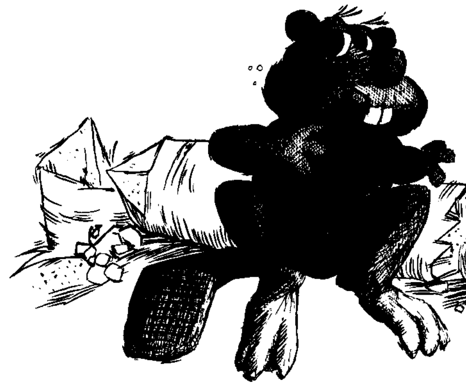
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



A General System for Managing Videotex Information Structures

David Adrien Tanguay

Data Structuring Group
CS-86-23

June, 1986

A General System for Managing Videotex Information Structures†

David Adrien Tanguay

University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

Most videotex systems organize their information in a hierarchy, also called a **tree**. The tree is an inadequate structure, however, consequently several systems augment it with **cross-links** or some form of **keyword access**. A considerable amount of research has been doneon improvements and additions to the tree, but relatively little work has been devoted to searching for and testing alternatives. Some basic criteria for the design of videotex structures are discussed. Several existing structures and auxiliary access methods are presented in light of these criteria.

To facilitate research in new structures, a system called **Stred** was designed and implemented. Stred facilitates the design and implementation of videotex structures and databases: a structure is described with a **structure program** written in Stred's **command definition language**, and this program is interpretted in an environment provided by Stred. Stred, its command definition language, and experiences with Stred, are described.

† This work was supported in part by grant G1154 from the Natural Sciences and Engineering Council of Canada.

Table of Contents

1. The Videotex Structure Problem 1
2. Structural Issues 4
    - 2.1. Navigational Structures 4
    - 2.2. Personalization 6
    - 2.3. Separation of Structure from Content 7
    - 2.4. Structural Heterogeneity 8
3. Some Information Structures 9
    - 3.1. The Tree 9
    - 3.2. The Tree with Cross-links 10
    - 3.3. The Multiple Parent Hierarchy 11
    - 3.4. Relative Page Labels - The Grid 12
    - 3.5. Multiple Context Trees 13
    - 3.6. The Relational Schema 14
4. Auxiliary Structures and Access Methods 15
    - 4.1. Direct Addressing 15
    - 4.2. Keyword Search 15
    - 4.3. Linear List 16
    - 4.4. Multi-Menus 16
5. Stred - A System to Design Browsable Structures 18
    - 5.1. A Model of Browsable Information Structures 18
    - 5.2. Hence, Stred 19
    - 5.3. An Overview of Stred 19
    - 5.4. The Command Definition Language 21
        - 5.4.1. Variables 21
        - 5.4.2. Program Organization and Operations 22
    - 5.5. Experience with Stred 26
        - 5.5.1. An Interface System 26
        - 5.5.2. Structures 28

5.6. Comments on Stred 29
5.6.1. General 29
5.6.2. The Model 29
5.6.3. Locales 30
5.6.4. Separation of Structure from Interface 31
5.6.5. The Command Definition Language 31
5.7. Summary 32
6. References 34
Appendix: Stred Manual 38
Appendix: Sample Programs 51
Appendix: New Stred Manual 74

1. The Videotex Structure Problem

Videotex presents its users with a library of information. It has the potential for rapid information perusal and retrieval by utilising the data processing power of a computer. One of the major obstacles towards fulfilling this potential is the human-machine interface. A powerful videotex system is not of much use if it is so unwieldy that the user dreads a session.

The user interface includes more than the human-machine communication protocol: it also includes the underlying organization of the information which the human must understand to use the system effectively. I will call the information organization the **structure** or **access method** (a more specific definition will be given in section 2). The actual information contained will be called the **data**.

The structure chosen for a videotex system must facilitate access to the data. To accommodate diverse user needs, the system should support more than one form of access. In a traditional library the most common form of information retrieval is **direct search** [Schabas, Tompa 83]: the user knows what he wants and uses the library structure (e.g., the card catalogue) to find and retrieve the data. We have several centuries of experience in designing libraries to suit this activity, and these designs have been transported to computer systems. In videotex, however, a common form of access is **browsing** [Godin, Saunders, Gecsei 84] [Schabas, Tompa 83]: either the user wanders through the information looking for anything interesting, or else the user needs specific data but is not sure exactly which data (but will recognise it when found). Direct search is also important in videotex, but structures that suit direct searching do not necessarily support browsing.

In addition to this problem of cross purposes, videotex imposes other constraints upon its structures. The implementation of a structure must be efficient. The user quickly gets impatient if he sees nothing happening. This leads to a lack of motivation to use the system [McCracken, Acksyn 84] [Newell 77].

The typical videotex display is small, limiting the amount of structure that can be displayed (relative to its entirety) [Engel, Andriessen, Schmitz 83] [Robertson, McCracken, Newell 79] [Schabas, Tompa 83]. This gives the user either a very narrow view of the structure or else a wider but less detailed view. This limitation is compounded by a restriction on the amount of new structural information that the user can assimilate at one time [Schabas, Tompa 83]. Too much information will swamp the user — overwhelm his perceptions — resulting in the retention of relatively little information.

The physical storage medium of the data also imposes constraints. It costs money to store information, and so the amount of information that can be stored is limited. This implies that data duplication should be avoided whenever possible [Hitchcock 81] [Tompa, Gecsei, Bochmann 81]. On the other hand, in some instances, a structure may be easier to design, and the database easier to maintain, if duplication of data is allowed; without duplication, a cross-link mechanism must be built into the structure.

Videotex users span the continuum of computer literacy. The access method must be easily usable by a naive user (even a child) and yet provide an experienced, sophisticated user with sufficient power [Newell 77] [Rhein 83]. A powerful system must hide any additional complexity from the naive user.

What structure should be used to provide access to data? The answer depends upon the purpose for the structure, the users of the structure, and the database itself to a limited degree. The intended use of the structure — the amount of browsing versus direct search — may determine if the structure should be **navigational** (see section 2.1) or not. The intended users may be unfamiliar with the system or they may be experienced with the system (or at least with computer systems in general); they may be one-time users, casual users, or frequent users; they may have a mixture (in varying ratios) of these qualities (e.g., many experienced computer users accessing the system casually mixed with many naive computer users who use the system frequently, with both groups being experienced with the system itself). The complexity and power of the structure provided must suit its users. The size and degree of generality of the database may impose structural constraints (e.g., a very large, general database would exclude a simple **grid**, described in section 3.4, as its structure).

The area of databases has been rich in research and development, but mostly for direct search uses. This development has led to systems that can be used effectively by both naive and sophisticated users, on both specific and general data. The development of videotex, where browsing is a major use, has increased the need for effectively browsable structures.

So far, most videotex systems (and other data browsing systems) have used the **tree** (see section 3.1) as their structure, often with **cross-links** (see section 3.2). The tree, however, is not always a good structure for this purpose. Unfortunately, relatively little work has been done to find a replacement. I have developed a system, **Stred**, which is intended to be a system that allows its users to design their own structures, and to implement these structures quickly and easily. This encourages the design of novel structures, leading, it is hoped, to some structures that are superior to the tree-based structures currently in use.

ZOG [McCracken, Ackscyn 84] [Newell 77] [Robertson, McCracken, Newell 79], an information system developed at Carnegie-Mellon University, uses network based structures. Each ZOG node, or **frame** in the ZOG vernacular, is constructed independently of the structure. This means that for any restrictive network, such as a tree, the restrictions are enforced only by convention (on the part of the network constructor). Also, there is no means of conveying information from one node to another. There are no state variables: variables which can contain node-independent information, such as the experience level of the user. It would be very difficult to implement a multiple context forest (of trees) in ZOG because of this lack of state variables (which could be used to indicate the current context).

The French CCETT Teletel/Star videotex system [Henriot, Yclon 79] attaches a program to each node which can be used to control navigation (as well as other actions). There are limited state variables (an array of boolean flags, and stacks of page identifiers). This is more flexible than the ZOG system, but once again structure is dependent upon convention: each page has a unique

program, written by the information provider at the time of page creation. This precludes the information provider from being a casual or naive user.

Stred is designed to be a system to implement a wide range of structures: the information provider (not necessarily the person who designs the form of the structure) does not need any intimate knowledge of Stred. The structural type is enforced by Stred, and does not rely on a node by node convention on the part of the information provider. Stred allows the application of conventional computer science data structures to videotex facilities [Tompa, Gecsei, Bochmann 81]. A Stred structure, therefore, can be seen as an example of a conventional abstract data type [Linden 76].

A key goal of Stred is **personalization** (see section 2.2). The guiding philosophy is that any aspect of structuring, even the database itself, that could be personalized should be personalizable. The database user can make private alterations to a public database, including the alteration of the data. A fluent user can even make changes to the structure itself.

Another important goal of Stred is **simplicity**. The design and implementation of a structure should be as simple as possible; the expertise required by the structure designer should be minimal. It is hoped that a typical user of the database will be able to design a structure, or, at the least, to make modifications to an existing structure.

Stred structures should be browsable, since they are for use in videotex systems. The spectrum of possible structures, however, is too broad to meet the simplicity goal of Stred, so limitations must be imposed on the types of structures Stred can create. These limitations manifest themselves as a generic model of browsable structures within which Stred works. The model described in section 5.1 is a template for the accepted structures.

Aesthetic considerations were ignored in the design of Stred: the design and implementation of structures is the goal; the particular user interface to operate upon the structure is another problem. However, since any system must have an interface, Stred is designed to be operable by another system whose function is to beautify Stred output and input: to make Stred user-friendly [Fraser 79].

Section 2 will outline some design criteria for structures. These criteria form the basis for the design of Stred. Sections 3 and 4 will look at some structures and access methods. These sections display the range of structures that Stred should be able to handle. Section 5 will informally describe and discuss Stred. The appendix contains a few examples of Stred structure definitions and a formal Stred manual. A manual for a new version of Stred is also contained in the appendix.

2. Structural Issues

In this section I will discuss some characteristics of structures. First, however, it is necessary to define what a structure is.

A **structure** is an **abstract data type**. It is used to manage a collection of variables, having types defined by the structure, via a set of functions which manipulate these variables. One of the parameters to these functions (it may be an implicit parameter) must be a variable of type **database**, which is the principal type defined by the structure. Henceforth, any instance of a structure will be called a **database**; the term **structure** will refer to the organizational schema used to construct a database. A database can be described by some ordered list of function invocations (although this list is not necessarily unique).

The functions which define a structure can be classified as either **maintenance** functions or **perusal** functions. A maintenance function has the effect (or side-effect) of changing the database in some way. A perusal function leaves the database unaltered, although other variables may be changed.

2.1. Navigational Structures

Navigation through a database has been likened to a spaceship travelling about the cosmos [Lochovsky, Tsichritzis 81]. Navigation is described as a "step-wise approach to information retrieval" [Raymond 84]. This description appeals to intuition, but is, strictly speaking, a rather broad definition: what kind of access cannot be seen as navigational? Even a relational database (see section 3.6) can be considered navigational: access to desired data comes after a series (possibly of length one) of increasingly refined queries, where each refinement can be considered a step. I will refine the definition of navigational to make it less all-inclusive.

A key requirement for a structure to be navigational is for the structure to present to the user a concept of **position**. A position in a database is a value of a special state variable which is used as a parameter for the structure's functions. All retrievable atoms of data (in the database) are then associated with some position. In other words, the selection of data is equivalent to (accomplished by) the positioning of the user within the database: the data that a user can retrieve is determined by the user's position.

A function which alters the value of the position variable is said to be a **traversal** function. The set of possible positions in a database (under a navigational structure) are determined by the value of the database variable, and are thus altered only by maintenance functions. The position values assigned by any traversal function (under a given database) must be closed with respect to this set. Also, the value assigned by a traversal function must only depend upon the values of the position and database variables, and be independent of the values of any other state variables. Let a **route** be a pair of positions, (*source, destination*), such that there is some traversal function, under some database value, which will change the position from *source* to *destination*. Given the set of all meaningful positions in a database, we can then generate a set of routes for that database, and this set can only be altered by a maintenance function (note that it is possible for a maintenance function to alter the set of routes without altering the set of positions). This invariance of the sets of positions and routes under a given value

of the database variable helps to enforce the concept of a stable "landscape" through which a "read only" user [Bochmann, Gecsei, Lin 82] can navigate.

A navigational structure is defined to be **spatial** if it appeals to the user's "intuitive notions of dimensions, distance, and nearness in the physical space" [Godin, Saunders, Gecsei 84]. This means that the positions and routes of a structure fit into some metric space, where distance can be related to number of traversal function invocations. To appeal to a user's intuition, however, the number of dimensions of the metric space should be restricted to three or fewer, since most users would have difficulty understanding a larger dimensional space.

Some structures may have subsets that are navigational, but include functions that violate the navigational conditions. A structure is **navigationally-based** if it contains a navigational subset and if all databases that could be created with the structure could also be created with the navigational subset. Henceforth any reference to a navigational structure will implicitly include navigationally-based structures.

Navigation is an everyday experience. The user, therefore, will have a ready, natural understanding of a navigational structure and will need to learn only the system's notation for already familiar operations — the user does not have to learn new search strategies. A spatial structure increases the user's ability to visualize position in a database and motion through it, and, importantly, to remember places in the database [Engel, Andriessen, Schmitz 83] [Newell 77]. This helps prevent the user from becoming lost in the database (see below).

A navigational structure readily allows the exploration of new information (browsing), especially if the structure is spatial [Engel, Andriessen, Schmitz 83] [Godin, Saunders, Gecsei 84] [Schabas, Tompa 83]. The available navigational routes from any position in a database provide the user with alternatives from which to select. This is easier than having to construct a search strategy creatively, provided that the number of alternatives is not too large. By restricting the alternatives for the next data selection, a navigational structure guides the user through a database. New information can be found by exploring unfamiliar routes. The user can apply experience and strategies from the real world — exploring a new city or shopping mall — directly to the database.

One of the problems with exploring a navigational database, as with a new city, is becoming lost [Godin, Saunders, Gecsei 84] [Newell 77] [Raymond 84] [Viszlai 81]. This may be caused by a drastic shift in context [Godin, Saunders Gecsei 84]. The user is suddenly removed from previous surroundings and may have difficulty returning. Drastic context shifts, however, are indicative of a strain on the spatiality of the database. In the real world it is not (usually) possible to find oneself suddenly transported from one locality to another. In a database this is very easy to arrange, either by accident or design, and the user may not immediately recognise the shift since the new context may have a valid interpretation in terms of the old context, and thus may not be recognised as being new. User ignorance of the new context may preclude recognition. For example, "Edberg", "Willander", and "Borg", under the category "Swedish", might be construed to be cities by someone unfamiliar with the tennis scene. A carefully constructed database may eliminate most of the potential for this disorientation, but it is unlikely to eradicate the problem since what may seem to

be a clear, natural shift in context for one user (or the database builder) may be unfathomable to another because of different experiences and associations.

Finally, the limited output capabilities of videotex allows only a small section of a navigational map to be shown at one time. There is little, if any, room on the display for peripheral vision, by which the user can orient the current position within the entirety of the structure [Godin, Saunders Gecsei 84] [Robertson, McCracken, Newell 79]. This tunnel vision makes it more difficult for the user to determine a context for any database position. This problem may be solved by allowing the user to "zoom out" to get a broader, but less detailed, overview.

2.2. Personalization

To build any database the information provider must choose criteria by which the data can be partitioned to conform to that structure. If the data is sufficiently general, then the information provider must, at some point, make arbitrary structuring decisions which, although reasonable, will be confusing for at least some users [Engel, Andriessen, Schmitz 83] [Godin, Saunders, Gecsei 84] [Leclerc, Zucker, Leclerc 82] [Newell 77]. For example, consider the assignment of keywords to a document, written for aspiring pilots, about the theory of flight. The average pilot would prefer keywords such as "lift" and "drag". A pilot who has an engineering background, however, might prefer "fluid dynamics" since the engineer would already be familiar with the subject under that term. All reasonable keywords could be assigned, but this could easily result in an excessive amount of keywords, and it also demands that the information provider has the diverse knowledge necessary to determine all such reasonable keywords. It is also possible that some users would prefer keywords that are judged by the information provider to be unreasonable.

Not only the vocabulary, but also the preferred organization for a particular user is dependent upon that user's experience [Engel, Andriessen, Schmitz 83] [Newell 77] [Rhein 83] [Schabas, Tompa 83]. A stranger to the information (e.g., a student) is unlikely to organize information in the same way (under some particular structure) as one familiar with the data would (e.g., a professor). Organization may also depend upon the user's goals and expectations [Newell 77] [Robertson, McCracken, Newell 79] [Raymond, Canas, Tompa, Safayeni 86]. For example, a scientist is likely to prefer a different organization of European history than would a literary scholar (even assuming they are equally well versed in the other's field of expertise). This type of organizational conflict has been called the **naming problem** because it is often apparent when a user has to select an item from a menu [Raymond, Tompa 86]. We must bear in mind that both a user's experience and goals will change over time [Rhein 83]. Thus, even for a single user, one organization, or even one structure, may not be suitable forever.

A different structural problem arises when a user wants to add information to a public database. For example, a user might want to include upcoming weddings and other parties in the same locality as public entertainment (concerts, shows, community events). A user might also want to integrate personal knowledge with the data in the database. This does not necessarily involve a change in the organization — a change in the content of one specific database document may be sufficient.

The ability to personalize a database will solve the above problems. This personalizing must be implemented in such a way as not to change the public database for other users. Ideally, the distinction between the public and private elements of the database should be transparent to the user [Robertson, McCracken, Newell 79]. This also allows a pleasant integration of a public database with more personal services, such as electronic mail and forums (electronic bulletin boards): these services can be added to the database as private additions (electronic mail, for example, could be retrieved in the same way for all users, but they only would get their own mail since the database addition to implement mail would be a personal addition for each user).

2.3. Separation of Structure from Content

Many current videotex systems have their structural information embedded within the content of the database. In Telidon, for example, the menu is an integral part of the data [Godfrey, Chang 81]. A separation of the structural information from the content, in navigational structures, has several advantages [Leclerc, Zucker, Leclerc 82] [Raymond 84].

Personalization of databases is much easier with structure/content separation. Modifications to the organization, including errors, cannot corrupt the content. Fewer storage resources are needed since only the organizational changes must be noted: there is no need to copy and alter the content as well. Similarly, a content update cannot damage the database's organization.

Separation allows more flexible structures. The structural information is more easily retrievable, and since it can be retrieved independently of the content, which can be very large in terms of machine resources, more structural information can be held and manipulated by the system at one time. For example, separation facilitates dynamic menu construction (see section 4.4 on multi-menus) since the menu display sub-system is not bound by the content.

In some applications (structuring systems), separation implies that data does not have to be modified to be incorporated into the system [Leclerc, Zucker, Leclerc 82]. A flexible structuring system can have this property in general. This allows the content to be created independently, thus eliminating the need for special (to the videotex system) editors. This data can be updated by a system non-specialist (i.e., a librarian is not needed to oversee content modifications).

Apparent content duplication is feasible in a separated system. In an integrated system, putting the same content in two (or more) places in the database requires duplication of the content. A separated system only requires extra structural information, which is usually much smaller than content. When multiple copies exist, there is the danger of inconsistent updating of that content. In a separated system, separate copies can be implemented as multiple references to the same content, thus reducing the update problem [Hitchcock 81].

2.4. Structural Heterogeneity

A **homogeneous** structure is one that follows one structural schema throughout. Alternatively, it is possible to create a **heterogeneous** structure wherein some parts follow one schema, and other parts follow another (see below for an example), or, two or more structures can exist in parallel (see section 3.5 on multiple context trees and section 4 for examples).

A **mixed** structure (which is an example of a heterogeneous structure) allows each subset of the information to be organized with a structure that is most suited for that subset. For example, consider a database on manufacturing. It may be fundamentally organized as a hierarchy (see section 3.1) with some subtrees replaced by a structure better suited to the data. For example, a section on automobile manufacturing could be organized as a grid (see section 3.4) with dimensions recording manufacturer and type (e.g., [**Ford, half-ton pickups**], [**Toyota, compact**]). This alternate structure may not be practical for the organization the entire database, but may work well with a subset.

Parallel structures can be used to provide scoping (i.e., zoom out — a bird's eye view of the database): one structure can organize key elements of a database — the "landmarks" — and the other structure can provide the detailed organization throughout the whole database. They can also provide an alternative structure more suited to sophisticated users without alienating the naive users [Forbes 83] [Newel 77] [Santo 83]. Parallel structures can also be designed for different search goals (see section 3.5 on multiple context trees). Parallel structures create a heterogeneous structure when there is more than one type of structure in parallel.

With mixed and parallel structures there is the danger that the user may get confused as to which structure is currently active [Schabas, Tompa 83]. The system must be designed to remind the user which structure, and which type of structure when there are several, is currently active. In multiple context trees (see section 3.5), for example, it was found that users occasionally forget which context they are in and attempt to give traversal commands to another context [Schabas, Tompa 83]. If many kinds of structures are used (especially in a mixed structure), the user should be fluent in manipulating each of the various structures. In general, this requires either greater skill by either the user (more than one structure, and the associated traversal commands, must be known) or by greater skill and care by the information provider (to provide a common interface).

3. Some Information Structures

3.1. The Tree

The original structure for browsable databases is the **tree**, also called a **strict hierarchy**. It is a navigational structure. Each position has exactly one route to a **parent** position, and one route to each **descendent** position (also called a **child**). There is one special position, the **root**, which has no parent. A position with no children is called a **leaf**. The root, therefore, is the top of a hierarchy, with the leaves being on the bottom of the hierarchy. The user browses through the tree by moving from a position to one of its children or to its parent. Figure 3.1 shows an example tree.

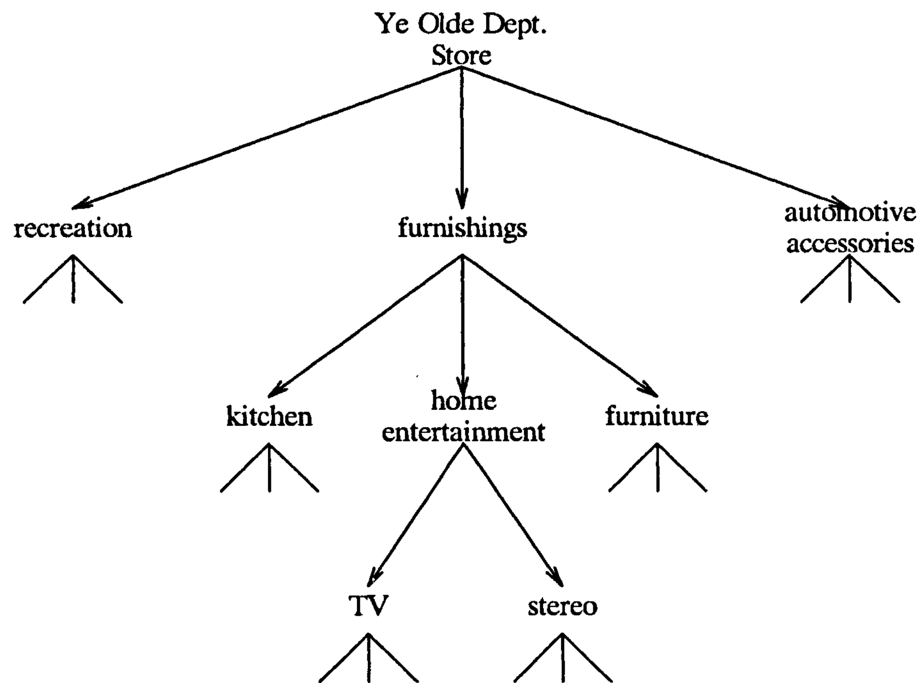


Figure 3.1: An Example Tree

Trees are simple and efficient to implement. More importantly, they are simple and natural to understand [Schabas, Tompa 83]. This is partly due to their spatiality (they can be drawn on paper), and also due to the user's familiarity with the structure from non-computer life: a user will see tree structures in chains of command (government, business, and military), in tournaments (e.g., hockey playoffs), and, of course, in the living trees they are named after. Also, navigation through a tree is easy to present to the user in terms of a menu: the user can select from the children presented in the menu, or the user can select the parent.

There is only one parent for each position, therefore there is only one path from the root to any particular position. Furthermore, there is only one path from any position to any other position. This means that the user must back up if a mistake is made while trying to get to particular information. Backing up implies that the user was defeated by the system, which causes user frustration and discontent [Godin, Saunders, Gecsei 84].

Information often does not fit into a hierarchical organization [Engel, Andriessen, Schmitz 83] [Leclerc, Zucker, Leclerc 82] [Schabas, Tompa 83] [Tompa, Gecsei, Bochmann 81]. In the example shown in figure 3.1, where would a user find the position corresponding to "car stereos"? It might be put under either "automotive accessories" or "stereo", or even under "recreation". This is an example of the naming problem described in section 2.2.

The videotex display (and human perception/cognition) limits the outdegree at each node (i.e., the number of children at each position). For example, a tree-structured database may at some point have to partition its information into separate categories for each of the 50 American states. Putting all the states under a single position is not feasible for a menu display, although that is exactly what is desired. Instead, artificial hierarchies must be created to keep the number of children at each position relatively small. The division might be alphabetical, geographical, or historical (e.g., by date of founding). These artificial hierarchies can confuse users because they do not fit their expectations of the organization.

As the database gets large, the average length of the path from the root to an arbitrary position gets longer (assuming the outdegree is bounded). A longer path leads to a greater chance of a user error: either a cataloguing error, a typographical error or an error in the selection of a child [Engel, Andriessen, Schmitz 83] [Schabas, Tompa 83]. Errors, in turn, lead to the user becoming lost. Even without errors, the existence of long paths makes retrieval of data more tedious. This problem can be solved by adding special traversal features to the tree (see sections 3.2 and 4).

3.2. The Tree with Cross-links

A common addition to the tree is the **cross-link**. This is a route between a pair of positions that defies the hierarchical organization. For example, if "car stereos" were placed under "automotive accessories" in figure 3.1, a cross-link could be made from "stereos" to "car stereos".

Cross-links allow information that belongs in many hierarchical classes to be easily accessible from each. They can be used to provide an escape route from common selection errors [Raymond 84]. They cut down retrieval time by allowing more paths to information [Raymond 84] [Schabas, Tompa 83].

As well as preventing errors, cross-links can also help cause them [Raymond 84] [Schabas, Tompa 83]. Traversing a cross-link can result in a drastic change of context. Consider a possibly natural cross-link from "recreation" to "home entertainment". At "recreation" the user sees tennis racquets and water skis, and then at "home entertainment" sees stereos and VCRs. To some users this would be disconcerting since in many department stores home entertainment is in the furniture or camera section, and sporting goods is nestled between toys

and hardware. The result can be user confusion which, aside from upsetting the user, can result in further mistakes while disoriented. This drastic change of context is indicative of a strain on the spatiality of the cross-link structure (as opposed to the underlying tree structure). If the cross-link maintains spatiality, there is a reduced likelihood of the user becoming disoriented.

Although a cross-link might be conceptually clear in one direction, it may not be as clear when the user is backtracking across it later. Retreating from "home entertainment" to "recreation", in the above context, can be even more confusing a context switch than the original traversal. The system (or user) might not remember to back up across the cross-link, in which case a brief side excursion into "home entertainment" can leave the user far from the original path.

The concept of cross-links seems reasonable, but how does the information provider decide where to put one in a tree? Often an arbitrary decision is made [Newell 77] [Schabas, Tompa 83]. The information provider must have an extensive knowledge of the database and its contents to place cross-links effectively. Any holes in the information provider's knowledge (or intuition) could result in the lack of an important cross-link. The display limitations make the decision even more difficult: a menu choice for a cross-link takes up display space just like a hierarchical descendant, and so the information provider is usually severely limited in the number of cross-links from any position [Schabas, Tompa 83].

Cross-links do alleviate some problems with trees, but they do not solve any. They may be sufficient for a small database, but eventually, as the database grows, they are unable to accommodate the increased relationships. They are useful, however, for a user familiar with the underlying tree structure. In a system where the user creates personal cross-links, the chance of the user getting lost (due to cross-links) would be reduced.

3.3. The Multiple Parent Hierarchy

A **multiple parent hierarchy**, another navigational structure, is a directed, acyclic graph with a single source node (the **root**) [Leclerc, Zucker, Leclerc 82]. In other words, it is a tree modified so that each position (except the root) can have more than one parent, under the restriction that no cycles are thereby created (i.e., no position can be the descendent of one of its descendents). It cannot be modelled in the plane (in general), but it can be modelled in three dimensional space. A particular database may have many spatial distortions, but most of the database will be spatial, particularly in a local region (i.e., the set of routes between all positions within a few traversal steps of the current position).

This structure is more flexible than a tree, since particular data can be placed in several hierarchical categories. It is still simple to use, but the choices presented in traversing the database to the root can cause confusion and can cause users to become lost.

The multiple parent hierarchy eases the naming problem by allowing the information provider to include a position under several parents. This increases the number of paths to data, and thus improves data retrieval. The information provider, however, must still be intimately familiar with the database to

determine which nodes should be the children of a given node, and so there is still the naming problem, although greatly reduced.

The multiple parent hierarchy is a good structure for more experienced users since it provides many paths to data and a form of sideways traversal [Godin, Saunders, Gecsei 84]. This is achieved by descending to a child node, then ascending to a different parent node. Repeating this several times illustrates that this is a property not shared by a simple tree.

Godin et al. [Godin, Saunders, Gecsei 84] implemented a keyword based lattice. Their system automatically builds a multiple parent hierarchy from the data, based on keywords assigned to each document (i.e., given the documents and associated keywords, a database is built). Descending routes are labelled by the keywords. This system eases the construction burden. The information provider does not need so thorough a knowledge of the entire database: based on the assigned keywords, the system builds all the appropriate routes. It is still the responsibility of the information provider, however, to make an intelligent selection of keywords for each data object.

3.4. Relative Page Labels - The Grid

The **grid** [Tompa, Gecsei, Bochmann 81] assigns to each document specific coordinates in some conceptual n-space. Each dimension consists of an unordered set of labels (the coordinate values). The specification of a label moves the user from the current document to a document with the same coordinates except that the named label becomes the new value for the appropriate (co-dimensional) coordinate. For example, let the dimensions be "producers" (*RCA, Sony, Philips*), "product" (*TV, stereo, VCR*), and "retailer" (*The Bay, Sears, Eaton's*). If the user is at the coordinates [*RCA, TV, The Bay*] (i.e., a description of RCA TV's sold at The Bay), moving to the label *Sears* would bring the user to [*RCA, TV, Sears*]. If the user had been at [*Sony, VCR, Eaton's*], the same label *Sears* would have brought the user to (*Sony, VCR, Sear's*). The meaning of a label, therefore, is relative to the current position.

The above example does not completely specify the structure, since it ignores some difficulties: is it possible to have the same label in two (or more) dimensions? If so, what action is taken if this label is specified? What if some coordinate is not defined or unreasonable (e.g., maybe **RCA** does not make **VCRs**)? A particular system must solve these problems (e.g., do not allow the information provider to create a label in a dimension if it already exists in another).

Whether the grid is navigational or not depends on the particular implementation. If the addition of a new label to a dimension does not implicitly create all of the positions thus generated, the structure will be non-navigational since all of the positions are not contained in the database: the routes to these positions are implicitly created with the addition of the new label (so the position exists, from the user's viewpoint) but the destination position of the routes do not yet exist in the database. Conversely, a particular implementation might implicitly create the positions (when a new label is added to a dimension, all positions thus made possible are created), and so the structure would be navigational. This grid is non-spatial since the dimensions are unordered. If there are **n**

dimensions, all positions are a maximum of **n** steps from any other, thus violating any natural sense of distance.

This structure is not suitable for a large database, since the number of dimensions and the size of each dimension must grow with the number of positions in the database. It is not entirely suitable for browsing (in a large database), either, since a user is presented with too many routes from which to select. Rather than helping to guide a user, as a good browsing system should, the grid provides a vast expanse (at each position) through which the user wanders with very little guidance from the organization. Despite these short-comings, the structure can be good for a regular database (i.e., a database with a narrow range of topics) which can be a section of a larger database (as part of a heterogeneous structure).

3.5. Multiple Context Trees

Data does not often fit nicely into one tree, but it could be organized into several trees. Each tree can be separately built top-down according to different organizing biases (viewpoints), or bottom-up by grouping related sets of positions under a single parent. The bottom up method facilitates the construction of multiple trees by identifying at each stage positions with several potential parents. The different trees, or **contexts**, are then superimposed to create one structure [Schabas, Tompa 83] [Yhap 83].

For example, consider a database on personnel of a government ministry. One possible organization is by authorative position with the minister at the root, and summer students at the bottom. Another possible organization is by location: by province, county, city, and office. With a multiple context tree, both of these organizations could be made available.

The user can navigate through a context just as in a regular tree, and additionally can switch to another context when the current context is not satisfactory. The structure is then not navigational, however, since the result of the application of a traversal function at a given position depends on more than just the database and the current position — it also depends on the current context. It is possible, however, to view context as an element of position, making the multiple context tree appear navigational. The structure is not spatial since the contexts are unordered, but each context is spatial and context switching leaves the user at the same position, so the structure maintains most of the benefits of spatiality. Each context represents a different hierarchical ordering, so there is an increased chance (versus that in a single tree) that the ordering that a particular user would like is present in one context or another (i.e., one of the contexts is likely to match the user's expectations, given enough contexts).

Multiple contexts result in more paths to given positions than there are in a simple tree (which has only one path). This increases the effectiveness of a user's search for data. Also, since there can be cycles in the paths (as a result of context switches), a user may have a second chance at getting to a position that was "missed" in one context. The user may not even notice that a mistake was made and thus may avoid frustration.

The multiple context idea can be extended to other structures. A system can have multiple context multiple parent hierarchies, for example. Different structures can also be mixed (this is similar to existing systems that combine a tree structure with, say, relational database facilities).

3.6. The Relational Schema

The relational schema is a familiar, well-developed, non-navigational information organization scheme. It is very good at helping a user retrieve desired information. It is also easy to understand and use — an interface that mimics natural language is commonly used [Hitchcock 81]. Unfortunately, relations are not well suited to videotex.

A relational system assigns a set of valued **attributes** to each element of content and organizes the data into relations having identical sets of attributes. The user then retrieves data by specifying a combination of attributes and attribute value constraints. A table of elements that matches the request is then presented to the user. For example, consider a relational database on a city's commercial establishments. The user could then specify a relational query such as:

> **business**=restaurant
> **AND type**=italian
> **AND** (**area**=my_suburb **OR area**=nearby_suburb)

The user would then be presented with a list of all nearby Italian restaurants.

As the number of topics in a database gets very large and general, the number of relational attributes needed to differentiate the data increases. In a general videotex database, the number of attributes needed would become too large for a browsing user to manage comfortably. The relational database also imposes a rigid format on the content of the database, and this is not always suitable for videotex content [Tompa, Gecsei, Bochmann 81b].

A relational database is non-navigational (there is no concept of position), and it is not well suited for browsing. The entire database is directly available to the user at each stage (user query), and the user is swamped with possibilities for the next selection: the system provides very little guidance, compared with, for example, a tree [Raymond, Tompa 86].

4. Auxiliary Structures and Access Methods

This section will describe structures and access methods that are intended to be used in addition to another navigational (or navigationally based) structure (such as described in section 3). These structures and access methods aid the user in accessing data via the primary structure, but without the primary structure they are not useable.

4.1. Direct Addressing

With **direct addressing** [Gecsei 83] [Tompa, Gecsei, Bochmann 81], each position in the database is given some unique label. The user can then go directly to that position by naming that label. The label may be user-definable. This is not a structure, but a common feature added to other structures. Most tree structures allow the user to go directly to the root at any time, a limited usage of direct addressing. In the **Prestel** system [Reid 80], the user can go directly to any position of which he knows the label.

Direct addressing is not very useful for a system neophyte, but it helps an experienced user get to a frequently visited area of the database quickly. It can also help the experienced, but lost, user return to some well-known position (preferably one in the vicinity of the current search).

4.2. Keyword Search

A position in a database can be given a set of keywords that describes the content at that position. The user can then search for data by asking the system to find all positions that have some set of keywords (possibly using boolean expressions or form filling) in a given subset of the database [Raymond 84] [Santo 83] [Viszlai 81]. The search may involve several steps of refinement of the keyword set [Godin, Saunders, Gecsei 84]. The lattice multiple parent hierarchy formalizes these stages of refinement, petrifying them into a navigational structure.

This **keyword search** capacity helps the user retrieve specific data. It is less useful for browsing than for direct search [Engel, Andriessen, Schmitz 83], but it is still helpful. The computer can search a section of the database at a much greater speed than could the user, and yet still maintain a high level of confidence that all relevent information is found. The utility of this feature, of course, depends upon the information provider's selection of keywords for each position. The naming problem arises again, but with reduced impact since each node can have several keywords [Furnas, Landauer, Gomez, Dumais 82].

Another form of keyword search is based on a series of stored directories of keywords, where each keyword "points" to a set of positions [Gecsei 83] [Bochmann, Gecsei, Lin 82]. When the user then specifies a keyword, the directories are searched (according to some method) for that keyword. It may also be possible to use boolean operations on the keywords (set intersection and union resulting) [Boggild 86].

4.3. Linear List

A **linear list** structure records a sequence of positions that the user can traverse. One linear list implementation is the **FAST TRACK**† in the **Venture One**† system [Santo 83]. The user can add favourite positions (selected from the database) into the list in order. For example, in a news database, a user might put stocks, the weather report, current news, and a favourite baseball team in the FAST TRACK. The user can then view these favourite positions by simply flipping through the FAST TRACK, avoiding completely any tedious structure traversal. The FAST TRACK is used to read an ordered list of favourite pages, but motion through a list could be made similar to structure traversal (i.e., the data does not have to be read as part of the traversal), especially in a system with separated structure and data. With a linear list there is no need for a user to have to remember "direct address" labels to get quickly to a favourite document.

The time needed to get to a position in the list depends on the length of the list, so there is a maximum useful length for the list. Beyond this maximum, flipping through the list would require more effort than traversing the database. The maximum useful length is a function of the system's speed of flipping through the list, and of the individual user's preference (an unskilled user would prefer to avoid structure traversal as much as possible, and so would probably accept a longer list than a skilled user).

This is a highly personal structure that is ideal for a casual user of a videotex system. The user only has to find interesting areas once, and can then return to them quickly and with little effort.

4.4. Multi-Menus

Multi-menus [Raymond 84] are not a particular structure, but, rather, a method of traversing a navigational structure. For this description I will follow Raymond's description based on the example of a tree, although any navigational structure could be used.

The multi-menu presents several levels of the structure at once. Several levels of grandchildren may be displayed. This gives the user a larger viewport onto the database - the "zoom out" ability mentioned in section 2.1. It also increases the traversal speed, since a grandchild can be selected directly. By seeing the grandchildren, the user has a better idea of the information provider's definition of the children, and so the user can better avoid selection errors.

The multi-menu also allows the user to descend more than one of the subordinate structures. The system then dynamically combines the descendants of the selected positions into a new multi-menu. This means that the user can be positioned at several places in the database at one time, or, viewed another way, the user is at a dynamically created position. For example, if a user cannot decide which of two subtrees to descend, both can be chosen and accessed in parallel. Instead of having to choose the best of several promising candidates,

---

† FAST TRACK and Venture One are trademarks of CBS.

the user can select them all. A tree with a multi-menu addition is not itself navigational, but rather an example of a navigationally-based structure (section 2.1).

The limitations of the display device restrict the number of selections that can be made. To allow more selections, the information provider must limit the size of the labels for each position.

Error recovery with multi-menus is not significantly better than with trees, and, in some instances, may be worse since the user can become confused about the previous position. On the other hand, the multi-menus help the user avoid errors, and so there is a reduced need for error recovery.

5. Stred - A System to Design Browsable Structures

5.1. A Model of Browsable Information Structures

A Stred **structure** is an abstract data type. It is formed from a set of **nodes** and **links** (possibly dynamically generated), along with a set of rules governing traversal and maintenance. This definition gives the model a strong navigational bias, suitable for the browsing nature of videotex.

A node is just an atom of structure. It can have associated content, which is called the node's **page**. A node may also have parameters as required by the particular structure of which it is a part. A page is not specified here: it is just a name by which data can be referenced. Pages are displayed, processed, or implemented by means of **readers**. For example, the reader could be a system editor and the page could be a text file; the reader could be a chess program and the page could be a file from which to restore the current position and move; the reader could be a database program that operates on its own specific data, and the page could identify a record in that database.

To effect personalization, all access is indirect. A node's name is really only a part of its full name which is computed at the time of access (see below). A link, therefore, does not actually contain a pointer to another node, but just the name of the target node. A link also contains a title which can be used to label the link (in a menu, for example). An ordered list of **places**, called an **access hierarchy**, is searched to find a name (an example is given below). A complete name, therefore, is a place/node-name pair. Personalization is achieved by allowing the user to create and update the access hierarchy. There are three separate access hierarchies: one for nodes, one for pages, and one for page readers.

The rules for traversal and maintenance of the structure are specified in the form of a program implementing the data type. Each action on the structure is defined by a routine in the program. The program contains variables which act as temporary copies of nodes during changes to the structure. The real nodes are found, under their given names, in a place indicaed by the node access hierarchy.

An example will help to explain the operation of an access hierarchy. Suppose **[A B]** is an access hierarchy for nodes, where **B** is publicly accessible and **A** is accessible only by the user **U**. Let node **C** exist, initially, in place **B**: its complete name is then (for example) **B.C.** When **U** first accesses **C** (obtains a copy of it), it is first sought as **A.C.** This fails since there is no node named **C** in place **A. B.C** is then tried, and the read is successful. Now suppose that **U** has an updated, personal version of **C** (created by a structure command). The access (to add the personalized **C** to the structure, a write) is first tried as **A.C.** This succeeds, since **U** can write into **A.** Thereafter, any attempt to read **C** by **U** will result in **A.C. U**, therefore, has made a personal modification to the public structure. Other users (who cannot access **A**), however, who try to read **C** will still find the original, unmodified **C** in place **B.**

5.2. Hence, Stred

Stred implements the structure model in a **UNIX**† environment. Positions in the access hierarchy are implemented as UNIX directories, and standard UNIX access conventions are used to determine hierarchical access (i.e., who can access what). The name of a node (or page or reader) is its filename in the directory. Each node is a separate file, and a reader is an executable file. Each page is also a single, separate file (although the reader may interpret its contents to find access to other files). The contents of a page are completely undefined by Stred; it is up to the reader to interpret the contents (if any) in a meaningful way.

Stred uses standard input and output to facilitate its being driven by an interface process. Its output, although legible, is ugly, as is its input. All stored data, such as nodes, is human readable. This leads to excessive file access and so Stred is not as efficient as it could be.

Stred can handle more than one database at a time. Each database is defined by a **locale**, which is implemented as a file which contains all the information needed to define a particular Stred database. A locale contains the access hierarchies (three lists of directories wherein the data, the organization information, and readers can be found, respectively); the name of the **command definition file** which contains the **user program** which describes the structure; name of a structure command to execute as an initialization command, which can be used to move the user to a starting point in the database. A locale represents one personalized form of a database (which is considered a separate database in itself); each user (or group of users) that uses a personalized form of the public database — which is represented by a locale — will be represented with a separate locale.

5.3. An Overview of Stred

A structure is defined by a structure program contained in the command definition file. A locale contains all the parameters necessary to create a particular database. Stred has a set of commands to create and modify locales (see the Stred Manual in the appendix for details). The command definition file is created (and maintained) with a conventional text editor.

All locales are completely independent: no action in one locale can affect the parameters of another locale. (Since locales can share nodes and pages, however, it is possible for one locale to change the structure upon which another locale is operating.) Furthermore, within a locale, the structure program cannot change its locale's parameters (such as making changes in an access hierarchy).

The initialization routine, as specified by the locale, is used to set up the initial state of the database. This usually involves initializing some global parameters and moving to an initial node. It may also be used to synchronize or initialize the interfacing process (the **driver**). For example, this may include output to tell the driver to bind function keys to frequently used commands.

---

† **UNIX is a trademark of Bell Laboratories.**

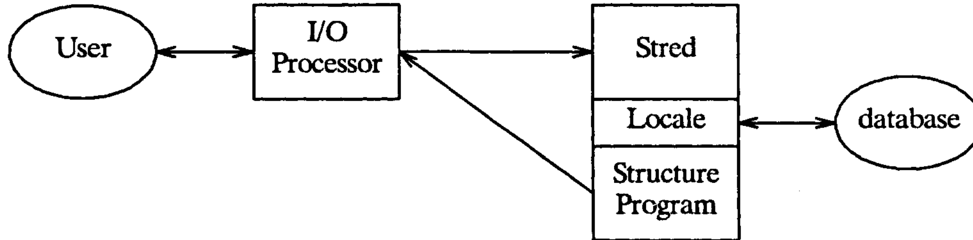


Figure 5.1: Stred System Layout

The output produced by Stred follows a consistent format. Output is initially flagged with a line starting with **#keyword**, where **keyword** indicates what type of information is to follow. For some keywords the information follows immediately and is terminated by the newline. Other information will continue onto succeeding lines and is terminated by a line starting with a **#End** (see the Stred Manual in the appendix for details on the output formats). The structure program output may follow any convention and may mimic the standard Stred output.

Stred input is in the form of a command followed by parameters. The command is first checked to see if it is a standard Stred command (e.g., a locale manipulating command). If it is not, it is assumed to be a structure program command. If it is not a structure program command, an error message is issued.

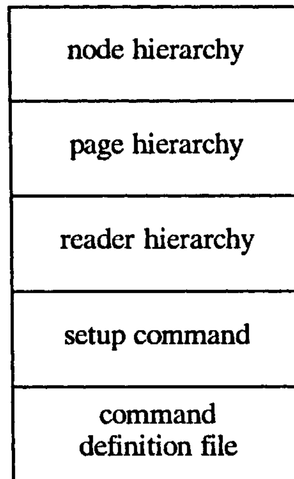


Figure 5.2: Format of a Stred Locale

5.4. The Command Definition Language

In this section I will outline the capabilities of the command definition language, with minimal discussion of details. A complete description of Stred, the Stred Manual, is included in the appendix.

5.4.1. Variables

A Stred structure program is a collection of commands and implementation procedures that can share global variables. Since only one routine is executed at a time, there are no synchronization problems. A routine can have local automatic variables and input arguments (as in C or Pascal).

The most basic data type for variables is the **string**. There are escape sequences provided to include some common unprintable characters, the string's delimiters, and the escape character itself. A string is of dynamic length.

A **link** is a representation of a directed edge. A link consists of three string fields: the **file** field which contains the name of the node being pointed to (subject to the node access hierarchy); the **title** field which contains a string to be used to elaborate the link (e.g., in a menu display); the **selector** field which is a general purpose field that may be used to contain display or selection information for the interface system. For example, a link's fields may be, respectively, "fred", "Deposits and Withdrawals by Fred (for June)", "#display reverse". The target node of the link is named "fred". A menu display would include both "fred" and the title. The selector field contains information that, in this example, indicates to the interface system that this menu entry should be displayed in reverse video.

A **page** is a representation of content. Since Stred knows nothing about the details of content, a page must also include the information needed to read the content. A page has four string fields: the **file** field, which contains the name of the content (subject to the page access hierarchy); the **title** field, which contains a description of the page for display purposes; the **reader** field, which contains the name of a process to read the content (subject to the readers access hierarchy); the **args** field, which contains any additional parameters that should be passed to the reader.

Nodes are a class of types defined in the structure program. Every node type has the following fields: the **file** field, a string which is the name of the node (subject to the node access hierarchy); the **title** field, a string which contains a description of the node for display purposes; the **page** field, a page record which represents the content associated with the node (if any). In addition to these fields, other fields may be defined that are strings, links, or lists of either (see below). A particular set of defined fields creates a particular node type that is named in the structure program. Node variables are declared by this given name; there is no generic **node** type.

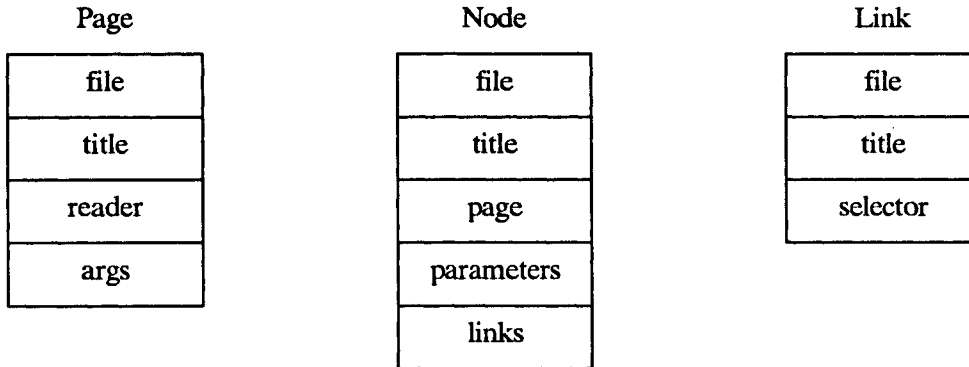


Figure 5.3: Stred Record Types

The above described types may be considered to be the basic types. In addition to these there are also **aliases** for the basic types. An alias is a pointer with implicit dereferencing. There is also a facility to create homogeneous **lists** of basic types. This is implemented within the language as an abstract data type (section 5.5.2 includes a description of lists). Lists of lists, lists of aliases, aliases of lists, and aliases of aliases are not provided.

The routines are divided into two classes: **commands** and **procedures**. Both may be used recursively. A command is a user callable routine and may only have arguments of type string. Commands, therefore, form the user-visible part of the abstract data type that is a structure. When a command is invoked by the user, missing arguments are allowed. A procedure is a routine that can only be invoked by other routines, and it may have arguments of any type. Procedures, global variables, and node definitions are the user-hidden part — the implementation details — of the structure. When a routine is invoked from another routine, all declared arguments must be passed.

Routines can have local variables (passed arguments and automatics). This may lead to a name conflict with global variables. The scope rules enforced by Stred cause local variables to be accessed in preference to global variables. Of course, no two local variables (and no two global variables) can have the same name. All routine names are global identifiers, and all routines are globally accessible. A Stred structure program, therefore, has two-level static name binding similar to that of a single C source file.

5.4.2. Program Organization and Operations

A Stred structure program consists of three sets of named fields. The first set consists of fields to define each of the defined node types. The second set is a single field, labelled with the keywords **globals**, that declares all of the global variables and routines. The arguments for the routines are also declared here. The third set has several fields, one to define each declared routine. A routine declaration field contains the local automatic variable definitions and the

execution statements (thus implementing the **rules** mentioned in section 5.2). There are several example structure programs in the appendix which illustrate the three sets of fields.

There are four basic types of Stred command definition language statements: the **simple** statement, the **repeat** statement, the **do-while** statement, and the **control** statement. If any statement fails (described below), then the routine executing that statement immediately terminates with failure. If all statements executed succeed, then the routine terminates with success. Success or failure is the only status returned by a routine.

A simple statement consists of an **expression**, an optional **success clause**, and an optional **failure clause**. The success clause is a list of statements, as is a failure clause. If the expression succeeds (see below), the success clause is executed. If there is no success clause, or if the success clause statements are all executed with success, then the simple statement succeeds and the next statement is executed. If any statement executed in the success clause fails, then the simple statement fails. If the expression fails and if there is a failure clause, the failure clause is executed, with the same results as with the success clause. If there is no failure clause, then the simple statement fails. A simple statement is reminiscent of a Snobol-4 statement with success and failure labels. Figure 5.4 is an example of a simple statement.

```
$ this is a comment
$ read in a node named 'root'
node1 <- 'root'  ?(
      $ this is the success clause
      current <- node1;  $ node assignment
):(
      $ this is the failure clause
      $ print out a error message
      % '#Error: unable to read root node\n';
      :!  $ fail
);
```

Figure 5.4: A simple statement

A repeat statement is similar to a simple statement. As long as the expression succeeds and the statement succeeds (i.e., the optional success clause succeeds), then the repeat statement is re-executed. If the expression fails, the loop is exited and the failure clause is executed to determine the success or failure of the statement as a whole (if there is no failure clause, then the statement fails). Figure 5.5 is an example of a repeat statement.

A do-while statement must have a success clause. This clause is first executed, and then the expression is executed. If the success clause ever fails then the do-while statement fails. Otherwise, as long as the expression succeeds the statement is re-executed. The failure of the expression causes the same results as an expression failure in the repeat statement. Figure 5.6 is an example of a do-while statement.

```
$ run through a 'list' of nodes
$ done is a string field of node1
\ node1.done = 'no'  ?(
      $ read in the next node
      $ next_node is a string field of node1
      node1 <- node1.next_node;
):(); $ empty failure clause
```

Figure 5.5: A repeat statement

```
$ run through a list of strings looking for
$ a match for the value of string variable search_string
?(
      $ compare and break loop if match
      l = search_string  ?( \! )  :();
)  l <- +l  :(  $ get next element of the list
      $ couldn't find it
      $ print an error message and fail
      % '#Error: could not find ';
      % search_string;  % '\n';
      :!  $ fail
);
```

Figure 5.6: A do-while statement

There are three control statements: one causes a routine to terminate with success (immediately); one causes a routine to terminate with failure (immediately); one causes the innermost executing loop statement to terminate with success (immediately). The loop termination control must be within the success clause of some loop statement (possibly nested inside enclosing statements). Figure 5.7 shows all of the control statements.

```
:!   $ fail statement
?!   $ succeed statement
\!   $ break loop statement
```

Figure 5.7: Control statements

The expressions do the actual work. Assignment, comparison, list operations, node and page operations, and output are all accomplished by expressions. Expressions operate on **objects**. An object can succeed or fail, causing the expression to succeed or fail, respectively. An object fails if it does not exist: for example, the third element of a list may be referenced, but the list may (at that particular time) have less than three elements. Additionally, an object fails if it has no value. A string has no value if it was never given a value or if it was explicitly set to have no value (the empty string is a value). The other basic types will have no value if their **file** field has no value.

Assignment itself is an expression. In general, the types of both sides must match exactly (there are exceptions decribed below). If an alias is the destination for the assignment, then the alias becomes an alias for the source (which must be a variable based object, not a constant). There is a very primitive string concatenation expression: two strings (only) may be assigned to a string, resulting in concatenation. The destination of the assignment may be one of the sources.

The only relational tests are equality and inequality of strings. If a node, page, or link is involved in a comparison, the comparison reduces to a comparison of the **file** fields.

```
$ string variable declarations
string  a, b, string1, string2;

$ assignment
a <- b;

$ string concatenation
string1 <- string2 + 'string number 3';

$ relationals
a = b;  $ equality
a ~ b;  $ inequality
```

Figure 5.8: Some Stred Expressions

Lists are an abstract data type. The first and last element can be referenced, as well as the previous and next element of any given list element. Aliases are used to march through lists: for example, each step sets the alias to the next list element. Lists contain copies of objects, not the actual objects specified for insertion. A copy can be inserted at the head of a list, or may be inserted following any referenced list element. Any referenced list element may be removed from the list. Figure 5.9 is an example that displays the list operations.

There is a special expression that finds a link that goes from one node to another. If there are more than one link between the two nodes, the first link (using the order in the node definition) is the one found. Another special operation, alluded to above, is the destruction of an object. This explicitly causes the object to have no value. It will fail only if the object initially had no value (or did not exist).

There are special expressions to perform special tasks. A node variable may be saved (written). The value of the **file** field is the name of the node, and the node access hierarchy determines exactly in which place the node should be saved. A node may also be restored, again through the node access hierarchy (via an assignment statement with a node destination and a string source). Both page content and nodes may be removed (from their permanent places, somewhere in the appropriate hierarchy). The page content may also be copied to another place (via an assignment statement with a page source and a string destination). The reader for a page may be invoked to act on the page's content.

```
$ a routine to display some list operations
#list_example {
      string list*;   $ declare a list of strings
      string alias^;  $ declare a string alias

      $ first assign the first five numbers to the list
      list <- '3';   $ places 3 at the head of the list
      list -> '4';   $ places 4 to follow the first element
      list* -> '5';  $ places 5 to follow the last element
      list <- '2';
      list <- '1';

      $ print out the list in order
      % list;      $ prints first element
      % +list;     $ prints second
      % --list*;   $ prints third last
      % -list*;    $ prints second last
      % list*;     $ prints last element

      $ remove the third element from the list
      ++list ->;

      $ walk through the list with a do-while loop
      alias <- list;   $ alias references the first element
      ?(  % alias; )  alias <- +alias  :();

      $ make the list empty
      \ list->  :();
}
```

Figure 5.9: List operations

Stred will invoke the reader itself if it is not being driven by an interface system; otherwise, Stred just informs the driver system that the computed command line should be executed. Figure 5.10 shows the special expressions.

5.5. Experience with Stred

5.5.1. An Interface System

A small system, called Scri, was written to provide an interface between Stred and an ANSI terminal. This section gives a brief description of Scri to illustrate the nature of the communication between Stred and its interface system.

Scri divides the terminal screen into five windows. The top line displays the name of the current locale. The next line displays the file and title of the current node, where the values are obtained from the last **#Locale** and **#Node** outputs issued by Stred (see the Stred Manual in the appendix).

```
#special_ops {
      $ network_node is a defined type
      $ (the definition is not shown here)
      network_node  n1, n2;
      link  ln;

      n1 <- 'root_node';  $ restore (read in) a node
      n2.file <- 'search_node';
      ln <- n1.n2;  $ finds a link from n1 to n2
      <-n2;  $ save a node
      n2->;  $ remove a node
      <-n1.page;  $ read a page
      n1.page->;  $ remove a page
      n1.page.file <- 'new_page';  $ set the new page
      n1.page.reader <- 'ed';      $ and set the reader
}
```

Figure 5.10: Special Expressions

The next line down is the command input window. It is entered by striking the escape key twice in succession or as the result of special escape sequences and key bindings described below. Characters are gathered until a carriage return, and the collected input is sent as a command to Stred. Scri then waits for the Stred input prompt before accepting any further input from the terminal.

The next line down is the message window, which displays messages from Stred. Stred sends a messages either by emitting the #**Error** output, or by a Scri-specific #**Message** line. In the former case, an audible beep is also generated by Scri.

The final lines of the terminal screen (usually 20 lines) form the display window. All output from Stred that is not recognised by Scri is paged through this window. Very little processing is done on this output, other than truncation of the output lines to fit the terminal screen, and simple pagination (i.e., when the display is filled, Scri waits for an input character before continuing to display the rest of the output).

Scri provides special escape sequences to generate Stred commands. For example, the sequence "*escape r*" causes the command window to be entered with the Stred command *Read-Locale* already entered. Stred itself can specify that Scri is to set single key bindings for user commands. For example, suppose Stred has output the lines

    #Bind 1 cd 1$
    #Bind g goto_child

Now if the user enters a "1", the command line "cd 1" is immediately sent off to Stred (the dollar symbol expands to a carriage return). If the user enters a "g", the command input window is entered with the string "goto_child" already typed, and Scri awaits further input to complete the command line (up to a carriage return).

Scri is a very simple system. There are many ways to extend it, such as increasing its control of the display window. For example, it could accept menu entry directives from Stred and then format this information into a nice menu in the display window. Special processing could also be added to the command input window to reduce typing, such as having some escape sequence expand to the current node name. The #Bind mechanism is sufficiently flexible, however, to enable Scri to provide an acceptable, although still somewhat crude, interface to all of the structures implemented so far.

5.5.2. Structures

A tree, a multiple parent hierarchy, a general network, and a grid have been implemented with Stred. All of these were simple to implement, but they are all very simple structures. The structure programs for some of these are in the appendix.

The main discovery from this experience was the ease with which changes could be made to a structure. Existing commands could be changed to operate in slightly different ways. This made the decision to choose between some design alternatives easy: just try one way for a while, then the other, and choose the favourite. For example, consider the removal of a child in a tree. What happens to the children of the child (the grandchildren)? There are several possibilities: they could be removed from the tree, they can be promoted to children of their grandparent, or they could be placed in a predefined orphanage. With Stred it was easy (for a user fluent with the command definition language) to change the structure to implement any of these schemes (in the first program in the Sample Programs appendix, the routines *_check* and *rm_only* promote grandchildren, whereas *_geno* and *rm* remove the subtree — note the shared code between *rm* and *rm_only*). Another example is the creation of a node in a multiple parent hierarchy. The node is first declared as a child of some other node. The problem then, is to specify other parents. One way is to have the user explicitly specify a new parent by its name. Another way is to select a node and traverse the structure adding parents to the selected node. The reverse can also be done: select a node and traverse the structure adding children. In the multiple parent hierarchy I designed, the third method was first used, but was later changed to the second method. Another simple example applies again to the multiple parent hierarchy. After a structure traversal, should the new set of parents be displayed (with the new children), or should the structure wait for a user command? Structural refinements like these are easy to make with Stred. These refinements can be made by a user who does not like the method used by the structure designer.

A new command can be easily added to perform some special modification. The new command may be a special command only needed by one user, and may not warrant inclusion into the structure for users in general. For example, this thesis was created using Stred with a tree structure. The leaves of the tree contained the sections, and the internal nodes were sections that had subsections. All went well until a hard copy was desired. In order to produce a hard copy, I wrote a command that traversed the tree and read all the pages, thus concatenating the sections into a whole. Headings, taken from node titles, were inserted as well. The concatenated output was then suitable for shipping through the text formatter to the printer (which could have been done by the

traversal command, too). This command would not be needed by many users of the tree structure, especially since the headings are dependent upon the typesetter (and macro package) used. A general tree traversal command might be useful enough to include as part of the tree structure, but the specific operations to be done in the traversal would vary from application to application, and so some small changes would have to be done by each user.

5.6. Comments on Stred

5.6.1. General

Most of the general goals of Stred were satisfactorily met. Stred allows the construction of structures, although not as easily as originally desired. The ability to create personal structure commands and the illusion that the user owns the database make for a comfortable environment for the user.

Stred's main failings are in the details of the implementation of the model — the command definition language in particular. It was intended to be usable by a person with little programming experience, but the complexity of the structure design problem forced the language to be too complicated for a non-programmer, and yet it is too limited for a programmer. The efficiency of Stred, in both time and space, is too poor for Stred to be used as part of a real videotex system, but this was expected.

5.6.2. The Model

The structural model used by Stred was adequate. It is sufficiently flexible to accommodate a wide variety of structures, and yet simple to learn.

The highlight of the model was the access hierarchy. It is very easy to understand, and yet it is powerful enough to create a flexible access system. It is well suited to the implementation of invisible personalization. It does, however, have a problem with database changes: if a user makes a private change to a public database (thus creating, in effect, a private database), and the public database is subsequently changed, the private database becomes outdated. It is even possible that the private database will be invalid (under the given structure). One way to alleviate this problem is to have Stred give a warning when the accessed entity is not the most recently updated.

The abstraction of content (pages) allows Stred to be flexible (recall section 2.3 on structure/content separation), but it also limits Stred. A structure may often be guided by its contents. Since Stred cannot access any information in the contents, information that could be used to build a database (e.g., keywords) must be duplicated, almost certainly laboriously, by the database builder. There is a trade-off here between content flexibility and structural flexibility. A solution might be to allow Stred to have content types: one type of content would be undefined and behave like content does now; other types of content would be recognised by Stred, and the information contained in these types could be directly manipulated by Stred. This would allow Stred to implement relational structures, and to simplify keyword-based access structures. The cost would be increased complexity in Stred, as well as a need for support software (e.g., a special editor for Stred content files).

The content abstraction allows Stred to implement **gateways** easily. The page reader is set to be the foreign system, and reading the content effects the transfer through the gate. An interesting scenario has Stred itself being the reader. This can be used to implement a heterogeneous system by a user with little expertise with the command definition language. For example, one node of a multiple parent hierarchy could have Stred as its page reader and a Stred locale, implementing a grid, as its page. This technique can also be used to combine independent Stred structures (e.g., by different information providers) into one super-structure.

Stred has no relational capabilities. There should be some facility for them, even if in some limited form. Keyword search and form-filling would be a valuable addition to Stred. This would increase the complexity of the Stred structural model, but it would probably be worthwhile. Such a feature might be added by using some type of Stred recognised content, as mentioned above.

5.6.3. Locales

A Stred locale is a convenient way to package a database. It is concise and simple. Despite this simplicity, there are some outstanding questions about its potential use.

Currently, Stred allows more than one database to be active (i.e., a Stred session may involve many locales, and, therefore, databases). Is there any reason to allow multiple locales? A naive user would only be confused by the concept of multiple locales. An experienced user might use them to implement parallel traversal of a database, but the effectiveness of this is limited because there is no way to communicate between locales. For example, to fork off a parallel search at some node the user must activate the new locale (on the same database) and traverse it to the current position. This work overhead probably neutralizes any gain from the parallel search. The user may also get lost in the context switch. Even if this capability was desired, it is probably better implemented as a structure onto itself (i.e., all done in a single command definition file).

The initialization of a locale is very primitive. Stred only allows a routine to be called with no arguments. It would be better either to allow the locale to contain its own setup routine, or to allow arguments to the initialization routine. This could aid the naive user: a pre-defined set of slight personal modifications could be made without having to leave Stred to change the command definition file. I think it wise, in general, to insulate the database user from the command definition file.

A related problem is the insulation of the locale from the structure program. A structure command cannot, in any way, change a locale. An intelligent structure might want to adjust some default global parameters to suit the observed usage by a particular user. This could be accomplished in a limited way by the up-keep of a special node that records global parameters, but it would be better if the structure program could change the initialization routine called by the locale file (especially if the above initialization scheme were implemented). There might be some use for the manipulation, by the structure program, of the

access hierarchies, but I think the complexity and danger of such scheming would create more problems than they solve.

5.6.4. Separation of Structure from Interface

The separation of Stred from any human interfacing responsibilities made the design of Stred much easier. This also increases the range of Stred applications: the interface can be changed to suit a variety of structures and a variety of input and output hardware. This suits Stred's purpose (the exploration of new structures), but, still, an interface is needed for any practical testing.

A suitably flexible interface system could handle a very wide range of structures and hardware, but any particular instance requires some degree of communication between the Stred structure program and the interface system. The result is a protocol between the structure program and interface system where the interface communicates with the structure program by command invocations and the structure program communicates with the interface system by the output of proper textual sequences (i.e., interface commands). This protocol can result in the non-portability of the structure program between various interface systems, since different interface systems will have different protocols. It may be possible to design an abstract interface model that would suit all needs, both hardware and structural (i.e., formatting of structure information), but I think this is unlikely. A good model should be able to handle a wide range of hardware and structures, however, so a small set of models (each with its own protocol) might be feasible. A Stred structure could be made portable among all of these models by having an internal switch and by having all protocols available by changing the value of this switch.

5.6.5. The Command Definition Language

The greatest weakness of Stred is the command definition language. This reflects the emerging realization — during design, implementation, and practice — that the complexity of structure design is too great to allow a naive user to be able to design and implement a structure. The result of my under-estimation is a lack of orthogonality and a dearth of capabilities.

Statements (expressions, objects, and routines) are driven by success and failure, reminiscent of Snobol and, to a lesser degree, Prolog. The original intent was to have a command be a short sequence of declarative statements, where the failure of any would cause the failure of the command. In certain cases, of course, recovery from failure was desired. To reduce the complexity of the logic for some commands, the success clause was added. The loop statements were added much later. It soon became apparent that the language would be too complex to be easily mastered by a non-programmer, but the success driven nature remained. It seems to fit the nature of the application well: even though the logic may often get involved, it still usually comes down to a sequence of operations, the failure of any voiding the entire command.

There are too few operators and types. One of the biggest problems with structuring is finding a name to identify each node uniquely. In a multiple parent heirarchy, for example, it should be possible for different nodes to have children of the same name. The ability to change the set of parents of a node,

however, prevents identical names from being resolved by context (i.e., the path from the root, which can be used in a tree structure to identify each node uniquely). There is probably a devious method whereby Stred, as it exists now, could distinguish such nodes, but it would require more expertise, by the structure designer, than should be needed. Full integer capabilities and a wide range of string processing capabilities would simplify this problem by allowing the easy generation of serial numbers for nodes.

The pre-defined records (nodes, pages, and links) seem too contrived. A node class of records is probably useful, but the others are probably better handled by allowing the user to create general records types.

Many operations are achieved in a fashion that seems indirect. For example, the copying of content is done by assigning a variable of type page to a string. This is ridiculous. My intent was to keep the language as small as possible, but such obfuscation does not really save in language size. A richer, more explicit set of operations is needed, and, among the existing operations, more orthogonality is needed.

The syntactic style of the language is not conventional (i.e., it is not a Pascal clone). There is a small set of keywords, covering the declarations of variables, and a relatively large set of funny tokens. This reflects a BCPL-derivative background, and I make no apologies for the choice of this style. There is, however, excessive symbol overloading and context sensitivity. A larger set of tokens (either keywords or symbols) is needed.

5.7. Summary

Stred makes a very good general personal information system. It is more useful as a system for experimentation with various structures than it is as a commercially viable videotex system, but this is to be expected from a research project. Even so, Stred would be good as a small in-house documentation retrieval system.

My experience with Stred suggests that the description and construction of an information structure is too complicated to be done by a person with little programming experience. It was a mistake to try to design the Stred command definition language to accommodate such users. The language should be made more conventional; it should have more generality, completeness (e.g., integers), and orthogonality.

The model for structures is good. It is simple and yet sufficiently general to handle a wide variety of structures. The use of access hierarchies seems to be a good method of implementing personalization. The only major drawback is the complete absence of any relational capabilities, or any sort of built-in keyword capabilities. It is questionable as to whether they should be included in a browsing system, but they would greatly extend the power of Stred.

Stred has no specifications for its pages and readers. This flexibility enables Stred structures to be useful outside of the videotex domain. Document creation structures, document filing structures, and program maintenance structures are all possible applications using Stred.

I cannot consider Stred to be a final product. It does point the way for a successor system that could be truly usable. My experience with Stred suggests that the idea of a personal videotex system has merit, from the individual's viewpoint, although it is probably not feasible for a large user community with current hardware capabilities and costs (unless personal computers become so commonplace that their use in videotex can be assumed without including their purchase cost).

6. References

[Bochmann, Gecsei, Lin 82]
Keyword Access in Telidon: An Experiment
G. V. Bochmann, J. Gecsei, and E. Lin
Videotex '82, New York, N.Y., June 28-30, 1982

[Boggild 86]
The Use of Boolean Operations on Keywords in Videotex
Vilhelm Boggild
Master of Math Thesis, University of Waterloo, 1986

[Engel, Andriessen, Schmitz 83]
What, where and whence: means for improving electronic data access.
F. L. Engel, J. J. Andriessen, and H. J. R. Schmitz
International Journal of Man-Machine Studies, v.18, 1983

[Forbes 83]
Getting to the Source
Mary Lou Forbes
Videotex '83, New York, June 27-29, 1983

[Fraser 79]
A Compact, Portable CRT-based Text Editor
Christopher W. Fraser
Software - Practice and Experience, vol. 9, 1979

[Furnas 81]
Pyschological Structure in Information Organization and Retrieval: Arguments for More Considered Approaches and Work in Progress
G. W. Furnas
Workshop/Symposium on Human Computer Interaction, Atlanta, Georgia, 1981

[Furnas, Landauer, Gomez, Dumais 82]
Statistical Semantics: Analysis of the Potential Performance of Keyword Information Systems
G. W. Furnas
T. K. Landauer
L. M. Gomez
S. T. Dumais
Bell Systems Technical Journal, 1982

[Gecsei 83]
The Architecture of Videotex Systems
Jan Gecsei
Prentice-Hall, 1983

[Godfrey, Chang 81]
The Telidon Book
edited by David Godfrey and Ernest Chang,
Press Porcèpic Ltd., Toronto, Ontario, 1981, pp. 179-190

[Godin, Saunders, Gecsei 84]
Lattice Model of Browseable Data Spaces
Robert Godin, Eugene Saunders, and Jan Gecsei
Document de Travail 157,
Département d'informatique et de recherche opérationnelle,
Université de Montréal, Novembre 1984

[Henriot, Yclon 79]
Language de Programmation des Bases de Données Star
A. Henriot and J. Yclon
CCETT Note Technique RSI/41/443/79

[Hitchcock 81]
Storing Pages: Introduction
Peter Hitchcock
in *The Telidon Book*, edited by David Godfrey and Ernest Chang,
Press Porcèpic Ltd., Toronto, Ontario, 1981, pp. 179-190

[Leclerc, Zucker, Leclerc 82]
A Browsing Approach to Documentation
Yvan Leclerc, Steven W. Zucker, and Denis Leclerc
IEEE Computer, June 1982

[Linden 76]
The Use of Abstract Data Types to Simplify Program Modifications
Thomas A. Linden
Proceedings of Conference on Data: Abstraction, Definition and Structure
ACM SIGPLAN Notices, Volume 8, Number 2, 1976

[Lochovsky, Tsichritzis 81]
Interactive Query Languages for External Databases
F. H. Lochovsky and D. C. Tsichritzis
Telidon Behavioural Research 5, Government of Canada,
Department of Communications, Behaviour Research and Evaluation,
November 1981

[McCracken, Akscyn 84]
Experience with the ZOG Human-Computer Interface System
Donald L. McCracken and Robert M. Akscyn
CMU-CS-84-113
Computer Science Department, Carnegie-Mellon University,
Pittsburgh, Pennsylvania, February 1984

[Newell 77]
Notes for a Model of Human Performance in ZOG
Allen Newell
Department of Computer Science, Carnegie-Mellon University
Pittsburgh, Pennsylvania, August 5, 1977

[Raymond 84]
Personal Data Structuring in Videotex
Darrell R. Raymond
Data Structuring Group, CS-84-7, University of Waterloo, Ontario
February 1984

[Raymond, Canas, Tompa, Safayeni 86]
Structuring Personal Databases
Darrell R. Raymond
Alberto J. Canas
Frank Wm. Tompa
Frank R. Safayeni
Technical Report, Department of Computer Science,
University of Waterloo, 1986

[Raymond, Tompa 86]
The Naming Problem in Videotex Fact Retrieval
Darrell R. Raymond
Frank Wm. Tompa
Technical Report, Department of Computer Science,
University of Waterloo, 1986

[Reid 80]
Prestel 1980
Edited by Dr. Alex Reid
Prestel, Telephone House, 1980

[Rhein 83]
User-Friendly is a Moving Target
Ezra Rhein
Videotex '83, New York, June 27-29, 1983

[Robertson, McCracken, Newell 79]
The ZOG Approach to Man-Machine Communication
G. Robertson, D. McCracken, and A. Newell
CMU-CS-79-148, Department of Computer Science, Carnegie-Mellon University,
October 23, 1979

[Santo 83]
An Eye on the CBS Database: A Case Study of Information Access Methods
Sherril L. Santo
Videotex '83, New York, June 27-29, 1983

[Schabas, Tompa 83]
Trees and Forests: User Reactions to Two Page Access Structures
Ann H. Schabas and Frank W. Tompa
Videotex '83, New York, June 27-29, 1983
also:
Trees and Forests: Experimental Results
Ann H. Schabas and Frank W. Tompa
Supplement for Videotex '83, New York, June 27-29, 1983

[Tompa, Gecsei, Bochmann 81]
Data Structuring Facilities for Interactive Videotex Systems
Frank W. Tompa, Jan Gecsei, and Gregor V. Bochmann
IEEE Computer, August 1981

[Tompa, Gecsei, Bochmann 81b]
Alternative Database Facilities for Videotex
Frank W. Tompa, Jan Gecsei, and Gregor V. Bochmann
in *The Telidon Book*, edited by David Godfrey and Ernest Chang,
Press Porcèpic Ltd., Toronto, Ontario, 1981, pp. 179-190

[Viszlai 81]
TELIDON Protocol Supporting DBMS
John Viszlai
in *The Telidon Book*, edited by David Godfrey and Ernest Chang,
Press Porcèpic Ltd., Toronto, Ontario, 1981, pp. 190-203

[Yhap 83]
Path Access Structures for Videotex
Christine B. Yhap
M.Math Essay, Department of Computer Science, University of Waterloo, 1983

Stred Manual

David Adrien Tanguay
University of Waterloo

1. The Stred System

1.1. Overview

Stred is a system that allows the user to design and implement personal information access structures for use in videotex systems. The current system runs under a UNIX 4.2bsd operating system. The design is achieved via a simple programming language that manipulates a basic model of information structure. **Stred** then interprets this language, thus achieving implementation.

Stred itself makes no concession to user friendliness, although its output is human readable and its input human operable. It is expected that a driver program will mate **Stred** output and user input with the available hardware devices to create an aesthetically pleasing, user friendly display.

A key goal of **Stred** is the personalization of public structures. This is achieved by means of an **access hierarchy** combined with separation of structure from content. The hierarchy is a list of places to search for elements of structure, or, with a separate hierarchy, of content. To access something, the first location in the list is tried. If that fails, the next is tried, and so on, until a location is found within which access can be made or with resulting failure of access. The hierarchy may be avoided: if the name being searched for begins with a "/", then the hierarchy is not employed and the name is considered to be an absolute pathname (all hierarchically accessible objects resolve to files) for that object.

Each database is represented by a **locale**. A locale contains the access hierachies, the structure design program (in the **command definition file**), and, optionally, the name of a routine (in the command definition file) to be used to initialize the database. **Stred** provides a small set of commands to create and manipulate locales. All structure-related commands are defined in the command definition file.

1.2. The Structure Model

The basic structural model used by **Stred** is that of a set of nodes interconnected by links. A **node** is an element of structure — a location in the database. A node does not represent, directly, any data content. Optionally associated with each node is a **page**, which is the data content of the database at that node.

Content is undefined by **Stred**. A page is simply a file of data which must be interpreted by a separate process. To read a page, therefore, **Stred** invokes a **reader** process with the page file as an argument. The reader process is a part of

the definition of a page. There are separate access hierarchies for nodes, pages, and page readers.

Links are connections between nodes. They operate on a by-name basis, using the node access hierarchy. For example, suppose node **A** has a link to node **B**. Then, when the user traverses a link from **A** to **B**, the actual node that is arrived at is subject to determination by the access hierarchy. It is possible to move from node to node without using links: only the name of the destination node needs be known; links are provided only as a convenient way of packaging this information.

This model places a strong navigational bias upon **Stred** structures, although certain non-navigational structures are possible. The most straight forward implementations with **Stred** are graph type structures, such as trees, directed acyclic graphs, and networks.

1.3. Commands

The **Stred** command line arguments are all taken to be locales that are to be used in the session. They are activated in reverse order so that the first given locale is active when the command prompt is first displayed.

The following commands are those accepted by **Stred**. They are described using the standard abbreviation syntax of the University of Waterloo: all upper-case letters of a command must be present, as well as non-alphabetic characters; all lower-case letters (including the underbar _) are optional. Optional arguments are enclosed in square brackets ([ ]). Where a choice of exactly one character is required, the characters are presented in parentheses and separated by vertical bars (eg. (+|-)). If an argument can be present zero or more times, an asterisk (*) is appended to the argument.

`List-Locale [locale]`
: This lists the named `locale`. If no locale is given, a list of active locales is presented.

`Goto-Locale locale`
: Switches to the named `locale`.

`Copy-Locale name`
: A copy of the current locale is made under the given `name`. **Stred** then switches to that locale.

`Read-Locale locale`
: The named `locale` is read into **Stred**. The command definition file is parsed and the initialization routine, if present, is executed. The named `locale` becomes the current locale.

`Delete-Locale [+PERManent]`

The current locale is deleted from **Stred**. **Stred** then switches to the last active locale (next in the locales list) if there is one. If the `+PERManent` option is invoked, the file containing the locale is removed.

`Edit-Locale [options]* [locale]`

Makes changes to either the current locale or the named `locale`. Options are as follows:

`(+|-) CReate`
Creates a new locale or clears an existing locale.

`(+|-) CLear`
All fields of the locale are cleared.

`(+|-) PERManent`
The edit changes are made permanent (i.e. the locale file is updated).

`(+|-) List`
Lists the locale verbosely after the edit.

`Init=routine`
Sets the initialization routine.

`-Init`
Resets the initialization routine.

`Comm_Def_File=file`
Set the command definition file.

`Director_List=(Readers|Pages|Nodes)`
Set the hierarchy to be modified (see below).

`List_ADD=dir`
Add `dir` to the end of the hierarchy.

`List_INS=dir`
Add `dir` to the head of the hierarchy.

`List_DEL=dir`
Delete `dir` from the hierarchy.

`Help-Locale`

This prints out a short list of **Stred** commands (not user defined commands).

!system command

If a ! is the first character of the line then the rest of the line is passed on to UNIX via the **system** function.

user defined commands

An unrecognised command is assumed to be user defined in the current locale. If it is not, an error message is produced.

1.4. Output

The following are the various forms of output **Stred** will produce. The user commands may produce any kind of output (even simulations of these).

Input prompt:

_>

This (with a trailing space, no newline) is the input prompt. It indicates that **Stred** is idle and ready to accept input.

Failure message:

`#Failed`

This indicates a failure of a user command (see section 2).

Error message:

`#Error: string`

This indicates that an error has occurred in one of the **Stred** commands, as detailed by `string`. For example, an attempt to `Goto-Locale` to a non-existant locale will produce an error.

Execution request:

`#Execute command_string`

If **Stred's** standard output is not to a terminal, **Stred** will produce this message in place of executing `command_string` itself.

Locales listing:

```
#Locales
     name1
     ....
     namen
#End
```

This is the list of all locales as produced by the `List-Locale` command.

Locale display:

```
#Locale name
     CommandDefinition=file
     Initialize=routine
     Node dir ...
     Page dir ...
     Readers dir ...
#End
```

This is the verbose listing of a single locale as produced by `List-Locale`. If any hierarchy is empty it is omitted. If there is no initialization routine or command definition file, the respective line is omitted. This is also produced by `Read-Locale` after a successful read (including the reading of command line locales), and by the `+List` option of `Edit-Locale`.

Page output:

```
#Page 'name'
     Title='string'
     Reader='string'
     Args='string'
#End
```

This is the output produced by the command definition language's output expression when its argument is a `page` (see section 2). If any of the strings are non-existent, the respective line is omitted.

Node output:

```
#Node 'name'
     Title='string'
#End
```

This is the output produced by the command definition language's output expression when its argument is a **node** (see section 2). If the title string is non-existent, the `Title` line is omitted.

Link output:

```
#Link 'name'
     Title='string'
     Selector='string'
#End
```

This is the output produced by the command definition language's output expression when its argument is a `link` (see section 2). If any of the strings are non-existent, the respective line is omitted.

Help response:

```
#Help
     text
     ....
#End
```

This is the format of the output produced by the `Help-Locale` command.

2. Stred Command Definition Language

2.1. Data Types

There are three basic data types in **Stred**: the `string`, the `page`, and the `link`. There is also the class of user defined types, called **nodes**. A particular node type has a standard specification with additional specification added by the user, including the type name.

All identifiers in **Stred** are composed of any sequence of alphanumerics and the underbar (_). Identifiers (and keywords) are case sensitive. Some example identifiers are `_hi`, `TopOfTree`, `1`, `84_nov`, and `_`.

One-dimensional, homogeneous (all elements of the same type) lists of basic types or node types can be made and manipulated. Also, aliases can be made for variable of either a basic type or a node type. Lists of lists, lists of aliases, and aliases of lists are not allowed.

There are two types of routines: commands (`comms`) and procedures (`procs`). Commands are user-callable and only allow `string` arguments. If the user does not provide all the required arguments, then the unspecified arguments are passed as non-existent (see below). Procedures are only callable from other procedures or commands and can have any type of arguments. Recursion is allowed for both.

The data types:

`string`
A simple character string with dynamic length. A `string` is non-existent if it has no value (a string with no characters — i.e., of length zero — has a value).

`page`
A `page` contains four `string` fields:
- `file` - This is the name of the `page`. If this field is non-existent then the `page` has no value.
- `title` - This specifies the title of the `page`. It is not required.
- `reader` - This is the name of the reader process of the `page`. It is not required for the `page` to have a value, but it is required if the `page` is to be read.
- `args` - This contains any options which should be passed to the `reader` in addition to the `page` name.

`link`
A `link` is a representation of a connection from one node to another. It has three `string` fields:
- `file` - This is the name of the target node. If this field is non-existent, then the `link` is considered to have no value.
- `title` - This is the title of the `link`. It is useful for menu selection.
- `selector` - This is an extra field which can be used by the

process which drives **Stred**. For example, it can contain information to map the selection of this `link` onto a particular display hardware feature.

node

A node is intended to be a location in the database, although it can be used to contain any structural information. The node can contain user defined fields, but all node types contain the following fields:

`file` - The name of the node, a `string`.

`title` - A title for the node.

`page` - This is the content associated with the node. It is a `page` structure (see above).

User defined fields can be of either `string` or `link` types, or lists of either. The creation of a node type is described in section 2.2. Note that variables of node type are not declared as **nodes**, but as the name given to the specific instance of node type.

2.2. Program Format

A **Stred** command definition file consists of a set of named fields which can be divided into three sections:

node type definitions:

This is the first section. It is a set of fields which define the various node types. The name of the field names the node type. The user-defined fields of the node are defined. An example field:

```
#tree {
     link a, b*;
     string  s1;
}
```

global definitions:

This single field, identified by the `globals` field tag, declares all of the global variables used, as well as all of the commands and procedures and their respective arguments. The order of the declarations is arbitrary. An example `globals` field:

```
#globals {
     tree     current;      $ a node
     string   s1, s2*, s3^;$ string, list, alias
     page     def_reader;
     link     temp, ptr^;
```

```
        comm create(name, titl),
             destroy(name);
        proc p1(),
             p2(n:tree, l:link^, list:string*);
    }
```

command and procedure definitions:

Following the `globals` field is a list of fields defining each command and procedure declared in the `globals` field. The field tag matches the name of the routine it defines. The routine definition consists of a (possibly empty) list of local, automatic variable declarations, similar to the global variable declarations, followed by the execution statements. If a local variable (either input argument or automatic) has the same name as a global variable, all references to that name will be to the local variable. If a user-defined field of a node type has the same name as either a local or global variable, it has access priority (only in node field selection, of course). An example command field:

```
    #destroy { $ name:string
        tree    n;

        n <- name;     $ read the node
        current.n->; $ destroy link from current to it
        <-current;    $ update the current node
        n->;       $ destroy the node
    }
```

2.3. Statements

A **Stred** routine either succeeds or it fails. Each statement in a **Stred** routine either succeeds or fails, and the failure of the statement causes the routine to fail (immediately). The routine succeeds if all statements executed in it succeed. There are four types of statements, described below. In the descriptions, an optional section is enclosed in bold square braces (**[ ]**). A list of statements is specified by <**stats**>. An expression, which can also succeed or fail, is specified by <**expr**>.

Simple

<**expr**> [?(<**stats**>)] [:(<**stats**>)];

If the expression succeeds then the statements in the ? clause are executed. If there is no ? clause then the statement succeeds, otherwise the statement succeeds only if all of the ? clause statements succeed. If the expression fails then the statements in the : clause are executed. If there is no : clause then the statement fails, otherwise the statement succeeds only if all of the : clause statements succeed. If this statement succeeds, the next statement is executed.

Repeat Loop

\ **<simple>**

This is similar to the simple statement except that if the expression (in the simple statement) succeeds and the simple statement itself succeeds, then it is re-executed, instead of continuing on to the next statement. If the expression fails, then this behaves the same as the simple statement.

Do-While Loop

?(**<stats>**) **<expr>** [:(**<stats>**)];

This is similar to the repeat loop except that the expression is executed after the ? clause. The ? clause statements, therefore, are always executed at least once.

Control

\! causes the immediate termination, with success, of any loop. It may only appear inside the ? clause of a loop statement. It terminates only the innermost enclosing loop.

?! This statement causes the immediate termination, with success, of the routine.

:! This statement causes the immediate termination, with failure, of the routine.

2.4. Expressions

Every expression results in either success or failure. The types of expressions are:

object
: This fails if the object fails (see below). Otherwise, no action is taken.

`object = object`
: This compares two objects for equality. This is an exact comparison for `strings`; for nodes, `pages`, and `links` only the `file` field is compared. The expression fails if either object fails or if the two (resultant) strings are not equal.

`object ~ object`
: This tests objects for inequality. It is otherwise identical to the equality test.

`object <- object`
: This assigns the right object to the left object. It fails if either object fails (and as specified below). The two objects must be of the same type except for the following exceptions:

  node <- string
  : This reads the node named by the value of `string` into the variable **node**. This fails if the node is unaccessible or if the saved node type is different from **node**'s node type.

  string <- page
  : This copies the content (i.e. the file) of `page` into the file named by `string` (as determined by the page access hierarchy). This fails if either page (content) is unaccessible.

  list <- object
  : This inserts a new copy of `object` at the head of `list`. The base type of `list` must be identical to the base type of `object`.

  alias <- object
  : This makes `alias` an alias for `object`. The base type of `alias` must be identical to the base type of `object`.

`object <- string1 + string2`
: This concatenates two strings into one string and assigns the result to `object` (under the above assignment conditions). Both `string1` and `string2` must succeed if the expression is to succeed.

`% object`
: This outputs `object`. This fails if `object` fails. `strings` are output directly. Nodes, `pages`, and `links` are output in the format specified in section 1.4.

`<-node`
: This writes `node` (in the file `node.file` as determined under the node access hierarchy). This fails if the `file` field of `node` fails or if the access fails.

`<-page`
: This reads the contents of `page`. This fails if the `file` or `reader` fields fail. It also fails if either the reader (under the binaries hierarchy) or the

content (under the page hierarchy) is inaccessible. The execution command is structured as "`reader args file`". If the output of **Stred** is to a terminal device then **Stred** itself will perform the execution with the standard **system** function; otherwise, an `#Execute` request is output (see section 1.4).

`proc(arg1, ..., argn)`
: This invokes a procedure or command. Each of the given arguments are assigned to the corresponding defined arguments as if by an assignment expression (with the attendent success or failure), except if the argument is declared as a list. A list can only receive another list, and the declared list argument becomes an alias for the passed list argument. The success or failure of the invoked routine then determines the success or failure of the expression. Note that if the declared argument is an alias, the passed argument must be a variable of that type (thus implementing a pass by reference).

`lobject -> robject`
: This appends a copy of `robject` following `lobject` in a list. `lobject` must be a list element, and `robject` must be an object of the same type. This fails if either object fails, with one exception: if `lobject` references the last element of a list (see below), this operation becomes equivalent to the assignment "`list <- object`".

`object->`
: This destroys and/or removes `object`. If `object` is a list element, it is removed from the list. `object` is cleared (i.e. any following reference to it or any of its fields will fail) and any associated permanent object (file of a page or node) is removed (unlinked from the file system).

2.5. Objects

An object can either succeed or fail. Failure occurs if the named object does not exist (eg. the successor of the last element in a list - see below) or if it has no value. The following are objects.

A variable:
: A simple variable reference, `var`, is an object. If `var` is a list, the `var*` references the last element in the list and `var` references the first element in the list (except on the left side of assignments and as a passed argument to a declared list).

A string constant:
: A string constant is a series of characters delimited by a matching pair of quotes (either single, double, or grave). There cannot be a line break within the string. The following escape sequences are recognised within the string:
: `\n` - a newline character
: `\t` - a horizontal tab
: `\b` - a backspace character (non-destructive)
: `\e` - the escape character

\' \" \` - a string delimiter
\\ - a backslash
any other sequence "\c" results in "c"

A selected field:

A field is selected from an object by `object.field`. If `field` is a list then the first element is referenced, and if there is a * suffix the last element is selected (as with a simple variable above). There is a special operation that selects a link from one node to another, using the syntax `node1.node2`. If there are more than one link, the link selected is the first encountered in the definition of the links of `node1`. If there is no link found, the object fails.

Link neighbours:

If `object` is an element of a list, then `+object` is the next object in the list, and `-object` is the previous element in the list. If there is no next (previous) element, this object fails.

Comments:

A comment is initiated by a `$` and terminated by the end of the line. It is, of course, ignored during execution.

```
$
$                    Example 1: A Tree
$
$ This is the Stred program for a tree.  The commands
$ for traversal and maintenance are inspired by UNIX,
$ but there is a strong Port influence.
$ This program was used to create this thesis.
$

#tree {
    string  name;
    link    parent, child*;
}

#mrk {
}

#globals {
    string  p_reader, p_args,   $ default reader
            root,     $ name of root node
            sep;      $ hierarchical separator for names
    tree    current, $ current position
            clip;     $ for set .. move, et al.

    comm    lc(dir),
            cr(dir, titl),
            cd(dir),
            rm(dir),
            rm_only(dir),
            r(dir),
            set(dir),
            move(nm, titl),
            copy(dir),
            head(dir),
            tail(dir),
            retitle(titl),
            set_reader(r, a),
            mark(),
            help();
    proc    _lc(n:tree),
            _geno(n:tree),
            _check(nc:tree, np:tree);
            _rename(old:string, par:string, new:string),
            _setup();
}

$ the setup routine
#_setup {
    mrk   m;

    root <- 'thesis';
    sep <- '_';
    p_reader <- 'vi';
    $ look for a marked position from previous session
    m <- '__mark__'  ?(
        current <- m.title;
        m->;
    ):(
        $ no marked session, so start at root
```

```
        $ build it if necessary
        current <- root  :(
            current.file <- root;
            current.page.file <- root;
            current.page.reader <- p_reader  :();
            current.page.args <- p_args  :();
            <-current;
        );
    );
    $ print current location
    % current;
    _lc(current);
}

$ mark the current node
$ next session will start at the marked node
#mark {
    mrk   m;

    m.file <- '__mark__';
    m.title <- current.file;
    <-m;
}

$ list the children of the passed node
#_lc {
    $ n:tree
    link   l^;

    l <- n.child  ?(
        ?(
            % l.selector;  % '\t';
            % l.title :();  % '\n';
        ) l <- +l  :();
    ):(% '\n';);
}

$ remove a subtree, recursively
#_geno {
    $ n:tree
    tree   m;
    link   l^;

    \ l <- n.child  ?(
        m <- l.file  ?(
            _geno(m)  :();
        ):();
        l->;
    ):();
    n.page->  :();  $ get the page, too
    n->;
}
```

```
$ rename a node
$ this changes the pathnames of all its children, so
$ they must be changed (moved) too (recursively)
#_rename {
    $ old:string   the old name of this node
    $ par:string   the new name of its parent
    $ new:string   the new name of this node
    tree   n, o;
    link   l^;

    o <- old;   $ get the old node
    n <- o;     $ and make an update copy
    n.parent.file <- par;
    n.name <- new;
    n.file <- par + sep;
    n.file <- n.file + new;
    n.page.file <- n.file;
    $ rename all the children recursively
    l <- n.child  ?(
        ?(
            _rename(l.file, n.file, l.selector)  :();
            l.file <- n.file + sep;
            l.file <- l.file + l.selector;
        ) l <- +l  :();
    ):();
    $ install the new node
    <-n;
    n.page.file <- o.page  :();
    $ outstall the old node
    o.page->  :();
    o->;
}

$ list the children of a node
$ must first figure out what directory the user wants
#lc {
    $ dir:string
    tree   n;

    dir  ?(
        $ first try relative to current
        n.file <- current.file + sep;
        n.file <- n.file + dir;
        n <- n.file  ?(
            _lc(n);
        ):(
            $ try global pathname
            n <- dir  ?(
                _lc(n);
            ):(
                % '#Error: No such directory\n';
                :!
            );
        );
```

```
    ):(
        $ no given directory means current
        _lc(current);
    );
}

$ create a new child
#cr {
    $ dir:string
    $ titl:string
    tree   n, t;
    link   nl;

    dir  :(
        % '#Error: syntax: cr name [title]\n';
        :!
    );
    n.file <- current.file + sep;
    n.file <- n.file + dir;
    n.name <- dir;
    current.n  ?(
        % '#Error: Already exists\n';
        :!
    ):();
    $ redundant: previous check should be sufficient
    t <- n.file  ?(
        % '#Error: Already exists\n';
        :!
    ):();
    $ build the node
    n.title <- titl  :();
    n.page.file <- n.file;
    n.page.reader <- p_reader  :();
    n.page.args <- p_args  :();
    n.page.title <- titl  :();
    nl.file <- n.file;
    nl.selector <- dir;
    nl.title <- titl  :();
    n.parent.file <- current.file;
    $ link to parent
    current.child* -> nl;
    $ update the database
    <-n;
    <-current;
    _lc(current);
}

$ move to a new node
$ let '.' be the current node
$ let '..' be the parent
#cd {
    $ dir:string
    tree   n;

    dir  :(
        $ no node given, assume '.'
        % current;
        _lc(current);
        ?!
```

```
    );
    dir = '..'  ?(  $ parent
        dir <- current.parent.file  :(
            % '#Error: At '; % root; % ' already\n';
            :!
        );
    ):(
        dir = '.'  ?(   $ current
            % current;
            _lc(current);
            ?!
        ):();
    );
    $ a real name was given
    $ first try relative
    n.file <- current.file + sep;
    n.file <- n.file + dir;
    n <- n.file  :(
        $ not relative, try global
        n <- dir  :(
            % '#Error: No such directory\n';
            :!
        );
    );
    $ now move
    current <- n;
    % current;
    _lc(current);
}

$ remove a child
$ this throws out the whole branch
#rm {
    $ dir:string
    tree   n;

    dir  ?(
        $ make sure child exists
        n.file <- current.file + sep;
        n.file <- n.file + dir;
        current.n  :(
            % '#Error: No such child\n';
            :!
        );
        n <- n.file;
    ):(
        $ no child given, so remove current node
        current.file = root  ?(
            % '#Error: Can't delete '; % root; % ' \n';
            :!
        ):();
        n <- current;
        $ move to parent
        current <- current.parent.file;
    );
```

```
    $ perform the removal
    _geno(n);
    current.n->;
    <-current;
    % current;
    _lc(current);
}

$ read contents (if any)
#r {
    $ dir:string
    tree   n;

    dir  ?(
        $ read a child
        n.file <- current.file + sep;
        n.file <- n.file + dir;
        current.n  :(
            % '#Error: No such child\n';
            :!
        );
        n <- n.file;
        <-n.page;
    ):(
        $ no child given => read current
        <-current.page;
    );
    _lc(current);
}

$ set the current node as source node
$ for a copy or move (a la Port)
#set {
    $ dir:string
    tree   n;

    dir  ?(
        $ set a child
        n.file <- current.file + sep;
        n.file <- n.file + dir;
        current.n  :(
            % '#Error: no such child\n';
            :!
        );
        clip <- n.file;
    ):(
        $ no child given => set current
        clip <- current;
    );
}
```

```
$ move a subtree
$ everything must be renamed
#move {
    $ nm:string
    $ titl:string
    link   l;
    tree   p;

    clip  :(
        % '#Error: No node was set to move\n';
        :!
    );
    clip.file ~ root  :(
        % "#Error: Can't move "; % root; % '\n';
        :!
    );
    $ clip the branch
    p <- clip.parent.file;
    p.clip->;
    p.file ~ current.file  ?(
        <-p;
    ):(
        current <- p;
    );
    nm  :(nm <- clip.name;);
    titl  ?(
        $ retitle it, if a title was given
        clip.title <- titl;
        <-clip;
    ):(
        $ no title, so use old (if any)
        titl <- clip.title  :();
    );
    $ make the link from the future parent
    l.selector <- nm;
    l.file <- current.file + sep;
    l.file <- l.file + nm;
    l.title <- titl  :();
    $ rename everything
    _rename(clip.file, current.file, nm);
    clip.file->;    $ unset
    $ attach to the new parent
    current.child* -> l;
    <-current;
    % current;
    _lc(current);
}

$ copy the content from one node to another
$ build a new node if the destination doesn't exist
#copy {
    $ dir:string
    tree    n;
    string  name, fname;

    clip.page;  $ a source node must be set
    $ have to determine the destination
    dir  ?(
        dir = '.'  ?(
```

```
            n <- current;
        ):(
            name <- dir;
        );
    ):(
        $ use old name if none was given
        name <- clip.name;
    );
    name  ?(
        $ get the destination node
        fname <- current.file + sep;
        fname <- fname + name;
        n <- fname  :(
            $ it doesn't exist => make it
            clip.title  ?(
                cr(name, clip.title);
            ):(
                cr(name, '');
            );
            n <- fname;
        );
    ):();
    $ copy the contents
    n.page->  :();
    n.page <- clip.page;
    n.page.file <- n.file;
    n.page.file <- clip.page;
    <-n;
    n = current  ?(
        current <- n;
    ):();
}

$ change the title of the current node
#retitle {
    $ titl:string
    link   l^;
    tree   p;

    current.title <- titl;
    <-current;
    $ also change the link title from the parent
    $ (if root => no parent)
    p <- current.parent.file  :(?!);
    l <- p.current;
    l.title <- titl;
    <-p;
    % current;
}
```

```
$ move a child to the head of the list
$ this is for display
$ there are better ways to do this ...
#head {
    $ dir:string
    tree   c;
    link   l, p^;

    dir  :(
        % '#Error: syntax: head child\n';
        :!
    );
    c.file <- current.file + sep;
    c.file <- c.file + dir;
    p <- current.c  :(
        % '#Error: No such child\n';
        :!
    );
    l <- p;
    p->;
    current.child <- l;
    <-current;
    _lc(current);
}

$ move a child to the tail of a list
#tail {
    $ dir:string
    tree   c;
    link   l, p^;

    dir  :(
        % '#Error: syntax: tail child\n';
        :!
    );
    c.file <- current.file + sep;
    c.file <- c.file + dir;
    p <- current.c  :(
        % '#Error: No such child\n';
        :!
    );
    l <- p;
    p->;
    current.child* -> l;
    _lc(current);
}

$ change the reader of the current page
#set_reader {
    $ r:string
    $ a:string

    r  :(
        % '#Error: no reader was given\n';
        :!
    );
```

```
    current.page.reader <- r;
    current.page.args <- a  :();
    <-current;
}

#help {
    % '\tlc   [dir]\n';
    % '\tcr    dir  [title]\n';
    % '\tcd   [dir]\n';
    % '\trm   [dir]\n';
    % '\tr    [dir]\n';
    % '\tset  [dir]\n';
    % '\tmove [name] [title]\n';
    % '\tcopy [dir]\n';
    % '\tretitle title\n';
    % '\thead  dir\n';
    % '\ttail  dir\n';
    % '\tset_reader reader [args]\n';
}

$ check that it is okay to promote grandchildren
$ to children
#_check {
    $ nc:tree
    $ np:tree
    link   lc^, lp^;

    lc <- nc.child :(?!);
    lp <- np.child :(?!);
    ?(
        lp <- np.child;
        ?(
            lc.selector ~ nc.name  ?(
                lc.selector ~ lp.selector;
            ):();
        )  lp <- +lp  :();
    )  lc <- +lc  :();
}

$ remove a child
$ this promotes all the grandchildren
#rm_only {
    $ dir:string
    tree   n;
    link   l^;

    dir  ?(
        $ make sure child exists
        n.file <- current.file + sep;
        n.file <- n.file + dir;
        current.n  :(
            % '#Error: No such child\n';
            :!
        );
        n <- n.file;
    ):(
        $ no child given, so remove current node
        current.file = root  ?(
            % '#Error: Can't delete '; % root; % ' \n';
```

```
                :!
            ):();
            n <- current;
            $ move to parent
            current <- current.parent.file;
        );
        $ perform the removal
        _check(n, current)  :(
            current <- n;
            % '#Error: cannot promote children\n';
            :!
        );
        current.n->;
        l <- n.child  ?(
            ?(
                _rename(l.file, current.file, l.selector);
                l.file <- current.file + sep;
                l.file <- l.file + l.selector;
                current.child* -> l;
            )  l <- +l  :();
        );
        <-current;
        % current;
        _lc(current);
}
```

```
$
$                    Example 2: A Grid
$
$ This is a Stred program that implements a flavour of
$ grid structure. It allows only 3 dimensions.
$

#grid {
    string  x1, x2, x3;
}

#save { $ the keys of the various dimensions
    string  1*, 2*, 3*;
}

#globals {
    string  saved_name, sep;
    string  p_a1, p_a2, p_a3, p_c1, p_c2, p_c3;
    grid    current;
    save    Keys;
    comm    move(1, 2, 3),
            wmi(),
            add(1, 2, 3),
            del(1, 2, 3),
            see(),
            set_reader(rdr, arg),
            new1(k),
            new2(k),
            new3(k),
            remove(k),
            keys(),
            help();
    proc    _setup(),
            _find(l^:string, k:string),
            _file(g^:grid),
            _init(),
            _parse(),
            _place(k:string),
            _fin(),
            _sad(s^:string, k:string);
}

$ the setup routine
$ get the keys, if any
$ if none, start with a few defaults
#_setup {
    string      start;

    sep <- '.';
    saved_name <- 'saved_keys';
    $ retrieve the keys
    Keys <- saved_name  :(
        Keys.file <- saved_name;
    );
    Keys.1  :(
        Keys.1 <- 'root_1';
    );
    Keys.2  :(
        Keys.2 <- 'root_2';
```

```
    );
    Keys.3  :(
        Keys.3 <- 'root_3';
    );
    <-Keys;
    $ nodes are named by a concatenation of their
    $ coords start at the first indicated label of
    $ each dimension
    start <- Keys.1 + sep;
    start <- start + Keys.2;
    start <- start + sep;
    start <- start + Keys.3;
    current <- start  :(
        $ create the node if it doesn't exist
        current.file <- start;
        current.x1 <- Keys.1;
        current.x2 <- Keys.2;
        current.x3 <- Keys.3;
        <-current;
    );
    % current;
}

$ look for a key in a list
#_find {  $ l^, k : string
    ?(
        l = k  ?(?!)  :();
    )  l <- +l;
}

$ build a file name, given coords
#_file {  $ g^ : grid
    g.file <- g.x1 + sep;
    g.file <- g.file + g.x2;
    g.file <- g.file + sep;
    g.file <- g.file + g.x3;
}

#_init {
    p_c1-> :();  p_c2-> :();  p_c3-> :();
    p_a1-> :();  p_a2-> :();  p_a3-> :();
}

#_fin {
    p_c1  :(p_c1 <- current.x1;);
    p_c2  :(p_c2 <- current.x2;);
    p_c3  :(p_c3 <- current.x3;);
}
```

```
#_place {  $ k : string
    _find(Keys.1, k)  ?(
        p_c1  ?(:!)  :(p_c1 <- k;);
    ):(
        _find(Keys.2, k)  ?(
            p_c2  ?(:!)  :(p_c2 <- k;);
        ):(
            _find(Keys.3, k)  ?(
                p_c3  ?(:!)  :(p_c3 <- k;);
            );
        );
    );
}

#_parse {
    p_a1  ?(
        _place(p_a1);
        p_a2  ?(
            _place(p_a2);
            p_a3  ?(
                _place(p_a3);
            ):();
        ):();
    ):();
    _fin();
}

$ move to another grid coordinate
$ only if there is a node there
#move {  $ 1, 2, 3 : string
    grid    c;

    _init();
    p_a1 <- 1 :();  p_a2 <- 2 :();  p_a3 <- 3 :();
    _parse()  :(
        % "#Error: Bad grid coordinates\n";
        :!
    );
    c.file <- ' ';
    c.x1 <- p_c1;  c.x2 <- p_c2;  c.x3 <- p_c3;
    _file(c);
    c <- c.file  :(
        % "#Error: No such node\n";
        :!
    );
    current <- c;
    % current;
}
```

```
$ make a node at a grid coordinate
#add {  $ 1, 2, 3 : string
    grid    t, g;

    _init();
    p_a1 <- 1 :();  p_a2 <- 2 :();  p_a3 <- 3 :();
    _parse()  :(
        % "#Error: Bad grid coordinates\n";
        :!
    );
    g.file <- ' ';
    g.x1 <- p_c1;  g.x2 <- p_c2;  g.x3 <- p_c3;
    _file(g);
    t <- g.file  ?(
        % '#Error: node '; % g.file;
        % ' already exists\n';
        :!
    ):();
    <-g;
}

$ delete a node at a grid coord
#del {  $ 1, 2, 3 : string
    grid    g;

    _init();
    p_a1 <- 1 :();  p_a2 <- 2 :();  p_a3 <- 3 :();
    _parse()  :(
        % "#Error: Bad grid coordinates\n";
        :!
    );
    g.file <- ' ';
    g.x1 <- p_c1;  g.x2 <- p_c2;  g.x3 <- p_c3;
    _file(g);
    g.file ~ current.file  :(
        % "#Error: Can't delete current node\n";
        :!
    );
    g <- g.file  :(
        % '#Error: No node at those coordinates\n';
        :!
    );
    g->;
}

$ print out current grid coords
#wmi {
    % current;
}

$ set the reader for the current node
#set_reader {  $ rdr, arg : string
    current.page.reader <- rdr;
    current.page.args <- arg  :();
    current.page.file <- current.file;
    <-current;
}
```

```
$ read the contents of the current node
#see {
    current.page.file  :(
        % '#Error: Page is not set for this node\n';
        :!
    );
    <-current.page;
}

$ add a new label to dimension 1
#new1 {  $ k : string
    _place(k)  ?(
        % '#Error: Key is already in a dimension\n';
        keys();
        :!
    ):();
    Keys.1* -> k;
    <-Keys;
    keys();
}

$ add a new label to dimension 2
#new2 {  $ k : string
    _place(k)  ?(
        % '#Error: Key is already in a dimension\n';
        keys();
        :!
    ):();
    Keys.2* -> k;
    <-Keys;
    keys();
}

$ add a new label to dimension 3
#new3 {  $ k : string
    _place(k)  ?(
        % '#Error: Key is already in a dimension\n';
        keys();
        :!
    ):();
    Keys.3* -> k;
    <-Keys;
    keys();
}

#_sad {  $ s^, k : string
    \ s ~ k  ?(
        s <- +s;
    ):(
        s->;
    );
}
```

```
$ remove a label
$ labels must be unique, so there is no problem
$ re which dimension to remove from
#remove {  $ k : string
    _sad(Keys.1, k)   :(
        _sad(Keys.2, k)   :(
            _sad(Keys.3, k)   :(
                % '#Error: No such key\n';
                :!
            );
        );
    );
    <-Keys;
    keys();
}

$ on-line doc
#help {
    % 'Commands are:\n';
    % '\tmove coord [coord [coord]]\n';
    % '\tadd  coord [coord [coord]]\n';
    % '\tdel  coord [coord [coord]]\n';
    % '\twmi\n';
    % '\tnew1   coord\n';
    % '\tnew2   coord\n';
    % '\tnew3   coord\n';
    % '\tremove coord\n';
    % '\tsee\n';
    % '\tset_reader reader [args]\n';
    % '\tkeys\n';
    % '\thelp\n';
}

$ display all the labels (keys) of all dimensions
#keys {
    string  s^;

    % 'Dimension 1:\n';
    s <- Keys.1;
    ?(
        % '\t'; % s; % '\n';
    )  s <- +s  :();
    % 'Dimension 2:\n';
    s <- Keys.2;
    ?(
        % '\t'; % s; % '\n';
    )  s <- +s  :();
    % 'Dimension 3:\n';
    s <- Keys.3;
    ?(
        % '\t'; % s; % '\n';
    )  s <- +s  :();
}
```

```
$
$       Example 3: Directed Acyclic Graph
$
$ This is a Stred program describing a dag with a
$ single source node. All nodes in the database must
$ have unique names.
$

#dag {
    link    parent*, child*;
}

#globals {
    dag     current,
            target;     $ for making links
    string  root,       $ name of starting source
            path*,      $ path from root to current
            p_reader,   $ default reader for pages
            p_args;     $ default page arguments
    comm    lc(),
            lp(),
            gc(c),
            gp(p),
            back(),
            mark(),
            new(c, t),
            mklink(t),
            unlink(c),
            read(),
            set_reader(r, a),
            page_reader(r),
            page_args(a),
            wmi(),
            help();
    proc    _setup(),
            _find(list^:link, it:string),
            _path(p:string),
            _check(p:string, c:string),
            _unlink(p:string, c:string);
}

#_setup {
    root <- 'root';
    p_reader <- 'vi';
    current <- root  :(         $ start at root
        current.file <- root;   $ create it if I must
        <-current;
    );
    path <- root;   $ initialize path
    % current;
    lc();
}
```

```
$ list children of current node
#lc {
    link    l^;

    l <- current.child  ?(
        % 'Children:\n';
        ?(
            % '\t'; % l.file; % '\t';
            % l.title :(); % '\n';
        )  l <- +l  :();
    ):(
        % 'No children\n';
    );
}

$ list parents of current node
#lp {
    link    l^;

    l <- current.parent  ?(
        % 'Parents:\n';
        ?(
            % '\t'; % l.file; % '\t';
            % l.title :(); % '\n';
        )  l <- +l  :();
    ):(
        % 'No parents\n';
    );
}

$ find a particular link in a list
#_find {  $ list^:link, it:string
    \ list.file ~ it  ?(
        list <- +list;
    ):(?!);
}

$ adjust the path for a move
#_path {  $ p : string
    string  s^;

    $ back to root then re-init the path
    p = root  ?(
        \ path->  :();
        path <- root;
        ?!
    ):();
    $ if the next node is the previous in
    $ the path, back up the path
    s <- -path*  :(
        path* -> p;     $ it's not: add to path
        ?!
    );
```

```
    s = p  ?(
        path*->;        $ it is: back up
    ):(
        path* -> p;     $ it's not: add to path
    );
}

$ go to child
#gc {  $ c : string
    dag     d;

    _find(current.child, c);    $ it must be a child
    d <- c;                     $ preserve current
    current <- d;
    _path(c);
    % current;
    lc();
}

$ go to parent
#gp {  $ p : string
    dag     d;

    _find(current.parent, p);   $ it must be a parent
    d <- p;                     $ preserve current
    current <- d;
    _path(p);
    % current;
    lc();
}

$ back up the path
#back {
    dag     d;

    path*->;    $ back up the path
    d <- path*  :(
        % "#Error: Can't back track\n";
        $ something weird happened: re-init path
        \ path*->  :();
        path <- root;
        path -> current.file;
        :!
    );
    current <- d;
    % current;
    lc();
}

$ make a new child for current
#new {  $ c, t : string
    dag     d;
    link    l;

    d <- c  ?(
        % '#Error: Already exists\n';
        :!
    ):();
    $ make the node
```

```
    d.file <- c;
    d.title <- t  :();
    d.page.file <- c;
    d.page.reader <- p_reader  :();
    d.page.args <- p_args  :();
    d.page.title <- t  :();
    $ link it to current
    l.file <- c;
    l.title <- t  :();
    current.child* -> l;
    l.file <- current.file;
    l.title <- current.title  :();
    d.parent* -> l;
    <-d;
    <-current;
    lc();
}

$ mark the current as the target for a new link
#mark {
    target <- current;
}

$ check that a link doesn't create a cycle
#_check {  $ p, c : string
    dag     d;
    link    l^;

    p ~ c;           $ make sure it's not a tight cycle
    d <- p  :(?!);  $ doesn't exist yet
    l <- d.child  :(?!);
    ?(
        _check(l.file, c);  $ recurse on children
    )  l <- +l  :();
}

$ make a link from current to target
#mklink {  $ t : string -> the link title
    link    l;

    target  :(
        % '#Error: No target node was set\n';
        :!
    );
    current ~ target;
    _find(current.child, target.file)  ?(
        lc();
        ?!
    ):();
    _check(target.file, current.file)  :(
        % '#Error: That would make a cycle\n';
        :!
    );
    $ make the link
    l.file <- target.file;
    t  ?(
        l.title <- t;
    ):(
        l.title <- target.title  :();
```

```
    );
    current.child* -> l;
    <-current;
    l.file <- current.file;
    l.title <- current.title  :();
    target.parent* -> l;
    <-target;
    lc();
}

$ destroy a link - if this results in an isolated
$ child, the node is removed and links to its children
$ are deleted (proceeding recursively)
#_unlink {  $ p, c : string
    dag     pd, cd;
    link    l^;

    $ break the link
    pd <- p;
    cd <- c;
    pd.cd->;
    <-pd;
    cd.pd->;
    cd.parent  ?(
        <-cd;
    ):(
        $ it's isolated: unlink children
        l <- cd.child  ?(
            ?(
                _unlink(c, l.file);
            )  l <- +l  :();
        ):();
        $ throw the node (and page) away
        cd.page  ?(
            cd.page->;
        ):();
        cd->;
    );
}

$ unlink a child from current
#unlink {  $ c : string

    _find(current.child, c);     $ make sure it's a child
    _unlink(current.file, c);    $ unlink it
    current <- current.file;     $ update current
    lc();
}

$ read the current page
#read {
    <-current.page;
    lc();
}
```

```
$ set the page reader and arguments for current
#set_reader {
    $ r, a : string - reader process and the arg string
    current.page.reader <- r;
    current.page.args <- a  :();
    current.page.file <- current.file;
    current.page.title <- current.title  :();
}

$ change default page reader
#page_reader {  $ r : string
    r  ?(
        p_reader <- r;
    );
}

$ change default page reader arguments
#page_args {  $ a : string
    a  ?(
        p_args <- a;
    );
}

$ show the path to current
#wmi {
    string  s^;

    s <- path;
    ?(
        % s; % ' ';
    )  s <- +s  :();
    % '\n';
}

#help {
    % 'Dag commands are:\n';
    % '\tlc\n';
    % '\tlp\n';
    % '\tgc  child\n';
    % '\tgp  parent\n';
    % '\twmi\n';
    % '\tback\n';
    % '\tnew  name [title]\n';
    % '\tmark\n';
    % '\tmklink [title]\n';
    % '\tunlink  child\n';
    % '\tpage_reader  reader\n';
    % '\tpage_args    args\n';
    % '\tset_reader   reader [args]\n';
    % '\tread\n';
    % '\thelp\n';
}
```

Stred Manual

David Adrien Tanguay
University of Waterloo

1. The Stred System

1.1. Overview

Stred is a system for designing and implementing structures for use in videotex databases. The major emphases are flexibility of the implementation and the personalization of the databases created with Stred. Stred also supplies a standard operating environment for its structures. The design of a structure is specified in the **command definition file**, in the form of a **structure program** written in a special language (see section 2). This program describes the commands and datatypes which specify the structure.

Stred itself makes no concession to user friendliness, although its interface is human operable. It is expected that a **driver** system will mate the Stred interface with the available hardware devices to create an aesthetically pleasing, user friendly interface.

1.2. Environment

Stred views a database as a collection of nodes and state variables, with commands, input interactively, that manipulate the organization of the database. The format of a node, the state variables, and the precise definition of the commands constitute the structure, the scheme of organization. The structure is defined in the **command definition file** by the **structure program**.

A node does not (usually) contain the contents of the database. The data (if any) that is associated with a node is stored in a separate file, called a **page**. Pages are not interpreted by Stred; a **reader** process must be supplied to read, modify, and/or format the data. The node, however, will usually contain the information necessary to access the page (the page's name, for example).

Nodes, pages, and readers are accessed indirectly: the actual name is modified by an **access hierarchy** (separate hierarchies for each). The access hierarchy is an ordered list of places that are searched to find the object (under the given access capabilities) (this is similar to the search for executable files done by the UNIX shell on the **PATH** environment variable). Stred uses UNIX access permissions to determine access type. For example, let the node access hierarchy be /usr/public/db_nodes, /usr/telidon/nodes, and /u/me/n. Now suppose we want to access the node **root_2_4** with write permissions, and that this node exists in /usr/public/db_nodes with read-only permissions and in /u/me/n with full permissions. First, /usr/public/db_nodes is searched and the node is found, but access fails since that **root_2_4** does not have the required permissions. Next,

/usr/telidon/nodes is searched, and again access fails, this time since the node does not exist there. Finally, /u/me/n is searched, the node is found with the required permissions, and the access terminates in success. The access hierarchy is not applied to names that begin with a slash (/). These names are assumed to be absolute pathnames and are accessed "as is" (i.e., without modification by the access hierarchy).

Different structures can operate one the same database, in certain conditions. The node formats (see section 2) must be compatible, and the commands of one structure must preserve the consistency of the database under the other structures. As a simple example, a command that performs a pre-order walk of a tree can be added to an existing tree structure, creating a technically different structure, but one that is still compatible with the original. More radical changes could be accommodated, but the risk of incompatibility will naturally increase.

A database is described by a **locale**. A locale contains the access hierarchies (node, page, and binary), the name of the command definition file (i.e., the structure of the database), and an initializing command line (executed immediately after loading the structure program). A locale is stored as a file, but it is created and modified with Stred commands. In one Stred session, several different databases may be activated (i.e., several locales may be loaded).

1.3. Commands

The following commands are those accepted by Stred. They are described using the standard abbreviation syntax of the University of Waterloo: all upper-case letters of a command must be present, as well as non-alphabetic characters; all lower-case letters (including the underbar _) are optional. Optional arguments are enclosed in square brackets ([ ]). Where a choice of exactly one character is required, the characters are presented in parentheses and separated by vertical bars (eg. (+|-)). If an argument can be present zero or more times, an asterisk (*) is appended to the argument.

Three types of arguments are allowed on the Stred command line:

(+|-)EchoLocale

When on, this causes a verbose listing of the current locale after every change of locale. It is off by default.

(+|-)Slave

When on, this indicates to Stred that it is being driven by another process. Stred will issue **#Execute** statements instead of executing processes itself (see section 1.4). The default is off.

(+|-)Repeat

When on, then this causes the input lines to be echoed. When off, the input prompt is not output. The default is to not echo input lines and to output a prompt. If Stred is a slave, setting this option has no effect.

All other arguments are assumed to be the names of Stred locales. They are loaded in reverse order so that the first given locale is active when the command prompt is first displayed.

The following are the commands accepted interactively:

List-Locale [locale]
This lists the named **locale**. If no locale is given, a list of active locales is presented.

Goto-Locale locale
Switches to the named **locale**.

Copy-Locale name
A copy of the current locale is made under the given **name**. Stred then switches to that locale.

Read-Locale locale
The named **locale** is read into Stred. The command definition file is parsed and the initialization routine, if present, is executed. The named **locale** becomes the current locale.

Delete-Locale [+PERManent]
The current locale is deleted from Stred. Stred then switches to the last locale (next in the locales list) if there is one. If the **+PERManent** option is invoked, the file containing the locale is removed.

Edit-Locale [options]* [locale]
Makes changes to either the current locale or the named **locale**. Options are as follows:

(+|-) **CReate**
Creates a new locale or clears an existing locale.

(+|-) **CLear**
All fields of the locale are cleared.

(+|-) **PERManent**
The edit changes are made permanent (i.e. the locale file is updated).

(+|-) **List**
Lists the locale verbosely after the edit.

Init=routine
Sets the initialization routine.

–Init
Resets the initialization routine.

Comm_Def_File=file
Set the command definition file.

Binary_ADd=dir
Binary_INs=dir
Binary_DEl=dir
Node_ADd=dir
Node_INs=dir
Node_DEl=dir
Page_ADd=dir
Page_INs=dir
Page_DEl=dir

Add (to end), insert (at head), or delete hierarchical places (directories) from the appropriate hierarchy.

Make-Place dir

Make a hierarchical place (a directory). The access permissions are those set in your umask.

Delete-Place dir

Delete a hierarchical place (remove the directory). The directory must be empty.

Help-Locale

This prints out a short list of Stred commands (not user defined commands).

!system command

If a ! is the first character of the line then the rest of the line is assumed to be a UNIX command line. If Stred is a slave process, then it outputs a **#Execute** request (see section 1.4). Otherwise, Stred itself executes the command (via the **system** function).

user defined commands

An unrecognised command is assumed to be user defined in the current locale. If it is not then an error message is produced.

1.4. Output

The following are the various forms of output Stred will produce.

Input prompt:

_>

This (with a trailing space - no newline) is the input prompt. It indicates that Stred is idle and ready to accept input. It is not output if **–Repeat** is in effect when Stred is not a slave.

Failure message:

#Failed
#Failed on line *n*

This indicates a failure of a user command (see section 2). Usually, the line number (of the command definition file) that was being executed (that caused the failure) is displayed (the second form).

Error message:

#Error: string

This indicates that an error has occurred in one of the Stred commands, as detailed by **string**. For example, an attempt to **Goto-Locale** to a non-existant locale will produce an error.

Execution request:

#Execute command_string

If Stred is being driven by another process, Stred will produce this message in place of executing **command_string** itself.

Locales listing:

#Locales
name**1**
....
name**n**
#End

This is the list of all locales as produced by the **List-Locale** command.

Locale display:

#Locale name
CommandDefinition=file
Initialise=routine
Node dir ...
Page dir ...
Binary dir ...
#End

This is the verbose listing of a single locale as produced by **List-Locale**. If any hierarchy is empty it is ommitted. If there is no initialization routine or command definition file, the respective line is ommitted. This is also produced by **Read-Locale** after a successful read (including the reading of command line locales), and by the **+List** option of **Edit-Locale**.

Help response:

#Help
text
....
#End

This is the format of the output produced by the **Help-Locale** command.

2. Stred Command Definition Language

2.1. Data Types

There are two basic data types in Stred: **string** (a character string), and **int** (signed integer). There is also the class of user defined types, called **nodes**. A particular node type has a standard specification with additional specification added by the user, including the type name.

Comments in Stred start with a commercial at sign (@) and continue to the end of the line.

All identifiers in Stred are composed of any sequence of alphanumerics, the underbar (_), and the octothorpe (#) that does not comprise a valid number constant (see section 2.4). Identifiers (and keywords) are case sensitive. Some example identifiers:

#hi
TopOfTree
84_nov
#1

One-dimensional, homogeneous (all elements of the same type) lists of basic types or node types can be made and manipulated. Also, aliases can be made for variables of either basic or node types. Lists of lists, lists of aliases, aliases of aliases, and aliases of lists are not allowed.

There are three data types:

string - A simple character string with dynamic length. A **string** is non-existent if it has no value (a string with no characters — i.e., of length zero — has a value).

int - An integer. **int** variables are initially undefined (a special internal value), and may be explicitly set to undefined by deleting them (see section 2.5).

node - A node is intended to be a location in the database, although it can be used to contain any structural information. The node can contain user defined fields, but all node types contain a **file** field, a **string**, which is the name of the node as it is stored in the access hierarchy. User defined fields can be **string**, **int** types, or a previously defined node type, or lists of these. The creation of a node type is described in section 2.2. Note that variables of node type are not declared as **nodes**, but as the name given to the specific instance of node type.

Variables are declared with a statement of the form:

type var*1*, var2, ..., var*n*;

where **type** is the base data type and **var*i*** is the identifier for the variable. The identifier may be followed by a carret (^) to indicate that the variable is an alias,

or by an asterisk (*) to indicate that it is a list. The following are example declarations:

```
string root, list_of_names*, reader, list_ptr^;
int count;
node_type current*, save_node;
```

2.2. Program Format

A Stred command definition file consists of a set of named fields which can be divided into three sections:

node type definitions:

This is the first section. It is a set of fields which define the various node types, one type per field. The name of the field names the node type. For each node type, the user-defined fields of the node are defined as a set of variable declarations. Aliases are not allowed in field declarations. Only previously defined node types are allowed (there is no forward declaration of types). An example node definition:

```
tree {
    stringreader, args;
    stringparent, child*;
}
```

global definitions:

This single field, identified by a null field tag, declares all of the global variables used, as well as all of the commands and procedures and their respective arguments. The order of the declarations is arbitrary. Commands and procedures are declared in a fashion similar to variable declarations, with their arguments also declared at the same time (see section 2.3 for command and procedure definitions). An example globals field:

```
{
    @ see also section 2.3
    tree current;              @ a node
    strings1, s2*, s3^;     @ string, list, alias
    comm      create(name, title),
              destroy(name);
    proc      p1(),
              p2(tree n; link l^; string list*);
}
```

command and procedure definitions:

Following the globals field is a list of fields defining each command and procedure declared in the globals field. The field tag matches the name of the routine it defines. The routine definition consists of a (possibly empty) list of local, automatic variable declarations, followed by the execution statements. If a local variable (either input argument or automatic) has the same name as a global variable, all references to that name (within that field) will be to the local variable. If a user-defined field of a node type has the same name as either a local or global variable, it has access priority (only in node field selection, of course). An example command field:

```
@ see sections 2.4, 2.5, and 2.6
@ This command destroys a child, passed as name, of
@ the current node in a tree structure.

destroy { @ string name;
        tree    n;
        stringlp^;

        #read n <– name;     @ read the node
        @ find the link from current to name
        lp << current.child;
        \ lp ~ name  ?( lp << ^+lp; ) :();
        @ nuke the link
        ! lp;
        #save current;   @ update the current node
        #remove n;       @ destroy name
}
```

2.3. Routines

There are two types of routines: commands (**comms**) and procedures (**procs**). Commands are user-callable and only allow **string** arguments. Commands also allow the last declared argument to be a list of **string**, and all excess arguments passed are placed in order in this list. If the user does not provide all the required arguments, then the unspecified arguments are passed as non-existent (see below). Procedures are only callable from other procedures or commands and can have any type of arguments. Both allow recursion.

Commands and procedures are declared in the globals field in a similar fashion to variable declarations. Following the routine's identifier, however, there must be a list of declarations of routine arguments (possibly empty) enclosed in parentheses.

Command arguments must be **strings**, so the list is just a comma separated list of identifiers. They may not be aliases or lists, except for the last identifier, which may be a list — any excess arguments to the command are put at the end of this list.

Procedure arguments can be of any type and can include aliases and lists. Their declarations follow the same format as global declarations, but the last semi-colon (;) may be omitted (see section 2.2, under global fields). If an argument is declared as an alias, then only a variable may be passed to it, not a constant or computed expression (either arithmetic or concatenated — see section 2.4). This implements a pass by reference. If an argument is declared as a list, then a list must be passed to it, and the argument becomes an alias for that list. Any manipulation of the list argument results in manipulation of the passed argument, since they both refer to the same list. Parameters passed as procedur) arguments do not have to have a value (see section 2.4).

2.4. Objects

A Stred object is a primitive form of expression. It can either succeed or fail. Failure occurs if the named object does not exist (e.g., the successor of the last element in a list — see below) or if it has no value (undefined integers, non-existent objects, etc. — see below). The following are objects.

string constant:

A string constant is a series of characters delimited by a matching pair of quotes (either single, double, or grave). There cannot be a line break within the string. The following escape sequences are recognised within the string:
n - a newline character
t - a horizontal tab
b - a backspace character (non-destructive)
g - an audio bell character
\^*c* - a control character
e - the escape character
\' \" \` - a string delimiter
\\ - a backslash
any other sequence "\c" results in "c"

number constant:

A number constant is a contiguous string of decimal digits (0-9), with an optional leading minus sign (–) to specify a negative constant.

list end dereference:

If **var** is a list, the **var$** references the last element in the list and **var** references the first element in the list (except on the left side of assignments and as a passed parameter to an argument declared as a list — see section 2.3).

variable:

A simple variable reference, **var**, is an object.

A selected field:

A field is selected from an object by **object.field**. If **field** is a list then the first element is referenced, and if there is a **$** suffix the last element is selected (as with a simple variable above).

link neighbours:

If **object** is a reference to a list element (i.e., a variable, possibly with selected fields — not a constant or computed object), then ^+**object** references then next element in the list and ^−**object** references the previous element in the list. The object fails if the next (or previous) element does not exist (i.e., at the end of the list).

arithmetic expressions:

Standard arithmetic expressions are allowed. The operations are +, −, /, and *. Multiplication and division have higher priority than addition and subtraction, but parentheses may be used to alter order of evaluation. If a string is found in a numerical context, it is cast into a number. If the string does not represent a valid number (i.e., it has non-numerical characters, excluding a leading minus sign), then the expression fails. Any other data types (nodes or routine invocations) result in failure.

concatenated expressions:

Consecutive string type objects (or arithmetic expressions, which are then cast into strings) result in a string which is the concatenation of those values. An example of a concatenated expression:

'The total number of widgets is:' wid_a+wid_b '\n'

2.5. Expressions

A Stred expression is a more complicated form of expression than a Stred object. Whereas objects are declarations of *value*, expressions generally perform operations on these values, and drive Stred statements (see section 2.6). Every expression results in either success or failure. In the definitions below, I will use the following short-hand for various classes of objects:

var - a variable object, including list dereferencing and field selection
alias - a variable declared as an alias
list - a variable declared as a list
node - a **var** that has a user defined type
list_obj - a reference to an element of a list (possibly via alias)
string_obj - an object that can be resolved or cast into a string value
proc - a command or procedure, defined in the globals field

The types of expressions are:

object

This fails if the object fails (see below). Otherwise, no action is taken.

string_obj <relop> string_obj

Where *<relop>* must be one of = (equality), ~ (inequality), <=, >=, <, or >. This compares two strings (lexicographically) or numbers (the latter only if both objects are numeric constants or variables, otherwise number to string conversion is used). The expression fails if either object fails or if the comparison fails (i.e., is not true).

var <-- object

This assigns the value of **object** to **var**. It fails if either object fails (and as specified below). The two objects must have the same type.

alias << var

This sets an alias.

% string_obj

This outputs **string_obj**. This fails if **string_obj** fails. The output is immediate (there is no buffering).

#save node

This writes **node** (in the file **node.file** as determined under the node access hierarchy). This fails if the **file** field of **node** fails or if the access fails.

#read node <-- string_obj
#read node

This reads in a node, from the name specified by **string_obj**, from the node access hierarchy. The saved node must have the same node type as **node**. In the second form, the value of the **file** field of **node** is used as the name.

#read string_obj with string_obj, string_obj
#read string_obj with string_obj

This reads the contents of a page. The first argument is the page name, the second is the reader process, and the third is the arguments for the reader It fails if either the reader (under the binaries hierarchy) or the content (under the page hierarchy) is inaccessible. The execution command is structured as "**reader args file**". If the output of Stred is to a terminal device then Stred itself will perform the execution with the standard **system** function; otherwise, an **Execute** request is output (see section 1.4).

#remove node

This removes the node named by the **file** field of **node** from the access hierarchy. This fails if the name (**file**) or the access fails.

#remove string_obj

This is similar to node removal, except that the page named by **string_obj** is removed from the page hierarchy.

proc(object, ..., object)

This invokes a procedure or command. Each of the given arguments are assigned to the corresponding defined arguments. The success or failure of the invoked routine then determines the success or failure of the expression.

list_obj –> object

This appends a copy of **object** following **list_obj** in a list. **object** must have the same type as **list_obj**. This fails if either object fails, with one exception: if **list_obj** references the last element of an empty list (i.e., it has the form **list$**), this operation becomes equivalent to the assignment "**list <– object**".

! var

This destroys (clears) **var**. If **var** is a list element, it is removed from the list. Hierarchy copies (when appropriate) are not effected.

2.6. Statements

A Stred routine either succeeds or it fails. Each statement in a Stred routine either succeeds or fails, and the failure of the statement causes the routine to fail (immediately). The routine succeeds if all statements executed in it succeed. There are four types of statements, described below. In the descriptions, an optional section is enclosed in bold square braces (**[]**). A list of statements is specified by <stats>. An expression (see section 2.5) is specified by <expr>.

Simple

<expr> **[**?(<stats>)**]** **[**:(<stats>)**]**;

If the expression succeeds then the statements in the **?** clause are executed. If there is no **?** clause then the statement succeeds, otherwise the statement succeeds only if all of the **?** clause statements succeed. If the expression fails then the statements in the **:** clause are executed. If there is no **:** clause then the statement fails, otherwise the statement succeeds only if all of the **:** clause statements succeed. If this statement succeeds, the next statement is executed.

Repeat Loop

\ <simple>

This is similar to the simple statement except that if the expression (in the simple statement) succeeds and the simple statement itself succeeds, then it is re-executed, instead of continuing on to the next statement. If the expression fails, then this behaves the same as the simple statement.

Do-While Loop

?(<stats>) <expr> **[**:(<stats>)**]**;

This is similar to the repeat loop except that the expression is executed after the **?** clause. The **?** clause statements, therefore, are always executed at least once.

Control

\! causes the immediate termination, with success, of any loop. It may only appear inside the ? clause of a loop statement. It terminates only the innermost enclosing loop.

?! This statement causes the immediate termination, with success, of the routine.

:! This statement causes the immediate termination, with failure, of the routine.