represented by real-world patterns are finite and thus regular sets, context-free grammars are frequently preferred over regular expressions. This is because their hierarchical formulation provides a more concise description of a pattern and because their description may model structural hierarchies occurring in nature. Another advantage of context-free grammars over regular expressions is that context-free parsing methods generally provide more information about the matched data in the form of a parse tree than is available from a deterministic finite automaton. Due to the complexity of many real-world patterns, special matching strategies have been developed. Stochastic grammars [4, p. 124], for example, assign probabilities to the productions of a context-free grammar in order to select the most likely parse from a set of valid parse trees. In order to improve matching performance, and to represent context-sensitive pattern features, *ad hoc* rules frequently supplement the formal specification of a context-free grammar. Not all syntactic pattern matching uses context-free grammars. In searching for pattern notations that are most appropriate for particular problems, some syntactic pattern matching has, for example, been based on transformational grammars [4, p. 53]. Syntactic pattern-matching researchers have also experimented with two-dimensional [5,6], plex [7,8], web [9], graph [8,10], and tree grammars [11-14]. Other researchers have focused on lower level abstractions such as cellular automata [15] and perceptrons [16].

Most real-world pattern-matching applications perform pattern matching on data values arriving from an external physical environment that are processed in several phases. The first phase, called preprocessing, provides an interface to the external environment, frequently converts data from one notation to another, and attempts to improve the general quality of the incoming signal. For example, the preprocessor may convert analog voltage levels to digital values and may filter out high-frequency electronic noise or provide some form of image enhancement. In the aircraft collision detection example described above, preprocessing consists of scanning each division of the radar screen in order to determine the integer values representing the number of planes flying in that division. The second phase of pattern matching converts the preprocessor's output into a language-like structure. This corresponds, in this example, to converting the array of integer values into a string of pattern primitives. The final phase of pattern matching performs syntactic analysis on the language-like structure produced by the previous processing phase. For the collision detection system, this consists of parsing the string of pattern primitives.

1.2 Real-Time Programming

Real-time programs generally process data representing characteristics of some physical environment as the data becomes available from the environment. An important characteristic of real-time programs is that they must be able to process data as fast as it is produced by the environment. In contrast, the data processed by other systems is generally stored in files where it may wait long periods of time until the computer is ready to process it. In real-time systems, scheduling constraints are imposed by the external environment. Generally, data that arrives at time t must be processed by time $t + \Delta$ where Δ is determined by the implementors of the real-time system and generally depends on physical characteristics of the environment and the purposes of the pattern-matching software.

In order to perform pattern matching in real time, the time required for each phase of pattern matching must be bounded by a small constant. Consider the aircraft collision detection system described above. The preprocessor repeatedly scans the entire radar screen, one square at a time, counting the number of aircraft appearing in each square. In order to guarantee a certain minimum scan rate, a fixed bound must be placed on the time required to count planes in each square of the radar screen. This time bound could be justified, for example, if each square contains a fixed number of pixels, each of which is represented by a fixed number of bits. The next phase of pattern matching replaces each count with one of the three pattern primitives described in §1.1. Clearly, the computation required to recognize these pattern primitives is bounded by a constant. Syntactic analysis, for this application, requires that each symbol of the string of primitives be examined. If the symbol is 0, no further work is required to process that symbol. Likewise, if the symbol is @, danger is recognized immediately, without any additional processing. If, however, the symbol is 1, then a maximum of four other symbols must be examined to determine if any of these is either 1 or @. The time required to perform this work also is bounded by a constant.

Frequently, the stream of information that is processed or scanned by each phase of pattern-matching software is conceptually divided into segments, each of which is passed through a particular phase of processing as an entire unit. For example, the continuous sequence of voltage readings representing electrocardiogram output might be divided into segments representing single heartbeats and digitized voice input might be divided into segments representing sentences. When pattern matching requires segmentation of the data stream, processing of data is delayed by the amount of time required to collect a complete segment. This time is bounded by a constant if the size of the segment is bounded and if a lower bound on the incoming rate of data exists. An upper limit on the size of a segment can be used to place an upper bound on the problem size for a particular phase of pattern matching. Based on this bound, it is possible to determine the maximum amount of time required for execution of that pattern-matching phase. By restricting the problem size in this way, it is possible to obtain constant response time even for algorithms of exponential complexity.

There is a wide variety of patterns that can be matched in constant time. Detailed measurements and analysis of programs and of the environments in which real-time systems operate are generally required to guarantee that particular software systems are capable of processing as much data as the physical environment provides as fast as the data becomes available. This dissertation focuses on the general idea that real-time programs must process each data value in constant time. Whether the constant time required is sufficiently small to satisfy real-time constraints depends in most cases on the computing and physical environments. Analysis of these relationships must be performed for each application independently.

1.3 High-Level Abstractions and Programming Languages

People use natural language not only to communicate with other people, but also to represent and organize ideas within their minds. Likewise, computer programming languages serve not only to communicate with computers, but also to represent and reason about algorithms within the mind of a programmer. Because language is such an important part of a person's reasoning ability, it is important that, for particular application domains, language provide abstraction mechanisms to facilitate description and solution of problems.

The linguist Benjamin Whorf was the first (or certainly the most famous) to suggest professionally that a person's natural language has a significant effect on how that person conceptualizes experiences and solves problems [17]. The ability to describe abstract ideas using language facilities is an important part of being able to understand those ideas. Some languages, for example, have words that cannot be translated into single words or even phrases in other languages. For example, Tahitians have no word for loneliness. Not only is the word missing from their vocabulary, but also the concept of loneliness is missing from their conceptual framework [18, p. 305]. Differences between languages are more than just distinctions in vocabulary. The Hopi language uses, for example, the suffix *-ta* to denote that the characteristics of the root word to which it is appended repeat or vibrate [19]. Regarding this linguistic mechanism, Benjamin Whorf concluded that it "practically forces the Hopi to notice and observe vibratory phenomena, and furthermore encourages them to find names for and to classify such phenomena."

Researchers Kay and Kempton recently hypothesized that English speakers would solve a particular problem in a different way than would speakers whose native tongue was Tarahumara

[17]. They suspected that the differences between notations used for reasoning or thinking about that problem would lead speakers of the different languages to arrive at different conclusions. They found that, for 29 out of 30 English-speaking subjects, the biases of the English language used to solve the problem led to incorrect conclusions consistent with their hypothesis. However, errors made by Tarahumara speakers were randomly distributed. To demonstrate that the different behaviors were in fact due to differences in language and not cultural biases, the English-speaking subjects were asked to solve the same set of problems again, but in the second formulation of the problem, care was taken to present the problem to the subjects in such a way that the subjects would avoid the biases of English when reasoning about the problem solution. Here, the performance of English speakers was roughly equivalent to the performance of the Tarahumara speakers on the first problem.

So a person's language determines to some degree how that person organizes abstractions. In general, it is not possible to claim that one form of linguistic abstraction is better than another. Frequently, one can argue the benefits of each of a large number of alternative abstractions for a particular idea. To understand the behavior of electricity [20], for example, it is sometimes useful to think of electricity as the flow of a fluid from high to low potential. In other situations, however, treating electricity as a crowd of electrons is a more appropriate abstraction since for particular problems, this model leads more directly to an understanding of how electricity works. And to understand the behavior of certain semi-conductors, it becomes useful to think of electricity as a crowd of empty holes representing positive charges.

Concerning the large numbers of natural languages that exist in the world today, the linguist George Lakoff wrote [21, p. 306]:

> There is a wide variety of reaction to the idea that other human beings comprehend their experiences in ways that are different from ours and equally valid. Some people are scared by the idea: *How can we ever communicate with people who think differently?* Others are repelled by the idea: *There should be a universal language so that we can minimize misunderstanding and conflict.* Others are attracted by the idea: *Maybe I could learn to think differently too. Maybe it would make my life richer and more interesting.*

Lakoff's observations apply equally well to computer programmers faced with choices between the many programming languages that are available to them. Computer scientists who study programming languages are, for the most part, members of Lakoff's third category. The goals of these computer scientists are to find new conceptual foundations that are especially useful in dealing with particular classes of problems. The expectation is that proper choice of abstractions can significantly reduce the complexity of a problem and simplify implementation of a solution.

Early computers were programmed using notations that, by today's standards, were very primitive and awkward to use. The design of these notations was motivated not by a desire to simplify the writing of computer programs, but instead by a need to minimize the efforts required to translate and run the programs. Programming languages that are motivated principally by implementation concerns are generally referred to as low-level languages. In contrast, high-level languages attempt to provide programmers with a good programming notation[2], regardless of the perceived complexity of implementing the language. Clearly, the distinction

[2] In this context, the word "notation" denotes both syntax and semantics.

between high- and low-level languages is one of degree. Likewise, the evolution of programming languages has been a gradual, uneven process marked by occasional setbacks and a large number of diverging paths.

One of the principle motivations for the current trend toward the design and use of high-level programming languages has been the great difficulty within the software industry of producing quality software in a timely and economical manner. The observation that the capabilities of computer hardware and the expectations of sophisticated computer users far exceed the ability of the software industry to deliver the necessary software capabilities has been labeled a "software crisis". Numerous solutions, including the design and implementation of new high-level programming language features, have been proposed to resolve this crisis [22-24]. Ultimately, all of these proposed solutions will likely contribute in some way to solving the problem.

High-level languages benefit programmers in at least two ways. First, they simplify programming by automatically taking care of many of the details involved in describing an algorithm to a computer. The details managed by a programming language may include the allocation and reclamation of system memory using garbage collection, automatic conversion between different types or representations of data, and automatic type and subscript checking to verify that operations to be performed on data are consistent with the data supplied as arguments to those operations. The second principal benefit of high-level programming languages is that they offer programmers an opportunity to conceptualize algorithms in new and different notations. Icon [25], for example, evaluates expressions using a goal-directed evaluation mechanism. Programmers who learn to think in terms of this mechanism have increased their repertoire of tools for modeling solutions to programming problems.

Most programmers who learn to use high-level languages find that their productivity is greatly increased. However, high-level languages are often criticized because programs written in these languages generally run slower than programs written in lower-level languages. This is the result, in part, of the fact that high-level languages do for the programmer much of what the programmer must do for him or herself in low-level languages. The costs of performing these services for the programmer are incurred in the implementation of a high-level language partially when a program is translated and partially when it is executed. Research efforts dedicated to reducing the implementation costs of high-level languages have made and will undoubtedly continue to make advances in reducing the overhead of high-level languages.

For the moment, however, the costs of programming in high-level languages are real and significant. And convincing arguments suggest that, although there is room for improvement, high-level languages will always run slower in general than well-written programs developed in low-level languages. These arguments are based on the inherent complexity of the services provided by high-level languages and the inherent computational complexity of many of the problems that must be solved in order to perform code optimizations.

The decision to use a high-level language for software development depends therefore on balancing the tradeoffs between the time and efforts required to develop software and the anticipated costs of running that software. Programs that run only once, for example, but may require a significant amount of effort to develop, generally would benefit from use of a high-level language. In contrast, a small program that is easy to develop and must process large amounts of data in minimal time should probably be written in a low-level language.

System prototypes, implemented in order to experiment with design decisions, are the sorts of programs that may require a large amount of development effort and process only a small amount of data in their typically short lifetime. This makes prototype development a good candidate for high-level language programming. It is common for the feedback provided by experimentation with the prototype to require modification of the prototype in order to experiment with alternative design decisions. Many of the same facilities that make the initial implementation of a system easier in a high-level language also make modifications to an existing implementation easier than in a low-level language.

1.4 Pattern Matching in Real Time using High-Level Language

Prototypes are used not only to evaluate new products in commercial environments, but also to investigate new algorithms for solving particular problems in research environments. Since many real-world patterns are not yet well understood, considerable research effort has been dedicated to experimenting with new algorithms for recognizing particular classes of patterns. The patterns associated with voice recognition, for example, have been the focus of much research over a period of many years, and still it is not technologically feasible for a computer program to recognize arbitrary words spoken by arbitrary speakers. Research in real-world pattern matching consists, in large part, of implementing prototypes, collecting data on how well the prototypes perform, and adjusting the pattern model for further prototyping. This is an application domain where high-level languages excel. Not only can high-level languages shorten the time required to build a prototype in order to gather feedback, but because of the relative ease of prototyping in high-level languages, many important problems that might otherwise have been judged too difficult to study may also be investigated.

Because of the close relationship between real-time programs and their physical environment, the throughput of a real-time system is usually determined not by the capabilities of the computer, but instead by the rate at which data values are produced in the physical environment. Real-time programmers have traditionally focused a great deal of attention on the requirement that their systems be able to process data at the maximum rate at which it might arrive from the environment. However, they may inadvertently overlook the fact that once these performance requirements are satisfied, further efforts to increase the speed of their systems provide no benefits. Programmers who write in high-level languages frequently justify the time penalty their programs incur by arguing that the benefits of obtaining faster solutions are not worth the efforts required to implement them. In many circumstances, these arguments are subjective and may not be very convincing. However, for many real-time applications, arguments such as these are quantifiable and supported by real measurements. For example, high-performance 32-bit microcomputers are now available for the same price as much slower 8-bit computers of just five years earlier. Today's newer, cheaper processors are capable of executing within tighter real-time constraints high-level language implementations of the same computations described in low-level notations for earlier generations of computers. As progress in the engineering and manufacturing of microprocessors continues, increasing numbers of real-time applications can benefit from the advantages of high-level languages.

Chapter 2: Control Requirements of Real-Time Pattern-Matching Systems

Much research has been dedicated independently to linguistic facilities supporting the problem domains of real-time and pattern-matching systems but only limited attention has been focused on the relationship between these two problem domains. Each of these domains has special needs for control abstractions. For example, many pattern-matching algorithms make use of lookahead and backtracking. These tools of pattern matching are used so frequently that special language mechanisms to facilitate their implementation might be justified. Likewise, real-time programs generally deal at a fairly low level with the external environment and must pay close attention to the times at which events occur and the time required to respond to those events. Real-time programming might benefit from special abstractions that represent changes in the environment, or special facilities for interacting with system clocks and timers.

2.1 Goal-Directed Expression Evaluation

When cast in the framework of grammars, many real-world patterns are represented by languages that are inherently ambiguous and require non-deterministic parsing. Most patterns describing natural languages, for example, are plagued by the same ambiguities that complicate human understanding of spoken speech. Backtracking, an important tool used in the implementation of non-deterministic parsers, is supported by special language mechanisms in, for example, SNOBOL4[26], Prolog[27], and Icon[25].

Backtracking in SNOBOL4, Prolog, and Icon is guided by goal-directed expression evaluation. In Icon, the goal of expression evaluation is to produce a result. A complicated expression may consist of several subexpressions, each of which must produce a result in order for the compound expression to produce a result. In other words, an Icon goal may consist of multiple subgoals. For example, the goal representing a valid English sentence in a natural language parser might be comprised of subgoals representing individually a subject and a predicate. In Icon, when parts of a subexpression fail to produce results[3], backtracking to previous subexpressions occurs automatically and those subexpressions are given the opportunity to satisfy their subgoals in alternative ways.

The mechanisms provided by Icon have proven themselves useful in performing pattern matching on string data. The same control structures could also be applied to more general data types. High-level language facilities to implicitly support goal-directed evaluation greatly simplify the implementation of many pattern-matching applications.

2.2 Synchronization

An important characteristic of real-time programs is their need to respond quickly to changes in their physical environment. As changes occur in the environment, programs must become aware of these changes and, in general, produce output that is a function of these changes within some small fixed time Δ. Real-time programs become aware of changes in the environment by means of sensors which measure characteristics of the environment, reporting these characteristics either at fixed time intervals or whenever changes occur. The requirement that data be

[3] Subexpressions fail only after they have exhausted all available alternatives for producing a result. Subexpressions that execute infinite loops do not fail in this sense.

processed within time Δ of becoming available is essentially a problem of synchronizing the program with its external environment.

In many real-time pattern-matching applications, the domain of patterns is the sequence of values produced by sensors in the physical environment. For example, pattern-matching software might have the responsibility of monitoring the output produced by an electrocardiogram. In this application, a sequence of values representing voltage readings taken at fixed time intervals would be reported to the application program from the remote sensors. It is important that the program be able to process each value of this sequence as soon as it is produced by the external environment.

The sequence of values available from an external sensor can be processed only after it has been read into memory from the device representing the sensor. In most programming languages, explicit instructions to read data from the operating system must precede all attempts to process that data. This complicates pattern matching because it requires that the pattern-matching system read data prior to processing. The difficulty with this is that, in general, it is not known prior to execution of the pattern-matching software how much data will be matched by a particular pattern. If the system reads less data than would be matched by the pattern, then the pattern-matching process must be interrupted in order to read additional data. If more data than is matched by a pattern is read into memory, then the excess data must be buffered for subsequent processing. Furthermore, many pattern-matching algorithms make use of lookahead and backtracking, both of which require that the program buffer data that has been read into memory without having been fully processed. It is inconvenient for programs to represent the single sequence of values available from an external sensor partially as an internal buffer of unprocessed values, and partially as an operating system device. Another undesirable consequence of reading more data than is matched by a pattern is that this causes unnatural delays in the processing of data. For example, if an attempt to match a single word from a stream of digitized voice input is preceded by an operating system call to read an entire minute of digitized speech, matching of that word may be delayed by nearly a full minute from the moment it was spoken.

These problems motivate a special data type to simplify the synchronization of pattern-matching software with events occurring in the external environment. This data type, called a stream, provides the following functionality:

- The stream data type represents in a single abstraction both data that is buffered in memory and values that have not yet been read from the operating system.
- Patterns are defined directly on the data represented by a stream, eliminating the need to precede pattern matching with explicit requests to read data from the operating system.
- The stream data type provides to pattern-matching procedures as much data as is required, as soon as that data becomes available from the operating system.
- The stream data type supports lookahead and backtracking in order to facilitate pattern matching.
- Fixed constants bound the time required to access values within a stream that have already arrived from the operating system.

2.3 Concurrent Processes

An important characteristic of real-time systems is that the flow of data in a real-time system is generally driven by the external environment, whereas in most other systems, data moves at whatever rate the CPU is able to process it. Conceptually, in a real-time system, the external environment is a concurrent process that produces values via sensors for consumption by pattern-matching software. In real-time systems, information is pushed from the environment into the computer as the information becomes available. With most other systems, information is pulled from the operating system into main memory when the CPU is ready to process it. In other words, real-time systems are generally sensor-bound, not CPU-bound.

As discussed in §1.1, pattern matching generally consists of multiple processing phases. The first phase of pattern matching, for example, is responsible for removal of electronic noise and general image enhancement, and may also convert information from one notation to another. The time required for information to flow from the external environment through the pattern-matching phases must be bounded by a small constant in order to satisfy real-time constraints. Each phase of pattern matching requires a small amount of time to execute. In order to satisfy real-time constraints, the time required for execution of each phase must be minimized.

The flow of data from external sensors to the output of the pattern-matching system can be compared to the flow of water down a river. Each phase of pattern matching is analogous to an obstruction in the river which causes a slight delay in the water's forward progress. Some pattern-matching operations process each object of the data stream in approximately the same small fixed amount of time. This type of processing might be compared to a water wheel powering an old-fashioned sawmill. Other pattern-matching operations collect up sequences of objects, process them as a group, and make all of the results of processing available to the next phase of pattern matching at essentially the same time. This is analogous to operation of a lock with gates that raise and lower boats between water levels. Still other pattern-matching algorithms maintain a window on the stream of data, passing data values to the next phase of pattern matching when they are no longer needed in the window representing the current focus of attention within the data stream. This type of algorithm is analogous to a dam placed in a river, with valves set so that the rate at which water is released from the dam is the same as the rate at which new water arrives from upstream.

Imagine that it were possible to stop the flow of water at the river's source. Then it would be possible to release a fixed quantity of water from the source and trace its flow to the ocean. The time required for the water to flow from the river's source to a water wheel and the delay imposed by the water wheel could be measured, as could the amount of time required to fill a lock to its threshold depth in order to cycle through the lock's operational states. As water arrives at a dam, the time required to fill the dam to its desired level and the amount of water released to the other side could also be measured. At both the lock and the dam, some of the water that was originally released from the river's source would be left behind, because the lock forwards only a fixed quantum of water at a time, and because the dam maintains a reservoir. These measurements are analogous to the analysis that must be done on a program in order to guarantee real-time response.

Suppose that a pattern-matching system processes segments of five data values at a time, that data values arrive at a constant rate of five per second, and that the time required to process each quantum of five values is one second. The worst-case response time for this system is 1.8 seconds (0.8 seconds from the arrival time of the first data value to arrival of the fifth, and one second to process the segment). Note that data values continue to arrive during the time that the preceding data segment is being processed. It is important that these new data values be buffered for subsequent processing. If the data values are produced by a hardware device with an output buffer that can hold only one value, then it is necessary to interrupt processing of the preceding segment each time a new value becomes available in order to explicitly buffer it. This requirement is easily satisfied by creating a high-priority process that reads directly from the hardware device, blocking whenever there are no data values to be read, and buffers each value until the pattern-matching software is ready to process the next quantum of data. In §2.2, the stream data type was introduced and motivated as a means of synchronizing pattern-matching software with the external environment. The stream data type can serve equally well to synchronize concurrent pattern-matching phases with each other. The low-level device driver process that reads directly from the hardware device simply writes values to an internal stream from which the pattern-matching process extracts values. Pipelined concurrent processes, therefore, provide a convenient notation for description of problems in which different phases of pattern matching are constrained to execute with different response times.

In general, the output of one pattern-matching phase may be sent in several conceptually different directions. This is useful when a single stream of information is analyzed for more than one kind of pattern. For example, atrial rhythm, ventricular rhythm, evidence of previous heart attack, and the effects of certain medications are all available from detailed analysis of electrocardiogram output. Since extraction of each of these characteristics from the stream of information is essentially an independent activity, simultaneous monitoring of a single data stream for different characteristics is greatly simplified by an ability to create separate processes to independently examine the common data stream. This same capability is useful in simulating non-deterministic parsing. Instead of choosing between alternatives and backtracking if a particular alternative fails, processes might be created to evaluate all of the non-deterministic choices concurrently. Parallel concurrent processes are, therefore, useful in describing solutions to problems where several different analyses are performed on the same data.

By providing programmers with language mechanisms to describe concurrency, the use of multi-processor architectures is greatly simplified. For certain problems, a multi-processor computer may be the only way to obtain the throughput needed to meet the demands imposed on the real-time system by the external environment. Although it is possible for compilers to automatically detect certain potential for concurrency in programs that do not explicitly specify it, the analysis is very difficult. Furthermore, as with traditional compiler optimizations for sequential programs, much potential for concurrency is disallowed for particular programs by the semantics of the language, even though the high-level problem specification may have permitted a large amount of concurrency. Providing language facilities for describing concurrency supports the idea that programmers, who have a better understanding of the problems to be solved than the compiler, must make most of the decisions regarding the order in which various components of a large software system are to be executed.

2.4 Timeouts and Interrupt Handling

Real-time pattern-matching systems typically process a steady stream of information supplied by the physical environment. Many real-time applications must also be able to deal with data that arrives from other than the standard information stream. For example, the pattern-matching software that might parse natural language in order to improve the quality of an automated telephone solicitor's synthesized speech might have to deal with an occasional irate customer. If, for example, the customer hangs up the phone, then the speech synthesis should be interrupted in order to dial another phone number.

The need to interrupt work in progress is fairly common in real-time applications. Cardiac monitor pattern matching, for example, may be interrupted by commands typed at a dedicated system's control panel. Scanning of radar screens for potential aircraft collisions might be interrupted by particular voice commands spoken by an air traffic controller. Likewise, a communication protocol is typically interrupted when the modem used for communication loses carrier. In some situations, interruptions require that the interrupted work be aborted. In other cases, the interrupted work must simply be suspended until some higher priority activity has completed. The activity that is triggered by an interrupt is typically represented by a high-priority process. Processes that execute in response to interrupts are called interrupt handlers.

Timeouts are internally generated interrupts that occur at times specified within the program. Timeouts are typically used within programs to detect failure in a distributed system of devices and computers. They may also be used to force compliance with real-time response constraints. A programmer may realize, for example, that a computer cannot process all possible data values in the limited amount of time allowed by real-time constraints. The programmer may choose in this case to ignore any data that cannot be processed within the imposed time limits. This can be accomplished by processing each data value with a timer running in the background. If the timer expires before processing completes, processing of that value is aborted.

Timeouts, interrupts, and interrupt handlers are important parts of almost all real-time systems. A high-level language designed to facilitate real-time programming must offer capabilities that provide the functionality of these programming paradigms. Interrupts are easily modeled by a stream of information that receives one value each time an event requiring interruption of control flow occurs. Interrupt handlers are simply high priority processes that spend most of their time waiting for interrupts to arrive. In response to an interrupt, the process carries out some programmer specified sequence of actions and then blocks itself, waiting for another interrupt. A timer is modeled by a concurrent process that sleeps for a period of time and then writes a data value to the stream representing the timeout interrupt. Activities that must be aborted in response to particular interrupts are modeled by processes that are killed by the interrupt handlers. High-level language mechanisms may be provided to simplify description of these sorts of process relationships.

It is important, in a real-time software system, that the time required to begin processing of an interrupt (the interrupt latency) be bounded by a constant. The implementation of a high-level language for real-time programming must be such that a reasonable upper bound on interrupt latency is available. Once processing of an interrupt begins, it is, in general, the application programmer's responsibility to ensure that the time required to process the interrupt is bounded (so that, for example, processing of an interrupt is known to complete before the next interrupt arrives). The programmer must also argue that the computer system is sufficiently fast to provide the throughput required by a particular application. The specification of high-level language features that are used by interrupt handling code must include reasonable and uncomplicated upper bounds on execution times in order to simplify the programmer's analysis of the interrupt handling code. Note that most high-level languages do not support these needs. For example, with many high-level programming languages, the cost of creating a new data object ranges from the small number of instructions required to allocate and initialize the new object from an existing pool of available memory to the thousands of instructions required to collect garbage if the free pool is empty.

The execution costs of each of the high-level language features proposed in this dissertation are described in symbolic terms[4]. Most of the language features execute in constant time. For those features that do not execute in constant time, the parameters that govern execution time are described so that programmers can analyze the processing requirements of the code they write. All operations that require more than a small constant amount of time to execute can be interrupted whenever higher priority activities need to take place. This is described in detail in Chapters 6 and 7.

[4] Since the current implementation of these language features is only a prototype, and since the actual times required to perform various functions depends heavily on the implementation and on the host computer, the real time required to execute these functions is not reported. A commercial implementation of these language features would have to measure the execution times of each language feature and report these measurements to the real-time programmers who are using the language.

Chapter 3: The Stream Data Type

The stream data type, discussed in §2.2, was motivated by a need to synchronize pattern-matching software with the external environment and with concurrent pattern-matching processes. In order to experiment with the utility of concurrent processes and the stream data type, a dialect of Icon called Conicon has been implemented. Icon was selected as a foundation for experimentation because it already provides many pattern-matching capabilities that have proven useful for processing string data and because the implementation of Icon is well understood and relatively easy to modify. As discussed above, Icon evaluates all expressions using a goal-directed evaluation mechanism. This greatly simplifies descriptions of rule-based patterns consisting of multiple rules, each of which might be satisfied in multiple ways. Throughout the remainder of this dissertation, it is assumed that the reader is familiar with Icon [25] and its goal-directed evaluation mechanisms.

A stream is an abstract representation for a potentially infinite sequence of values and a current focus of interest within that sequence. Values within the stream are referenced relative to the stream's current focus. A process that attempts to read from a stream values that are not yet available blocks until the values become available. Attempts to read beyond the end of a finite stream result in failure. Processes that attempt to write to a stream more than the stream is capable of buffering are blocked until additional buffer space becomes available. This is described in greater detail below.

3.1 Creation of Streams

In Conicon, streams replace files and stream scanning replaces string scanning. For example, &input, &output, and &errout are all streams. Scanning in Conicon has been modified to operate on streams instead of strings and &subject initially represents &input. Conicon has no &pos keyword. The open function in Conicon returns a stream either for reading or writing, depending on the mode supplied as its second argument. As in Icon, open's first argument is a string representing the name of a file. The second argument is a string representing the mode with which the stream should be opened. If no mode is specified, the default of "r" is supplied. The available modes for opening system files are:

r	open for reading
w	open for writing
a	open for writing in append mode

Streams are also created by coercion of other objects. This coercion occurs, for example, if a string or list is supplied as the argument to a scanning expression. Streams coerced from strings consist of a representation for the original string and a pointer into that string representing the current focus of scanning. Likewise, streams coerced from lists consist of a representation for the original list and an offset into the list representing the current scanning focus. Any data that can be coerced to a string can be coerced to a stream representing that string. Explicit conversion to streams is provided by the stream function. stream fails if it is unable to coerce its argument.

The work performed by Conicon in opening streams that represent interfaces to operating system files executes in constant time. Streams coerced from strings or lists are created in time proportional to the length of the string or list. The internal representation of streams and the algorithms that manipulate this representation are described in Chapter 6.

3.2 Writing to Streams

Writing to a stream opened for output behaves similarly to Icon. If the first argument to write is a stream, all subsequent arguments are written to that stream provided it is opened for writing. If the stream is not opened for writing, Conicon aborts with a run-time error message. If write's first argument is not a stream, then all arguments are written to &output. write returns the number of values written. Unlike Icon, no newline is automatically written to the stream after the last argument. Because of this, there is no writes function in Conicon. Also, Conicon's version of write does not allow users to change streams within the argument list by supplying another stream argument.

Each invocation of write represents an amount of computation that is proportional to the number of values to be written. Assuming that the output buffer is able to receive all of these values without blocking the writing process, each value is written to the stream in constant time. If a process must wait for buffer space to become available, it is put to sleep until the buffer is ready. While a process is sleeping, it does not require any computation to be performed by the host computer. Thus the CPU time required to write a value to a stream is always constant. The real time required to write a value to a stream depends, however, on availability of buffer space, which in turn depends on the behavior of concurrent activities. If it is necessary to bound the real time required to write values to a stream, then an analysis of the concurrent activities that consume values written to the stream must guarantee that as much buffer space as is required by the writing process is always available.

3.3 Reading from Streams

Reading from streams opened for input is accomplished by two functions: probe and advance. probe allows the user to view the contents of a stream without advancing the current position within that stream. Positions within a stream are numbered relative to the current position in the same way that positions are numbered within strings and lists. Negative and zero-based subscripts are specified relative to the end of a stream. This means that, immediately after converting a string or list to a stream, the positions within the stream are numbered exactly as they had been numbered in the string or list from which the stream was derived. The second argument to probe, which defaults to &subject, specifies the stream on which probe operates. For streams of characters, probe returns a string representing the data between the stream's current focus and the position named by its first argument. For example, the following expression assigns "fee" to the variable s:

```
"fee fi fo fum" ? (s := probe(4))
```

For streams of objects (as might be created from a list), probe returns a list representing the data between the current focus within the stream and the position specified by its first argument. The probe invocation below, for example, returns a two-element list containing the strings "while" and "(".

```
probe(3, ["while", "(", index, "+", "4", ">", "12", ")"])
```

With either type of stream, if a third argument is supplied to probe, it represents an initial offset to which the current focus for the stream is temporarily advanced before determining the absolute position named by probe's first argument. The following code assigns "fum" to s.

```
"fee fi fo fum" ? (s := probe(4.,-3))
```

probe fails if its finite stream argument does not have enough remaining items to produce the requested string or list.

advance expects the same arguments as probe and produces the same result, but advance has the side effect of advancing the current position for the stream to the position named by its first argument. Notice that Conicon has no read function. It is a simple matter to build read[5] out of advance and upto, which behaves similarly to the upto function of standard Icon:

```
procedure read(s)
  /s := &input
  return s ? 1(advance(upto('\n')), advance(2))
end
```

If advance is resumed, it restores the stream's focus to its previous value.

Invocations of probe and advance represent an amount of computation that is proportional to the distance of the last referenced stream value from the current scanning focus. This computation includes dynamic allocation of the buffers required to implement the function call. As discussed in Chapter 7, allocation of memory executes in time proportional to the size of the allocation. Backtracking through advance executes in constant time. If the data requested by probe or advance is not yet available, the process is put to sleep until the data values become available. Analysis of reading processes is similar to analysis of writing processes, which is described above.

3.4 Stream Programming Examples

Suppose that &subject represents a stream of tokens, each token stored as a string. The code fragment below might be used to parse the header of a C while statement:

```
if advance(2)[1] == "while" then {
  advance(2)[1] == "(" | stop("expecting left parenthesis")
  (parse_expr() & advance(2)[1] == ")") | stop("expecting right parenthesis")
  }
```

In this example, advance(2) advances the stream to position 2 relative to its current focus and returns a list of the data that is found between the stream's old and new points of focus. Note that the returned list is of length one. If the single entry in the list represents the while token, the body of the controlling if expression is executed with the stream focused on the next token. On the other hand, if advance fails, the body of the if expression is not executed. It is also possible for advance to succeed but for the comparison to fail. In that case, advance is automatically resumed, restoring the stream's focus to its previous position.

Following the while token, only a left parenthesis may appear. Assuming that tokens representing the while keyword and a left parenthesis are matched, an attempt is made to parse an expression. This goal-directed parser simply allows parse_expr to generate from shortest to longest each point at which a valid subexpression has been parsed. For example, in parsing the

[5] This read procedure differs slightly from the standard read function provided in Icon. Unlike the standard function, this read requires that a newline terminate the file.

expression:

```
index + 4 > 12
```

parse_expr treats index, index + 4, and index + 4 > 12 each as valid subexpressions. If this expression is supplied as the controlling expression of a while statement, parse_expr would suspend three times. Since neither of the first two suspensions is followed by a right parenthesis, parse_expr would be resumed in order to find alternative ways to satisfy its goal[6]. Most programming languages are designed such that parsing does not require all of the generality of a backtracking parser. However, certain real problems require more powerful techniques than are available from standard parsing algorithms. For example, the expressiveness of Conicon for the implementation of goal-directed parsers might be useful in experimenting with algorithms for real-time parsing of natural language.

In Icon, certain syntactic units are recognized as bounded expressions. After control leaves a bounded expression, backtracking into that expression is not possible. Since the control clause of an if expression is bounded, it is not possible to backtrack into that subexpression once the body of the if expression has been entered. Thus, in this example of a goal-directed parser, it is not possible for the stream's focus ever to return to the while token that was produced by advance. Conicon recognizes this situation and its garbage collector reclaims stream memory that is no longer accessible to the program.

As another example of programming with the stream data type, a procedure that counts the number of times each word occurs in an input file is provided below. This solution, which uses stream scanning, strongly resembles a string scanning solution to the same problem. In the traditional solution, a loop that extracts words from a line of input is nested within another loop that reads each line from the input file. The higher level view of data files afforded by the stream abstraction eliminates the need for nested iteration:

```
procedure main()

  wchar := &lcase ++ &ucase ++ '\'-_'
  words := table(0)

  while advance(upto(wchar)) do words[advance(many(wchar))] +:= 1

  wlist := sort(words, 1)
  every pair := !wlist do
    write(left(pair[1], 15), right(pair[2], 3), "\n")
end
```

Note that this program takes advantage of the initial scanning environment being automatically established with &subject set to &input.

3.5 Destruction of Streams

Though Conicon's garbage collector is capable of reclaiming much of the memory used to represent a stream that is no longer being accessed, the garbage collector does not communicate to the operating system that the file associated with the stream is no longer needed. The close function serves this purpose. After a stream has been closed, further attempts to write to the

[6] This example demonstrates some of Conicon's capabilities. It is not intended to suggest the most appropriate implementation of a parser for the C language.

stream are treated as fatal run-time errors. Any attempt to read beyond the end of the data available at the moment[7] the stream was closed results in failure. probe and advance requests succeed if they access only data that had already arrived from the operating system at the time the stream was closed. For example, after closing a stream with the following expression, a minimum of 80 additional characters can be read from the stream s.

```
advance(81, s) & close(s) & &fail  # read-ahead 80, close, then backtrack
```

Closing of a stream executes in time proportional to the number of processes that are blocked waiting to read from or write to the stream.

3.6 Real-Time Stream Applications

The ability to restrict backtracking through the use of Icon's bounded expressions is important when programming with the stream data type. This language feature limits the amount of history that must accompany each stream, allowing scanning of unbounded streams of data that might be sent, for example, between an earth station and a weather satellite, if matching expressions perform only a limited amount of backtracking. This characteristic is used in the following code to search for a C-style comment in a stream of text:

```
&input ? {
  while advance(2) ~== "/" | {advance(2); probe(2) ~== "*"}
  if advance(2) then {
    while advance(2) ~== "*" | {advance(2); probe(2) ~== "/"}
    if advance(2) then
      write("found comment\n")
    }
  }
```

The code shown above examines each character of the stream, looking for a slash followed by an asterisk. When the first comparison in the control clause of the first while expression fails, advance is resumed, backtracking the current stream focus. The first action performed in evaluating the right-hand subexpression of the alternation operator is to advance over the slash that has already been matched. If the following character is not an asterisk, the while expression continues to loop. Note that there are multiple sources of failure in the while expression's control clause. Looping halts when either the beginning of a comment has been recognized or when an attempt is made to read beyond the end of the stream. For this reason, the search for the end of the comment is preceded by a test to verify that at least one character remains in the stream to be processed. The character returned by this advance invocation, assuming the call succeeds, is the asterisk that introduces the comment. This could be rewritten to use scanning procedures:

```
&input ? {
  if advance(upto('/')) & advance(match("/*")) then
    if advance(upto('*')) & advance(match("*/")) then
      write("found comment\n")
  }
```

In contrast with the previous solution, this approach keeps more stream history in case

[7] Access by concurrent processes to the internal representation of a stream is essentially serialized by a mutual exclusion lock. This is described in Chapter 6.

backtracking is required. For example, while looking for the start of a comment, all string data from the the start of &input to its current focus is retained for possible backtracking. Although the second solution is much cleaner conceptually, it is less efficient in terms of memory usage if long segments of the input file contain no comments. One approach that might be taken to give the second solution the same efficiency as the first is to revise the definitions of the scanning functions.

This is the approach taken in the following example. Here, a procedure searches for valid data packets and returns each data packet that is found, treating any data that does not fit the definition of a data packet as noise and ignoring it. This activity, carried out by many communications protocols, is analogous to lexical analysis in a compiler. Each data packet is prefaced by a special start symbol, represented below by SOH. The data packet itself is comprised of header and data components, each of which is protected by a CRC checksum. The header has a fixed length of eight bytes, but the size of the data component is specified by the first three bytes of the header.

```
record packet(hdr, data)

procedure next_packet()
  local hdr, data

  return &input ? {
    while skipto(SOH) & chk_crc(hdr := advance(10)[2:0]) do
      if chk_crc(data := advance(hdr[1:4]+1)) then
        break packet(hdr, data)
    }
end
```

In the example above, chk_crc performs an internal consistency check on its string argument, succeeding only if the string's CRC checksum has a particular value. skipto is similar to upto, but it automatically advances the stream focus as it goes. An implementation of skipto is provided below:

```
procedure skipto(c, s)
  local char

  c := cset(c)
  /s := &subject

  repeat {
    char := probe(2, s) | fail        # fail if stream is exhausted
    if any(c, char) then
      suspend 1
    advance(2, s)
    }
end
```

next_packet is written in high-level Conicon, yet processes each byte in constant time and uses only a small amount of memory to represent the incoming stream of data. Constant-time performance is guaranteed by the fact that each block of code that is executed consumes at least one byte from the incoming stream. skipto consumes one character each time it executes its loop. Once skipto finds a character in c, it suspends, allowing the next subgoal to execute. The next subgoal attempts to verify that the eight bytes following the SOH character represent a packet header. If the CRC check, which is performed in time proportional to the length of its string argument, succeeds, the body of the while statement is entered. If the CRC check fails, control backtracks into the skipto procedure, returning the stream's focus to the SOH character. skipto advances past this character and searches for another SOH. Even this error condition is handled in constant time. The cost, in this case, of consuming the SOH character is the cost of executing one iteration of skipto's loop and failing to verify that the following eight bytes have a correct CRC checksum. This continues until a valid packet header is found. Upon entry into the while statement's body, the backtrack point left by advance when the argument to chk_crc was computed is automatically discarded. Inside the while statement's body, a second CRC check is performed. If this check fails, next_packet assumes that data was corrupted at some point following the most recently verified packet header. The while expression loops, looking for a new SOH starting with the character following the preceding packet header. In terms of the real-time cost analysis, the time spent failing to match the data component of a packet after successfully matching a header is charged to the nine characters that were consumed by the packet's preface and header. If, however, the CRC check of the data component succeeds, the complete data packet is returned to the calling environment. Since chk_crc executes in time proportional to the length of its string argument, this execution path also satisfies constant-time constraints.

To facilitate experimentation with modified versions of the string scanning procedures as demonstrated above with the skipto procedure, the functionality of Icon's built-in scanning functions is provided by a library of Conicon procedures. Since match is not a built-in function in Conicon, Icon's unary = operator is simply a macro that expands into:

```
advance(match(arg))
```

where *arg* represents the argument of =. An implementation of find is shown below:

```
procedure find(s1, s2)
  local str, len, i

  s1 := string(s1) | stop("bad argument to find")
  len := *s1

  /s2 := &subject

  every i := seq(1) do {
    str := probe(len+1, s2, i) | fail          # fail if the stream is exhausted
    (s1 == str) & suspend i
    }
end
```

Note that the parameters to find shown above are slightly different from those of Icon's function by the same name. This version of find takes only a string argument representing the string for which to search, and a stream in which to search for it. As implemented in Conicon, the parameters of other scanning procedures have been modified similarly.

3.7 Other Stream Applications

The stream abstraction is useful not only in real-time programming but also in many pattern-matching applications that do not require real-time response. For example, certain aspects of lexical analysis are somewhat awkward to implement in traditional Icon. This is because the string scanning environment that might be established inside of the procedure that looks for tokens is lost each time the procedure returns. But with stream scanning, this is not a problem. When the lexical analyzer advances its focus within its input stream, this change is stored as part of the stream's internal state. Subsequent attempts to view data from that same stream automatically start where the last advance left off. Below, for example, is code for a simple lexical analyzer that reads from standard input:

```
procedure get_token()
  static firstchars, idchar, digit

  initial {
    digit := '0123456789'
    idchar :=  &lcase ++ &ucase ++ '_' ++ digit
    firstchars := idchar ++ digit ++ '*/–+()'
    }

  return &input ? {
    skipto(firstchars) &              # skip nonsense characters
    if any(digit) then                # scan an integer
      advance(many(digit))
    else if any(idchar) then          # scan an identifier
      advance(many(idchar))
    else                              # scan an operator
      advance(2)
    }
end
```

Streams integrate naturally with Icon's goal-directed expression evaluation. For example, the scanning expression above could be rewritten as:

```
&input ? (skipto(firstchars), advance(many(digit | idchar) | 2))
```

It is even possible to simulate non-deterministic lexical analysis using the backtracking conventions of Icon. A lexical analyzer for C needs, for example, to recognize +, +=, and ++ as valid tokens. Traditional compilers usually match whichever alternative pattern consumes the longest substring of the input stream. However, Icon provides a concise notation that allows each of possibly many alternative matches to be recognized as tokens, providing alternative matches only if failures in subsequent parsing result in backtracking. The following code, for example, implements a simple non-deterministic lexical analyzer.

```
&input ? (skipto('+'), advance(match("+" | "+=" | "++")))
```

This capability is useful if, for example, the input stream represents syllables of speech which lexical analysis must assemble into words.

Chapter 4: Concurrent Processes

Concurrent processes were motivated by several special needs of real-time applications. One of these needs was to improve system throughput and response time for programs implemented on multi-processor computers. A second need was to simplify the creation of programs that are conceptually comprised of multiple independent activities. Finally, the need of real-time applications to deal with multiple sources of information, timeouts, and interrupts is also satisfied in part by an ability to describe concurrent processes.

4.1 Internal Streams

As discussed in Chapter 3, the stream data type provides synchronization and communication between the external environment and a Conicon program. Streams serve this same purpose between concurrent processes by providing message queues for the exchange of information. Streams connecting concurrent Conicon processes are called internal streams. They are created explicitly by calling open with a null value in place of a file name as its first argument. The following expression, for example, creates an internal stream of characters.

```
char_pipe := open()
```

Whenever the first argument to open is null, open's second argument specifies whether the internal stream represents arbitrary Icon values or only characters. Values are read from streams using the built-in functions probe and advance. These functions return strings when reading from streams of characters, and lists when reading from streams of arbitrary Icon values. The open modes are abbreviations for the types returned by probe and advance for each of these classes of streams respectively. To represent the idea that streams of arbitrary Icon values are referenced as lists or arrays, these streams are opened with an "a" mode. Streams of characters, which represent string data, are created using the "s" mode. The default mode is "s".

As with streams that communicate with the operating system, messages are inserted into a stream using write. The following expression, for example, inserts nine characters into the stream char_pipe:

```
write(char_pipe, "some text")
```

With streams of characters, write returns the number of characters written to the stream.

open, if called with a null value as its first argument and the string "a" as its second argument, opens an internal stream of arbitrary Icon values. Values of any type can be written to or read from the stream. Each argument of write other than the first is appended to the end of the stream. When writing to streams of values, write returns the number of values actually written.

```
write(object_pipe, [1,2], "string", object_pipe)
```

The expression above, for example, writes three values to the end of the stream referenced by object_pipe. As with the other structured types of Icon, a stream value is a pointer to an internal data object. Assignment of this value does not make a copy. In the example above, object_pipe is written to itself, creating a pointer cycle.

Values are extracted from a pipe of this form using the advance function which, as with streams coerced from lists, returns a list instead of a string. advance(1, object_pipe), for example, returns an empty list, leaving the current stream focus unchanged. The line below produces the next two items in the stream as a list, advancing the stream focus to position 3 relative to its previous focus:

```
    two_items := advance(3, object_pipe)
```

With either type of stream, an attempt to advance the focus beyond the end of the stream results in failure. This can be used, for example, to control looping:

```
while x := advance(2, object_pipe) do process(x[1])
```

This loop terminates when advance is called following closure of object_pipe. Attempts to advance an open stream's focus beyond the end of the data currently available from that stream simply blocks the process until additional data becomes available. Because the size of the internal buffer that holds data piped between processes is fixed[8], writing processes are blocked whenever they attempt to write to a stream more data than the stream is capable of buffering. These characteristics are used in the following example to implement semaphore operations[9]:

```
procedure P(sem)
  advance(2, sem)
end

procedure V(sem)
  write(sem, &null)
end
```

Whenever processes must wait for stream data or buffer space, they are placed on internal process queues associated with the events for which they are waiting. All internal process queues are ordered first by process priority, and then in FIFO order. Process queues are described more fully in Chapter 6. This organization of process queues allows a higher priority process that arrives on the queue later than some other process to advance more quickly to the front.

Conicon's built-in functions for accessing the data of a stream are atomic with respect to other operations accessing the same stream. This means, for example, that the two values written to the stream mailer by the following write invocation appear contiguously when read from the stream, even though other processes may be writing to the stream at the same time this code is executing.

```
write(mailer, "robert", msg)
```

The same is true for advance and probe invocations. Once a process has gained mutually exclusive read or write locks to a stream, that process retains the lock until it is done accessing the stream, even if the process must wait for new data values or for buffer space to become available.

Occasionally, processes that need to read from or write to a stream must be able to do so without becoming blocked on internal queues waiting to perform the requested I/O. Special non-blocking versions of the standard stream accessing routines provide these capabilities[10].

[8] The default buffer size for streams of either characters alone or of arbitrary Icon values is 256. This value can be modified using the bound function described below.

[9] This implementation of a semaphore is only valid as long as the number of V operations does not exceed the number of P operations by more than the size of the stream's internal buffer.

[10] Invocation of conditional stream manipulation routines may block the current process on queues associated with read or write locks for the specified stream but never on queues associated with the arrival of new data values or availability of additional buffer space.

cadvance, cprobe, and cwrite are conditional versions of advance, probe, and write respectively. cwrite returns the number of values written without blocking. cadvance and cprobe return as much of the requested stream as is available.

The close function, described in §3 as it relates to streams representing operating system files, behaves similarly for internal streams. Attempts to write to a closed internal stream result in a fatal run-time error. Reading from a closeↄ stream succeeds only if the requested data was available from the stream at the time it was closed. If, at the moment a stream is closed, processes are blocked waiting to write to the stream, close terminates with a run-time error.

4.2 Process Creation

Icon's create operator, which creates co-expressions out of its expression argument, has been modified in Conicon to instead create a concurrent process out of its expression argument. The create operator executes in constant time. In the following program, processes are created to intertwine two sequences of integers, the first sequence increasing from 1, and the second decreasing from −1.

```
procedure main()

  # write positive integers
  p1 := create every write (&output, seq(1), "\n")

  # write negative integers
  p2 := create every write (&output, -seq(1), "\n")

  every deathwatch(p1 | p2)
end
```

The deathwatch function in the above program waits for its process argument to die. In this example, deathwatch never returns because p1 and p2 never terminate.

Several of Icon's keywords have been redefined in Conicon. For example, &main and &current refer to the main and current processes respectively instead of referring to co-expressions, and there is no &source keyword. The meaning of &subject has also been changed. Each process has its own private &subject, which is set, at process creation time, to the value of &subject in the creating process. &subject for the main process is set initially to &input.

Concurrent processes often serve as filters. A filter might be used, for example, to replace in a stream of characters each pair of consecutive as with the single letter b. A second filter, reading the output of the first, might replace every pair of consecutive bs with a single c. A solution to this problem that uses co-expressions to implement each filter is described in §13.4.2 of [25]. The solution presented here uses instead an internal stream of characters to connect the two filters, and creates processes to act as each of the filters.

```
pipe := open()
create compact("a", "b", &input, pipe)
create compact("b", "c", pipe, &output)
```

The first filter reads from &input and writes to the internal stream pipe. The second filter reads from pipe and writes to &output.

```
procedure compact(s1, s2, in, out)
  local c1

  c1 := cset(s1)
  s1 ||:= s1
  in ? {
    while write(out, advance(upto(c1))) do {
      if advance(match(s1)) then write(out, s2)
      else write(out, advance(2))
      }
    write(out, advance(0))
    }
end
```

In the above application, the two filters execute independently of each other. If pipe empties, the second filter is automatically blocked until new characters become available. If the pipe reaches its full capacity, the first filter is blocked until characters are extracted from the pipe, making room for more characters to be written.

4.3 The Yield of a Process

Many string processing problems can be divided naturally into multiple filters connected to each other by internal streams. For example, a lexical analyzer might be connected to a parser by a stream of tokens. Here, the lexical analyzer executes as a process that reads from &input and writes tokens to an internal pipe. For this application, the stream that connects the two processes represents arbitrary Icon values. It might be opened using the following expression:

```
tokens := open( ,"a")
```

Suppose a procedure named get_token has been implemented by the programmer. On each invocation, this procedure simply reads from &input to the end of the next available token and returns the token as a string. Given the existence of get_token, the following expression starts up a process to perform lexical analysis:

```
create while write(tokens, get_token())
```

A backtracking parser of the tokens stream is described briefly in §3. Because the expression argument of the create operator is evaluated in a goal-directed fashion, the process is capable of producing a sequence of results. At the time of its creation, a special output stream is allocated for each process[11]. This stream automatically receives the results produced by the expression. The following line, for example, creates a process that writes to its output stream each token returned by the lexical analyzer.

```
p := create |get_token()
```

The function yield produces the output stream of a process. For example, yield(p) represents the output stream of process p. The expression shown below reads the next available token from the lexical analyzer.

[11] Every process has a standard output stream created in this way. Processes can have more than one output stream by explicitly creating them using open and explicitly writing to them using write.

```
tok := advance(2, yield(p))[1]
```

Created processes are automatically killed when the expressions they are evaluating fail to produce further results. When the process is killed, its output stream is automatically closed. Closing of the output stream does not discard buffered values. It simply prohibits the writing of additional data to the stream[12], and causes any attempt to read data that is not available from the stream to be treated as an end-of-stream condition instead of blocking the reading process.

These capabilities provide the ability to mimic the behavior of coexpressions. For example, the Icon expression:

```
every write(find(s1, s2))
```

can be rewritten as:

```
p := create find(s1, s2)
while write(advance(2, yield(p))[1])
```

Because it is common to ask for only one result at a time from a process created in this manner, Icon's co-expression activation operator has been redefined to provide similar functionality in Conicon. The meaning of

```
@p
```

is simply:

```
advance(2, yield(p))[1]
```

Thus, yet another notation for the every expression above is:

```
p := create find(s1, s2)
while write(@p)
```

Note that executing an expression as a separate process is not necessarily equivalent to execution within an every expression. For example, the code below writes the time at which each invocation of compute begins. This might be used to measure the performance of compute.

```
every write(|&time) do
  compute()
```

According to the pattern established above, this could be rewritten as:

```
p := create |&time
while write(@p) do compute()
```

However, the output of this program is somewhat deceptive. Because a stream of buffered results connects the two processes, it is possible that a considerable delay might occur between invocation of &time and execution of the @ operator that produces the time for printing.

Values are suspended from processes by writing to the process's yield. This executes in constant time provided that the stream's buffer is not full, as discussed in §3.3.

[12] Typically, the killed process is the only process that would write to this stream. However, it is possible for other processes to explicitly write to the stream using, for example, write(yield(p), x).

4.4 Concurrent Alternation

Icon's alternation operator generates all results of its left-hand expression followed by all results of its right-hand expression. Conicon's concurrent alternation operator generates all results of both its left- and right-hand expressions, but both expressions are evaluated concurrently, and results are produced in whatever order the expression arguments produce them. Concurrent alternation is represented by the binary ! operator, which has the same operator precedence as Icon's more traditional alternation operator.

In standard Icon,

```
every write(1 | 2 | 3)
```

outputs the three lines:

```
1
2
3
```

Use of the concurrent alternation operator in place of Icon's standard alternation operator yields the same three lines, but in undefined order. For example, the output of:

```
every write(1 ! 2 ! 3)
```

might be:

```
3
1
2
```

The concurrent alternation operator creates a separate Conicon process to evaluate each of its expression arguments. The processes created by this operator share a common output stream. Each expression is evaluated in a goal-directed fashion, producing all possible values and writing them to the shared output stream. The following expression, for example, writes all of the integers from 1 to 20, but the order of the written numbers depends on how the two processes are scheduled with respect to each other:

```
every write((1 to 10) ! (11 to 20), "\n")
```

As processes exhaust the result sequences of the expressions they are evaluating, the processes kill themselves, decrementing a reference count on their shared output stream. When a process decrements this reference count to zero, it closes the stream. Remember that closing the stream only prevents other processes from writing additional data to the stream. It does not prevent other processes from reading the stream data values that have already been buffered. Concurrent alternation therefore provides a concise notation for the creation and destruction of short-lived processes and communication with those processes while they are active.

In many contexts, only one result from an expression is desired. In standard Icon, if no more results are required from an expression capable of producing multiple results, evaluation of the expression simply aborts. For example, the following expression obtains only one result from the upto generator:

```
write(upto('aeiou', "a fool and his money are soon parted"))
```

However, the line below writes each position at which a vowel appears in upto's string argument.

```
every write(upto('aeiou', "a fool and his money are soon parted"))
```

The difference between these two examples is that the first invocation of upto appears within a bounded expression that requires only one result. In Icon, expression boundaries, which are determined at translation time, serve to limit the interaction between the goal-directed evaluation of separate subexpressions. Once control crosses an expression boundary, it is not possible for subsequent failure to backtrack into the bounded expression [28]. This same distinction is made for concurrent alternation. Whenever a bounded expression terminates, any processes created by a concurrent alternation operator within that bounded expression are automatically killed. For example, only one result is required from the expression below:

```
s := prompt_user() ! sleep(10000)
```

In this example, prompt_user displays some prompt to an interactive user and waits for a response. If this response comes within ten seconds (10000 milliseconds), it is returned as a string by prompt_user and assigned to s. Since only one result is required from this expression, sleep is automatically killed as control leaves this bounded expression. However, if sleep produces its result before prompt_user, then prompt_user is killed. This idiom provides a clear concise notation for exception handling. A similar construct might be used, for example, to monitor a modem's carrier signal while a concurrent process communicates via the modem.

```
if (watch_carrier() ! communicate()) \ 1 = -1 then
  write("lost carrier\n")
```

The example above expects watch_carrier to sit quietly until it detects that carrier has been lost, at which time it returns -1. communicate, in this example, must return some numeric value other than -1.

This paradigm can be carried even further. Because of the high computational complexity of many real-time problems, it is not possible to obtain solutions to these problems that work for all possible combinations of input data in a constant amount of time. However, it is possible to create solutions that work in constant time for some subset of the possible domain. Consider, for example, natural language parsing of voice phonemes. A procedure might be written to scan a stream of these phonemes, returning a parsed English sentence. If this procedure is able to find a sentence within, for example, 100 milliseconds, the real-time constraints imposed on the algorithm are satisfied. However, if no sentence can be found in that amount of time, the algorithm may be forced to simply ignore some initial sequence of the phonemes remaining to be parsed. This is modeled as:

```
if ((s := get_sentence()) ! sleep(100)) \ 1 === &null then
  skip_to_pause()
```

In the above expression, skip_to_pause advances the focus within the stream of voice phonemes to the next period of silence exceeding some minimum duration. Before giving up entirely on parsing the phonemes, however, a quick check of the system status may reveal that computing resources are available for additional parsing. If, for example, after 100 milliseconds of unsuccessful parsing, few if any additional phonemes have arrived for processing, the decision might be made to continue parsing the phonemes that were previously available. This is achieved with the following:

```
every s := (get_sentence() ! |sleep(100)) do
  if \s | heavy_load() then
    break
```

This loop terminates only if get_sentence parses a complete sentence or if the system is perceived to be heavily loaded at one of the timeout points. Note that a result becomes available from the concurrent alternation operator at least once every 100 milliseconds.

Similar constructs might be used in voice synthesis. In order to give proper duration, intonation, volume, and accents to the synthesized reading of English text, some natural language parsing is required. However, if some sentence is particularly difficult to parse, or some word is not found in the system's dictionary, then the program might attempt a simpler, less capable algorithm. Suppose the simple algorithm is known to execute in constant time. The following code might be used to obtain a list of parameterized phonemes representing a single English sentence:

```
phonemes := compile_sentence(s) ! (sleep(75), simple_compile(s))
```

In this code, the system vocalizes the list of phonemes produced by whichever parser terminates first. Because the output from compile_sentence is preferred over the output from simple_compile, it is given a 75 millisecond head start. Because simple_compile is known to execute in constant time, no additional timing restrictions need be imposed on this expression.

Initialization of a concurrent alternation operator consists of creating two concurrent processes. As mentioned above, process creation executes in constant time. When control leaves a bounded expression within which concurrent alternation processes are still active, those processes must be killed and their shared yield must be closed. In general, closing a stream executes in time proportional to the number of processes that are waiting to read from or write to the stream. For the special-purpose streams used to implement concurrent alternation, there are typically no more than two processes blocked at a time[13]. Since the number of blocked processes is bounded, closing this shared stream executes in constant time. The time required to kill the two child processes is in general not so easily bounded. As discussed below, the run-time costs of killing a process are proportional to the sum of the number of processes waiting for the given process to die and the number of descendent processes. Because of the restricted way in which concurrent alternation processes interact with one another, there are generally no processes waiting for a particular coalternation process to die. The number of descendent processes, however, is potentially unlimited. In order to bound the time required to terminate a concurrent alternation expression, a programmer must determine an upper bound on the number of descendent processes spawned by nested concurrent alternation expressions.

4.5 Process Manipulation

Several new functions are provided for accessing and manipulating information about processes and internal streams. bound, for example, allows programmers to set an upper limit on the number of messages that may be stored in a stream. bound takes as arguments a stream and an integer limit. Following execution of bound, a process that attempts to place more than the specified number of values into the internal buffer is blocked. Once a message has been viewed by a recipient process, that message no longer occupies a slot in the bounded buffer representing the stream. If backtracking occurs, the message is restored to a special buffer

[13] If the parent process is blocked attempting to read from the stream, then writers cannot be blocked. If both writers are blocked, the parent cannot be blocked.

whose size is not affected by bound. If the buffer, at the time that bound is called, holds more than the specified number of values, all of these values are retained in the buffer until they are extracted by a reading process. If the limit is set to zero, future message passing requires a rendezvous between reading and writing processes. bound executes in constant time.

The kill function destroys its process argument, blocking the current process until the destruction is complete. Any processes spawned by the specified process in order to evaluate a concurrent alternation expression are likewise killed. Processes created by the killed process using the create operator are not killed. If a killed process owns mutually exclusive read or write locks to a stream at the time it is killed, these locks are released when the process is killed. After a process is killed, all other processes that are waiting for this process to die (because they executed kill or deathwatch specifying this process as an argument) are unblocked. In all, the time required to kill a process is proportional to the sum of the number of descendent processes and to the number of processes that are waiting for this process to die.

Process priorities range from 0, being the highest priority, to 15. When processes are created, they inherit the priority of the process that creates them. &main initially has priority 0. The priority function allows a program to change the priority of a process. priority takes two arguments: an integer priority and a process. The default process is &current. priority executes in constant time.

The sleep function blocks the current process for the number of milliseconds specified by its single integer argument and returns the null value. sleep executes in two phases. When sleep is first invoked, an amount of work proportional to the *log* of the number of other processes that are sleeping is performed. When this process must be awakened, an amount of work proportional to the *log* of the number of processes that are sleeping at that time is performed. When a process is put to sleep, the work required to subsequently awaken the process is scheduled to take place at the appointed time. This work will be performed even if the process is killed before its time to awake arrives[14]. Additionally, the implementation of sleep depends on a high-priority daemon process that receives interrupts at regular intervals from the system's hardware timer. This process spends most of its time blocked waiting for interrupts. In analyzing system throughput, it is necessary to subtract a fixed amount of CPU time from each time quantum to represent the time spent executing this sleep daemon.

The deathwatch function blocks the current process, delaying until the process specified as deathwatch's single argument dies. If the specified process is already dead, deathwatch returns immediately. deathwatch represents a constant amount of computation.

Each process has associated with it a special output stream that accumulates the results produced by the process. yield, which expects a single process argument, returns the output stream of that process. yield executes in constant time.

4.6 Concurrent Programming Examples

In addition to the examples discussed above, two classical concurrency problems are presented and solved here. These programs demonstrate, among other things, that even though Conicon offers only a small number of simple concurrent processing mechanisms, the language

[14] Analyzing the performance of sleep is somewhat difficult because of the global nature of its dependencies. This analysis might be simplified by more formal methods to describe response time and throughput requirements and good software tools to assist with the analysis.

is expressive enough to allow elegant solutions to traditional concurrent programming problems[15].

One famous concurrency problem, selected because it represents a variety of real scenarios in which multiple processes share access to a limited number of resources, is the "Dining Philosophers Problem" [29]. In this problem, *n* philosophers sit at a round table with a bowl of rice in the center of the table. To eat rice, each philosopher needs two chopsticks. However, there are only a total of *n* chopsticks on the table, so not all of the philosophers can eat at the same time. Each chopstick is placed on the table between a pair of philosophers, and is shared only between the two adjacent philosophers.

A solution to this problem must implement some protocol that allows each philosopher to repeatedly eat, then think. While certain philosophers are thinking, other philosophers are given the opportunity to eat. The solution presented here ensures that every philosopher is given equal opportunity to eat and prevents deadlock from occurring. This solution uses a stream to represent each chopstick. Below is the main procedure:

```
procedure main()
  local i, n, chopsticks

  write("how many philosophers? ")
  n := read()

  chopsticks := []
  every 1 to n do {
    put(chopsticks, open(,"a"))
    write(chopsticks[-1], &null)
    }

  every i := 1 to n - 1 do
    create philosopher(i, chopsticks[i], chopsticks[i+1])
  deathwatch(create philosopher(n, chopsticks[1], chopsticks[n]))
end
```

The main procedure simply creates one stream for each of the *n* chopsticks, and sends to each of those streams a single value. Then main creates the *n* philosopher processes. The philosopher procedure takes as parameters its own identification number, and the streams representing each of the two chopsticks that it must eat with. philosopher's second argument represents the first chopstick that the philosopher must pick up. It is important to specify for each philosopher the order in which it should pick up its two chopsticks. By requiring that the *n*th process pick its chopsticks up in reversed order from the other processes, the possibility of deadlock is avoided.

```
procedure philosopher(id, firstchopstick, secondchopstick)

  repeat {
    getchopstick(firstchopstick)
    getchopstick(secondchopstick)
```

[15] It can also be demonstrated that streams are capable of simulating semaphores and monitors. Note that, in terms of classical interprocess communication methods, the stream data type is classified as a message passing system. The examples above are intended to demonstrate not only that Conicon is capable of describing solutions to these problems, but that it does so elegantly.

```
        write("philosopher ", id, " is eating\n")

        putchopstick(firstchopstick)
        putchopstick(secondchopstick)

        write("philosopher ", id, " is thinking\n")
        }
    end
```

putchopstick and getchopstick are simple stream accessing procedures:

```
    procedure getchopstick(queue)
      advance(2, queue)
    end

    procedure putchopstick(queue)
      write(queue, &null)
    end
```

The dining philosophers problem represents situations where multiple processes compete for limited resources. In another large class of concurrent problems, a single server process satisfies the needs of many client processes. These problems are represented by the ``Sleeping Barber Analogy'' [30]. In this analogy, a small barbershop with a single barber and a single barber's chair serves customers as they arrive. Within the barbershop, a special waiting room is provided where customers generally take short naps, waiting until the barber is ready to cut their hair. If the barber finds his waiting room empty after finishing a hair cut, he likewise sleeps in the waiting room, awaiting arrival of a new customer. A program that models this situation is described below. In this program, the barber and each customer are represented by individual processes.

The barber repeatedly waits for a customer, cuts the customer's hair, opens the exit door for the customer, and waits for the customer to leave. This is implemented by the following procedure:

```
    procedure barber()

      repeat {
        # get a customer
        signal(barber_ready)
        wait_til(chair_occupied)

        cut_hair()

        # show the customer out the door
        signal(door_opened)
        wait_til(customer_gone)
        }
    end
```

Meanwhile, each client to the barber executes the following procedure:

```
procedure client()

  wait_til(barber_ready)
  signal(chair_occupied)

  # sit quietly while barber cuts hair

  wait_til(door_opened)
  signal(customer_gone)
end
```

In this program, wait_til blocks the current process until the condition supplied as its argument is signaled by some other process. Processes blocked on these conditions are queued in first-come, first-served order. Each time a condition is signaled, only the first process on the queue is wakened. Using streams to represent each of the special conditions, signal and wait_til are implemented as shown below:

```
procedure signal(s)
  write(s, " ")
end

procedure wait_til(s)
  advance(2, s)
end
```

The main procedure shown below initializes shared variables and creates processes to represent the barber and each of the clients.

```
global barber_ready, chair_occupied, door_opened, customer_gone

procedure main()
  local n, clients

  # create condition queues
  barber_ready := open()
  chair_occupied := open()
  door_opened := open()
  customer_gone := open()

  create barber()

  write("how many clients?")
  n := read()

  # create client processes
  clients := []
  every 1 to n do
    put(clients, create client())

  # wait for client processes to die
  while wait(get(clients))
end
```

Chapter 5: Sample Applications

This chapter describes two real-time applications and how their implementations are simplified by the high-level pattern-matching capabilities of Conicon. The first application is an automatic login scripting program that provides functionality similar to the L.sys facility used by UNIX uucp [31]. The other application analyzes electrocardiogram output in order to label the phases of each heartbeat.

The intent of this chapter is to demonstrate the expressiveness of the proposed high-level language features in dealing with concerns that are typical of real-time programming problems. The benefits of this expressiveness are several fold. First, implementation of applications is simpler. Second, maintenance and enhancement of existing applications is easier. Third, because of the ease of implementation and maintenance of Conicon programs, application designers and implementors are able to focus more on the users' requirements and less on implementation concerns. Ideally, this results in development of better software at a reduced cost.

5.1 Automatic Login Package

Many long-haul network connections between UNIX computers use the same serial-line connections that are available to human users of the same computers. By default, these serial lines are configured for humans and prompt for interaction with these users. In order to establish a connection between two UNIX computers, one of the computers must dial into the other using telephone modems and emulate a human user long enough to initiate the more formal handshaking provided by a communication protocol. For most UNIX systems, emulation of a human user consists of interacting with a port contention unit to select a host computer, dealing with the automatic baud selection of the host computer, and responding appropriately to login and password prompts from the remote computer.

A standard scripting approach has been designed for describing these interactive sessions. In general, an interactive session consists of several independent actions. Each action must complete successfully in order for the next action to be attempted. Actions generally consist of sending a string of characters and then waiting for a particular response from the remote system. Whenever an expected response does not arrive within a predefined amount of time (generally 15 seconds), the search for that string is aborted. Within the specification of an action, it is possible to describe alternative subactions to be carried out when timeouts occur.

The session script is simply a string of text. Independent actions are separated within the script by spaces. Each action consists of a sequence of strings separated by hyphens. The odd-numbered strings represent strings to be sent to the remote system. By default, a newline character is appended to the end of each sent string. Even-numbered strings represent patterns for expected responses. Interpretation of an action's specification proceeds from left to right until an expected response is matched. For example, the specification shown below sends a single newline and expects a login: prompt. If the prompt is not received within the default timeout period, another newline is sent and the timer is reset to search once again for the login: prompt. This action is repeated a maximum of three times.

```
    "-login:--login:--login:"
```

The action described above would typically be combined with other actions to comprise the entire login sequence. For example, a complete login session might be represented by the following script:

"-login:--login:--login: kelvin-password: mypwd-TERM~s=~s(ansi) vt100-%~s"

The intention of this script is to wait for a login: prompt, type the user name kelvin and wait for a password: prompt, type the password mypwd and wait for the TERM = (ansi) prompt, respond with vt100 and wait for the shell prompt %. Timeouts are used when searching for the login: prompt in order to coordinate with the automatic baud selection of the remote system. Escape sequences within session scripts are used to represent special characters. The tilde escape character introduces each sequence. The following sequences are understood:

~~	the escape character (~)
~–	a dash within a send or expect string
~b	a backspace
~n	a newline (linefeed)
~r	a carriage return
~s	a space
~o[o[o]]	any ASCII character with the specified octal representation (note that 1, 2, or 3 octal digits may follow ~)

Also, within strings to be sent, special meaning is given to the following sequences:

~B	transmit a BREAK character
~d	delay 2 seconds
~c	do not append a newline to the end of the string

The do_script procedure, shown below, divides a script into individual actions by searching for white space and invokes do_action with each action's string representation as an argument. The script string is passed to do_script as its first argument. do_script's istream and ostream arguments represent input and output streams respectively. istream is assumed to be in RAW mode[16]. This allows scanning to match received text even before a newline has been received. do_script fails if do_action fails. do_script succeeds after do_action succeeds for each of the actions specified in its script.

```
procedure do_script(script, istream, ostream)
  static not_space
  local command

  initial not_space := ~' \t'

  return not (script ?
    while command := advance(many(not_space)) do {
      if not do_action(command, istream, ostream) then break
      advance(many(' \t'))
      })
end
```

[16] The ioctl function provides the capability of selecting between, among other things, RAW and COOKED modes and allows selection of baud rates and other communication parameters.

The procedure do_action repeatedly transmits a string and attempts to match an expected response. do_action returns successfully if an expected string is matched. do_action fails if none of the expected strings is found. The procedure do_send outputs a string or fails if there are no more strings to be sent. do_expect searches for a specified pattern, failing if the pattern cannot be found within the default timeout period. The implementation of do_action is shown below:

```
procedure do_action(cmd, istream, ostream)
  return ("-" || cmd) ? (|do_send(ostream) & do_expect(istream))
end
```

Note that do_send and do_expect execute within the scanning environment established by do_action. This environment scans the string specification of the action. Within this string specification, strings are separated by hyphens. Since do_send expects a hyphen to introduce the string to be sent, do_action prepends an initial hyphen to the command string before creating this scanning environment. If the next character in the command is not a hyphen, do_send fails, indicating that the list of backup subactions has been exhausted. This is shown below:

```
procedure do_send(ostream)
  static not_dash
  local newline, c

  initial not_dash := ~'-'

  if advance(2) == "-" then {
    newline := "\n"
    repeat {
      write(ostream, advance(many(not_dash)))
      if advance(2) == "~" then
        write(ostream,
              case c := advance(2) of {
                "-": "-"
                "~": "~"
                "b": "\b"
                "s": " "
                "n": "\n"
                "r": "\r"
                "B": (ioctl(ostream, "BREAK"), &null)
                "d": sleep(2000)
                "c": newline := &null
                !"01234567": do_digits(c)
                })
      else {
        write(ostream, \newline)
        return
        }
      }
    }
end
```

The invocation of ioctl shown in the code above sends a break to the serial line represented by its stream argument.

do_expect first calls get_pattern, which advances the scanning focus to the end of the expected string, and then starts up processes to concurrently sleep for 15 seconds while searching for the specified pattern. do_expect is implemented below:

```
procedure do_expect(istream)
  local pattern

  pattern := get_pattern()
  return \((sleep(15000) ! try_match(pattern, istream)) \ 1)
end
```

If the sleep process terminates before the pattern is found, do_expect fails because sleep returns the null value. If, however, try_match completes first, it returns a non-null value. Note that care has been taken to ensure that do_expect advances the scanning environment's focus to the end of the expected string before failing. If the focus were advanced instead within try_match, then it would have been possible for the timer interrupt to arrive before the focus had been shifted[17].

get_pattern checks for a hyphen to introduce the expected string and searches for the next unescaped hyphen or for the end of the command string, whichever comes first, translating all of the intervening escape sequences accordingly. If the first character in the scanning environment is not a hyphen, get_pattern simply returns an empty string, indicating that there is no pattern to be matched. The implementation of get_pattern is provided below:

```
procedure get_pattern()
  static not_dash
  local result, c

  initial not_dash := ~'-'

  result := ""
  if advance(2) == "-" then {
    repeat {
      result ||:= advance(many(not_dash))
```

[17] It is highly unlikely that scanning of the expected string requires more than 15 seconds. However, it is good defensive programming style to prevent all race conditions, no matter how unlikely they might be.

```
          if advance(2) == "~" then
            result ||:=
              case c := advance(2) of {
                "-": "-"
                "~": "~"
                "b": "\b"
                "n": "\n"
                "r": "\r"
                "s": " "
                !"01234567": do_digits(c)
                }
          else break
          }
        }
    return result
  end
```

try_match searches for the specified pattern and advances the stream focus to the end of the matched text. Its implementation is shown below:

```
  procedure try_match(pattern, istream)
    return advance(find(pattern, istream) + *pattern, istream)
  end
```

In the introduction to this chapter, the claim was made that the high-level pattern-matching facilities of Conicon simplify solutions to many real-time problems. This application is one example for which comparison with an implementation in another language is fairly easy. A C implementation of this same application is provided as part of the standard distribution of the kermit file transfer protocol for UNIX[18]. After stripping comments and debugging code from both versions, the C implementation is over twice as big as the Conicon implementation (measured in terms of number of lines of code, number of individual words, or number of characters in the respective source files). Though it is hard to evaluate the ease of code readability or maintenance, it is worth noting that the C implementation uses four goto statements, two invocations of setjmp, and one invocation of longjmp. The C version implements timeouts using the alarm and signal system calls. Because of the unstructured nature of programming with timeouts and other signals in C, it is difficult to localize logical concerns to contiguous sections of C code. Also, presumably because of the difficulty of describing arbitrarily long strings in C, the C implementation imposes the somewhat arbitrary restriction that only the last seven characters of each pattern string are actually matched against the input stream.

The scripting facility presented here is rather limited in its capabilities. Note that there is no way to change the default timeout value or to condition future actions based on the prompts provided by the remote system. In writing a script to interact with an intelligent modem, for example, it may be desirable to execute one sequence of actions if the modem's response to a dial command indicates that the line is busy and another sequence of actions if the response indicates that the remote system simply did not answer its ringing telephone line. In other situations, it might be desirable to describe patterns that are more flexible. For example, regular expressions

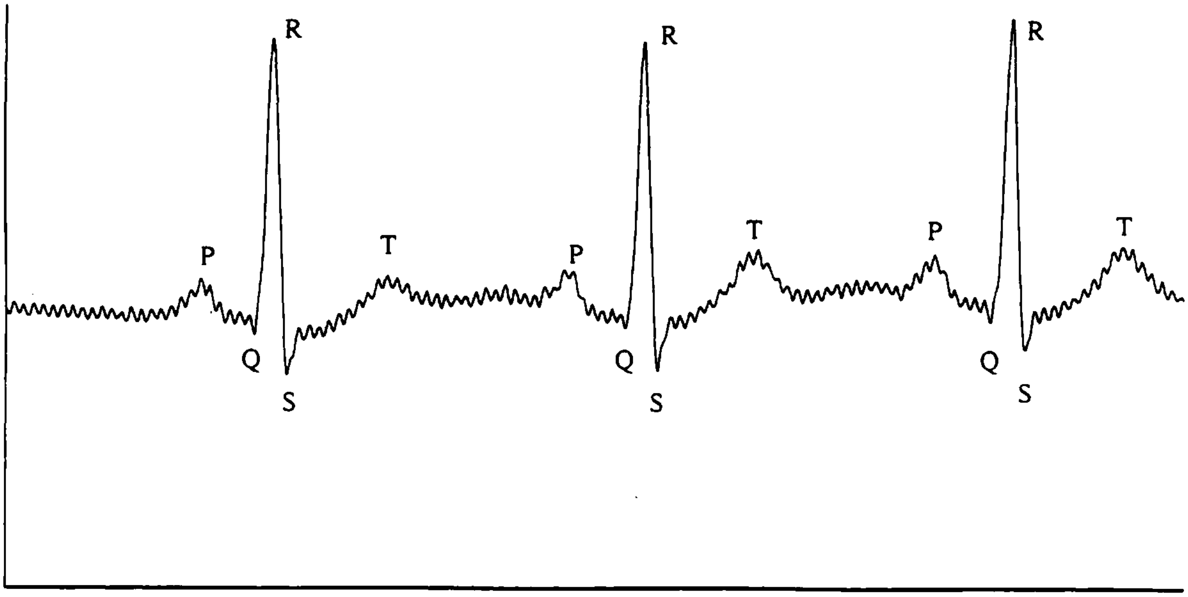
[18] There are some small differences in the syntax used for script specification in the two applications. For example, the C implementation uses the keyword BREAK instead of ~B to represent a break character.

might replace the simple string patterns. Most of these suggested enhancements require changes to the grammar for script specifications. Experimentation with these enhancements likely consists of experimenting both with the scripting grammar and with the implementation of the semantics represented by the grammar. Thanks to the high-level string data type, automatic garbage collection of parse trees, and goal-directed expression evaluation, implementation of the enhanced parser is much easier in Conicon than in C. And because of the more structured nature of dealing with timeouts and concurrent processes, experimentation with the implementation of the semantics is also easier in Conicon than in C.

5.2 Electrocardiogram Analysis

Electrocardiograms measure voltage differences between electrodes placed on a person's skin in order to determine the polarization of various chambers of the person's heart [32-34]. By observing the changes in voltage readings as a function of time, it is possible to determine not only when each heartbeat occurs, but also when each of five major phases of each heart beat occurs. These phases are labeled P, Q, R, S, and T. During the P phase, the heart's auricles depolarize. Together, the Q, R, and S phases represent the period of time during which the ventricles depolarize. The T phase represents repolarization of the ventricles. Repolarization of the auricles generally occurs during the same period as depolarization of the ventricles, but is normally not visible in the electrocardiogram output. On the electrocardiogram, P waves appear as rounded hills, the Q, R, and S phases appear as a sharp upward spike surrounded by downward spikes referred to collectively as the QRS complex. The T phase appears as another rounded hill. Below is the labeled electrocardiogram output for several heartbeats of a normal person.

Electrocardiogram Results (Voltage as a Function of Time)



For diagnostic purposes, physicians generally must determine the time at which each wave hits its peak or valley. Much diagnostic information is represented by the distances between these points and the shapes of the waves that connect the points[19]. Both analysis and tabulation of results are tedious, repetitive tasks that might best be left to a computer [35][20].

The data plotted in the display shown above arrives from remote sensors as a stream of characters representing integer values. The stream is divided into lines no longer than 80 characters in length, each line holding as many integer values as fit. Integers are separated from each other by spaces or newline characters. Since pattern matching is concerned only with the relationship between the integer values, the stream of characters is first converted into a stream of integers. This is accomplished by the following code, which executes as a process whose yield is the stream of integer values. The scanning environment of this process is set externally to the incoming stream of characters.

```
procedure scan_ints()
  local digits

  digits := '0123456789'
  while skipto(digits) do
    suspend integer(advance(many(digits))) \ 1
end
```

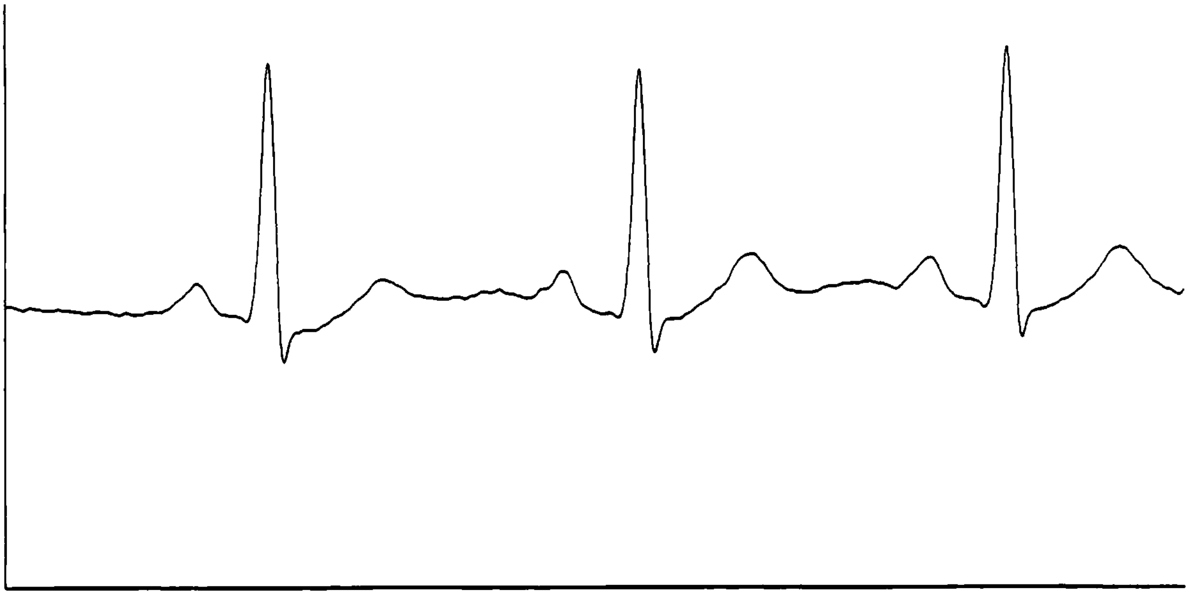
A second phase of processing filters high-frequency noise from the stream of integers by averaging each voltage value with neighboring voltage values. This is realized by the procedure shown below, which also executes as a process. The scanning environment of this process is the yield of the process shown above. The yield of this process is a stream of integers representing the filtered electrocardiogram results.

```
procedure smooth()
  local sum, values

  #  collect up the first 7 values
  values := advance(8)
  sum := 0
  every sum +:= !values

  while put(values, advance(2)[1]) do {
    sum +:= values[-1]
    suspend sum / 8
    sum -:= get(values)
    }
end
```

The output of this phase is plotted below. Note that this filtering method also removes some fine detail from the original signal that was not noise. However, the significant characteristics of the original signal are still strong enough to perform pattern matching.

[19] The software described here makes no attempt to characterize the shapes of the waves. That analysis might best be done by statistical pattern-matching strategies using Fourier series expansions for the wave segments that are isolated using the methods described here.

[20] Several commercial products are on the market for analyzing electrocardiogram output. Once again, the purpose of this discussion is not to present new algorithms, but to emphasize the ease with which existing algorithms can be expressed in Conicon.



Filtered Electrocardiogram Results

The pattern-matching algorithm described here assumes that each heartbeat exhibits each of the five major phases P, Q, R, S, and T. Any heartbeat not having these five phases is not recognized as a heartbeat. This software simply skips over the unrecognized data to the next legitimate heartbeat. In a real application, the pattern-matching system would have to recognize several alternative manifestations of a heartbeat. Attempts to recognize alternative heartbeat patterns could be carried out either in parallel by concurrent processes or by backtracking[21].

Recognition of the P, R, and T phases is accomplished by the hill procedure. The Q and S phases are recognized by the procedure valley. The hill and valley procedures are parameterized according to the shape and expected location of the searched wave form. hill, for example, takes parameters min_val, min_height, start_time, and end_time. min_val, if specified, represents the minimum voltage value that is allowed to precede the crest of the hill. Typically, min_val is the voltage at the time of the preceding valley. If, while looking for a hill, the voltage drops beneath this value, then the location of the valley should shift forward to the new low value. This is accomplished by failing inside of hill. min_height, if specified, represents the minimum height of the hill. The minimum height of the T wave depends, for example, on the height of the preceding P wave. start_time and end_time represent lower and upper bounds on the time at which the peak is expected to arrive. Any voltage peaks that arrive before start_time are considered to be electronic noise. If the peak has not been found before end_time, backtracking to previous goals occurs. Both start_time and end_time are specified in terms of the number of voltage values received since the preceding goal was satisfied. The analog to digital

[21] Analysis of electrocardiograms might benefit from an initial phase of configuration during which the pattern-matching software prepares itself for the peculiarities of a particular heartbeat or electrode placement.

converter used in this application produces voltage values by sampling the voltage at fixed time intervals[22].

Each heartbeat is recognized as one complete unit. After a heartbeat has been recognized. backtracking into the data representing that heartbeat is not allowed. This limits computational complexity by restricting the amount of backtracking that might take place, and limits memory requirements by discarding stream history that has already been processed. The goal of recognizing a heartbeat is comprised of goals representing each of the five phases of the heartbeat and a sixth goal that represents a voltage level after the T phase. The heartbeat goal is represented by the procedure shown below:

```
record coordinate(time, voltage)

procedure heartbeat()
  static tlevel                # time to level off preceding heartbeat
  local next_tlevel            # time to level off current heartbeat
  local p, q, r, s, t

  initial tlevel := 0

  return p := hill() &
    q := valley(p.voltage, p.voltage - 50, 5, 100) &
    r := hill(q.voltage, p.voltage + 1000, 5, 50) &
    s := valley(r.voltage, q.voltage + 50, 5, 50) &
    t := hill(s.voltage, p.voltage - 50, 20, 150) &
    next_tlevel := level_off(t.voltage, t.voltage - 30) &
    (p.time +:= tlevel, tlevel := nextlevel, [p, q, r, s, t])
end
```

Both hill and valley return coordinate records representing the points at which their goals are satisfied. Times are specified relative to the time at which the previous goal was satisfied. Note that the time of the P wave, measured from the time of the preceding heartbeat's T wave, is the sum of the time required for the preceding heartbeat to level off and the time to reach the P crest after leveling off. heartbeat returns a list of coordinate records, representing the time and voltage values that correspond to each of the five phases of the heartbeat.

The two parameters to level_off represent the maximum voltage value that is allowed to precede the point at which the voltage eventually levels off and the maximum voltage at this level. Code to implement level_off is provided below:

[22] Actually, pattern matching for this application did not take place in real time. The voltage values were stored on a floppy disk and later copied to a time-shared VAX, where the prototype implementation of Conicon performed the data analysis. For this application, the throughput of the Conicon system was not sufficient to support real-time processing. However, future implementations of Conicon may provide the necessary processing speed.

```
global fail_point

procedure level_off(max_val, max_height)
  local prev_val, val, i

  every i := 1 to 100 do {
    val := advance(2)[1] | fail
    /prev_val := val
    if val > max_val then {
      fail_point := i
      fail
      }
    if prev_val < val < max_height then return i
    prev_val := val
    }
  fail_point := i
end
```

In the code shown above, the global variable fail_point is set to the scanning offset for the current goal before failing. The procedure into which backtracking occurs will automatically advance itself to this failure point before suspending with an alternative solution. Note that level_off fails if its goal can not be satisfied within 100 subsequent voltage values.

In general, the procedure hill must make sure that the peak it finds represents the highest voltage value it has seen. Within hill, the variable hi_volts represents the highest voltage seen so far. This value is compared with the height of any potential hill before suspending so that hill suspends only if the current height equals the highest voltage seen. An exception to this rule is allowed when recognizing a P wave. This exception allows hill to find the start of a new heartbeat even though it may be invoked in the middle of a heartbeat and might even see the R wave for that preceding heartbeat. This special handling is required to allow initial synchronization of the software with the heartbeats and to allow error recovery whenever the software falls out of synchronization. The implementation of hill is shown below:

```
procedure hill(min_val, min_height, start_time, end_time)
  local hi_volts, i, val, prev_val

  every i := seq(1) do {
    val := advance(2)[1] | fail

    if /prev_val := val then hi_volts := val
    else {
      if val < \min_val then {
        fail_point := i
        fail
        }
      hi_volts <:= val
      }
```

```
        if i > \end_time then {
          fail_point := i
          fail
          }

        if i = \fail_point then fail_point := &null
        if /fail_point & not (\start_time > i) then {
          if /min_val then {
            # looking for P, ignore hi_volts
            if val < prev_val & not(\min_height > prev_val) then {
              # restore stream focus on backtracking
              suspend (advance(1), coordinate(i, prev_val))
              fail_point +:= i
              }
            }
          else if val < (prev_val = hi_volts) & not (\min_height > prev_val) then {
            # restore stream focus on backtracking
            suspend (advance(1), coordinate(i, prev_val))
            fail_point +:= i
            }
          }
        prev_val := val
        }
    end
```

Note that, before suspending, hill performs advance(1). This has the effect of remembering the current scanning focus without shifting the focus. When backtracking into hill becomes necessary, advance is resumed and the stream focus is restored to the point at which hill left off.

The implementation of valley strongly resembles that of hill. The main difference is that there is no need to supply default behavior in case argument values are unspecified. valley is shown below:

```
    procedure valley(max_val, min_depth, start_time, end_time)
      local lo_volts, i, val, prev_val

      every i := seq(1) do {
        val := advance(2)[1] | fail

        if /prev_val := val then lo_volts := val
        else {
          lo_volts >:= val
          if val > max_val then {
            fail_point := i
            fail
            }
          }
    if i > end_time then {
      fail_point := i
      fail
      }

    if i = \fail_point then fail_point := &null
    if /fail_point & start_time <= i then {
      if val > (prev_val = lo_volts) <= min_depth then {
        # restore stream focus on backtracking
        suspend (advance(1), coordinate(i, prev_val))
        fail_point +:= i
        }
      }
    prev_val := val
    }
end
```

All of the procedures discussed in this section are driven by the main procedure shown below:

```
procedure main()
  local p1, p2

  # scan_ints reads from standard input
  p1 := create scan_ints()

  p2 := create (yield(p1) ? smooth())

  yield(p2) ?
    while output(heartbeat())
end
```

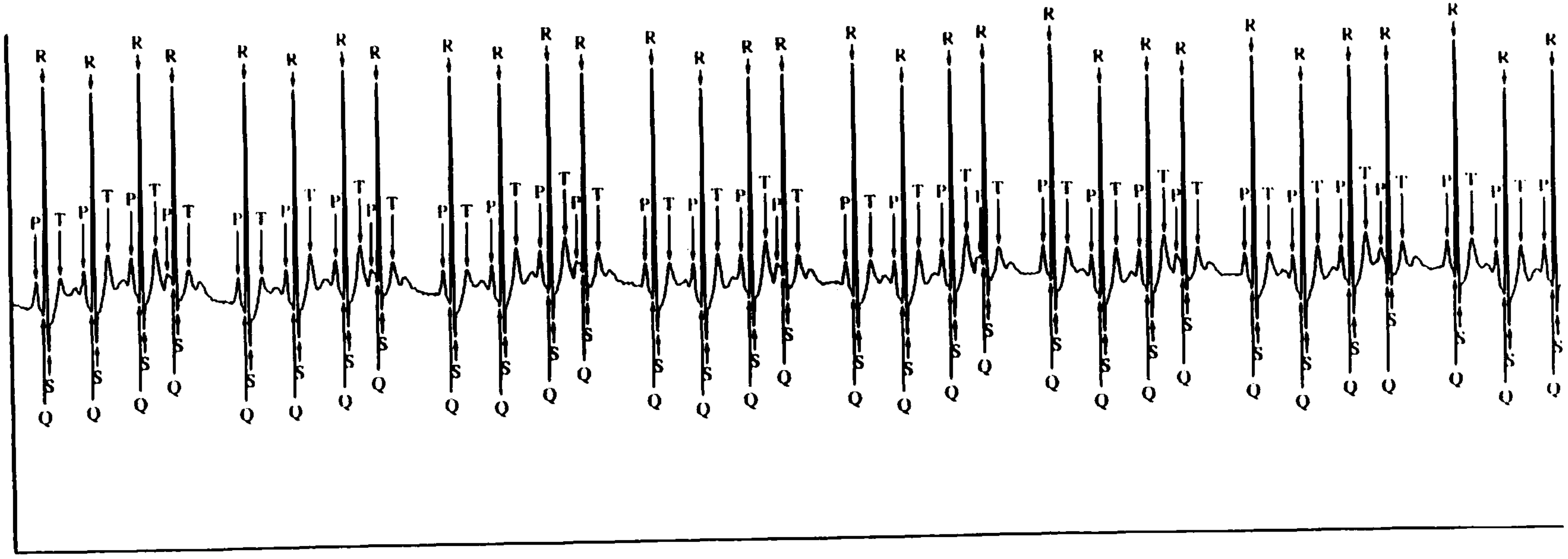
Note that, for this application, pattern matching is accomplished by three pipelined processes.

The output function simply writes the parameterization of each heartbeat to standard output, as shown below:

```
procedure output(a)
  write("p[", a[1].time, ", ", a[1].voltage,
        "]q[", a[2].time, ", ", a[2].voltage,
        "]r[", a[3].time, ", ", a[3].voltage,
        "]s[", a[4].time, ", ", a[4].voltage,
        "]t[", a[5].time, ", ", a[5].voltage, "]\n")
  return
end
```

When provided with real data collected from a human subject, the pattern-matching system described above successfully matched each component of each heartbeat. In the following figure, labels for the electrocardiogram output were generated automatically from the output of the program described above.

Automatically Labeled Electrocardiogram Results (Voltage as a Function of Time)



The pattern-matching system described above processes each voltage value in constant time. This is argued by first considering the costs of processing values within level_off. Since the loop body that processes each voltage value is simply straight-line code, level_off processes each voltage value in constant time. Next consider the costs of locating the T peak using the hill procedure. The loop body of hill is likewise straight-line code except for the two points at which the procedure suspends. In terms of complexity analysis, this suspension can be treated as an invocation of level_off. The level_off invocation processes a maximum of 100 values, each one in constant time. So the costs of processing each value in searching for the T peak is likewise bounded by a constant. Since each procedure except for the initial invocation of hill is given a maximum distance to scan forward, similar arguments bound the computation required to process each value in searching for each of the five phases of each heart beat.

Note that, though the voltage values can be processed in constant time, the constant is very large. Faster algorithms for electrocardiogram analysis probably exist. This algorithm, however, has the advantage of being very simple and understandable. This implementation provides a strong foundation upon which experiments to better understand the characteristics of electrocardiogram results might be constructed. The results of this kind of experimentation might suggest new, more efficient pattern-matching algorithms. Alternative algorithms, even those that perform no backtracking, would likely still benefit in their implementation from the high-level language features provided by Conicon. Note, for example, that neither of the first two processes in this pattern-matching pipeline perform any backtracking. Even though the work they perform is non-trivial, it is described using only a few lines of concise Conicon code.

Chapter 6: Implementation of Streams and Concurrent Processes

The prototype implementation of Conicon is based on the implementation of Icon. Streams and concurrency were added to Icon by modifying its implementation, which consists of three major components: a translator, a linker, and an interpreter. Each of these components is implemented primarily in C. The translator reads Icon source code and produces assembly language for an abstract machine. This machine, known as the Icon virtual machine, was designed to simplify the translation and execution of Icon programs. The Icon linker resolves external references in the intermediate code and assembles the translator output into machine language for the virtual machine. The interpreter is simply a virtual machine simulator. The complete implementation of Icon is described in detail in [28].

In the implementation of Conicon, a few modifications were made to Icon's translator in order to recognize and generate code for the concurrent alternation operator and to change the semantics of the create operator. However, most of the changes, such as redefinition of the scanning operator's semantics, were implemented by simply redefining the semantics of particular virtual machine instructions. This discussion of the implementation of streams and concurrent processes concentrates on the virtual machine implementation.

6.1 Review of the Conventional Icon Virtual Machine

A simplified description of Icon's virtual machine is provided here. Understanding Icon's virtual machine is essential to understanding the implementation of Conicon. The Icon virtual machine resembles real stack-based architectures except that its instruction set is highly specialized and represents a much broader spectrum of operations (as ordered by their complexity) than would likely be provided in any real machine. Some of the simple virtual machine instructions, such as pop, represent a small fixed amount of computation. Other instructions, such as subsc, which performs hash table lookups, represent an unbounded amount of computation. In between these extremes are instructions that represent large but bounded amounts of computation. An instruction of this type is create, which among other things, must allocate and initialize a large run-time stack. The virtual machine instructions, called icode, are stored in the simulated machine as an array of integers. A pointer into this array, called ipc, represents the current flow of control within the icode array. ipc stands for icode program counter. Most virtual machine instructions take arguments from the run-time stack and return results to the stack. For example, the virtual machine instruction:

```
int     10
```

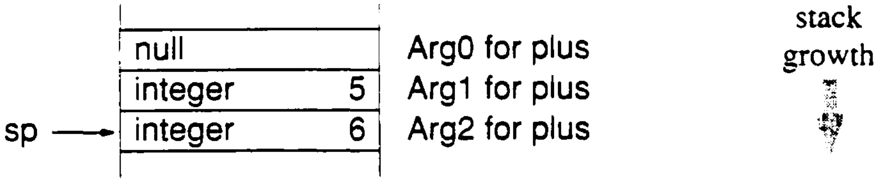
pushes the internal representation for the integer 10 onto the stack. Special virtual machine instructions simplify the creation of certain frequently used values. For example, the instructions push1, pushn1, and pnull place onto the stack the values 1, −1, and null respectively. By convention, virtual machine instructions that take arguments from and return results to the stack expect a stack location to be set aside in advance for the returned result. For example, plus expects its two operands to be supplied as arguments one and two, and places the sum of these two values on the stack in the position reserved for argument zero. The generated code for the expression

```
5 + 6
```

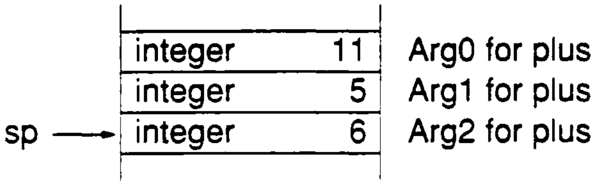
is, for example:

```
pnull
int    5
int    6
plus
```

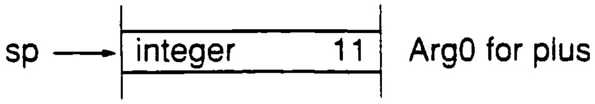
Immediately before the plus instruction is executed, the top of the virtual machine's stack appears as shown below. Since all internal values are tagged with their type, both types and values are labeled in this figure. Note that this stack grows from high to low addresses.



At some point during execution of this instruction, argument zero is overwritten with the result of performing the computation:



Upon completion of the plus instruction, unneeded values are removed from the stack:



Bounded Expressions

Bounded expressions in Icon provide a linguistic mechanism for restricting the depth of backtracking. After a bounded expression has produced a result, backtracking into that bounded expression is not permitted. An efficient implementation of bounded expressions must reclaim the resources (memory) allocated to evaluation of a bounded expression after evaluation of the expression has terminated.

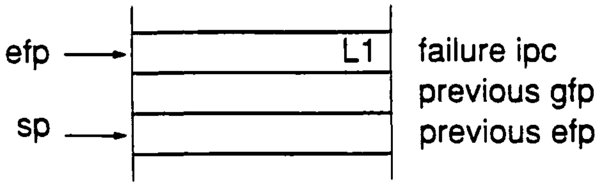
Within a compound Icon expression, any expression terminated with a semicolon is a bounded expression. The following Icon code, for example, represents a bounded expression:

```
        write(5 + x);
```

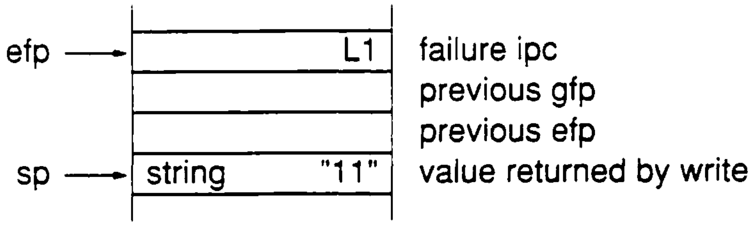
Assuming that write and x are global variables, this is translated into the following virtual machine code:

```
             mark    L1
             global  write
             pnull                  # Arg0 for plus, becomes Arg1 for write
             int     5              # Arg1 for plus
             global  x              # Arg2 for plus
             plus                   # plus produces Arg1 for write
             invoke 1               # invoke write with one argument
             unmark
        L1:
```

The virtual machine instructions mark and unmark surround the generated code for each bounded expression. The operand of mark represents the label (known as the failure label) to which control flows if the bounded expression is unable to produce a result. This is described in greater detail below. The mark instruction establishes on the stack an expression frame for evaluation of the bounded expression. The expression frame includes storage for the failure label and previous values of the virtual machine's efp and gfp registers. efp, which represents the current expression frame, is set to point to the newly established expression frame and gfp, which points to the current generator frame, is set to 0, indicating that no generators are active within this bounded expression (generator frames are described more fully in the following section). After execution of mark completes, the top of the machine's stack contains the following information:



Since Icon's write produces the last string written, following invocation of write, the top of the stack appears as:



When the unmark instruction is executed, the stack is cleared to the height marked by efp (removing the failure ipc) and both efp and gfp are restored to their previous values according to the information saved within the current expression frame.

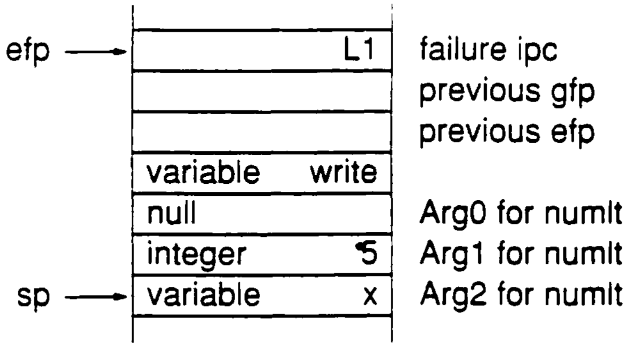
In the absence of generators, any conditional operation that fails causes evaluation of the enclosing bounded expression to abort. The following expression, for example, prints the value of x if it is greater than 5, but does not invoke write if the comparison fails. Consider:

```
write(5 < x);
```

The generated code for this expression is almost identical to the code from the previous example:

```
          mark    L1
          global  write
          pnull                 # Arg0 for numlt
          int     5             # Arg1 for numlt
          global  x             # Arg2 for numlt
          numlt                 # numlt produces Arg1 for write
          invoke 1              # invoke write with one argument
          unmark
L1:
```

The only difference between this code and the code in the previous example is that here, numlt replaces plus. numlt checks to see that its first argument is numerically less than its second. If this condition is satisfied, numlt copies its second argument onto the stack location reserved for its zeroth argument. If, however, this condition is not satisfied, then control flows to the failure ipc that is associated with the current expression frame, stack values up to and including the current expression frame are popped off of the stack, and both efp and gfp are restored to their previous values. At the time numlt is executed, the stack appears as shown below:



Generating Expressions

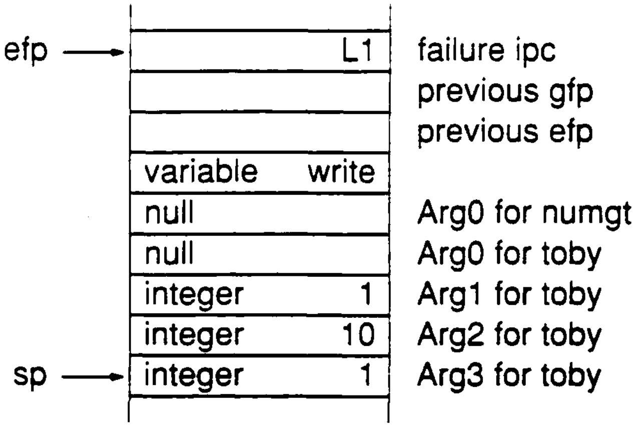
Certain machine instructions are capable of producing more than a single result. For example, the toby instruction generates integers ranging between its first and second arguments separated by its third argument. Consider evaluation of the following expression:

```
write((1 to 10 by 1) > x);
```

which is represented by the following virtual machine code:

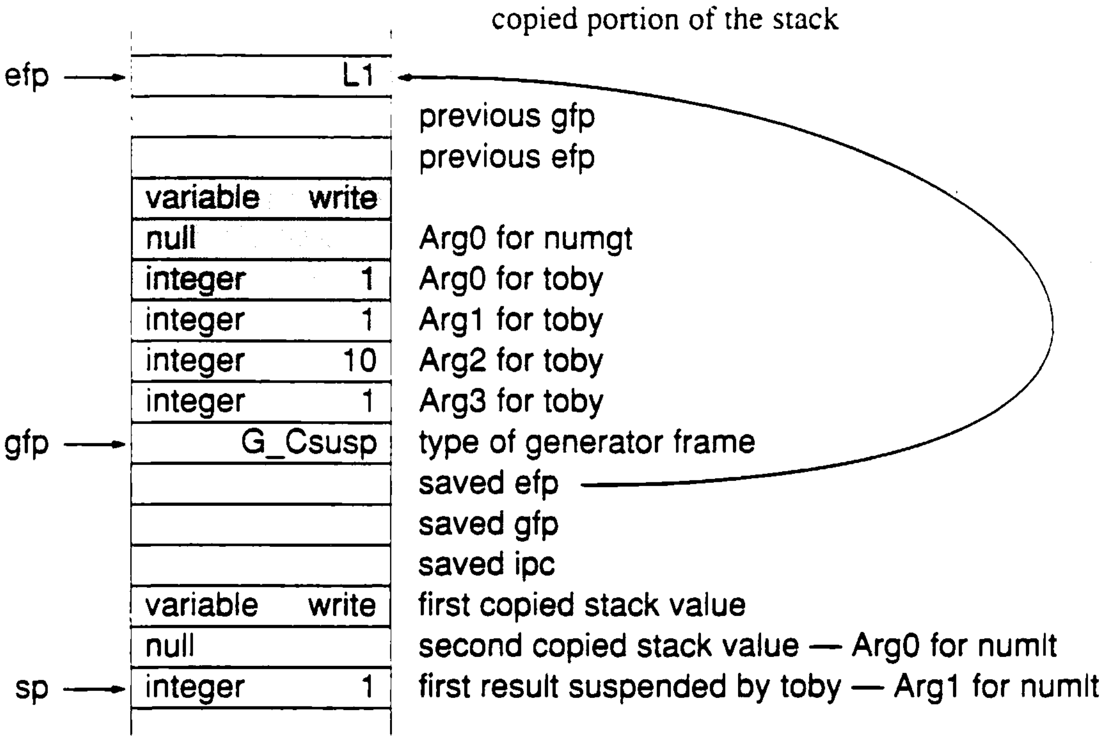
```
        mark   L1
        global write        # Arg0 for invoke
        pnull               # Arg0 for numgt
        pnull               # Arg0 for toby, becomes Arg1 for numgt
        push1               # Arg1 for toby
        int    10           # Arg2 for toby
        push1               # Arg3 for toby
        toby                # toby produces Arg1 for numgt
        global x            # Arg2 for numgt
        numgt               # numgt produces Arg1 for invoke
        invoke 1            # invoke write with one argument
        unmark
L1:
```

When toby is executed, the top stack elements are as shown below:

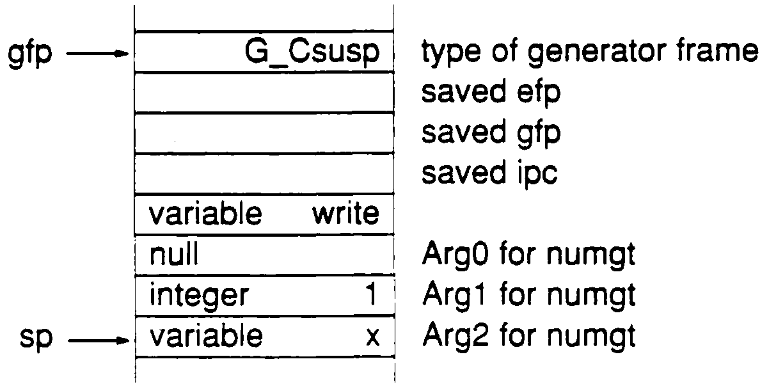


efp →		L1	failure ipc
			previous gfp
			previous efp
	variable	write	
	null		Arg0 for numgt
	null		Arg0 for toby
	integer	1	Arg1 for toby
	integer	10	Arg2 for toby
sp →	integer	1	Arg3 for toby

The first value produced by toby is the integer one. Since toby must be prepared to produce alternative results in case backtracking is required, it is not possible to simply discard toby's arguments in order to evaluate numgt. However, the numgt instruction expects the top three items on the stack to represent its arguments numbered zero, one, and two. In the current implementation of the virtual machine, instructions such as toby are suspended by building a generator frame and creating a copy of the portion of the stack that is relevant to further evaluation. The relevant stack information begins at the enclosing expression or generator frame and includes all data up to the zeroth argument of the suspending machine instruction. The interpreter then recursively calls itself in order to obtain a new set of local variables so that the current set of local variables can be preserved until the toby instruction is resumed. Following suspension of toby, the top of the stack contains the following values:



Expression evaluation now executes within the newly established generator frame instead of the expression frame. When the numgt instruction is executed, the top portion of the stack consists of:



If numgt succeeds, invoke calls write, the value of x is written, and unmark clears the stack to the depth marked by efp, removing the current expression frame. However, if numgt fails, then backtracking occurs automatically. The difference between this situation and the previous example in which failure of numlt resulted in branching to L1 is that here, numgt executes within a generator frame. In order to backtrack, the values of ipc, efp, and gfp are restored from the current generator frame, and sp is set to point at the stack element immediately beneath the generator frame. Then the toby instruction is resumed by returning a special signal from the current recursive invocation of the interpreter. If toby succeeds in suspending an alternative result, then the virtual machine code that follows toby is restarted with the alternative result on the stack (remember that the ipc was restored to its previous value when toby was resumed). If toby is unable to produce alternative results, this failure causes control to flow to label L1 and sp is set to point at the data value immediately beneath the current expression frame.

6.2 Conicon's Virtual Machine

Several changes to the internal organization of the Icon interpreter have been necessary in order to support concurrent processes and real-time response to interrupts. Some of these changes provide the new expressive capabilities required to describe concurrency and manipulate processes. The major motivation for changes, however, has been the need to comply with real-time performance constraints.

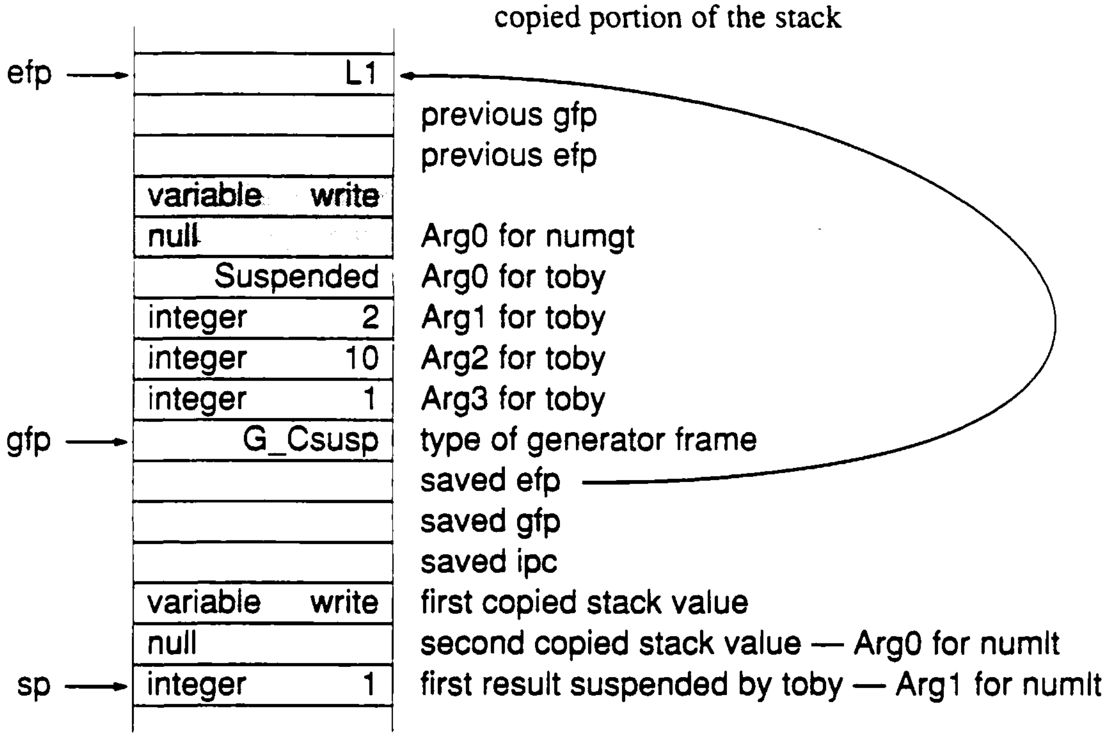
Removal of Recursion from the Interpreter

In order for multiple processes to share the single virtual machine, an internal representation for the machine state was designed and a mechanism for saving and restoring the virtual machine's state provided. Processes take turns running on the virtual machine. When one process's turn is over, the state of the virtual machine is stored as part of the abstract representation for that process. Then the next process that is ready to run is granted access to the virtual machine by setting the machine's state according to that process's state information. The process that is currently running on the virtual machine is called the current process.

One difficulty with saving the state of Icon's virtual machine is that part of its state is buried within C call frames for recursive invocations of the interpreter. Although Icon does not support concurrent processes, Icon does perform context switches when evaluating coexpressions. In the Icon interpreter, a context switch consists of saving the virtual machine's register values and the value of the host hardware's stack pointer, setting the host machine's stack pointer to use the stack of the new coexpression, and restoring the virtual machine's state from the new coexpression's data structure. Obtaining and setting the host hardware's stack pointer is accomplished by a small subroutine written in assembly language.

The approach taken in the implementation of Icon's virtual machine assumes that garbage collection of memory allocated for stacks never relocates a stack that is in use. Because the special real-time garbage collection algorithm used in the implementation of Conicon's virtual machine must be able to relocate all allocated objects, the Icon approach to context switching is not feasible. Instead, a small change was made to the interpreter so that recursion would no longer be necessary. Whenever a virtual machine instruction must suspend itself, the zeroth argument of that instruction is overwritten with a special value indicating that the instruction has been suspended and the ipc value that is saved in the new generator frame is set to point at the suspending instruction instead of pointing at the instruction that follows. When backtracking is required, the suspended instruction is simply restarted. Execution of a machine instruction that is capable of suspending begins with examination of its zeroth argument. If the value of this argument is null, the instruction attempts to produce the first of potentially many values. If, however, the value of this argument indicates that the instruction has been suspended, then the instruction behaves differently, producing alternative results instead of an original value.

Suspended instructions frequently need to retain a small amount of information to represent their intermediate state of computation. In Icon's virtual machine, this information is retained within the call frame of the interpreter, which recurses to obtain a new set of local variables. However, in Conicon's virtual machine, this state information is kept on the virtual machine's stack. For example, a suspended toby instruction must remember which integer value to produce on its next resumption. When toby suspends, it overwrites its first argument with a new lower bound on the sequence of integers to be generated. After suspending the first value in a sequence from 1 to 10, for example, the top of the stack contains:



efp →		L1	
			previous gfp
			previous efp
	variable	write	
	null		Arg0 for numgt
		Suspended	Arg0 for toby
	integer	2	Arg1 for toby
	integer	10	Arg2 for toby
	integer	1	Arg3 for toby
gfp →		G_Csusp	type of generator frame
			saved efp
			saved gfp
			saved ipc
	variable	write	first copied stack value
	null		second copied stack value — Arg0 for numlt
sp →	integer	1	first result suspended by toby — Arg1 for numlt

In some cases, it is not possible to simply overwrite arguments with this intermediate state information either because there is too much state information or because the values of the arguments must be retained. In these cases, the instruction is redefined to take additional null-valued arguments that provide storage locations for the intermediate state information that must be retained when the instruction is suspended. For example, the bang instruction, which generates all elements of its single argument, needs much more state information than can be stored in the single stack location reserved for this argument. In Conicon's virtual machine, bang is defined to expect five arguments and the translator's code generator has been modified to push four extra nulls onto the stack prior to executing bang. Built-in functions that must suspend multiple results are implemented in a similar fashion.

Conversion to RISC virtual machine

In real-time systems, context switches are often required in response to particular events such as successful communication between concurrent processes or between the system and sensors or effectors in the external environment. Real-time constraints require that such context switches occur within a small fixed amount of time of when the event requiring the context switch occurs. As with most real computer architectures, context switches in Conicon's virtual machine are permitted only between execution of machine instructions. Therefore, a lower bound on the worst-case time required to switch contexts is the maximum time required to execute any single virtual machine instruction. Since the time required to switch contexts must be bounded by a small constant in order to satisfy real-time constraints, the time required to execute each of Conicon's virtual machine instructions must be bounded by a small constant. As mentioned above, in Icon's virtual machine, certain instructions require unbounded amounts of time to execute. Machine instructions that do not execute within a small constant time are replaced or redefined in Conicon's virtual machine to comply with these real-time constraints. Instructions in the Icon machine whose execution times are not bounded by small constants are classified according to the reason for their non-compliance with real-time constraints. Methods of dealing with each group of instructions are described in following paragraphs.

One class of instructions includes all of the instructions that represent large, but bounded, amounts of computation. Instructions in this category are create, which creates a new process and a new stream representing the yield of that process, and coalt[23], which creates two new processes and a single shared output stream and then generates all of the values read from that stream. The general strategy for dealing with these instructions is to divide the responsibilities of these single instructions between several instructions, each of which is much simpler than the original instruction. For example, the actions taken by the create instruction are:

```
create:
        allocate memory for a new process data structure
        allocate memory for a new process stack
        allocate memory for a new stream data structure
        initialize memory for the new stream
        initialize memory for the new process stack
        initialize memory for the new process data structure
        enqueue the new process on the ready list
```

In a real-time implementation of Conicon's virtual machine, each of the steps described above is implemented by a single virtual machine instruction. As discussed in the following chapter, the garbage collection that accompanies each allocation executes in time proportional to the size of the allocation. Since the size of each of the objects allocated above is fixed, the time required to execute each of the allocation steps is bounded. Because the size of a process stack is large in comparison to the other allocated objects, it may be desirable to further divide the allocation and initialization of this object into several even simpler instructions. Initialization of a large data structure is easily divided into small steps, each one initializing a different portion of the data structure. Below, a method of incrementally allocating a large object is also described. A method of enqueuing a process in constant time is described below in the subsection entitled "Context Switching".

A second class of virtual machine instructions includes all of the instructions that allocate a bounded amount of memory while performing an unbounded amount of computation. Instructions in this category include lexical comparison instructions, which may need to individually examine every character in their string arguments of unbounded length, and the subscripting instruction which may need to perform a hash table lookup. The general strategy for dealing with these sorts of instructions is to provide the functionality of these instructions with blocks or sequences of simple virtual machine instructions that iterate. For example, the eqv instruction verifies that its two arguments are equivalent by value and fails if they are not. This instruction

[23] Actually, coalt executes in two phases. The first phase creates two processes and their shared yield. The second phase generates all of the values available from the stream. Only the first phase of execution represents a "large, but bounded, amount of computation."

can be divided into several steps as shown below. This code uses the machine semantics of expression frames and failure labels described above and assumes that an expression frame and corresponding failure label have already been established[24]. Note that expression frames can be nested. Each of the operations described below in italics represents a constant-time operation that could be implemented by a single virtual machine instruction. Operations that verify compliance with certain conditions fail if those conditions are not satisfied.

```
eqv:
        verify that the type of Arg1 is the same as the type of Arg2
        mark  L1
        verify that Arg1's type is numeric
        unmark
        verify that the numeric value of Arg1 equals the numeric value of Arg2
        goto  L5

L1:                    # type is not numeric
        mark  L2
        verify that Arg1 is a cset
        unmark
        verify that Arg1's cset is equivalent to Arg2's cset
        goto  L5

L2:                    # type is not numeric or cset
        mark  L4
        verify that Arg1 is a string
        unmark
        verify that Arg1's string length is same as Arg2's string length
        set temporary indexing variable offset to 0
Loop:
        mark  L5
        verify that offset is less than length of Arg1
        unmark
        verify that both strings have the same character values at position offset
        increment offset
        goto  Loop

L4:                    # type is not numeric, cset, or string
        verify that Arg1 points to same object as Arg2
L5:
```

Another interesting machine instruction in this category is subsc which performs string, list, and table subscripting. Strings are stored internally as arrays of characters. Given the starting address of the string and the desired offset within the string, simple constant-time addressing arithmetic yields the address of the desired character Lists, however, are stored as linked lists of list-element blocks. Each list-element block holds a finite number of the values stored in the list. Within the virtual machine, hash tables with chaining to resolve collisions are used to implement the table data type. Icon's subsc machine instruction is replaced by the following pseudo-code:

[24] The code presented here uses only virtual machine instructions that have been previously discussed. A more efficient implementation makes use of a computed goto based on the type of Arg1.

```
subsc:
        mark   L2
        verify that Arg1 is a table
        unmark
        compute hash value for Arg2
        use computed hash value to select appropriate hash chain
        store the first link of the hash chain in temporary variable link
L1:
        # note: if link is null, then fail in outer context
        verify that link is not null
        obtain the value represented by link
        mark   L6
        verify that the obtained value is not equivalent to Arg2 by executing
          the eqv code described above
        unmark
        set link to next element of hash chain
        goto   L1

L2:                        # Arg1 is not a table
        convert Arg2 to an integer value if it's not already an integer
        mark   L5
        verify that Arg1 is a list
        unmark

        adjust negative or zero-based subscript depending on size of Arg1
        verify that specified subscript is within bounds required by size of Arg1
        store pointer to the first list-element block in temporary variable block_ptr
L3:
        # note: if block_ptr is null, then fail in outer context
        verify that block_ptr is not null
        mark   L4
        verify that specified position is not found in block referenced by block_ptr
        unmark
        set block_ptr to next list-element block
        goto   L3

L4:
        obtain desired element out of block referenced by block_ptr
        goto   L6

L5:                        # Arg1 is not a table or list
        convert Arg1 to a string if it's not a string already
        adjust negative or zero-based subscript depending on length of Arg1
        verify that specified subscript is within bounds required by length of Arg1
        obtain the desired character out of string represented by Arg1
L6:
```

In the pseudo-code above, conversion to a string value aborts with a fatal run-time error if the conversion is not possible. Likewise, conversion to an integer aborts with a run-time error if that conversion is not possible. In either case, for all values that can legitimately be coerced, the coercion executes in constant time.

The third category of virtual machine instructions includes all instructions that allocate and initialize an unbounded amount of memory. Instructions in this category include cat and lconcat which catenate strings and lists respectively; diff, inter, and unions which perform set difference, intersection and union; and sect which builds a substring or sublist from its string or list argument. For some of these instructions, though the total amount of allocated memory is unbounded, this memory is allocated as a large number of small constant-sized objects. This is the case, for example, with set operations. For these instructions, the same techniques described above provide constant-time response to interrupts. However, some of these instructions, such as cat, require that an unbounded amount of contiguous memory be allocated. Since the time required to allocate memory is proportional to the size of the allocation request, this memory cannot be allocated by a single invocation of a virtual machine instruction. Since interrupts (or time slices) might occur between execution of consecutive instructions, it is not possible to allocate a large segment of memory by repeatedly allocating small fixed-size objects and assuming they are allocated contiguously[25]. Instead, a special virtual machine instruction is provided for allocating a variable amount of memory. This machine instruction blocks the current process until the desired amount of memory can be allocated in constant time[26]. Similar to execution of instructions that suspend multiple results, when virtual machine instructions require the current process to be blocked, the process's ipc is set to restart the blocked instruction when it becomes unblocked, and the instruction's zeroth argument is overwritten with a special value so that the restarted instruction can detect that it was previously blocked. Every memory allocation is potentially accompanied by an amount of garbage collection that is proportional to the size of the allocation. After the garbage collection completes, the requested memory is set aside by simply decrementing a special internal pointer value by the size of the request. Note that the allocation itself, which consists entirely of decrementing the special internal pointer, executes in constant time. A process that becomes blocked waiting for memory to become available must wait on a queue for the corresponding garbage collection to complete. Assuming that no other activities of equal or higher priority require access to the CPU, the maximum time that a process must wait on this queue is proportional to the size of the allocation request. In order not to starve processes that are waiting for memory, all allocation requests coordinate with this queue. All condition queues in the system are ordered first by priority and then by arrival time. As soon as enough memory to satisfy the first process on the memory queue becomes available, that process is unblocked and its requested memory is granted. Conceptually, garbage collection is performed by a concurrent process that executes with the priority of whatever process is at the front of the memory queue. If the memory queue is empty, then garbage collection executes with the lowest possible priority. Garbage collection is described in greater detail in Chapter 7.

A final category of virtual machine capabilities that must be restructured in order to comply with real-time constraints is the set of built-in functions. In the Icon virtual machine, all of these are represented by a single machine instruction called invoke which simply calls the C code that implements a particular built-in function. Certain functions, like sort, perform an unbounded amount of computation. The general strategy for restructu.ing these capabilities is to implement

[25] Since concurrent Conicon processes share global variables, all memory is allocated from a shared memory region. Consecutive allocations are contiguous unless a new pass of the garbage collector begins between two consecutive allocations. However, since more than one process may be allocating memory, consecutive allocations made by one process may be interleaved with allocations made by other processes.

[26] In analyzing the time required to execute a section of Conicon code, a programmer must account for the costs of allocating memory. As discussed in Chapter 7, the time required to allocate memory is proportional to the size of the allocation request.

each built-in function as a sequence of virtual machine instructions, each of which executes in constant time[27]. The same techniques discussed above for dealing with individual virtual machine instructions apply also to redesigning the implementations of built-in functions. In Conicon's virtual machine, built-in functions are provided as a library of virtual machine code instead of being part of the interpreter.

Context Switching

As discussed above, multiple processes share access to the virtual machine by taking turns. Each turn, called a time slice, consists of a limited amount of computation. Currently, this computation is measured in terms of numbers of virtual machine instructions executed[28].

Context switches occur whenever a process's turn expires, when the current process must be blocked or killed, or when a process with higher priority than the current process is unblocked. Processes become blocked when they must wait for particular resources (such as additional memory) or for particular events (such as communication with concurrent processes). Processes that are not running are maintained on queues associated with the resources or events for which they are waiting[29].

All queues are ordered first by process priority and then in first-come first-served order. When a time slice expires, the priority of the first process on the ready queue is compared with the priority of the current process. If the process on the queue has equal or higher priority[30], then it becomes the current process. When the virtual machine switches contexts, the previous current process is placed on the ready queue, waiting for access to the virtual machine's interpreter. Likewise, whenever a process becomes blocked waiting for a particular event, it is placed on the wait queue for that event. When a process is killed, it must be removed from whatever queue it is on. Since all queues are ordered by process priority, if a process's priority is changed, the relative location of that process within a queue must be changed to reflect the new priority. All of these operations must execute in constant time.

Process queues are organized as doubly-linked lists[31]. The last process in the queue is linked to the first in a circular fashion so that new processes can be appended to the end of the list without examining each process on the list. A second thread through the processes on the list links the first processes of each priority together. Since there are a total of 16 different priorities,

[27] This technique works for almost all of the existing Icon functions. Certain functions, like system, require operating system support that is inconsistent with the goals of a real-time programming language. Functions that cannot be implemented using these techniques are removed from the library of Conicon support routines.

[28] When analyzing the processing requirements of a real-time software system written in Conicon, it is necessary to consider the requirements of each process to determine whether it is possible to schedule all of the necessary computation. Since the overhead of time slicing is limited to a maximum number of context switches every second, each executing in constant time, these costs are accounted for by simply subtracting the constant costs of time slicing from each quantum of available CPU time.

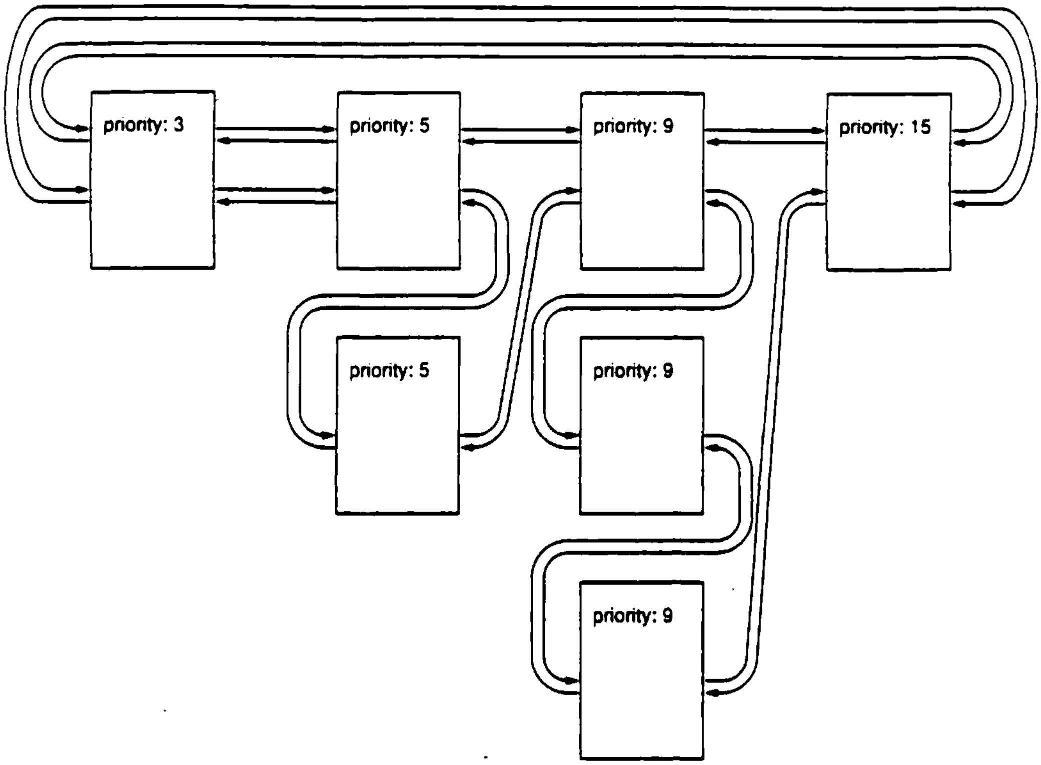
[29] Even the ready queue represents a list of processes waiting for access to a particular resource. In this case, the shared resource is the virtual machine.

[30] Priorities are numbered from 0 to 15, with priority 0 processes given scheduling preference over all other processes. Throughout this dissertation, high priority refers to scheduling preference and thus low priority numbers. Likewise, increasing priority corresponds to decreasing priority numbers.

[31] Constant-time constraints could be satisfied by a variety of alternative data structures. One advantage of the data structure proposed here is that the storage allocated to an empty process queue is only that required for the pointer to the first process in the queue plus the links allocated within each process descriptor. Note that every process except the current process is on a queue and that no process can be simultaneously linked on multiple queues.

the length of this thread is limited. In order to add a process to a queue, the thread that connects the leading processes of each priority is followed[32] until either the thread loops back to the list's first element, or a process of lower priority than the new process is found. In either case, the new process is inserted onto the queue immediately before the point at which the search ends.

A Process Queue:



The figure above illustrates the doubly-threaded doubly-linked organization of process queues within the run-time support system. The top thread through the processes links the leading process of each priority. The bottom thread links all processes on the queue into a single list. Not shown in the figure above is a pointer from each process in the queue to the descriptor that controls the queue. Whenever the first process in the queue changes, the controlling descriptor must be modified to point at the new first process in the queue. Given a pointer to any process in the queue, that process can be removed from the queue by updating a maximum of ten pointers[33]. Insertion of a process into the queue requires examination of no more than 16 different processes and updating a maximum of ten pointers. Changing the position of a process within the queue as a result of a change in its priority consists simply of removing the process from the queue and inserting it again. Process queues are used only by the kernel code, within which no more than one pointer ever refers to the same queue. Pointers representing a queue simply point to the leading process in the queue. A null-valued pointer represents an empty queue. The leading

[32] The algorithm described here executes in constant time. An improvement on this algorithm that yields a performance gain of 50% is achieved by examining the thread in reverse order for priority numbers greater than 7.

[33] Removing a process from the queue must update a maximum of four pointers for each thread through the process and may have to update the pointers that connect the queue's first process with the queue's controlling descriptor. Inserting a process into the queue requires similar manipulation.

process in the queue is removed by setting the queue pointer to the following process and adjusting links as with removal of any other process from the queue. Note that all of these operations execute in constant time. Note also that the number of priority levels in the system can be adjusted to tune worst-case response time.

In the single-tasking version of the interpreter, the virtual machine is represented by the following code outline:

```
repeat {
  fetch instruction and operands
  increment instruction pointer by size of instruction and operands
  execute the instruction
  }
```

In traditional machine architectures, a check for pending interrupts immediately precedes the instruction fetch. In Conicon's virtual machine, interrupts correspond to events that awaken blocked processes or events that require the current process to be put to sleep or killed. For example, if a process writes data to an internal stream from which a higher priority process is blocked waiting to read, then the interrupt flag is set so that a context switch occurs on the following instruction. As discussed above, an attempt by the current process to allocate a large amount of memory may require that the process be blocked until that memory is available. In order to block the current process, the allocation instruction sets the interrupt flag to True, the global variable why_interrupt to BlockProcess, and the global variable where_block to point at the queue of processes waiting for memory. Then ipc is decremented so that when this process runs again, it restarts the allocation instruction. schedule, on noticing that the current process must be blocked, inserts this process on the queue specified by where_block and sets the process status to Blocked. A context switch is also required if the current process must be killed. In this case, interrupt is set to True and why_interrupt is set to KillProcess. When schedule notices that it must kill the current process, it simply overwrites its pointer to the current process's data structure with NULL and then searches for a new current process. When called because of expiration of a time slice, schedule behaves as discussed in the description of process queues, comparing the priority of the current process with that of the leading process on the ready queue and acting accordingly. Each time schedule returns, it sets time_slice to SliceSize. The multi-tasking interpreter is modeled by the following:

```
global interrupt, why_interrupt, where_block

  ...

  time_slice := SliceSize
  repeat {
    time_slice := time_slice - 1
    if time_slice = 0 and not interrupt then {
      interrupt := True
      why_interrupt := TimeSlice
      }

    if interrupt then schedule()

    fetch instruction and operands
    increment instruction pointer by size of instruction and operands
    execute the instruction
    }
```

Note that, in the code outline shown above, an interrupt occurs at least every SliceSize iterations of the interpreter loop.

If schedule is called because the current process must be blocked, there may not be any processes ready to run. If the ready queue is empty in the UNIX implementation of Conicon, schedule checks to see if at least one process is waiting for expiration of a timer or for system input or output. If so, schedule invokes the UNIX select system routine with arguments specifying that the virtual machine should be suspended until either the specified time arrives or until one of the relevant files becomes ready for I/O. If the ready list is empty, and no processes are waiting for either timers or file access, then Conicon aborts with an error message indicating system deadlock.

Low-Level Process Synchronization and Manipulation Primitives

In most multi-tasking software systems, a run-time kernel provides interprocess synchronization and communication facilities. Conceptually, the kernel is a library of system functions that execute with special privileges. These privileges generally consist of access to regions of memory that are otherwise protected and uninterrupted access to the CPU. Because interrupts are typically disabled within the kernel code, the worst-case time required to respond to an interrupt can be no shorter than the longest time required to execute a kernel function. Since interrupt response time in Conicon's virtual machine already depends on the worst-case time required to execute a virtual machine instruction, each of the kernel services required to implement concurrent processes is provided by a single virtual machine instruction. There is no need to provide any security enforcement at run time because Conicon's translator is trusted to generate correct code and Conicon provides no mechanisms for dealing directly with internal kernel data structures.

High-level synchronization and communication capabilities such as those provided by the built-in functions advance, probe, and write are provided in libraries of icode. Within these libraries, calls to kernel routines are implemented by special virtual machine instructions. These instructions, collectively referred to as low-level process synchronization and manipulation primitives, are described in following paragraphs.

Process creation consists of allocation and initialization of several data objects. After the appropriate data structure has been constructed, a pointer to this data structure is passed to the kernel so that the process can begin to execute. The virtual machine's activate instruction takes a single argument which points to a new process data structure and places that process on the ready queue. The retire instruction takes a single argument that points to a process data structure and causes that process to kill itself[34]. A process that retires another process is automatically blocked until the retired process is completely dead. There is no bound on the amount of work that must be carried out in order to kill a process. For example, each of the process's descendents must be killed and every process actively retiring this process or simply waiting for it to die must be unblocked. When a process is retired, it is immediately unblocked from whatever wait queue it might be blocked on, and it is unlinked from its parent and sibling processes. Data structures representing process genealogy are described below. Then the process is given the priority of the highest priority process that has requested its retirement and its ipc is set to point at a library routine of icode called die that allows the process to carry out the unbounded amount of work associated with its death. The status of a process that is executing this subroutine is set to Retired so that subsequent requests to retire the process do not cause execution of this subroutine to restart. Processes retire themselves in this way instead of expecting the killing process to carry out this work because this approach eliminates complications that might exist if a killing process is itself killed before the work is completed. The last instruction in this subroutine is deactivate, which forces a context switch without placing the current process on the ready queue. A complete description of this subroutine is provided at the end of this subsection.

Processes can temporarily protect themselves from being retired by executing the shield instruction. Processes that have been shielded can only be retired if they are blocked on a queue other than the ready queue. An attempt to retire a shielded process that is not blocked causes the process that executed the retire instruction to be blocked until the shielded process either drops its shield or executes an instruction that would cause it to become blocked. At that moment, the shielded process is retired. Processes drop their shield by executing the unshield instruction. Within the code that implements high-level system functions, processes shield themselves whenever they must execute all or none of a particular sequence of virtual machine instructions. Typically, a process raises its shield and then blocks itself waiting for a particular event before executing the critical code. If the process is deactivated while on the blocked list, then none of the critical code is executed. However, if an attempt is made to retire the process after it has become unblocked, the retiring process will block until this process completes execution of its critical code and drops its shield. The reason that the process must shield itself before waiting for the event is that other processes may be aware that the shielded process has seen the event and based on that knowledge, may assume that this process will carry out certain actions prompted by that event. Note that it is acceptable for the shielded process to be killed before the event occurs since, not having seen the event, no other processes are expecting it to carry out any special work. Examples of the use of this mechanism are provided below.

In order to protect concurrent processes from seeing or creating internally inconsistent shared data structures, a method of guaranteeing mutually exclusive access to shared data structures is provided. The instruction create_guard creates a data object, called a guard, that provides functionality similar to semaphore. create_guard takes a single integer argument that

[34] The strategy described here is similar to the strategy used to kill UNIX processes using the exit system call [36, p. 212]. zombie status for a UNIX process is analogous to Retired status for a Conicon process.

represents the initial value of the semaphore. Guards are manipulated using the P, V, enter_guard, and exit_guard virtual machine instructions. enter_guard and exit_guard have semantics similar to P and V respectively. In addition to executing a P semaphore operation, enter_guard places the guard on a linked list of guards associated with the current process. This list is kept in last-in first-out order and, by convention, is limited in size to a maximum of two guards. The exit_guard instruction removes the most recent guard from the list of guards associated with the current process and executes a V semaphore operation on it. When a process is retired, any guards on this list are removed one at a time and a V operation is executed for that guard. By preventing a process from taking mutual exclusion to its grave, this mechanism prevents certain potential sources of system deadlock.

Although it is possible to implement both mutual exclusion and synchronization using only guards, low-level synchronization of processes in Conicon's virtual machine is implemented using a different abstraction called a condition queue. The virtual machine instruction create_queue builds a data object representing the queue. The wait instruction, which takes a condition queue as its single argument, blocks the current process on a wait queue until this process is at the front of the queue when another process executes a signal instruction on this queue. If, at the moment a process sends a signal to a condition queue, no processes are waiting on the queue, the signal is ignored. This is the major difference between condition queues and guards. A guard would remember that a signal had been sent and grant access to the next process attempting to wait for that particular signal. Frequently, it is convenient for processes to know whether a sent signal unblocked another process or was ignored. The signal machine instruction fails if no processes are on the signaled condition queue and succeeds otherwise. This gives signaling processes an opportunity to behave differently depending on the consequences of the sent signal.

Occasionally, it is necessary to temporarily release mutually exclusive access to a resource until a particular condition is signaled. If a process were to execute in sequence the exit_guard and wait instructions, then the process would miss a signal that arrived between execution of those two instructions. The special virtual machine instruction exit_guard&wait provides the necessary functionality. This instruction takes a single stack argument representing the condition queue on which to wait, and has the effect of atomically executing exit_guard on the most recently entered guard and wait on the specified condition queue. When some other process signals this condition, the process is removed from the condition queue and, without interruption, caused to reenter the same guard that had been exited by the exit_guard&wait instruction. Reentering a guard is similar to executing the enter_guard instruction except for a minor difference described below. Processes that reenter a guard are automatically given higher priority within the guard's wait queue than processes that execute the enter_guard instruction. This preferential treatment is implemented by providing each guard with two queues, an entry queue and a reentry queue. Access to the guarded resource is offered to the leading process on the reentry queue if there is one before being granted to the first process on the entry queue. A special flag in the status field of a blocked process informs the signaling process that the process blocked on the condition queue is executing an exit_guard&wait instruction. This information is used by the signaling process to determine whether it should enqueue the unblocked process on the ready queue or on a guard's reentry queue.

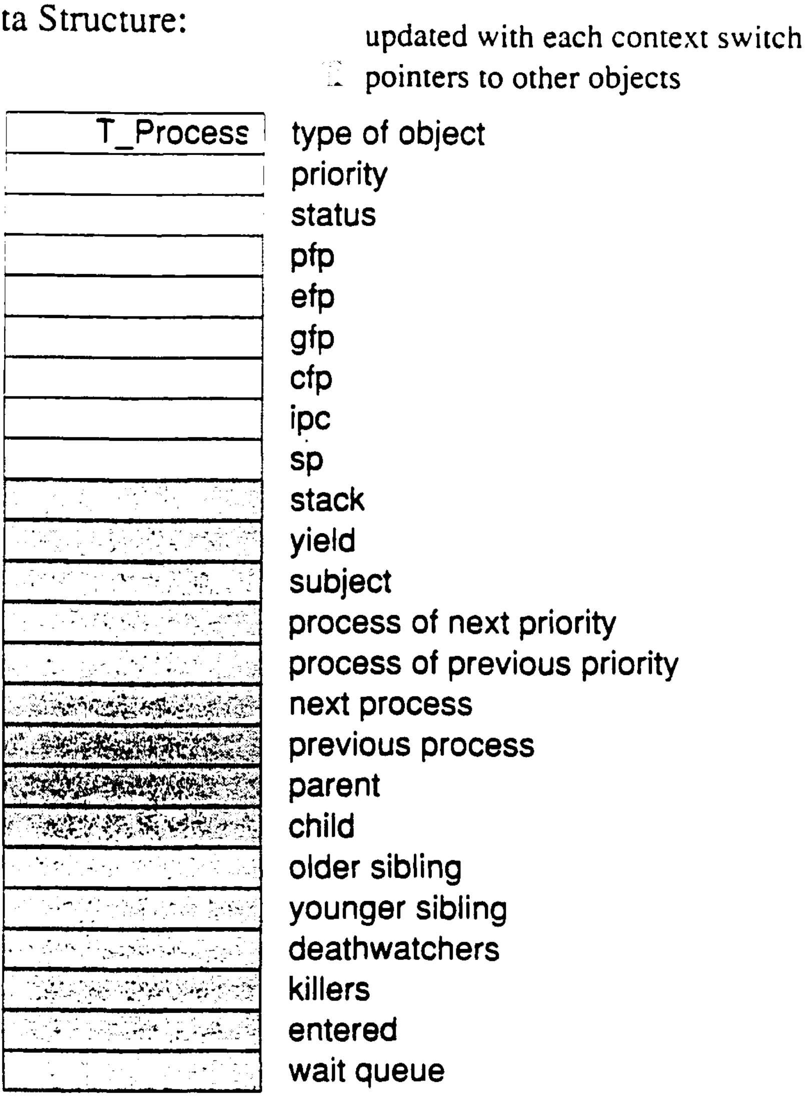
Frequently, it is necessary to kill a guard or condition queue, unblocking any processes on blocked process queues originating in those data structures. The virtual machine instructions kill_guard and kill_queue facilitate this action. Both of these instructions flag their guard or condition queue argument as inactive so that future attempts to manipulate the respective data structures can be treated as errors. Then kill_guard places pointers to the controlling descriptors of its two blocked process queues on the stack and kill_queue places a pointer to the controlling descriptor of its single process queue on the stack. The special instruction unblock, which operates on a single stack argument, awakens the leading process on the queue referenced by the top stack location and leaves the stack pointer unchanged. unblock fails if the queue is empty. Generally, killing a guard or condition queue consists of first executing the kill_guard or kill_queue instruction and then repeatedly executing unblock until it fails on each of the process queues produced by the respective kill instruction.

Another important synchronization capability is the ability to wait for a particular process to die. The deathwatch instruction blocks the current process until the process specified as its single argument is retired by placing the current process on a special queue associated with deathwatch's process argument.

In order to change the priority of a process, a pointer to the process and the integer priority value are passed on the stack to the priority machine instruction. This instruction first sets the priority field within the process data structure. Then, if the process is on a process queue, a pointer to that queue is obtained from the process data structure, and the process is removed from its current position within the queue and inserted into the same queue at the position corresponding to its new priority. As discussed above, both removal from and insertion into a process queue is accomplished in constant time. If the process whose priority has changed is the current process and, as a result of the change, the current process has lower priority than the first process on the ready queue, interrupt is set to True and why_interrupt is set to TimeSlice to force a context switch on the next iteration of the interpreter loop.

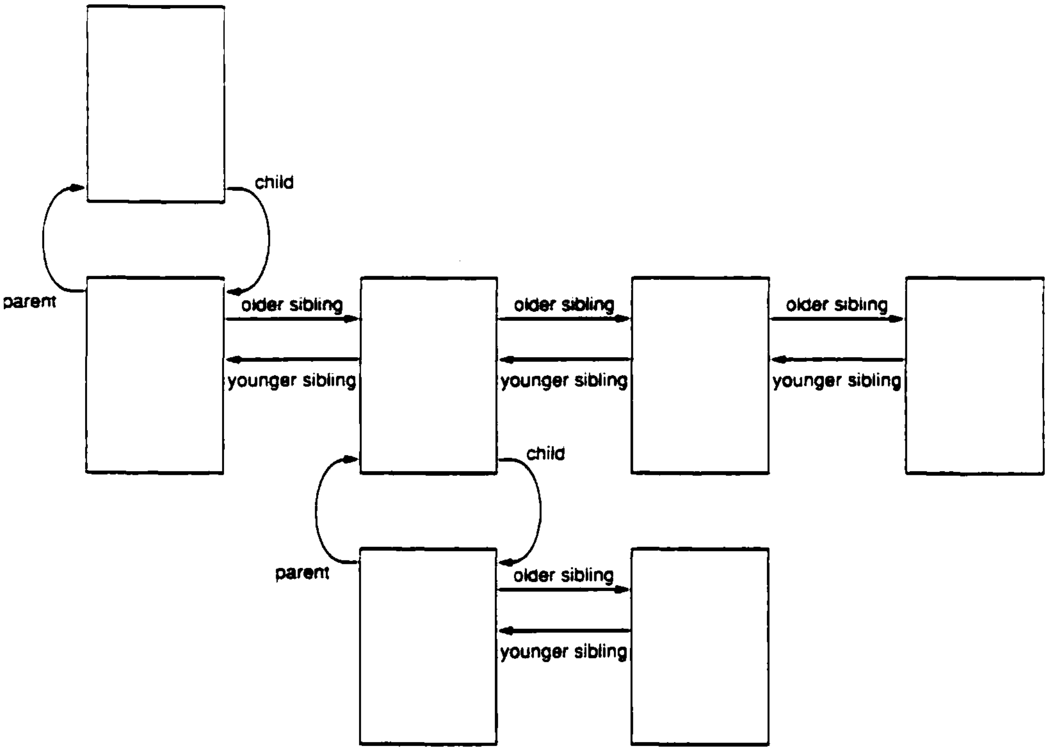
The process data structure must represent a variety of information, as shown in the figure below:

The Process Data Structure:



In this figure, the machine state consists of each of the virtual machine's registers. Most of these have been described above. pfp points to the current procedure frame and cfp points to the current coalternation frame. Coalternation frames are described below. Procedure frames are described in [28]. The stack field points to a large block of memory that serves as the stack for this process. yield points to the standard output stream for this process and subject points to the input stream that represents the current scanning environment. Blocked process queues are threaded and linked through the fields labeled process of next or previous priority and next . r previous process. Processes created for the purpose of evaluating concurrent alternation operators are considered to be child processes of the process that creates them. As described below, when a process is killed, so are all of its descendent processes. All of the children of a particular process are doubly-linked to their parent using the fields labeled parent, child, older sibling, and younger sibling as shown below:

Process Genealogy:



Only the most recently created child process points to its parent. Note that addition of a new child process or removal of any existing child process from this data structure executes in constant time. In the process data structure, the fields labeled deathwatchers and killers point to queues of blocked processes waiting for this process to die. The entered field points to a list of guards to which this process has been granted mutually exclusive access. The field labeled wait queue points to the descriptor that governs the queue of waiting processes on which this process is waiting or is null if this is the current process.

Having outlined the process data structure and introduced the data structures that represent process queues and process genealogies, it is now possible to more completely describe the subroutine that is executed by a retiring process. The subroutine is represented by the following pseudo-code. In this code, italicized statements that refer to particular processes or guards are assumed to fail if the processes or guards do not exist.

```
die:
        # Note: process status has been set to Retired
        mark   L2
L1:     remove first guard from entered queue and execute V operation
        goto   L1
```

```
L2:                      # no more guards on entered queue
        decrement reference count on process yield
        if reference count is zero, close the yield
        mark   L4
L3:     retire child
        goto   L3

L4:                      # no more children to retire
        change process status to Dead
        mark   L6
L5:     unblock first process on killers queue
        goto   L5

L6:                      # no more killers to unblock
        mark   L8
L7:     unblock first process on deathwatchers queue
        goto   L7

L8:                      # no more deathwatchers to unblock
        deactivate
```

Since closing the process's yield requires an unbounded amount of computation, this responsibility is actually implemented by another subroutine. This is described more completely in §6.5. Note that, after all descendents have been killed, the process status is changed to Dead. This prevents future deathwatch or retire instructions from placing new processes on the deathwatchers or killers queues. Instead, when these instructions see that the status of the specified process is already Dead, they simply proceed to the following instruction instead of becoming blocked.

6.3 High-Level Process Manipulation

Programs written in Conicon do not directly invoke most of the synchronization primitives described above. Instead, Conicon programs call built-in functions like advance, write, and kill and make use of the create and concurrent alternation (!) operators. This section describes how the high-level capabilities are implemented in terms of low-level primitives. To simplify the presentation of virtual machine code in this section, high-level machine instructions are described and used. As discussed in §6.2, high-level machine instructions must be replaced in Conicon's virtual machine by sequences of simpler instructions. Consider these high-level instructions to be macro expansions for sequences of simpler instructions.

Process Creation

Conicon's create operator is used to explicitly create a new process. A special virtual machine instruction called create implements this operator inside the interpreter. The Conicon expression:

```
create expr
```

is translated into the following virtual machine code:

```
        goto   L3
L1:
        mark  L2
        code for expr
        coret
        efail
L2:
        cofail
L3:
        create L1
```

In the virtual machine code shown above, create is given an operand of L1. This operand is a pointer to the code that is executed by the new concurrent process. The create instruction allocates and initializes memory for new stack and process data structures. This instruction also opens a new data stream to serve as the yield of the created process. Then it places the newly created process on the ready queue and proceeds to the following virtual machine instruction. As discussed above, create is a high-level macro that actually represents a sequence of simpler instructions.

The expression argument to Conicon's create operator is evaluated in a goal-directed fashion. For each result that the expression produces, this code fragment executes a coret instruction. coret returns a value from a concurrent process by writing the value on top of the stack to the stream representing the yield of the process. The details of writing to a stream are described below in §6.4 and §6.5. Eventually, evaluation of the expression may fail. When this happens, control passes to label L2 and the cofail instruction is executed. cofail kills the current process, closing the stream that represents its yield. Subsequent garbage collection reclaims the space allocated to the stack and process data structures.

Processes are also created by the concurrent alternation operator. The expression:

```
expr1 ! expr2
```

is translated into the following virtual machine code:

```
        goto   L5
L1:
        mark  L2
        code for expr1
        coret
        efail
L2:
        cofail
```

```
L3:
        mark   L4
        code for expr2
        coret
        efail
L4:
        cofail
L5:
        pnull
        coalt  l.1     L3
```

For example, the expression:

```
        write(3 + (4 ! 5))
```

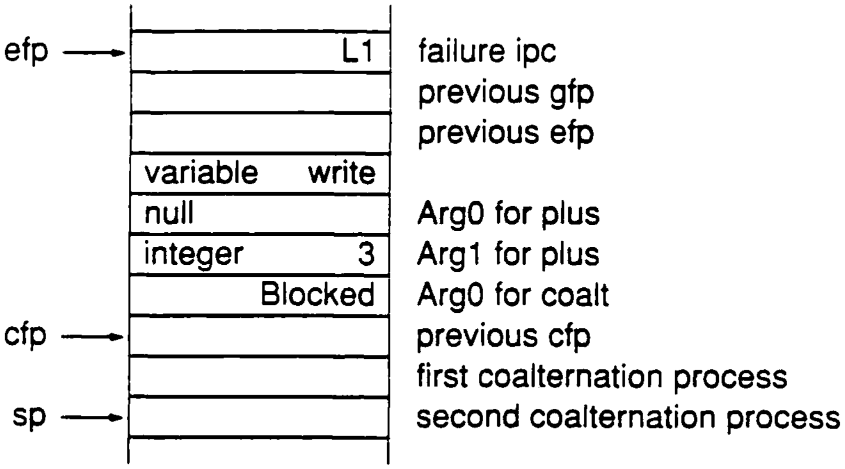
is translated into the virtual machine code:

```
        global write
        pnull
        int    3
        goto   L5
L1:
        mark   L2
        int    4
        coret
        efail
L2:
        cofail

L3:
        mark   L4
        int    5
        coret
        efail
L4:
        cofail
L5:
        pnull
        coalt  L1      L3
        plus
        invoke 1
```

The coalt instruction is similar to create except that it creates two processes, both of which share a single stream as their yield. The shared data stream is given a reference count of 2, indicating that the stream represents the yield of two different processes. When each of these processes executes the cofail instruction, it decrements this reference count. Upon decrementing the reference count to zero the shared stream is closed.

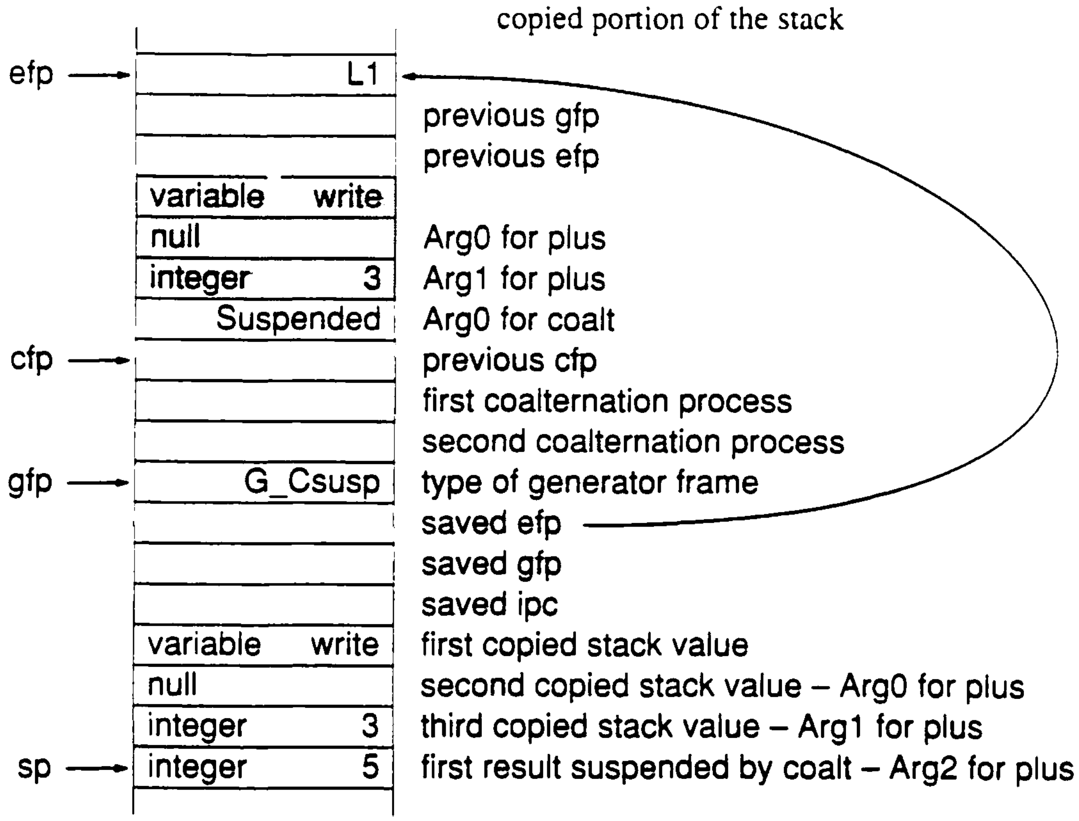
After starting up processes to evaluate each of the expression arguments of concurrent alternation, coalt becomes a generator, producing each of the values it reads from the shared yield of the two processes. coalt blocks the current process until a value is produced by one of the created processes.

Since coalt is a generator, each invocation of coalt causes a small amount of stack growth within which intermediate state information is stored. The portion of the stack dedicated to representation of this information is called a coalternation stack frame. The virtual machine register cfp points to the current or most deeply nested active coalternation stack frame. Each coalternation stack frame includes a pointer to the previous coalternation frame on the stack and pointers to each of the two created processes. In the example above, after the coalt instruction activates the new processes and blocks waiting to read from their shared output stream, the top of the stack contains the following information:



Suppose that the second process produces the value 5 before the first process produces 4. When the coalt instruction is restarted, it suspends the value 5. At that moment, the top stack values are[35]:

[35] Actually, coalt requires a small number of local variables to represent the internal state of the generator. As described in §6.2, these variables occupy locations on the stack. These local variables are omitted from the figure and discussion since they only distract from the ideas that are presented here.



If the coalt instruction is resumed through backtracking, the stack returns to its previous state.

Evaluation of the concurrent alternation operator is aborted if control leaves the enclosing bounded expression. When this happens, the virtual machine executes an unmark instruction. This instruction clears the virtual machine's stack to the height at which the corresponding mark instruction was executed. Range checks on stack pointer values are used to determine when unmark requires destruction of concurrent processes that are evaluating concurrent alternation operators. Notice, for example, in the stack picture shown above, that cfp is between sp and efp. When this condition is detected, the concurrent processes are killed and cfp is set to its previous value. As discussed above, killing of a process requires killing of each of its descendents. Since the number of coalternation stack frames contained within a bounded expression is arbitrarily large and since each coalternation process may have an arbitrary number of descendents, unmark is a high-level instruction that must be replaced by a sequence of simple instructions that loop.

Process Destruction

Processes kill themselves by executing the cofail instruction, and are automatically killed by their parent process whenever control leaves a bounded expression within which concurrent alternation operators are being evaluated as described above or when the parent process itself is killed. Processes are also killed when virtual machine code invokes the built-in kill function. In all of these cases, the low-level retire instruction described above is executed with the process to be killed as its argument.

Other Functions

The priority function allows reassignment of a process priority. After verifying that the types and values of its arguments are consistent with the definition of this function, these arguments are simply passed to the priority machine instruction described above.

The deathwatch function, given a process argument, suspends execution of the current process until the specified process has died. If the process is already dead when deathwatch is invoked, then deathwatch returns immediately. If, however, the specified process is not dead, then the current process is suspended and linked onto a queue of blocked processes that is associated with the process argument of deathwatch. This is accomplished by setting interrupt to True, why_interrupt to BlockProcess, and where_block to point at the deathwatchers field of the specified process. Later, when the specified process dies, all of the processes on this linked list are unblocked. The work described here is actually carried out by the deathwatch machine instruction, which is invoked by this function.

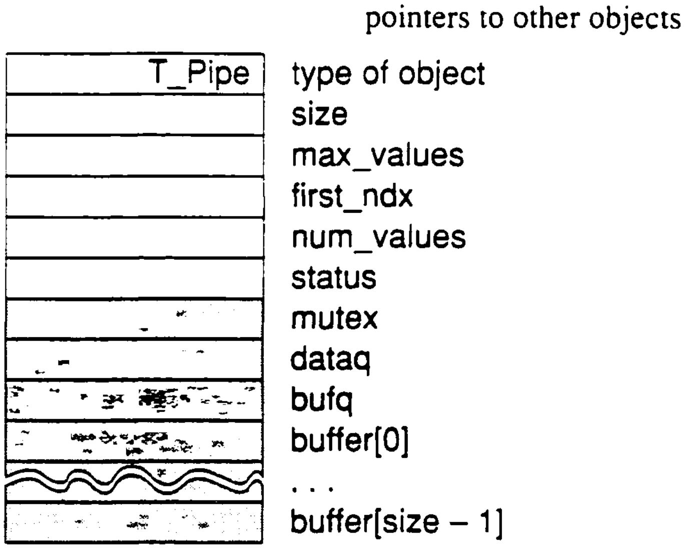
The yield function takes a process argument and returns the stream representing the yield of that process. A field within the process data structure represents the process's yield. Implementation of yield is consequently a simple record field lookup.

6.4 The Internal Pipe Data Type

Within Conicon's run-time system, streams are implemented on top of a lower-level communication mechanism called a pipe. Pipes provide bounded buffers for interprocess communication but do not support backtracking. Writing processes that attempt to place into a bounded buffer more data than the bounded buffer is able to hold are blocked until buffer space becomes available. Likewise, processes that attempt to read from a bounded buffer data values that have not yet been provided by writing processes are blocked until the data arrives. Mutual exclusion implemented by higher level routines guarantees that for a given pipe, no more than one reading process and one writing process are ever attempting to manipulate the pipe's bounded buffer and state variables at a time.

Pipes are represented internally by the data structure shown below.

The Pipe Data Structure:



In this figure, mutex represents a guard that provides mutual exclusion to this shared data structure. dataq and bufq represent condition queues upon which processes waiting for the arrival of new data and for additional buffer space respectively become blocked. size is an integer value representing the number of slots in the bounded buffer. max_values specifies the maximum number of values that can be inserted into the bounded buffer. Initially, size and max_values have the same integer value. However, execution of bound may decrease max_values without changing size. first_ndx represents the offset within buffer at which the next value to be read from the buffer is located. num_values represents the number of values currently held within the bounded buffer. Note that the number of slots in the buffer array depends on size. Values stored within buffer may represent either pointers to larger objects or scalar values such as integers.

Processes attempting to read from a pipe first obtain mutually exclusive access to the pipe and then determine whether a process rendezvous is required. The semantics of read and write pipe operations require a process rendezvous whenever max_values equals zero. Virtual machine code for pipe and stream manipulation functions is revealed in the displays below as it is discussed. To aid readers in assembling the code into a complete implementation, the listings are organized using WEB-style [37] annotations. The code presented throughout the remainder of this chapter represents a simplification of the actual implementation. No distinction is made, for example, between pipes of arbitrary values and pipes of only characters. Also ignored is much of the detail associated with type checking and coercion. And the implementations of read_pipe and write_pipe transmit only single values at a time, even though much better system throughput is achieved by dealing with larger blocks of information at a time. The emphases of this presentation is on concurrency control and synchronization methods. Details have been omitted in order to focus on the important aspects of the implementation. The implementation of read_pipe, which reads a value from its single pipe argument, is outlined below:

```
read_pipe:
        push pipe.mutex onto stack
        enter_guard

L1:     mark   L2
        verify that pipe.max_values is zero and pipe.num_values is zero
        unmark
        <rendezvous-with-writing-process>
        goto   L3

L2:                     # size of bounded buffer is non-zero
        <extract-leading-value-from-bounded-buffer>
L3:     exit_guard      # release mutual exclusion provided by pipe.mutex
```

In the outline above, the two operations enclosed within angle brackets represent macro expansions for virtual machine code that is described elsewhere. Many of the code fragments presented below are labeled according to their origin in this and other code outlines. Whenever the definition of an individual component is not presented entirely in a single display, this is indicated by appending a period and an integer to the end of the component name. Using this convention, the number represents the relative position of a particular section of code with respect to other sections representing the same component. For example, the label *<rendezvous-with-writing-process.3>* accompanies the third section of the component named *rendezvous-with-writing-process.*

If the bounded buffer is not empty when a process attempts to extract a value from it, the process simply obtains the leading value and adjusts the pipe's state variables to indicate that a value has been removed. If the bounded buffer was previously at its full capacity, then a signal is sent to pipe.bufq indicating that new buffer space has become available. If bound sets max_values to zero when num_values is non-zero, the next num_values attempts to read from the pipe obtain the previously buffered values without requiring a process rendezvous. The same signal is sent to pipe.bufq when the last value is removed from the buffer, indicating that a blocked writer may now proceed with its attempt to rendezvous with a reader. Note that a process awakened by this signal is not able to access the pipe data structure until this reading process executes exit_guard. Code representing these actions is shown below:

<extract-leading-value-from-bounded-buffer.1> ≡

```
        mark   L5
        verify that pipe.num_values > 0
        unmark
        shield
        obtain value at pipe.buffer[pipe.first_ndx]

        mark   L4
        verify that pipe.num_values = pipe.max_values or
            (pipe.num_values = 1 and pipe.max_values = 0)
        unmark
        push pipe.bufq onto stack
        signal
```

```
L4:     set pipe.first_ndx to (pipe.first_ndx + 1) % pipe.size
        decrement pipe.num_values
        unshield
        goto   L3
```

If, however, the buffer is empty when a reading process attempts to extract a value from it, the reader blocks itself on the dataq condition queue. This is shown below.

<extract-leading-value-from-bounded-buffer.2> ≡

```
L5:                     # the buffer is empty, wait for data to arrive
        push pipe.dataq onto stack
        exit_guard&wait
                        # either data has been placed into the buffer
                        #  or the bounded buffer has been resized.
                        # in either case, restart the read_pipe function.
        goto   L1
```

The bound function directly manipulates the pipe data structure, setting max_values to its new limit. Before manipulating the pipe, bound obtains mutually exclusive access to the pipe data structure by entering pipe.mutex. If the new buffer size exceeds the previous size, then a new pipe structure is allocated to accommodate the larger buffer and relevant data is copied from the old structure. Relocating the pipe data structure is possible because the only pointer to a particular pipe structure is stored as a field of the stream data structure passed as an argument to bound. If the new limit is smaller than num_values, the data already placed into the buffer is preserved to satisfy subsequent read requests, but writing processes are not allowed to insert additional data into the buffer until num_values is less than max_values. Whenever the new values of max_values and num_values are smaller than size, size is decreased and relevant buffer data is copied to the front of buffer[36]. This allows the next garbage collection pass to reclaim that portion of the original pipe data structure that is no longer needed. Because both reading and writing processes synchronize differently depending on max_values, bound sends signals to the dataq and bufq condition queues whenever the changes it has performed might require readers and writers respectively to revise their strategies for communicating. Note that in the code described above, an awakened reading process considers both possible reasons for it having been unblocked: either new data was placed into the bounded buffer or an invocation of bound required interruption of the reader's current approach to synchronization.

If bound sets max_values to zero, then subsequent transfer of information between writing and reading processes requires a rendezvous[37]. Even though max_values is zero, the buffer array is provided with at least one slot. This slot provides a shared memory location for passing data between writing and reading processes. The code executed by the reader is shown below:

[36] Because garbage collection assumes that memory consists of a contiguous sequence of objects, it is necessary to title the discarded portion of memory with its type (garbage) and size. This permits the garbage collector to find the beginning of the object that follows the garbage.

[37] If, at the moment bound sets max_values to zero, the bounded buffer is not empty, then num_values subsequent read operations extract data from the bounded buffer without requiring a process rendezvous.

<rendezvous-with-writing-process> ≡

```
        shield
        mark  L6
        push pipe.bufq onto stack
        signal
        unmark
        goto   L8
L6:             # signal(pipe.bufq) failed. wait for data to become available
        push pipe.dataq onto stack
        exit_guard&wait

        mark  L7
        verify that buffer size is zero
        unmark
        goto   L8
L7:             # bounded buffer was resized, restart the read operation
        unshield
        goto   L1
L8:     obtain the value from pipe.buffer[0]
        unshield
```

This code cooperates with similar code in the write_pipe function, which is outlined below:

<rendezvous-with-reading-process> ≡

```
        assign value to pipe.buffer[0]
        mark  Lx
        push pipe.dataq onto stack
        signal
        unmark
        goto   Ly
Lx:                     # signal(pipe.dataq) failed, wait for a reader to arrive
        push pipe.bufq onto stack
        exit_guard&wait
        # check to see if the size of the buffer is still zero
        #  if it has changed, then restart write_pipe
        mark  L1
        verify that pipe.max_values is zero
        unmark
Ly:
```

Whenever two processes must rendezvous via the run-time kernel, one process always reaches the rendezvous point before the other. If the reader arrives first, its attempt to signal a waiting writer fails (because the bufq condition queue is empty). When this happens, the reader relinquishes mutual exclusion and blocks itself on the dataq condition. Later, when a writer attempts to rendezvous with this reader, it sends a signal to the dataq condition, awakening the reader. If, however, the writer arrives first, its attempt to signal dataq fails and it blocks itself on bufq, waiting for a signal from a subsequent reader.

Note that writers place the value to be transmitted into buffer[0] before the process rendezvous and readers remove the value from the shared memory location after the process rendezvous. If a second invocation of write_pipe were allowed to overwrite buffer[0] before the reading process from the previous rendezvous had obtained the old value, the integrity of the data stream would be violated. New writers are prevented from overwriting buffer[0] before the old value has been read by the mutex guard. Since a reentering reader process has higher priority on a guard's queue than entering processes, a waiting reader is guaranteed access to the pipe data structure before either a subsequent writer or a process that is executing bound.

The remainder of the write_pipe function is detailed below:

```
write_pipe:
        push pipe.mutex onto the stack
        enter_guard

L1:     mark  L3
        verify that pipe.max_values = 0
        unmark

        # if pipe.num_values ≠ 0, wait for buffer to empty
        mark  L2
        verify that pipe.num_values ≠ 0
        unmark
        push pipe.bufq onto the stack
        exit_guard&wait
        goto  L1

L2:     <rendezvous-with-reading-process>
        goto  L4

L3:                     # size of bounded buffer is non-zero
        mark  L7
        verify that pipe.num_values < pipe.max_values
        unmark

        mark  L6
        verify that pipe.num_values = 0
        unmark
        push pipe.dataq onto stack
        signal

L6:     place value in pipe.buffer[(pipe.first_ndx+pipe.num_values) % pipe.size]
        increment pipe.num_values
        goto  L4

L7:     # wait for buffer space to become available
        push pipe.bufq onto stack
        exit_guard&wait
        goto  L1

L4:     exit_guard     # pipe.mutex
```

Implementation of Hardware Interrupts

All stream communication connects concurrent processes. The implementation shown above describes how concurrent processes running on a single time-multiplexed CPU communicate using pipe primitives. In many situations, however, the source or destination of stream data may be a concurrent process running on external hardware. For example, interrupts received from a disk controller might supply data values for an incoming stream. Likewise, values to be transmitted over a RS-232 connection might be written to a special output stream.

Conicon attempts to provide a high-level, machine independent view of physical devices. This requires that Conicon's run-time system provide low-level interfaces to physical interrupts and hardware devices. Pipes representing communication paths to or from the external environment are implemented using slightly different mechanisms than those described above. When these pipes are created, their status is flagged as either read-only or write-only. Attempts to read from a write-only pipe or to write to a read-only pipe are treated as fatal run-time errors. Outgoing pipes are easily implemented by providing special low-level driver processes that extract values from the pipe and output these values to the appropriate hardware addresses. Incoming pipes are more complicated. Values are inserted into the pipe by a low-level interrupt handler. Special reserved file names are used to name these types of streams. Associated with each of the reserved file names is a collection of low-level driver software including an initialization routine and interrupt handlers. The initial size of the bounded buffer is specified as part of the device driver software and bound is not allowed to modify pipes of this type. Mutual exclusion for these pipes is provided by inhibiting interrupts. Interrupts are disabled throughout execution of the device's interrupt handler and during execution of the special virtual machine instruction get_pipe that reads values from the pipe. The get_pipe instruction is invoked within read_pipe instead of executing the code described above whenever examination of the pipe's status reveals that the pipe represents system input from an external device. If the requested data is available, get_pipe simply obtains the value and modifies the pipe's state variables to indicate that a value has been removed from the bounded buffer. There is no need to signal pipe.bufq. If the requested data is not available, the current process is blocked on pipe.dataq. Each time the interrupt handler inserts a value into the bounded buffer, it signals pipe.dataq to awaken a blocked reader if there is one. The way that overflow of the bounded buffer is managed by an interrupt handler depends on the characteristics of the physical device and semantics of the information carried within the pipe. In some cases, ignoring certain interrupts is an acceptable approach. Ethernet receivers, for example, regularly drop packets that arrive faster than they can be processed. When packets are ignored, higher level protocols retransmit packets that are not acknowledged within a standard timeout period. In other cases, it might be more appropriate to generate a fatal run-time error. During testing and debugging of programs that are not intended to drop interrupts, for example, notifying the programmer of unexpected buffer overflow is necessary so that the programmer can revise the program. Interrupting a running program with a fatal error whenever buffer overflows are detected is one way of providing this notification.

Much of the discussion above assumes that Conicon is running on top of a bare machine. However, it is also possible to implement Conicon on top of an operating system such as UNIX. Application programs running under UNIX are not able to handle low-level physical interrupts. Instead, those details are handled by the operating system which abstracts the low-level details of a particular device as a system file. In the UNIX implementation of Conicon, all files are opened in non-blocking mode. Pipes representing UNIX files are flagged specially and the buffer array is replaced with the number of the file represented by the pipe. Instead of inserting values into the bounded buffer, write_pipe simply writes a value to the specified file. Instead of removing values from a bounded buffer, read_pipe reads from the file. Whenever an attempt to read or write a system file fails because it would require that the UNIX process be blocked, Conicon's run-time system simply blocks the current Conicon process and sets a flag indicating that access to a particular system file has caused a process to block. Inside schedule, which is called at least once every SliceSize iterations of the interpreter loop, this flag is tested to see if file input or output is pending. If so, a non-blocking call is made to the UNIX select function to see if any of the relevant files have become ready for more input or output. For each file that has become ready for input or output, schedule signals the appropriate bufq or dataq condition.

Closing Pipes

When a stream is closed, each of the processes blocked waiting for access to the pipe or for particular conditions to be signaled must be unblocked. Interrupted reading processes terminate in failure. If any writing processes are blocked when the process is killed, Conicon aborts with a fatal error message. The close_pipe function, which is called by close, executes the following sequence of instructions:

```
close_pipe:
        push pipe.mutex onto stack
        enter_guard

        set type of pipe to DeadPipe

        mark   L2
        push pipe.mutex onto stack
        kill_guard
        # Note that kill_guard leaves two process queues on the stack
        copy process queue pointer on top of stack to a temporary variable
                and remove from stack

L1:     unblock
        goto   L1

L2:     mark   L4
        copy saved process queue pointer from temporary variable to top of stack

L3:     unblock
        goto   L3

L4:     mark   L6
        push pipe.dataq onto stack
        kill_queue

L5:     unblock
        goto   L5

L6:     mark   L8
        push pipe.bufq onto the stack
        kill_queue
```

```
L7:     unblock
        goto    L7
L8:
```

In the implementations of read_pipe and write_pipe provided above, it is necessary to add a test immediately following every instruction that might cause the process to become blocked. If. after being blocked and then awakened, a reading process detects that the type of the pipe has been changed to DeadPipe, then it must terminate in failure instead of proceeding. Likewise, if an awakened writer detects that the pipe's type has changed to DeadPipe, a fatal error condition is signaled.

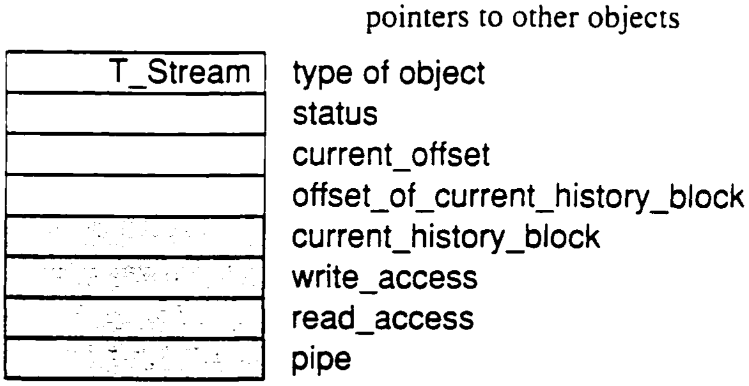
Since processes and streams exist independently, care must be taken to ensure that death of a process does not adversely effect operation of a stream. The semantics of Conicon's stream data type requires that operations that insert or remove values from a stream execute as atomic actions. A process that is killed while attempting to write a value to a stream, for example, must leave the stream state unchanged or must complete all of the steps associated with inserting a new value into a pipe's bounded buffer. Care has been taken in the code presented above to shield portions of code that represent atomic actions. As described in the following section, each value read from a pipe must be inserted into the stream's history buffer. In the code for reading pipes that is presented above, to "obtain" a value from the bounded buffer means to place it directly into the stream history. The high-level functions probe and advance that call read_pipe pass a pointer to the appropriate portion of the stream history so that this can be done in constant time. This is described in greater detail below.

6.5 The Stream Data Type

Streams differ from pipes in two major ways. First, streams support automatic backtracking. Second, each invocation of write, probe, or advance executes atomically with respect to other invocations of these functions on the same stream. This means, for example, that an invocation of probe that requests ten data values obtains ten consecutive values even though concurrent processes may be attempting to read values at the same time. Likewise, an invocation of write with five data values to be written places each of these data values into consecutive locations of the bounded buffer associated with the stream even though concurrent processes may be attempting to write different values to the same stream at the same time. These requirements are relaxed if a reading process is killed before it has obtained all of its requested data or if a writing process is killed before it has transmitted all of its arguments. In that case, as much of the requested transfer as completes prior to destruction of the process executes without interleaved access to the stream by other processes and the remainder of the transfer is aborted.

Stream backtracking is implemented by maintaining a history of all the values read from a stream that are within the current window of backtracking visibility. Atomicity of reading and writing operations is provided by two additional guards called read_access and write_access respectively. Before even examining the stream history, a reading process executes the enter_guard instruction with read_access as its argument. Before processing any of its data arguments, a writing process executes the enter_guard instruction with write_access as its argument. The stream data structure is illustrated below:

The Stream Data Structure:



In this data structure, current_offset represents the number of values that precede the current scanning focus within the stream. The stream history is organized as a linked list of buffers. each buffer holding a fixed number of data values read from the stream. current_history_block points to the buffer that contains the next data to be read from the stream. offset_of_current_history_block specifies how many values precede the contents of the current history block. Together, these three values represent the current focus of scanning within the stream. write_access and read_access point to the guards described above. pipe points to the pipe data structure described in §6.4.

Stream Creation

Streams are created whenever concurrent alternation or create operators are evaluated, type coercions require creation of a stream from a string or list, or when the open function is invoked. Creation of a stream consists of allocating and initializing the data structures associated with each stream that are described above. This includes creation of pipes to provide interprocess communication. Initialization requires, among other things, creation of the guards and condition queues that are used to implement mutual exclusion and synchronization.

Streams that result from coercion of strings or lists are much simpler than streams that connect concurrent processes. Since these streams are read-only, there is no need to provide the write_access guard. Since the entire history of these streams is known at the moment the stream is created, there is no need to create a pipe for interprocess communication. When streams are created as a result of coercion, the stream data structure is created and the entire stream history is allocated and initialized[38]. The current stream focus is set to point at the beginning of the history data structure. Then the end-of-stream flag within the stream's status field is set to prevent reading processes from attempting to extend the history by reading from the nonexistent pipe.

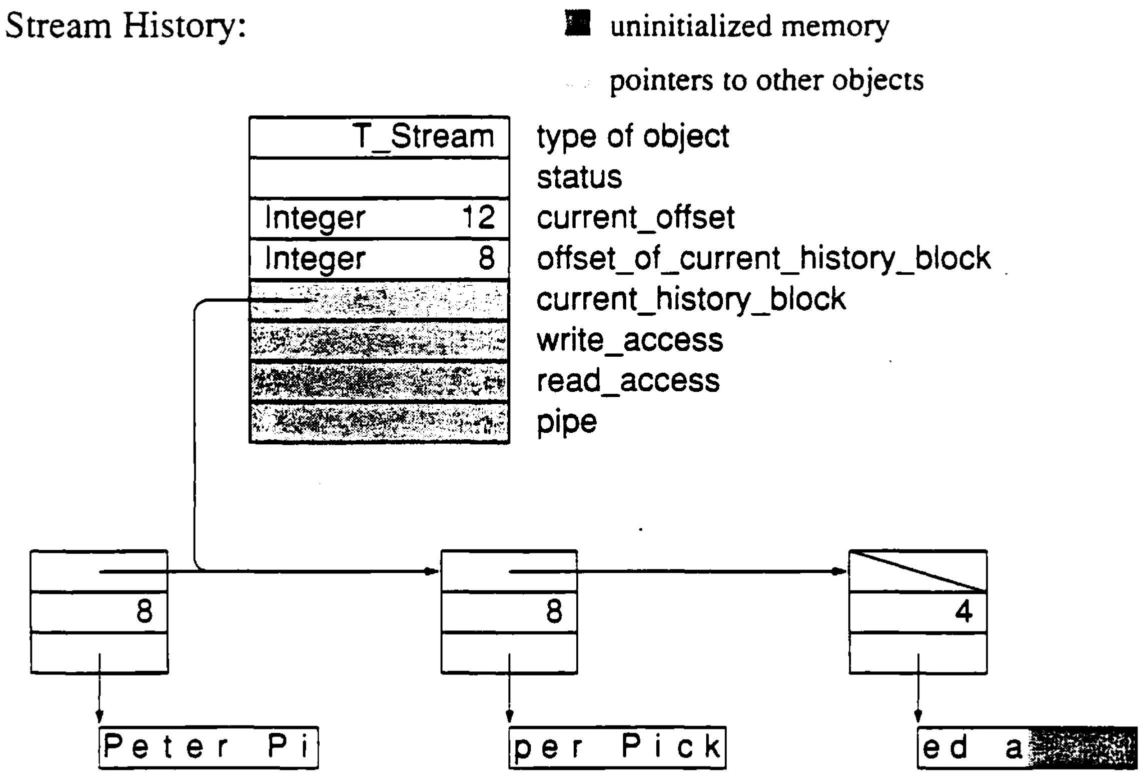
[38] An alternative approach would be to build the stream history using lazy evaluation techniques.

Writing to streams

Given the write_pipe function described above, implementation of the built-in write function is straightforward. write simply executes the enter_guard instruction on write_access and then, for each of its arguments, invokes write_pipe, passing it a pointer to the pipe[39]. After all of the arguments have been written to the pipe, write executes the exit_guard instruction.

Reading from Streams

As mentioned above, each value that is extracted from an internal pipe is inserted directly into the corresponding stream history. The stream history is organized as a linked list of buffers as shown below:



The stream history shown above stores eight characters in each buffer. In the actual implementation, each buffer holds 512 characters. In this example, the scanning focus points at the beginning of the word Picked. Each link of the history list maintains a count of the number of valid characters stored within its buffer. The last buffer in this figure contains only four valid characters. The other characters have not yet been read from the pipe. Note that, unless some other variable points to the first buffer in the history list, the next invocation of the garbage collector will reclaim the memory allocated to the first link and its buffer.

[39] The implementation of bound described above depends on the invariant that no more than one pointer to each pipe data structure exists. In order not to violate this constraint, write_pipe takes a stream argument instead of requiring a second pointer to the stream's pipe. write_pipe finds the pipe by examining the pipe field of its stream argument.

Both probe and advance execute enter_guard on the read_access guard. Then both functions search the stream history for the requested data. If the desired data is not currently in the stream history, read_pipe is repeatedly called with a pointer to the array slot within the relevant history buffer where the value produced by read_pipe is to be stored until all of the requested data has been copied into the stream history. If a new history buffer must be allocated, this is done by probe or advance before calling read_pipe. probe executes the exit_guard instruction before returning so that subsequent readers can enter.

Besides returning the requested string or list, advance has the side effect of shifting the current focus within its stream argument forward. Before modifying the stream data structure to reflect changes in its scanning focus, the old offset and a pointer to the previously current history block are saved in local variables by advance. Then advance exits read_access and suspends itself, returning the requested data. If backtracking is required, advance is resumed. When resumed, advance enters read_access, restores the scanning focus within the stream data structure by copying the values previously stored in local variables, and exits the read_access guard. Note that the suspended advance function keeps in local variables a pointer to the history block that represents the previous scanning focus within the stream. This pointer is required to implement backtracking and to prevent garbage collection of the history buffers that hold the data between the old and new focus.

The Close Function

Most of the work required to close a stream is implemented by the close_pipe function. The close function is outlined below:

```
close:
        set type of stream to ClosedStream
        call close_pipe with stream.pipe as its argument

        mark  L2
        push stream.write_access onto stack
        kill_guard
        copy process queue pointer on top of stack to a temporary variable
                and remove from stack

L1:     unblock
        goto   L1

L2:     mark  L4
        copy saved process queue pointer from temporary variable to top of stack

L3:     unblock
        goto   L3

L4:
```

Note that only the write_access guard is killed. This is because reading processes are still allowed to access the stream history even after the stream has been closed. The read_access guard is required to protect the history from being corrupted by multiple processes attempting to manipulate it simultaneously. It is important that the write, advance, and probe functions cooperate with close. advance and probe must fail whenever read_pipe fails. write must abort with a fatal run-time error whenever an attempt to write to a closed stream is detected.

Immediately after entering the write_access guard, write must verify that the stream's type has not changed to ClosedStream. If it has, then write likewise must abort with a fatal run-time error. Note that code to handle these special circumstances must be added to the implementations of write, advance and probe which were described above.

6.6 Timeouts and Delays

In Conicon, both timeouts and delays are implemented using the same high-level mechanism. Timed delays are implemented by calling the sleep function. Timeouts are implemented by executing sleep in one process while other computation is carried out in another concurrent process. To facilitate communication between the two processes, the processes are typically executed as two arms of a concurrent alternation operator.

sleep is provided as a library routine of icode that communicates with a special daemon process. This daemon receives messages from two streams. One of the streams, called sleep_requests, delivers records from processes that want to sleep. The other stream delivers interrupts from a hardware timer. These interrupts are assumed to arrive at regular timed intervals. An implementation of the sleep daemon in high-level Conicon is presented below. In this code, time_interrupt is a procedure that suspends &null whenever a timer interrupt arrives. sleep_request is a procedure that suspends a sleep_descriptor record each time a process executes sleep. This record provides fields representing the time at which the process wants to be awakened, a pointer to a private guard, two link fields and one bookkeeping field for use by the internal sort routines. These records are maintained by the sleep daemon using a leftist tree data structure [38, p. 150]. In the code below, two internal procedures manipulate the heap data structure. heap_insert takes a new record as its argument and inserts this record into the appropriate location within the heap. heap_extract removes the minimum element from the heap and reorganizes the remaining elements. Magnitude comparisons for this heap are based on the value of the time field. The heap_front function returns a pointer to the minimum element in the heap without modifying the heap data structure. heap_front returns &null if the heap is empty. If a sleeping process is retired, that process is automatically removed from whatever blocked queues it is on. When the record describing the retired process reaches the front of the heap, no process is awakened because the wait queue associated with its private guard is empty[40].

[40] Alternatively, it would be possible to remove records for retired processes from the leftist tree when the process is retired. The algorithm described here has the advantage of not complicating the work carried out by the die procedure.

```
record sleep_descriptor(time, guard, left, right, height)

procedure sleep_daemon()
  local first, current_time

  current_time := 0
  every x := time_interrupt() ! sleep_request() do
    if /x then {              # received an interrupt
      current_time +:= 50   # assume 20 interrupts per second
      while current_time >= (\heap_front()).time do {
        first := extract_heap()
        V(first.guard)
        }
    else {                     # received a sleep request
      x.time +:= current_time
      heap_insert(x)
      }
end
```

time_interrupt and sleep_request simply read values from the timer interrupt stream and from the dedicated sleep_requests stream respectively. The invocation of V shown above is simply a Conicon notation for the low-level machine instruction with the same name.

Processes that want to sleep for a period of time execute the following code:

```
procedure sleep(delay)
  local sd

  sd := sleep_descriptor(delay, create_sem(0))
  write(sleep_requests, sd)
  P(sd.guard)
end
```

In this code, a private guard with an initial count of 0 is created and sent as part of the sleep_descriptor record to the sleep daemon. After sending the record containing the guard, the sleeping process waits for the sleep daemon to execute a V operation on that guard.

Chapter 7: Garbage Collection in Real Time

The high-level nature of Conicon requires automatic garbage collection of memory allocated to strings and linked data structures. In Conicon, a string is simply a pointer to an array of characters and an integer that represents the string's length. Special garbage collection techniques are required to deal with string data for several related reasons:

- Multiple pointers to the same characters within a character array are allowed and encouraged.
- All possible character values are legitimate as data so it is not possible to "mark" a string by overwriting it with a reserved character.
- A character is generally much smaller than a pointer so it is not possible to overwrite a single character value with a forwarding pointer to a new location for a particular string.

Special garbage collection techniques are also motivated by the needs of real-time systems. The delay, imposed at arbitrary times during a program's execution, of a traditional stop-and-wait garbage collector violates the requirement that execution of each virtual machine instruction execute in constant time. The garbage collection algorithm presented in this chapter is real-time in the sense that the time required for allocation of each basic unit of memory is bounded by a constant.

Introduction

The string type provides an abstraction for a descriptor containing two fields: a character pointer and a length, plus the data referenced by the character pointer. The actual string data consists of as many consecutive characters as are specified by the length field starting with the character referenced by the character pointer field of the descriptor. The two string operations directly relevant to garbage collection are substring selection and catenation. String catenation creates a new string representing the sequence of characters in its first string argument followed by the sequence of characters in its second string argument. Substring selection creates a new string that is a substring of its string argument. Both of these operations are described in greater detail below.

In typical implementations[41] of the string data type, objects are allocated from two distinct regions of memory: a string region and a block region. Each of these regions is initially empty. Within each region, allocation requests are satisfied from low to high addresses. Each request returns a pointer to the next available memory within the region and increments the internal pointer representing the boundary between allocated and available memory by the size of the request. In the block region, blocks of memory representing records, lists, and a variety of other objects are allocated. Within the string region, only string data is allocated.

String catenation simply allocates enough memory in the string region to store the new string and copies the string data from each of its two arguments. Icon's implementation of catenation uses a heuristic to reduce string region memory requirements. Whenever its first argument extends to the end of the currently allocated string region, the catenate procedure allocates only

[41] This description is derived from the Icon implementation [28], which was based on the implementation of XPL [39]. Related work is described in Reference [40].

enough memory to copy the second argument into the string memory that immediately follows the first argument's string data. This results in sharing of string data between the resulting string and the first argument to the catenate operation. This heuristic has been measured to reduce the size of the string region by approximately 5% over a wide range of programs.

Substring selection simply builds a new string descriptor and sets the character pointer and length fields appropriately. No new memory from the string region is allocated. The resulting string shares string data with the string argument of this operation.

A variety of alternative data structures might be considered for representing strings. This particular solution has been chosen because it is conceptually simple, is reasonably compact, and interfaces easily to the operating system and to the string support routines of most machine architectures. For example, to write a string in a C implementation of an interpreter, simply pass the address of the string and its length to fwrite. Many architectures also provide machine instructions for copying and comparing string data. Note also that computation of a string's length is a simple constant-time operation.

Garbage collection of strings is slightly more complicated than traditional garbage collection. In the implementation of Icon [28], garbage collection consists of a traditional mark and relocate phase, and a string collection phase. During the first phase, an array of pointers to those string descriptors that are marked is constructed. The string collection phase sorts this array, ordering descriptors according to the magnitude of the character pointers within each string descriptor. By examining string descriptors one at a time in sorted order, it is easy to recognize regions of text that are shared by multiple string descriptors. Holes of unaccessed string data are compressed out of the string region to terminate this phase of garbage collection.

The remainder of this chapter describes how string and linked data types are implemented so as to guarantee that each allocation request (and the garbage collection that might accompany it), executes in time proportional to the size of the allocation. The algorithm is based on a garbage collection strategy originally proposed by Baker [42] in which two regions alternate as regions of allocation and garbage collection respectively.

7.1 Garbage Collection in Real Time

Baker's method of garbage collection distributes the work of garbage collection over many invocations of the storage allocator. In this section, his basic algorithm is outlined. In the sections that follow, extensions to the algorithm are proposed that permit the garbage collection of differently-sized objects and strings.

The Baker algorithm uses two regions for allocation, a *to* space and a *from* space. While allocation is taking place in the *to* space, garbage collection is performed in the *from* space. Non-garbage in *from* space is incrementally relocated into *to* space. Calculations performed at run time guarantee that garbage collection of *from* space completes before *to* space exhausts its free pool. When there is no longer enough memory available in *to* space to satisfy an allocation request, the names given to *to* space and *from* space are reversed. Baker provides algebraic justification for his claim that the cost of each allocation is bounded by a small constant.

Memory is allocated starting at the end of *to* space. Each allocation request is satisfied by decrementing the value of an internal pointer representing the end of available memory in *to* space by the size of the allocation and returning its new value. Although similar in concept, allocation is slightly more complicated than in systems not requiring real-time response. First, in order not to expand storage regions unless the behavior of a particular program requires additional storage, it is necessary[42] to verify at the point of each allocation that the currently allocated data need no more space than the current size of the allocation region to collect garbage. Second, if garbage collection is active, an amount of work proportional to the size of the allocation must be dedicated to garbage collecting *from* space.

Garbage collection begins when an allocation request either exceeds the amount of free memory in *to* space or would necessitate needless expansion of the regions in order to collect garbage. When this happens, all objects referenced by all tended descriptors are relocated into *to* space. This bootstraps the garbage collection process. By definition, tended descriptors are the only pointers from the interpreter into data space. Data that can not be reached by following some path of pointers originating at a tended descriptor is garbage, and its space can be reclaimed to satisfy future allocation requests. All objects are titled in their first field with an integer representing the type of the object[43]. Because multiple pointers to the same object are common, every time an object is relocated, the title of the old object is overwritten with a special mark and the second field of the object is set as a forwarding address to its new location. The algorithm proceeds by repeatedly relocating objects pointed to by objects that were just relocated themselves. This process, called scanning, results in a breadth-first mark and relocation of accessible memory. Since only accessible objects are relocated into *to* space, holes of unaccessed memory are automatically compressed out of the allocation region. After all pointers of all relocated objects have been followed and relocated, garbage collection is done. The old storage region is reclaimed to serve as a future allocation region.

In the Baker algorithm, each allocation of n words is accompanied by the scanning of $n \times K$ words where K is set at compile time. Since it is conceivable that all memory in *from* space needs to be relocated into *to* space, *to* space must be at least $(1+1/K)$ times as big as the amount of memory allocated in *from* space. At the moment a new garbage collection pass begins, a check is made to verify that *to* space is sufficiently large to collect the garbage in *from* space. If it is not large enough, *to* space is expanded before garbage collection proceeds.

During the time that garbage collection is taking place, many data structures are in a kind of limbo. For example, part of a linked list may reside in *to* space and part in *from* space. This apparent inconsistency is resolved by permitting the run-time system to see only that portion of a linked data structure that resides in *to* space. As mentioned above, all objects referenced by tended descriptors are relocated at the moment garbage collection begins. At any given time, the system can examine only objects referenced by the system's tended descriptors. Each assignment to a tended descriptor is preceded by a test to see if the value being assigned represents an object in *from* space, and relocation of the object if it does. This bookkeeping work is one of the major costs of distributing garbage collection over several invocations of the garbage collector. It results in increased code size and slower execution.

One minor point about the layout of *to* space is that objects relocated into *to* space are kept separate from objects newly allocated in *to* space. Since all fields of newly allocated objects are assigned values only by way of tended descriptors, it is not necessary to scan these objects. In order to keep new objects separate from relocated objects, new objects are allocated starting

[42] As described by Baker, this check is not made with each allocation, but instead at the start of each garbage collection pass. Unfortunately, that approach typically requires that storage regions expand with each pass of the garbage collector, which can quickly consume all available memory.

[43] Actually, in the Baker algorithm, which deals only with dotted pairs, objects are not titled and range checks on the value of the CAR field determine that an object has been relocated into *to* space. Conceptually though, Baker's approach is analogous to the description provided above.

from the rear of *to* space and relocated objects are allocated starting from the front. The pointer relocated points to the boundary between objects relocated into *to* space and free memory. The pointer scanned separates relocated objects that have been scanned from those waiting to be scanned. As objects are scanned, the objects they point to are copied into *to* space, incrementing the value of relocated by the size of the copied objects, and scanned is set to point to the neighboring object. When scanned catches up to relocated, garbage collection of linked data structures is complete.

7.2 Garbage Collection of Typed Objects with Different Sizes

Baker's work was somewhat simplified by the fact that his target language was LISP, where each data object is the same: a two-element cell containing a CAR and a CDR. Both the algorithm and the analysis are cluttered by additional detail when storage management must deal with a variety of data types and sizes. In order to satisfy the more general needs of Conicon, strategies for dealing with linked objects of different sizes and strings have been developed.

This chapter presents a C implementation of the Baker algorithm in its enhanced form. The code is presented using the same conventions described in the previous chapter. The implementation is comprised of the components listed below:

<global-declarations>
<alloc>
<scan>
<next_block>
<copy>
<sift_string>
<swap_spaces>

As mentioned above, the modified Baker algorithm is complicated by the greater complexity of the languages it is designed to deal with. For example, all linked structures in a Lisp system are constructed out of dotted pairs. However, Conicon provides a wider variety of building blocks out of which linked data structures might be constructed. For example, Conicon has a list type with primitives to provide doubly-ended queue access to the elements of the list, an associative table where items are indexed by keys, which can be objects of any other Icon type, and user-defined records for which the user specifies the number of fields in each record. This complicates garbage collection because the garbage collector must first determine the type and size of an object before it can do anything else. Real-time response is also affected by having a variety of differently sized data objects to garbage collect because the granularity of garbage collection is increased. Since it is difficult to relocate an individual object only partially[44], a lower bound on the time required for allocation depends on the size of the largest object referenced by a program.

The Baker algorithm scans K words for each word allocated. This algorithm scans and copies a combined total of K words for each word allocated. It was not necessary to make this

[44] As described below, strings are relocated incrementally. Since the string type is applicative (i.e. string data is never modified), pointers to string data are simply not adjusted until after the entire string region into which the pointer points has been copied. A similar strategy could be used to incrementally relocate large objects. The storage cost of this would be an extra word in each object that points to the old version of the object. Updating a value within the object would update both of its copies. Values would be read from the old copy until relocation of the object is complete.

distinction with LISP because the scanning of each word could result in, at worst, relocation of one dotted pair. However, in a mixed-type implementation, the object that might need to be relocated as a result of scanning one word typically ranges in size from two words to ten for built-in types, and may be even larger for a user-defined record type with a large number of fields. A state variable named scan_balance represents the degree to which allocation is ahead of garbage collection. Allocation of memory increments scan_balance, while garbage collection activities decrement it. Conceptually, the storage allocator pays for the garbage collection required to satisfy its request for memory in terms of scan_balance points. An allocation of n bytes costs the storage allocator $n \times K$ scan_balance points. Below are declarations of several of the global variables used by the garbage collector.

```
<global-declarations.1> ≡

/* by how much is allocation ahead of garbage collection */
int scan_balance;

char *new,          /* points to most recently allocated block in to-space */
  *scanned,         /* marks the division between scanned and */
                    /*  unscanned objects */
  *relocated;       /* marks the division between relocated objects and */
                    /*  free memory */

int scan_desc_cnt;  /* counts the number of contiguous descriptors */
                    /*  to be scanned */

int cur_size,       /* how big is to-space */
  next_size;        /* size of to-space for next pass of garbage collector */

int gc_active;      /* is garbage collection active? */

/* Ceiling(x)  - round float up to integer */
#define Ceiling(x)    ((int) (((x) > (int) (x))? ((x) + 1): (x)))
```

All storage requests are issued to the alloc function. alloc performs an amount of garbage collection proportional to the size of the allocation request, and increments next_size to reflect the increased amount of memory that the next garbage collection pass may need to relocate. Before returning a pointer to the newly allocated memory, alloc makes sure that there is enough memory in the current *to* space to satisfy the request. If there is not, swap_spaces is called to reverse the roles of *to* and *from* spaces.

```
<alloc> ≡
char *alloc(n)
     int n;
{
  if (gc_active) {
    scan_balance += n * K;
    scan();
    }
  next_size += n + Ceiling(2.0 * n / K);
```

```
    if (!gc_active && next_size > cur_size)
      swap_spaces(n);

    new -= n;
    return new;
  }
```

After exchanging the roles of *to* and *from* spaces, swap_spaces bootstraps garbage collection by initializing state variables and relocating all objects referenced by tended descriptors into *to* space. swap_spaces expands the new *to* space before proceeding if this is required in order to perform garbage collection.

So that the garbage collector can determine the types of objects that it is garbage collecting, every object is titled in its first field with an enumeration of all built-in types. Those types with sizes that cannot be determined by type alone accompany the type enumeration with a field specifying the size of the object. All objects recognized by the garbage collector are declared as possible field values of the union object, as shown below:

```
<global-declarations.4> ≡
union object {
  struct {                    /* A generic structure can overlay all others. */
    int title;                /* specifies the type of the object */
    int num_desc;             /* number of descriptors in the object */
                              /*  if its size is variable */
    } generic;
  struct {                    /* overwrite copied objects with this */
    int title;                /* Relocated */
    union object *ptr;        /* points to relocated object */
    } forward;
  struct {
    int title;                /* Integer */
    int value;                /* integer value */
    } integer;
  struct {
    int title;                /* Record */
    int num_desc;             /* the number of descriptors in this object */
    Descriptor data[1];       /* more descriptors are appended */
                              /*  to end of this array as needed */
    } record;
  struct str_obj string;
  ...
  };
```

Data objects are referenced in this system only by means of descriptors. All global and local variables, and intermediate results stored on the stack are, for example, descriptors. Record objects shown above also contain descriptors which may point to other objects. Below is the declaration for the programmer-defined Descriptor type.

```
<global-declarations.3> ≡
typedef struct descriptor {
  int title;                        /* holds type of object */
  union object *ptr;                /* points to the object */
  } Descriptor;
```

The scan function defined below makes use of three macros. DescOffset takes as an argument an integer representing the type of an object and returns the byte offset of the first descriptor from the beginning of the object. If objects of a particular type contain no descriptors. DescOffset simply returns the size of the object. NumDesc similarly expects a type argument and produces the number of descriptors in objects of that type. If objects of a particular type have a variable number of descriptors (such as objects of type Record). NumDesc returns the value -1. Both DescOffset and NumDesc are implemented as array lookups. The macro InFromSpace expects a descriptor as its argument and returns a non-zero value if that descriptor points to an object in *from* space. InFromSpace is implemented as a range check on pointer values.

scan examines objects already copied into *to* space and relocates objects referenced by these into *to* space. scan iterates until scan_balance is less than or equal to zero.

```
<scan> ≡
scan()
{
  int type;
  register Descriptor *dp;

  while (scan_balance > 0) {
    <initialize-string-regions>
    if (scan_desc_cnt > 0) {
      scan_desc_cnt--;
      dp = (Descriptor *) scanned;
      if (InFromSpace(*dp))
        dp->ptr = copy(dp->ptr);

      scan_balance -= sizeof(Descriptor);
      dp++;
      scanned = (char *) dp;
      }
    else if (scanned < relocated) {
      type = ((union object *) scanned)->generic.title;
      if ((scan_desc_cnt = NumDesc(type)) < 0)
        scan_desc_cnt = ((union object *) scanned)->generic.num_desc;

      scanned += DescOffset(type);
      if (type != String)
        scan_balance -= DescOffset(type);
      else
        <decrement-scan_balance-for-string>
      }
    <scan-string-regions>
    else {                  /* scanning is done */
      gc_active = False;
      return;
      }
    }
}
```

It is important that every operation contributing to garbage collection decrement scan_balance by an amount proportional to the work performed by that operation. Scanning of a descriptor decrements scan_balance by the size of the descriptor. Relocation of an object into *to* space decrements scan_balance by the size of the object[45]. The copy function, shown below, returns a pointer to a copy, residing in *to* space, of the object whose address is supplied as its argument. This function makes the copy if it does not already exist.

```
<copy> ≡
/* copy - copy *op into to space */
union object *copy(op)
     union object *op;
{
  int num_desc;
  int num_bytes;
  union object *rp;

  if (op == NULL)
    return NULL;

  if (op->generic.title == Relocated)
    return op->forward.ptr; /* object has already been copied */

  if ((num_desc = NumDesc(op->generic.title)) < 0)
    num_desc = op->generic.num_desc;
  num_bytes = DescOffset(op->generic.title) + num_desc * sizeof(Descriptor);

  movmem((char *) op, relocated, num_bytes);
  rp = (union object *) relocated;
  relocated += num_bytes;
  scan_balance -= num_bytes;
  next_size += num_bytes + Ceiling(2.0 * num_bytes / K);

  <set-up-string-block-for-scanning>
```

[45] Note that the *K* of the revised algorithm is half as potent as the *K* of the Baker algorithm. As a result, much of the analysis presented below in this chapter disagrees with Baker's figures by a factor of two.

```
        /* leave a forwarding address with the old object */
        op->forward.title = Relocated;
        op->forward.ptr = rp;

        return rp;
    }
```

The garbage collector can suspend itself in the middle of scanning, but not relocating, an object. Whenever a certain allocation request results in excessive garbage collection (because of the granularity of garbage collection operations), a subsequent invocation of the garbage collector performs less work than normal because the value of scan_balance persists between invocations.

7.3 Garbage Collection of Strings

To eliminate synchronization problems that would result from attempting to apply the real-time garbage collection algorithm to several distinct allocation regions, all data is allocated from the same region. This differs from traditional implementations of strings in which strings are allocated from a special region.

As string data is allocated, it is grouped into common regions that are allocated in increments of 256 characters. String objects contain pointers both to the string data and to the start of the region that contains the data. A declaration of the string object is shown below:

```
<global-declarations.2> ≡
struct str_obj {
  int title;                        /* String */
  char *start;                      /* whereabouts of string data */
  int len;                          /* length of string */
  union {
    struct str_obj *next;           /* forward link during garbage collection */
    struct str_region *region;      /* otherwise, points to string region */
    } u;
  struct str_obj *prev;             /* backward link during garbage collection */
                                    /* otherwise, field is NULL */
};
```

The string allocation and garbage collection routines attempt to pack multiple strings into a common region whenever this is convenient. By never allocating any more contiguous 256-character blocks of string data than are necessary to satisfy a particular allocation request, the allocator ensures that the data allocated to a string never exceeds its required length by more than 255 characters. By remembering the most recent allocation of a string region and the amount of extra space available in that region, the allocator is frequently able to squeeze new allocation requests into an existing string region.

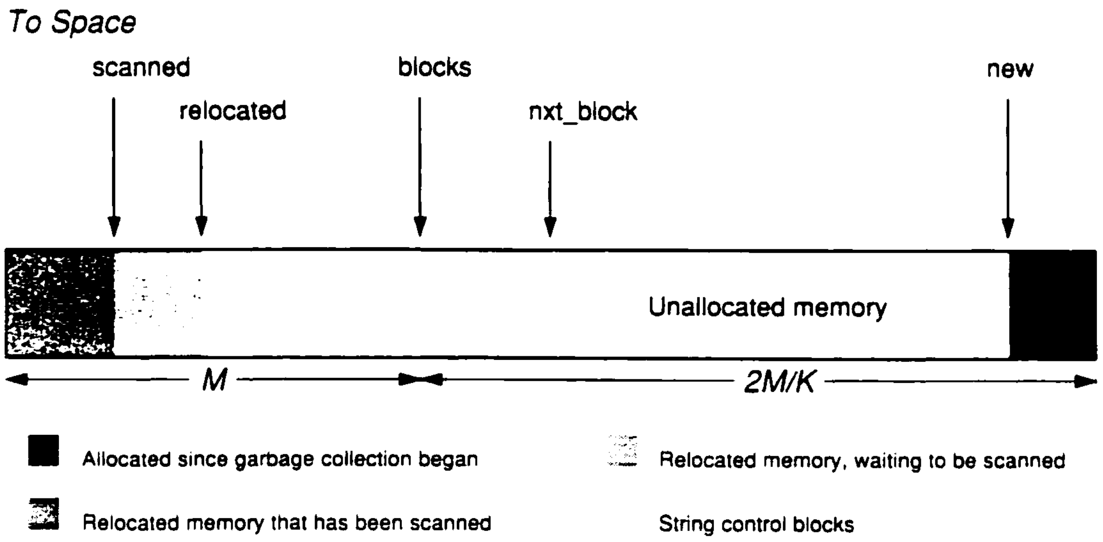
Each string region is prefaced with a small header that describes the region. Within the header are fields representing the length of the region and a mark flag:

```
<global-declarations.5> ≡
#define SubregionSize      256

struct str_region {
  union {
    int len;                     /* if unmarked, length of region */
    struct control_block *cbp;   /* if marked, points to control block */
    } u;
  char mark;                     /* zero means unmarked */
  char data[1];                  /* size of array is determined by need */
  };
```

During garbage collection, marking of a string region consists of flagging the mark field, allocating and initializing a string region control block, which provides temporary work space for the garbage collector, and overwriting the length field with a pointer to the region control block.

Region control blocks are allocated from the same region in *to* space as new data. However, these blocks have a limited lifetime. Since they need not persist after garbage collection terminates, they are positioned in *to* space such that after garbage collection terminates, they may be overwritten to satisfy further allocation requests. This allows the storage allocator to run for longer periods of time between passes of the garbage collector.

The layout of *to* space is outlined below. In this figure, the constant *M* represents the maximum amount of memory that might need to be relocated out of *from* space. This is simply the total amount of memory that had been allocated at the time this pass of the garbage collector began.


To Space
scanned
relocated
blocks
nxt_block
new
Unallocated memory
M
2M/K
Allocated since garbage collection began
Relocated memory, waiting to be scanned
Relocated memory that has been scanned
String control blocks

Region control blocks divide string regions into subregions of 256 characters each. As strings are marked, they are grouped together on doubly-linked lists according to the subregion into which they point. While on these lists, it is not necessary for the string to retain pointers to the string region, so the field is overwritten with a pointer to the next object on the list. An extra field is appended to the end of each string to serve as a reverse pointer for the doubly-linked list.

The storage overhead imposed by the real-time garbage collector on each string is therefore twice the size of a pointer value[46]. Declarations for the control block and subregion structures are shown below. Note that, because they are comprised of the same data types, a subregion structure can overlay a string object.

```
<global-declarations.6>
struct subregion {
  int title;                        /* for alignment with struct str_obj only */
  char *first;                      /* first accessible data in subregion */
  int len;                          /* distance of last accessible data */
                                    /*  from first */

  struct str_obj *next, *prev;
  };

struct control_block {
  struct str_region *str_data;
  int num_segments;                 /* how many subregions in this region? */
  struct subregion init;            /* head for list of strings that are */
                                    /*  marked before control block is */
                                    /*  completely initialized */
  struct subregion probes[1];       /* as many probes as are required to */
                                    /*  process a string region are */
                                    /*  appended to the end of this array */
  };
```

Strings are stored on these lists until their string data has been relocated. After the string data has been copied into *to* space, strings are unlinked from these lists and their pointers are adjusted to point to the copied string data.

Subregions do not, in general, align with the first character in the string region. Instead, they are offset from the beginning of the string region by the variable amount probe_offset. The first subregion is responsible for all strings starting within probe_offset of the region's first character. The second subregion controls strings starting within the next 256 characters, and so on.

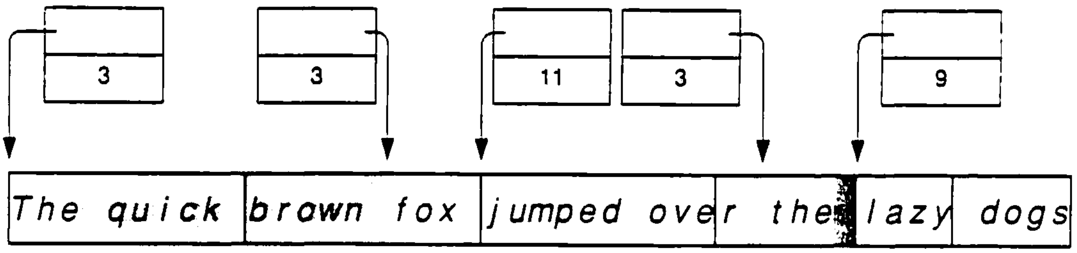
It is possible to modify an existing string so that it points to different data than previously. Whenever a string already on a list of strings pointing to a particular subregion is modified in this manner, that string is removed from the list associated with its old string region. The reason these lists are doubly linked is so that strings can be removed in constant time.

It is also possible for a newly created string to point at a string region that resides in *from* space. This occurs when a new string is created by subscripting an existing string. Whenever this happens, the newly created string is linked onto the appropriate list of strings waiting for their string data to be copied into *to* space so that their pointer values can be updated. Because this action increases the work of the garbage collector, a small amount of garbage collection effort is charged to the string subscripting operation whenever the source string points into *from* space.

The traditional algorithm for string garbage collection described in a previous section sorts strings by pointer value in order to locate overlapping regions of text. Real-time constraints

[46] The real-time garbage collection algorithm requires two more pointers for each string than the traditional garbage collection algorithm.

require that the addition of each new string descriptor to the list of those pending relocation be performed in a constant amount of time. Even an incremental heapsort would require *log(n)* time for each new string descriptor. The real-time garbage collection algorithm sacrifices exactness in locating garbage for improved time performance. The real-time algorithm finds string garbage only if it occurs between strings originating in different subregions.



Reclaimed by this pass of the garbage collector

▨ Unaccessible memory that is not reclaimed on this pass of the garbage collector

Note: this graph uses simplified string descriptors consisting only of character pointer and length fields

For example, in the figure above (which uses a subregion size of 10 bytes), the probe at the beginning of subregion 2 would detect that 7 bytes of data preceding the probe point and 6 bytes following are no longer accessed. No probe would detect the unaccessed data in the middle of the fourth region. Because any hole of unaccessed data that is at least as big as the size of a subregion would necessarily touch at least one probe point, this algorithm guarantees that the amount of data allocated to a single string never exceeds the size of the string by more than the size of a subregion.

In the example illustrated above, the probes failed to locate the unaccessed string data in the middle of region 4. In order to find all holes of unaccessed data, it is necessary to shift the probe points on subsequent passes of the garbage collector. The global variable probe_offset, which represents the offset at which subregions are aligned, is set for each pass of the garbage collector by swap_spaces.

After all relocated memory has been scanned, string data is incrementally relocated into *to* space. All of the string data that survives garbage collection is combined into a single string region by scan. This reduces the storage overhead of managing a large number of string regions.

Each pass of the garbage collector remembers the location of the common string region created by the preceding pass. This region is always the first region to contribute to the commor string region for the current pass. A consequence of this arrangement is that string data with a relatively long life time (surviving more than one pass of the garbage collector) gradually sifts forward in the common string region. The relative stability of the front of the region helps to organize the systematic searching by probes for holes of unaccessed data. On each pass of the garbage collector, a new probe value is chosen by indexing into a 256-entry array of next probe values. These values are chosen so as to locate large holes of unaccessed data quickly, probing first at the subregion's mid-point, then the quarter-point, then the three-quarters-point, and so on.

This would not make sense, of course, if every iteration of the garbage collector removed a random number of bytes from the front of the common subregion. For this reason the algorithm attempts to stabilize at least the front portion of the common string region.

This implementation of strings increases the amount of memory required to represent strings and adds complexity to the routines that access and manipulate string data. The string structure, for example, is larger than the corresponding data object in Icon's implementation of strings. String data also requires more memory than in the Icon implementation because requests for string memory rarely come in multiples of 256 bytes, and because each string region includes a small header which is not needed in the Icon implementation. A further disadvantage of the real-time algorithm is that strings, because of their increased size, can not be stack allocated as they are in Icon[47]. In terms of execution costs, the real-time algorithm requires, in general, one more level of indirection to access strings than in Icon (for the same reason that strings can not be stack allocated), and complicates the string subscripting operation by requiring that a pointer to the region of its string argument be copied to the string produced by subscripting.

In order to garbage collect strings, several new bookkeeping variables must be introduced:

```
<global-declarations.7>
int probe_offset;             /* offset at which boundaries between */
                              /*  256-character subregions are aligned */

struct control_block *cbp,    /* points to block being initialized, */
                              /*  or to next block to be allocated */
  *scbp;                      /* points to control block being scanned */

int sr_init_cnt,              /* number of subregions in *cbp to be */
                              /*  initialized */
  sr_scan_cnt;                /* number of subregions in *scbp to be */
                              /*  scanned */

char *blocks,                 /* points to array of control blocks */
  *nxt_block;                 /* location of next control block to be */
                              /*  allocated */

/* merge_region points at string region into which all string data that */
/*  survives garbage collection is combined */
struct str_region *merge_region;

char *next_mem;               /* points to next available memory */
                              /*  in merge_region */
```

copy, described partially above, must be enhanced to deal specially with strings. When a string is copied, the string region it points to is marked if it had not been marked previously, and the string is linked onto a list for one of the subregions of the region's control block. This code is outlined below:

[47] In the implementation of Icon, the run-time stack contains descriptors consisting of title and data words. The title word describes how the data word is to be interpreted. Integer values, for example, are stored directly in the data word. For records, however, the data word is a pointer into the dynamic allocation region. The most significant bit of the title word is zero only for descriptors that represent string data. For these descriptors, the remaining bits of the title word, interpreted as an integer value, represent the length of the string. The data word points to the string data.

```
<set-up-string-block-for-scanning> ≡
  if (op->generic.title == String) {
    int which_region;         /* which subregion does descriptor point into? */
    register struct str_obj *sp = (struct str_obj *) rp;

    if (sp->u.region != NULL) {
      if (!sp->u.region->mark)
        <mark-string-region-and-enqueue-string>
      else
        <enqueue-string>
      }
    }
```

Marking a string region allocates a string region control block and begins to initialize it. The size of a string control block is proportional to the length of the string region it controls. Because of this, it is not possible to fully initialize the control block in a constant amount of time. Instead, the block is only partially initialized by copy. Enough state information is maintained to allow users of the data found in a control block to recognize that certain fields have not yet been initialized and to permit scan to finish initialization of the control block. Within the control block, an initialization subregion maintains a list of the strings pointing to a string region that are copied into *to* space before the region has been completely initialized. Following initialization of all the subregions in the block, the strings on this list are distributed to their respective subregions. Below is the code that allocates the control block and places the string on its initialization subregion's list:

```
<mark-string-region-and-enqueue-string> ≡
    {
      int nr;                      /* number of subregions in new block */
      struct control_block *bp;    /* points to new block */

      /* mark the subregion */
      sp->u.region->mark = 1;

      /* allocate a string region control block */
      nr = sp->u.region->u.len / SubregionSize + 1;
      bp = sp->u.region->u.cbp = (struct control_block *) nxt_block;
      nxt_block += sizeof(struct control_block) +
        (nr - 1) * sizeof(struct subregion);

      /* Initialize the string region control block's header. */
      bp->str_data = sp->u.region;
      bp->num_segments = nr;

      /* Initialize the init subregion.  There's no need to initialize first */
      /*  and len fields in this subregion as they are not used. */
      bp->init.prev = bp->init.next = sp;
      sp->prev = sp->u.next = (struct str_obj *) &(bp->init);
scan_balance +=
  (sizeof(struct control_block) + (nr - 1) * sizeof(struct subregion)) * K
  - ((sizeof(struct str_region) + (nr - 1) * SubregionSize - 1)
      - (nr * sizeof(struct subregion)))
  + 2 * sizeof(struct str_obj *);

if (bp == cbp)          /* If this is the next control block to initialize */
  sr_init_cnt = nr;     /*  set sr_init_cnt to advise scan of condition. */
}
```

Above, scan_balance is adjusted to account for several different actions carried out by this code. First, since unlike most other objects copied into *to* space, string regions do not need to be scanned, the scan_balance points that would have been reserved for scanning this portion of relocated memory are available to pay for other activities of the garbage collector. The total size of this string region in bytes is: (sizeof(struct str_region) + (nr - 1) * SubregionSize - 1). Of this many available scan_balance points, (nr * sizeof(struct subregion)) are reserved to pay for initialization of each subregion in the control block. The remaining points are spent here, to pay for the allocation and partial initialization of the string control block. Note that the control block is allocated in *to* space. As with other allocations, scan_balance is incremented by the size of the allocation times K. The size of this allocation is (sizeof(struct control_block) + (nr - 1) * sizeof(struct subregion)). Above, the string being copied is linked onto the list for the initialization subregion of the control block. Eventually, at a cost of (2 * sizeof(struct str_obj *)) scan_balance points, this string must be removed from the initialization list and placed on the list for its corresponding subregion. This many scan_balance points are saved here for when that happens.

In order to guarantee that garbage collection does not loop indefinitely, it is important that the effect of the above adjustments to scan_balance result in a net decrease in its value. The value of K, the number of characters in a string subregion, and the sizes of the string region, subregion, and control block structures all contribute to the sign and magnitude of the change in scan_balance's value. Static analysis based on these values must guarantee that the change is negative. For example, on a 32-bit VAX computer, it can be shown that the desired conditions are met for values of K less than or equal to 3[48].

The string region control block allocated above was only partially initialized. Each of the subregions other than the one set aside for initialization must be initialized by scan. Initialization of string control blocks occurs in two phases. First, each of the subregions is given initial values. The pointer cbp points to the control block being initialized. Within that control block, the variable sr_init_cnt counts the number of subregions requiring initialization. After initialization of subregions completes, strings are removed one by one from the initialization subregion and placed on the list of the subregion to which they correspond. At the top of its loop, scan tests to see if there are any subregions within a control block that must be initialized. A template of this code is provided below:

[48] Note that nr ≥ 2, sizeof(struct str_region) = 6, sizeof(struct subregion) = 20, and sizeof(struct control_block) = 48.

```
<initialize-string-regions> ≡
    static int sort_strings = False;

    if (sr_init_cnt > 0)
      <initialize-a-subregion>
    else if (sort_strings)
      <sort-strings-to-respective-subregions>
    else
```

Initialization of a subregion consists of setting the first and len fields, and setting up the doubly-linked list to contain only its head, the subregion structure itself. After all subregions of a particular control block have been initialized, the variable sort_strings is set to True if there are strings from the initialization subregion to be sorted. Otherwise, initialization efforts are focused on the next control block.

```
<initialize-a-subregion> ≡
    {
      register struct subregion *srp = &cbp->probes[--sr_init_cnt];

      srp->first = NULL;
      srp->len = 0;
      srp->prev = srp->next = (struct str_obj *) srp;

      scan_balance -= sizeof(struct subregion);

      if (sr_init_cnt == 0) {                 /* control block is initialized */
        if (cbp->init.next != (struct str_obj *) &(cbp->init))
          sort_strings = True;                /* start sorting strings */
        else {
          cbp = (struct control_block *)
            &(cbp->probes[cbp->num_segments]);
          if ((char *) cbp < nxt_block)       /* start initializing next block */
            sr_init_cnt = cbp->num_segments;
          }
        }
    }
```

After all the subregions of a control block have been initialized, it may be necessary to sort the strings saved up in the initialization subregion into their proper subregions. In the code fragment above, this condition is signaled by setting the variable sort_strings to True. Immediately following the above code in the loop of scan is the following fragment. Note that this fragment is only executed if, at some time, a string was placed on the initialization list. Even if that string has since been removed from the list, scan_balance points were previously set aside to pay for this operation, so scan_balance is decremented here.

```
<sort-strings-to-respective-subregions> ≡
    {
      register struct str_obj *sp;
      int which_region;
```

```
  scan_balance -= 2 * sizeof(struct str_obj *);
  if ((sp = cbp->init.next) != (struct str_obj *) &(cbp->init)) {

    /* remove *sp from the initialization list */
    cbp->init.next = sp->u.next;
    sp->u.next->prev = (struct str_obj *) &(cbp->init);

    /* and insert it in the appropriate subregion */
    which_region =
      (sp->start - cbp->str_data->data + SubregionSize - probe_offset)
      / SubregionSize;
    sift_string(sp, &cbp->probes[which_region]);
    }

  if (cbp->init.next == (struct str_obj *) &(cbp->init)) {
    /* The initialization subregion is empty. */
    /*  Prepare to initialize next block. */
    sort_strings = False;
    cbp = (struct control_block *) &(cbp->probes[cbp->num_segments]);
    if ((char *) cbp < nxt_block)       /* start initializing next block */
      sr_init_cnt = cbp->num_segments;
    }
}
```

The sift_string function places a string on the list for a particular subregion and adjusts the state information for that subregion. The code is shown below:

```
<sift_string> ≡
sift_string(sp, srp)
     register struct str_obj *sp;
     register struct subregion *srp;
{
  sp->u.next = srp->next;
  sp->u.next->prev = sp;
  sp->prev = (struct str_obj *) srp;
  srp->next = sp;

  /* Update the state information for this subregion */
  if ((srp->first == NULL) || srp->first > sp->start) {
    if (srp->first)
      srp->len += srp->first - sp->start;
    srp->first = sp->start;
    }

  if (srp->first + srp->len < sp->start + sp->len)
    srp->len = (sp->start + sp->len) - srp->first;
}
```

Above, the history of the first string referencing a particular string region is traced. As the object is copied, its string region is marked and a control block is allocated, and eventually initialized. The processing of subsequent strings pointing to the same string region is generally simpler. If the string region referenced by a newly copied string has already been marked, copy needs only to insert the string on a list of strings pointing into that string region. Ideally, the new string is placed on the list that holds all strings pointing to a particular subregion. If, however, the desired subregion has not yet been initialized, the string is added to the list for the special initialization subregion. The following code performs this task. It is extracted from copy, immediately following the code from that same function that appears above.

```
<enqueue-string> ≡
    {
      register struct control_block *tcbp = sp->u.region->u.cbp;

      which_region =
        (sp->start - sp->u.region->data + SubregionSize - probe_offset) /
        SubregionSize;

      if (tcbp > cbp || (tcbp == cbp && sr_init_cnt > which_region)) {
        /* The specified subregion has not yet been initialized. */
        /* Place this string region in the initialization subregion for now. */
        sp->u.next = tcbp->init.next;
        sp->u.next->prev = tcbp->init.next = sp;
        sp->prev = (struct str_obj *) &(tcbp->init);

        /* Save scan_balance points for when this object will be sorted. */
        scan_balance += 2 * sizeof(struct str_obj *);
        }
      else                          /* the subregion has been initialized */
        sift_string(sp, &tcbp->probes[which_region]);
      }
```

In the code above, scan_balance is incremented to represent the idea that the string has not yet been completely copied. The string is not considered completely copied until the string has been moved from the initialization list to the proper subregion's list. Prior to execution of the above block of code, scan_balance had been decremented by the entire size of the string object under the assumption that the string would not need to be linked onto the initialization subregion's list.

After there are no more linked objects to scan, scan processes the string region control blocks. The variable scbp points to the control block being scanned. At the start of garbage collection, scbp is set to point at the base of the array of control blocks in the middle of *to* space. Scanning of a control block consists of locating regions of contiguous string data within the string region controlled by the block, copying this string data into the common string region of *to* space, and adjusting all strings that point at the contiguous region of string data so that they now point instead to the copied data. All of this is done incrementally. After each control block is scanned, scbp is incremented to point to the next control block. Scanning of control blocks terminates when scbp equals cbp. This code is organized according to the following template:

```
<scan-string-regions> ≡
  else if (scbp < cbp) {
    <local-variables>
    <macro-definitions>
if (shift_amt == 0) {
  <look-for-contiguous-region>
  }
else if (!found_end) {
  <attempt-to-extend-contiguous-region>
  }
else {
  <adjust-string-pointers-for-a-contiguous-region>
  }
}
```

Several local variables are introduced here. The register variable srp points to the subregion currently being scanned. shift_amt records the difference between the new and old addresses of the string data. A value of zero is used to signify that no block of contiguous data has yet been found. nxt_shift_amt represents the next value of shift_amt. After all strings pointing to the current region have been adjusted, nxt_shift_amt is assigned to shift_amt. The boolean variable found_end is set after the end of a contiguous region has been found. The character pointers scontig and econtig mark the beginning and end of the contiguous region respectively. In order to adjust each string's character pointer in constant time, all strings pointing to a particular contiguous region of data are linked onto the first subregion occupied by that contiguous region. The pointer fsubregion points to this subregion. The variables nxt_found_end and nxt_fsubregion hold the next values to be assigned to found_end and fsubregion respectively. These assignments are made when the scanner begins to look for another contiguous region if nxt_shift_amt is non-zero. As string data is relocated into the common string region of *to* space, the variable next_mem, which points to the next available memory in that string region, is incremented by the length of the string data appended to the region. numc simply represents the number of characters to copy at a time into *to* space.

```
<local-variables> ≡
    register struct subregion *srp = &scbp->probes[sr_scan_cnt];
    static int shift_amt = 0,              /* by how much are string */
                                           /*  pointers to be adjusted for */
                                           /*  this contiguous region */
      nxt_shift_amt = 0,                   /* next value of shift_amt */
      nxt_found_end;                       /* next value of found_end */

    char *scontig;                         /* start of contiguous string data */
    static char *econtig;                  /* end of contiguous string data */

    static int found_end;                  /* has end of contiguous region */
                                           /*  been found? */

    static struct subregion *fsubregion,   /* first subregion for */
                                           /*  contiguous block */
      *nxt_fsubregion;                     /* next value of fsubregion */

    int numc;                              /* how many characters */
                                           /*  to copy? */
```

Within this section of code, several new macros are introduced. Min, for example, produces the minimum value of its two arguments. Because subregions are aligned at index position probe_offset from the beginning of the string region to which they correspond. calculation of the beginning. end. and size of each region is somewhat involved. The macros RegionStart. RegionEnd. and RegionSize provide these values.

```
<macro-definitions> ≡

#define Min(a,b)             (((a) < (b))? (a): (b))

#define RegionStart(bp, rn) ((bp)->str_data->data + \
                             ((rn)? (probe_offset + SubregionSize \
                                     * ((rn) - 1)): 0))

#define RegionEnd(bp, rn)   ((bp)->str_data->data + (rn) * SubregionSize \
                             + ((((rn) + 1) == (bp)->num_segments)? \
                                0: probe_offset))

#define RegionSize(bp, rn)  (RegionEnd(bp, rn) - RegionStart(bp, rn))
```

String data is copied into *to* space one subregion at a time. Scanning of a subregion consists of determining which string data within that subregion is currently accessed. and copying that data into the shared string region of *to* space. Whenever a contiguous region of accessible string data is found to include multiple subregions, the lists of strings pointing to the contiguous region of string data are combined into a single list. Merging of two lists is a constant-time operation performed by scan whenever it scans subregions that share string data with previous subregions. In terms of the effect on scan_balance, scanning of a subregion decrements scan_balance by the size of the string data governed by that subregion. After all of the accessible memory in a contiguous block of string data has been copied into *to* space, the strings pointing to that data are removed one at a time from the linked list based at the first subregion for that block. and are adjusted to reflect the new location of their string data. As each string is updated in this manner. scan_balance is decremented by the size of a string minus the size of its integer title. The integer title of the string was considered scanned when the string was encountered by the scan function. Following is the line of code extracted from scan that serves this function:

```
<decrement-scan_balance-for-string> ≡
        scan_balance -= sizeof(int);
```

Real-time constraints require that the task of processing a control block be interruptable. The code provided below divides the task into the processing of each subregion's string data. which must be either copied into *to* space or ignored. and the updating of string objects to reflect the new location of the string data they point to. This code can be interrupted between the processing of each subregion or string object. Were it necessary to interrupt the code even more frequently. additional bookkeeping variables and work would make this possible.

If there is no possibility of a string that originates in some preceding subregion overlapping the current subregion. then the current subregion is processed by the following code. Note that this code is executed only if the garbage collector is not already looking at a block of contiguous string data.

<look-for-contiguous-region> ≡

```
/* look for some string data to copy */
if (srp->next != (struct str_obj *) srp) {
  if (merge_region == NULL) {
    /* Initialize merge region for this pass of the garbage collector. */
    merge_region = (struct str_region *) relocated;

    /* Initialize region. */
    /*  Length will be set after all string data has been copied. */
    merge_region->mark = 0;
    next_mem = merge_region->data;
    }

  scontig = srp->first;
  econtig = srp->first + srp->len;

  fsubregion = srp;

  shift_amt = next_mem - scontig;
  found_end = False;

  numc = Min(econtig, RegionEnd(scbp, sr_scan_cnt)) - scontig;

  movmem(scontig, next_mem, numc);
  next_mem += numc;
  }

scan_balance -= RegionSize(scbp, sr_scan_cnt);

if (++sr_scan_cnt >= scbp->num_segments) {
  /* done scanning this block */
  if (shift_amt == 0)
    next_block();   /* move scbp to next string region control block */
  else {            /* prepare to update strings */
    found_end = True;
    nxt_shift_amt = 0;
    }
  }
```

The code above calls next_block to advance scbp to the next string control block after scanning of the current block completes. If next_block detects that there are no more blocks to scan, it finishes off garbage collection of strings by setting the length of the merge region.

<next_block> ≡

```
next_block()
{
  scbp = (struct control_block *) &(scbp->probes[scbp->num_segments]);
  sr_scan_cnt = 0;
```

```
        if (scbp == cbp) {          /* garbage collection of strings is done */
          merge_region->u.len = Ceiling((next_mem - merge_region->data) /
            (double) SubregionSize) * SubregionSize;
          next_mem = &merge_region->data[merge_region->u.len];
          next_size += (next_mem - relocated) +
            Ceiling(2.0 * (next_mem - relocated) / K);
          }
      }
```

Because each subregion remembers only the lowest and highest addresses referenced by strings originating in that subregion, there can be no more than one contiguous region of non-garbage string data originating in each subregion. However, contiguous regions of string data may traverse boundaries between subregions. In general, the decision of whether the end of a contiguous region of string data has been found is not made until the following subregion has been examined. As shown in the code above, when a subregion containing strings is found, the variable shift_amt is set to the pointer difference between the new and old locations of the string data in memory, and found_end is set to False. The processing of subsequent subregions is performed by the following code, which appears immediately following the code described above:

```
    <attempt-to-extend-contiguous-region.1> ≡
      scontig = RegionStart(scbp, sr_scan_cnt);

      if ((econtig > RegionEnd(scbp, sr_scan_cnt)) ||
          (srp->first && econtig > srp->first)) {
        /* still haven't found end of contiguous region */
        if (srp->first + srp->len > econtig)
          econtig = srp->first + srp->len;

        numc = Min(econtig, RegionEnd(scbp, sr_scan_cnt)) - scontig;

        movmem(scontig, next_mem, numc);
        next_mem += numc;

        /* move list of strings, if it's not empty, to *fsubregion */
        if (srp->next != (struct str_obj *) srp) {
          struct str_obj *sp = fsubregion->next;

          srp->prev->u.next = sp;
          sp->prev = srp->prev;

          sp = srp->next;
          fsubregion->next = sp;
          sp->prev = (struct str_obj *) fsubregion;
          }
        }
```

The fragment above extends the contiguous region to include all string data referenced by this subregion also. All strings pointing into this subregion are linked onto the list of strings for the first subregion contributing to this contiguous region of string data. All the strings are placed on a common list so that later, after the end of the contiguous region of string data has been found, these strings can be individually found and modified in constant time.

Following is the code that executes when the end of the current contiguous region of string data has been found. Note that it is possible for one contiguous region of string data to terminate and another one to begin within a single subregion.

```
<attempt-to-extend-contiguous-region.2> ≡
  else {                        /* contiguous region has ended */
    found_end = True;

    if ((numc = econtig - RegionStart(scbp, sr_scan_cnt)) > 0) {
      movmem(scontig, next_mem, numc);
      next_mem += numc;
      }

    if (srp->next != (struct str_obj *) srp) {
      int numc;                 /* start another contiguous region */

      scontig = srp->first;
      econtig = srp->first + srp->len;

      nxt_fsubregion = srp;

      nxt_shift_amt = next_mem - scontig;
      nxt_found_end = False;

      numc = Min(econtig, RegionEnd(scbp, sr_scan_cnt)) - scontig;

      movmem(scontig, next_mem, numc);
      next_mem += numc;
      }
    else
      nxt_shift_amt = 0;
    }

  scan_balance -= RegionSize(scbp, sr_scan_cnt);

  if (++sr_scan_cnt >= scbp->num_segments) {
    /* done scanning this block */
    if (!found_end) {
      found_end = True;
      nxt_shift_amt = 0;
      }
    else if (nxt_shift_amt)
      nxt_found_end = True;
    }
```

After the end of a contiguous region has been found, all the strings pointing into that region must be updated. This is the task of the following block of code. Note that, after this task is completed, the values of shift_amount, fsubregion, and found_end are set according to whether or not the most recently processed subregion started a new block of contiguous string data.

```
<adjust-string-pointers-for-a-contiguous-region> ≡
    register struct str_obj *sp;

    if (fsubregion->next != (struct str_obj *) fsubregion) {
      sp = fsubregion->next;
      fsubregion->next = sp->u.next;
      sp->u.next->prev = (struct str_obj *) fsubregion;

      sp->start += shift_amt;
      sp->u.region = merge_region;
      sp->prev = NULL;
      }

    scan_balance -= sizeof(struct str_obj) - sizeof(int);

    if (fsubregion->next == (struct str_obj *) fsubregion) {
      /* there are no more strings to update */
      if ((shift_amt = nxt_shift_amt) != 0) {
        fsubregion = nxt_fsubregion;
        found_end = nxt_found_end;
        nxt_shift_amt = 0;
        }
      else if (sr_scan_cnt >= scbp->num_segments)
        next_block();
      }
```

swap_spaces is the routine that bootstraps garbage collection. Its definition is provided below:

```
<swap_spaces.1> ≡
char *tospace, *fromspace;

extern int nxt_probes[SubregionSize];

swap_spaces(n)
     int n;              /* how much memory is being allocated now */
{
  int allocated;         /* how much memory has been allocated */
                         /*  in from space */
  int i;
  char *cp;

  gc_active = True;
  allocated = ((merge_region)? next_mem: relocated)
    - tospace + (tospace + cur_size - new);
```

```
/* Recalculate nxt_size to reduce round-off errors */
/*  and to add the current allocation. */
next_size = allocated + Ceiling(2.0 * allocated / K) + n;
if (next_size > cur_size) {
  expand_regions(next_size);
  cur_size = next_size;
  }
```

Above, the amount of memory required to garbage collect the current *to* space is calculated. If more memory than is available in the current *from* space is required, the size of the regions is expanded. Code for expand_regions is not provided. The ability to guarantee real-time response depends on an ability to enlarge at least *from* space in a constant amount of time. Since it is not necessary that the new *from* space contain the same data that was stored in the old *from* space, the prototype implementation of expand_regions simply frees the current *from* space and allocates a new larger *from* space[49] using the C routines: malloc and free. To guarantee real-time response, either the host operating system must guarantee that allocation or expansion of memory is performed in constant time, or the storage allocator must treat any request to expand the allocation regions as a fatal error.

Below, garbage collection state variables are initialized for a new pass of the garbage collector. In alloc, before swap_spaces was called, scanning may have been charged to the current allocation request so it would not be fair to charge for this allocation also on the current pass of the garbage collector. Instead, the requested memory is simply set aside in the new *to* space without further incrementing scan_balance.

```
<swap_spaces.2> ≡
  /* Set up variables for the next garbage collection pass. */
  next_size = n + Ceiling(2.0 * n / K);

  cp = tospace;
  tospace = fromspace;
  fromspace = cp;

  new = tospace + cur_size;
  scanned = relocated = tospace;

  nxt_block = blocks = tospace + allocated;
  scbp = cbp = (struct control_block *) blocks;
  sr_scan_cnt = 0;

  probe_offset = nxt_probes[probe_offset];

  scan_balance = 0;
```

[49] In the prototype implementation, *to* and *from* spaces are not necessarily the same size. The code above, however, assumes they are the same size in order to simplify the presentation. In the actual implementation, whenever *to* space is enlarged in order to garbage collect *from* space, enlargement of *from* space is postponed until the next pass of the garbage collector, at which time *from* space is enlarged to become the new *to* space.

```
        if (merge_region) {
          <initialize-merge-region>
          }
        else
          sr_init_cnt = 0;

        merge_region = NULL;

        /* update tended descriptors */
        for (i = 0; i < TendedDescriptors; i++)
          if (InFromSpace(tend_desc[i]))
            tend_desc[i].ptr = copy(tend_desc[i].ptr)
      }
```

As mentioned above, all string data that survives garbage collection is combined into a shared string region. With each pass of the garbage collector, a new shared string region is created. For reasons discussed above, the first data copied into the shared region is the data that is still accessible from the previous shared region. swap_spaces marks the old merge region before any other region has been marked so that its control block appears first in the array of control blocks. This is shown below:

```
<initialize-merge-region> ≡
    /* allocate and begin initialization of control block for merge region */
    merge_region->mark = 1;
    sr_init_cnt = merge_region->u.len / SubregionSize + 1;

    nxt_block += sizeof(struct control_block) +
      (sr_init_cnt - 1) * sizeof(struct subregion);

    cbp->str_data = merge_region;
    cbp->num_segments = sr_init_cnt;

    cbp->init.prev = cbp->init.next = (struct str_obj *) &(cbp->init);

    merge_region->u.cbp = cbp;

    /* Adjust scan_balance to reflect allocation and initialization efforts. */
    scan_balance -=
      (sizeof(struct str_region) + (sr_init_cnt - 1) * SubregionSize - 1)
      - (sr_init_cnt * sizeof(struct subregion));

    scan_balance += (sizeof(struct control_block)
                    + (sr_init_cnt - 1) * sizeof(struct subregion)) * K;
```

7.4 Analysis of Worst-Case Response Time

In the expression that follows, K represents the proportionality constant discussed above that relates allocation to garbage collection. Each allocation of n bytes is accompanied by scanning and relocating a combined total of $n \times K$ bytes. Let γ be the size of the largest object referenced by the program. Define Γ to be the cost of copying a byte of data from one area of memory to another. Assume that this cost is the same as the cost of scanning a byte. Remember that scanning consists of looking at a pointer and modifying its value if the pointer points into *from* space.

Scanning frequently requires that objects be copied into *to* space but that operation and its costs are charged separately.

An allocation of δ bytes of memory executes in time

$$\alpha + \beta + (\delta \times K + \gamma - 1) \times \Gamma$$

where α is the constant time required to perform the allocation and the rest of the expression above represents the cost of garbage collection. β in the expression above represents the constant costs of garbage collection including the cost of bootstrapping the garbage collection algorithm. The bootstrapping cost includes the cost of updating all tended descriptors, which is simply the number of tended descriptors times $(\gamma \times \Gamma)$. In most real programs, garbage collection is only active a small fraction of the running time, so the actual cost of allocation is usually only α.

7.5 Analysis of Storage Throughput and Requirements

The applications that motivated the design of this real-time garbage collection algorithm typically run for long periods of time. Existing programs serving these application areas generally iterate, responding to asynchronous events as they occur. Most programs reach a steady state, finding a natural balance between allocation and garbage collection. In this steady state, the amount of memory accessed by the program at any time (the non-garbage) remains relatively constant.

Suppose that the steady-state amount of memory referenced by a program is M bytes. The garbage collector must be prepared to scan and copy all M bytes while satisfying new allocation requests. The amount of new memory that must be allocated in order to scan and copy M bytes is $(2 \times M)/K$. Thus the combined total of *to* and *from* space must be at least $2 \times (M + (2 \times M)/K) = 2 \times M \times (1 + 2/K)$. K in the above equation is the same constant discussed above. Note that memory requirements are $O(1/K)$ and worst-case response time is $O(K)$. For large K, response time increases but memory requirements decrease. Decreasing K improves response time at the expense of additional memory.

Since each invocation of the garbage collector executes in time proportional to the amount of garbage collection performed, garbage collecting a region of size N executes with time complexity $O(N)$. Note that, assuming the size of memory regions remains constant, the amount of memory allocated from a region of size N is given by $M = (K \times N)/(K + 2)$. This is just a constant factor times N. Likewise, it is common for the amount of memory copied into *to* space to be only a small fraction of the total amount allocated in *from* space.

In the analysis described above, M includes the space occupied by string data regions, which may include holes of unaccessible data that has not yet been reclaimed by the garbage collector. M also includes the storage overhead of titles on every object, and link fields within strings.

Though it is important to understand and recognize the storage overhead of this algorithm, only in rare occasions is this a practical concern. It is very uncommon for garbage collection systems (real-time or not) to operate at or near full capacity. Almost all real systems employ heuristics of some sort designed to keep plenty of breathing room in the allocation regions so as to reduce the frequency of required garbage collections. Being able to collect garbage in real time lessens the necessity to reduce the frequency of garbage collections. In some cases, this may even result in programs using less memory than with more traditional stop-and-wait garbage collection strategies.

7.6 Comparison with the Icon Algorithm

Since Icon performs all garbage collection with a single invocation of the garbage collector. Icon's performance can be compared only with the analysis of storage throughput described above. Icon's cost of garbage collecting multiple regions with a combined total of N bytes and a total of v strings is of time complexity $O(N + v^2)$. In this expression, the first term represents the cost of relocating objects in order to expand regions and compress holes from in between allocated objects. The second term represents the cost of sorting strings using the quicksort algorithm in order to recognize overlapping string segments.

Although the time complexity of the real-time algorithm compares favorably with Icon's algorithm, the real-time algorithm imposes a greater penalty on standard memory referencing operations. For example, assignments to tended descriptors must be prefaced by a test to see if garbage collection is active, and if so whether the referenced object has been relocated.

One advantage of the real-time algorithm is that, because it is designed to run concurrently with execution of a program, most of the work of garbage collection could be delegated to an auxiliary processor, and much of the bookkeeping work associated with indirection of pointers could be executed by microcode or even hard wired into a CPU. This assumes, of course, that dedicated hardware might be developed to support this garbage collection algorithm.

The prototype implementation of Conicon uses the real-time garbage collection algorithm described in this chapter. Implementation of the prototype garbage collection algorithm was motivated principally by a need to verify the algorithm's correctness. Efforts were not focused on obtaining an optimal implementation of the algorithm. No attempts have been made, for example, to hand-tune the implementation based on the results of run-time profiling. Also, in several places, descriptors are fixed in preparation for loading them into tended registers. Depending on the flow of execution, the same descriptor may be fixed several times, even though it needs to be fixed only once. This is one place where attention to detail was sacrificed in order to get the prototype up and running quickly.

For purposes of comparison, a program that generates random sentences of a small context-free grammar was implemented in both Conicon and Icon. This program is described in §7.6 of [25]. Because both Conicon and Icon use the same random number generator, each program generates the same sequence of sentences. For programs generating 100-20,000 random sentences, the traditional implementation consistently ran in 40% of the time required by the real-time implementation. For this program, the task of garbage collection is particularly easy for both algorithms because there is very little data to be retained by the garbage collector. After each random sentence is generated, it is simply discarded. The difference in run times for these measurements is due principally to the overhead imposed on normal execution by the real-time garbage collection algorithm. This overhead includes the increased size of strings, the storage overhead associated with string data, the extra level of indirection associated with accessing string data, and the extra work that accompanies each assignment to a tended descriptor. Though not described in this chapter, a small amount of bookkeeping work also accompanies every operation that decreases the height of the run-time stack.

When the program described above is modified to tabulate the number of times each generated string is produced as described in §5.2.1 of [25], the relative performance of the two implementations is similar. With both garbage collection algorithms, garbage collection for the tabulating version of the program must find and relocate all the data stored in the table used to tabulate results. This benchmark measures the time required to do non-trivial garbage collection.

Using either garbage collection strategy, the program's running time is roughly proportional to the number of strings tabulated. For this program, the traditional algorithm runs in from 35-45% of the time required by the real-time algorithm, depending on how many strings have been generated. The version of this program that tabulates 20,000 random strings was profiled in detail. After accounting for the overhead of the profiling tools, it was found that the real-time algorithm spends 10.6% of its time inside functions dedicated solely to the allocation of new memory, and 14.3% of its time inside garbage collection functions. In contrast, the traditional algorithm spent only 2.6% of its time allocating new memory, and 12.9% of its time performing garbage collection. The real-time algorithm collected garbage 65 times in running this test case compared to only 20 times for the traditional algorithm. This is presumably because of the increased burden placed on the real-time algorithm of allocating each string object from the dynamic allocation region instead of on the stack as is done in the traditional algorithm, and because the real-time algorithm is only able to allocate a small fraction of its available memory. The real-time algorithm, using a value of 3 for K, required almost ten times as much data space as the traditional implementation.

These measurements suggest that, in terms of system throughput, the major shortcoming of the real-time algorithm is the large amount of bookkeeping overhead that is distributed throughout the interpreter, the costs of which are incurred even for programs that do not perform garbage collection. These statistics support the idea that building special-purpose machine instructions or parallel circuitry to do the bookkeeping of real-time garbage collection would have a significant positive effect on the performance of this algorithm. Although these figures are not particularly good for the real-time algorithm, they demonstrate that it is possible to trade throughput for improved worst-case response time in situations where response time is a primary concern.

Chapter 8: Conclusions

This dissertation has briefly introduced some of the special needs of real-time programming applications and suggested how these needs can be satisfied by specially designed high-level language features. Likewise, the special needs of applications that perform structural pattern matching were discussed and language features to satisfy these needs were also presented. Algorithms for implementing the new language features have been described and analyzed. And several examples of how the new language features have been used were presented. In the discussion and analysis that accompanied these examples, it was argued that the proposed language features simplify the development of many applications that need to perform pattern matching in real time. Below, the findings of this dissertation research are summarized and future research directions are outlined.

8.1 Summary

One of the important findings of this research is that it is possible to provide useful high-level language capabilities that execute in constant time. These capabilities can greatly simplify the development of real-time software. For example, because of the perceived complexity of garbage collection and alternative dynamic storage management strategies, many real-time programs are written to use only statically allocated memory. Having to allocate all memory statically restricts selection of algorithms, may burden the programmer with explicit dynamic management of the statically allocated memory, and may result in somewhat arbitrary restrictions in the resulting program's capabilities. This dissertation provides an algorithm for allocating and garbage collecting blocks of linked memory and string data in constant time. Programs designed to take advantage of this garbage collection algorithm can ignore many of the storage management details that significantly complicate programs that do not use garbage collection. Other high-level language capabilities that can be provided in constant time are, for example, the creation of a concurrent process and reading of a single quantity from a stream of data values.

Concurrent processes provide a useful programming abstraction for describing multiple concerns that must be dealt with simultaneously. This is a common need of real-time systems. Concurrent processes also provide to real-time programmers an opportunity to describe the activity of multiple processors working together to solve a single problem. The syntactic mechanisms for describing concurrent processes and synchronization of processes are built into Conicon and integrate naturally with other language features.

The stream data type provides other useful high-level capabilities to real-time programmers. Streams are used to synchronize concurrent activities and to transfer information between concurrent processes. Stream-based synchronization or communication may connect concurrent Conicon processes or may interface Conicon processes with the external environment. Streams also provide a high-level view of physical interrupts. In general, the stream data type connects producer processes with consumers. Unlike the low-level pipe or file manipulation primitives provided in many other concurrent languages and operating systems, the stream data type is designed to allow efficient pattern matching on data streams as soon as data values are available, and supports implicit data backtracking whenever the application program requires it. Backtracking of the stream focus is a natural part of Conicon's goal-directed expression evaluation mechanism. Also, streams cooperate with the real-time garbage collection algorithm in order to reclaim memory allocated to stream values that can no longer be referenced.

8.2 Future Research Directions

Historically, the design of programming language features has been an iterative process consisting of evaluation of existing language features, invention and proposal of new language features, implementation of the new features, and experimentation with these features. The work described in this dissertation has already benefited from several iterations through this loop. However, the language features have so far been exposed only to a small number of persons whose principle concerns are in the area of language design. These features have not yet been put to the test in the application environments they were originally designed to serve.

In order to subject the proposed language features and implementation to real users, it is necessary to first obtain a more robust implementation. Greater attention to performance concerns must also be given. The current implementation is essentially a prototype. It provides an opportunity to experiment with the general feel of the language features but probably does not provide sufficient throughput for challenging real-time applications and it has not been thoroughly tested. Also, in order to provide true constant-time performance, it is necessary to build an implementation on top of a bare machine instead of running as an application within a larger time-sharing, virtual-memory operating system.

One aspect of the virtual machine described in Chapter 6 has not been implemented. Instead of using the RISC instruction set described there, the current implementation of Conicon uses the same instruction set used by Icon's virtual machine with a few additional instructions and a small number of changes to the existing instructions. Redefinition of the translator's code generator and of the virtual machine to conform to the specifications of Chapter 6 is needed in order to support actual real-time applications. Accompanying this work is the necessity to rigorously define the worst-case execution times for each instruction of the RISC machine and the worst-case sequence of instructions required to implement each high-level language feature. Most of the work required to obtain a robust real-time implementation of Conicon is well understood and does not constitute research in and of itself. However, this work is an important step toward gathering meaningful feedback from the user community that is actively involved in developing real-time pattern-matching software.

There are other aspects of the implementation where further development efforts could lead to new research topics. For example, it was suggested in Chapter 7 that much of the work of the real-time garbage collection algorithm might be delegated to special-purpose hardware circuits, microcode, or auxiliary processors. There are several interesting questions that might be answered by research in the design and analysis of a computer architecture to support this type of memory management. An important measure when analyzing this kind of special-purpose architecture is the cost in terms of hardware complexity of the proposed circuitry. It might be worthwhile, for example, to investigate strategies for locating all of the garbage collection hardware within a computer's memory modules in order to allow the use of general purpose microprocessors for central processing units.

Special computer architectures also have the potential of increasing Conicon's throughput by running concurrent Conicon processes on different processors. There are many details to be considered in providing these capabilities. For example, design and analysis of a shared run-time kernel that still provides real-time response is a challenging problem worthy of investigation. Strategies for implementing shared memory and efficiently pipelining streams of information between processors are other areas that would benefit from further research. As was mentioned in Chapter 1, real computerized pattern matching is usually carried out using a combination of statistical and structural pattern-matching methods. This dissertation has focused on linguistic support for real-time structural pattern matching. Methods of performing statistical pattern matching in real time have been investigated extensively [43,44] and generally make use of parallel circuits to carry out fine-grain parallelism. In designing a multi-processor implementation of Conicon, it is important to consider that it may be necessary to provide efficient interfaces between Conicon processes and special-purpose statistical pattern-matching hardware. New language features might also be designed to simplify the description of these sorts of configurations.

In order to evaluate the usefulness of the language features proposed within this dissertation, it is important to collect feedback from programmers who are using the language and to analyze the programs they write to determine how the features affect their programming style. Observed usage patterns might suggest new optimization techniques and changes to the language definition. For example, shared memory and shared access to streams are achieved at a significant cost to the run-time system. If these features are only rarely used, then their generality might not justify their cost. In that case, either the feature could be removed, or a method of incurring the costs of full generality only when they are actually needed might be developed. Whenever it is known, for example, that only one process will read or write a particular stream, then it is not necessary for that process to obtain read or write locks for the stream before accessing it. Special functions might reduce the costs of subsequent access to the stream by communicating to the run-time system the intended use of the stream. Feedback from users might also suggest that greater language expressiveness is needed. For example, Conicon's goal-directed expression evaluation backtracks in last-in, first-out order. Mechanisms for describing alternative backtracking order might be especially useful to real-time programmers. As another example, control over process scheduling is currently provided by controlling process priorities. For some real-time applications, a finer degree of control might be desired. Similar concerns might motivate special language facilities for scheduling access to shared resources such as network communications channels.

Some real-time computer systems are responsible for very expensive equipment and even human lives. It is very important that these systems function according to their design specifications. In these situations, it may be worthwhile to attempt to prove that the software system is "correct" or at least that it conforms to the design specifications. Currently, proof techniques are somewhat *ad hoc*. Proofs of this sort might benefit from the development of standard proof techniques or formalisms to simplify an understanding of Conicon programs. Furthermore, a Conicon programming environment with some knowledge of these proof techniques and formalisms could reduce the amount of detail required to compose a proof of correctness, time boundedness, and memory requirements. This programming environment might provide mechanisms for determining the worst-case execution times of particular expressions and might provide symbolic manipulation capabilities for combining calculated or measured execution times algebraically into time bounds for larger program components. Support for automatic analysis of a program would be helpful whenever a software system is modified, ported to a different host computer, or recompiled and possibly reoptimized.

Acknowledgments

I would like to thank Ralph Griswold for his support in this research. Throughout, he has extended freedom to pursue my own ideas, while at the same time providing very helpful guidance. His comments and probing questions frequently revealed aspects of my work that were not yet fully developed or that might benefit significantly from minor revisions. Professor Griswold has also been a great help in focusing my efforts on pursuits that are consistent with completion of this dissertation in a timely manner. In the preparation of this dissertation and of other documents related to this dissertation, Professor Griswold has assisted by quickly reviewing many preliminary drafts and suggesting a variety of improvements in both content and presentation.

I am also indebted to my other committee members: Saumya Debray and Peter Downey, who have carefully reviewed early drafts of the dissertation and provided a variety of suggestions for improving the presentation of this material. Their experience and knowledge in complimentary fields have also helped to clarify the capabilities and limitations of the problem-solving approaches discussed in this dissertation.

To the other members of the Icon project at the University of Arizona: Dave Gudeman, Janalee O'Bagy, and Ken Walker, I am indebted for their participation in several informal discussions regarding this dissertation work. Their feedback helped me to perceive the needs and expectations of programmers who might someday use the proposed language features to write real application software. Members of the Icon project also made several good suggestions for how the language features could be presented and described to others.

To Professor Kenneth Mylrea and graduate student Mohammad Navabi, both of the Electrical and Computer Engineering Department at the University of Arizona, I am grateful for the electrocardiogram data they collected for one of the applications described in Chapter 5 of this dissertation.

Finally, I am grateful to my parents and to my family. To my parents, I am especially grateful for their having taught me to be curious, and to seek answers to my questions through active investigation. I am also grateful that they have always encouraged me to pursue my interests and ambitions and have helped me to reach my worthy goals, whatever they might be. To my wife Lorrain, I am grateful for her support and encouragement. I am also thankful for the time and attention she has given to our children in my absence (be it physical or mental) and for her willingness to work around the somewhat demanding schedule of a graduate student.

REFERENCES

1. C. Chen. *Statistical Pattern Recognition*, Spartan Books. Hayden Book Company. Inc.. 1973.
2. W. A. Lea, The Elements of Speech Recognition, in *Electronic Speech Recognition*. G. Bristow (ed.), Collins Professional and Technical Books, 1986.
3. J. E. McNamara, *Technical Aspects of Data Communication*, Digital Equipment Corporation. 1977.
4. K. S. Fu, *Syntactic Methods in Pattern Recognition*, Academic Press, 1974.
5. R. A. Kirsch, "Computer Interpretation of English Text and Picture Patterns", *IEEE Transactions on Computers EC-13*, 4 (1964), 363.
6. M. F. Dacey, "The Syntax of a Triangle and some other Figures", *Pattern Recognition 2*, 1 (1970), 11-31.
7. J. Feder, "Plex Languages", *Information Sciences 3*, 3 (1971), 225.
8. A. C. Shaw, "Picture Graphs, Grammars, and Parsing", in *Frontiers of Pattern Recognition*, S. Watanabe (ed.), Academic Press, 1972.
9. J. L. Pfaltz and A. Rosenfeld, WEB Grammars, in *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, Washington D.C., 1969, 609-619.
10. T. Pavlidis, "Linear and Context-Free Graph Grammars", *Journal of ACM 19*, 1 (1972), 11.
11. J. E. Donar, "Tree Acceptors and Some of Their Applications", *Jounal of Computer and System Sciences 4*(1970), 406-451.
12. K. S. Fu and B. K. Bhargava, "Tree Systems for Syntactic Pattern Recognition", *IEEE Transactions C22*, 12 (1973), 1087-1099.
13. B. K. Bhargava and K. S. Fu, "Transformation and Inference of Tree Grammars for Syntactic Pattern Recognition", *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Dallas, TX, 1974.
14. J. Gips, "A Syntax-Directed Program That Performs a Three-Dimensional Perceptual Task", *Pattern Recognition 6*, 3/4 (1974), 189-199.
15. A. Rosenfeld, Parallel Image Processing Using Cellular Arrays, *Computer 16*, 1 (1983), 14-20.
16. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, 1988.
17. P. Kay and W. Kempton, "What is the Sapir-Whorf Hypothesis?", *American Anthropologist 86*, 1 (1984), 65-79.
18. R. I. Levy, *Tahitians: Mind and Experience in the Society Islands*, The University of Chicago Press, 1973.
19. B. L. Whorf, The Punctual and Segmentative Aspects of Verbs in Hopi, in *Language, Thought, and Reality: Selected Writings of Benjamin Lee Whorf*, J. B. Carroll (ed.), The MIT Press, 1956.

20. D. Gentner and D. R. Gentner. "Flowing Waters or Teeming Crowds: Mental Models of Electricity", in *Mental Models*. D. Gentner and A. L. Stevens (ed.), Erlbaum. 1982.

21. G. Lakoff. *Women, fire, and dangerous things*. The University of Chicago Press. 1987.

22. R. L. Glass. "Real-Time: The "Lost World" Of Software Debugging and Testing". *Comm. ACM 23*, 5 (1980), 264-271.

23. R. A. Foster. "It's Time to Get Tough About Computer Software". *Quality Progress 11*. 8 (1978). 10-12.

24. M. Schindler. "Software productivity needs tools for improvement". *Electronic Design 28*. 17 (1980). 45-48.

25. R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Prentice-Hall. Inc.. Englewood Cliffs. NJ. 1983.

26. R. E. Griswold. J. F. Poage and I. P. Polonsky. *The SNOBOL4 Programming Language*. Prentice-Hall. Inc., Englewood Cliffs. NJ. second edition. 1971.

27. W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag. New York. 1981.

28. R. E. Griswold and M. T. Griswold. *The Implementation of The Icon Programming Language*. Princeton University Press, 1986.

29. R. C. Holt. G. S. Graham. E. D. Lazowska and M. A. Scott. *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley. 1978.

30. E. W. Dijkstra, "Cooperating sequential processes", in *Programming Languages*. F. Genuys (ed.), Academic Press, 1968.

31. D. A. Nowitz and C. S. Gutekunst, Installation and Operation of UUCP 4.3BSD Edition. in *Unix System Manager's Manual*, USENIX, 1986.

32. G. E. Burch and T. Winsor. *A Primer of Electrocardiography*. Lea and Febiger. fourth edition, 1960.

33. F. F. Rosenbaum, *Clinical Electrocardiography*, Oxford University Press. 1950.

34. R. P. Grant. *Clinical Electrocardiography: The Spatial Vector Approach*. McGraw-Hill Book Company. Inc, 1957.

35. J. R. C. Jr., F. M. Nolle and R. M. Arthur. Digital Analysis of the Electroencephalogram. the Blood Pressure Wave. and the Electrocardiogram. in *Machine Recognition of Patterns*. A. K. Agrawala (ed.). John Wiley & Sons , 1977.

36. M. J. Bach. *The Design of the Unix Operating System*. Prentice-Hall. Inc.. Englewood Cliffs, NJ, 1986.

37. D. Knuth. "Literate Programming", *Computer J. 27*, 2 (May 1984), 97-111.

38. D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley. 1973.

39. W. M. McKeeman, J. J. Horning and D. B. Wortman. "*A compiler generator*", Prentice-Hall. Inc.. Englewood Cliffs. NJ. 1970.

40. D. R. Hanson. "Storage Management for an Implementation of SNOBOL4". *Software—Practice & Experience 7*(1977), 179-192.

41. R. Fenichel and J. Yochelson. "A LISP garbage-collector for virtual-memory computer systems". *Comm. ACM 12*, 11 (Nov. 1969). 611-612.

42. H. G. Baker Jr., "List Processing in Real Time on a Serial Computer". *Comm. ACM 21*, 4 (Apr. 1978). 280-293.

43. S. R. Steinberg. *Language and Architecture for Parallel Image Processing*. North-Holland Publishing Company. 1980.

44. S. Yalamanchili. K. V. Palem. L. S. Davis. A. J. Welch and J. K. Aggarwal. Image Processing Architectures: A Taxonomy and Survey. in *Progress in Pattern Recognition 2*. L. N. Kanal and A. Rosenfeld (ed.). Elsevier Science Publishers. 1985.