CedarPrimer.tioga
Copyright © 1990, 1992, 1993 by Xerox Corporation. All rights reserved.
Brian Oki, October 4, 1990 10:40 am PDT
Brent Welch November 5, 1990 11:59 am PST
Christian Jacobi, April 21, 1993 5:25 pm PDT
Chauser, November 12, 1993 4:50 pm PST

CedarPrimer 1993

How To Get Around Cedar - CDROM Edition

Brian Oki, Christian Jacobi et al.

Filed on: [Cedar10.1]<CedarDoc>CedarPrimerCDROM.tioga

Abstract: This document is a general introduction Cedar and is geared towards the needs and interests of newcomers to the Cedar programming language and environment. It describes starting the Cedar environment, printing on-line documents, programming in Cedar, and some lore about Cedar.

[**If you are looking at this document on-line from within the editor named Tioga, you might want to use the level functions to its overall structure. Click the ‘‘Levels’’ button in the top menu, then click ‘‘FirstLevelOnly’’ in the new menu that appears. That will show you the major section headings. Click ‘‘AllLevels’’ to read the details.**]

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

1.1 Foreword

The purpose of this document is to help newcomers adapt to the Cedar computing environment. We hope it will prove useful as a primer on getting started (and surviving) with Cedar on Sun SPARCstations. In particular, we describe how to start the Cedar environment (a *world*), printing on-line documents, and a bit about Cedar programming.

This document does *not* teach you the Cedar programming language, but it does provide a number of hints and summaries on the subject; these may or may not be documented elsewhere. See LanguageOverviewDoc.tioga for an introduction to the Cedar language.

A great deal of useful information is available on-line in the form of documents and source programs. Reading them is often very helpful, but finding them can be a nuisance. Throughout this document, references to on-line material are indicated by <*n*>, where *n* is a citation number in the bibliography at the end of this document. Standard citations to the open literature appear as [*n*]. To help the browsing reader, we pretty much have abandoned the custom of defining terms before using them. Instead, relevant terms, acronyms, and such are collected in <73>. You can easily jump to the definition of a reference by selecting <*n*> or [*n*] and clicking on the "Def" button. Include the brackets in the selection. You can jump back to where you were by clicking the "PrevPlace" button. You can experiment now with <*n*> or [*n*].

A starting point in your on-line documentation search is **CedarDocDoc.tioga**. It contains pointers to other useful documents to read. You can open it now by typing "open CedarDocDoc" at a commander prompt.

1.2 Roadmap

This document consists of six sections. Section 2 tells you what magic incantations you need to invoke to bring up a Cedar world on your SPARCstation, some details about the window for typing commands, how to tailor your system to suit your personal style, the meaning of the buttons, and a short introduction to editing using Tioga,.

Section 3 discusses how to print documents.

Perhaps the most important section is Section 4, which introduces the tyro user to the mysteries of programming in Cedar. In particular, it discusses the idiosyncrasies of naming, contains a brief history of the Cedar programming language, describes the file name space is organized, discusses system modelling using DF files, and contains a walk-through of a simple Cedar example.

Section 5 contains a bit more about some of the on-line documentation.

We conclude with a long list of references.

Note: To assemble this document I lifted portions of documentation from various sources without always acknowledging the sources or the authors; my apologies to those possibly offended.

Acknowledgments. Thanks to Ken Pier and Brent Welch for reading a draft of this document in its entirety and suggesting numerous changes that significantly improved its presentation. Thanks also to Jim Foote and Sharon Johnson for critiquing the sections on mail and programming. Finally, thanks to Craig Mudge, David Goldberg, John Coolidge, Vijay Saraswat, and countless others for reporting bugs.

2. Getting started

2.1 Bringing up the Cedar world from the CDROM

How do you bring up a Cedar world? The following assumes that you have already started the X Windows system and have mounted the CDROM as a unix file system somewhere in your network. You need to cd to the right place on the CDROM and set up an environment variable. Type the following to a csh shell prompt (if not using the C shell do whatever you have to do to set the environment variable):

```
% setenv XeroxCedar <cdrom-mount-point>/Cedar

% cd $XeroxCedar/solaris1

% x11v
```

Of course, if you are running on a Solaris2 machine use "solaris2" in the second command above.

Various things will scroll by in the window where the shell was running, and eventually a large window will be created that will contain the Cedar windowing system called *Viewers*. When activity stops, on the right side of the screen is a window (or ‘‘viewer’’ in Cedar lingo), called the **Commander** viewer. The **Commander** is the viewer in which you type your commands at the % prompt. There will also be a commander prompt in the X window you used to start Cedar.

Note that "killing a connection" won’t destroy a Cedar Viewers world. The X11ViewersWorld attempts to reconnect to an X server if it loses its connection. This is useful if your X server crashes. Just restart X and your Cedar world springs back into existence. To stop a Cedar Viewers world either use the Cedar command "ExitWord", the X11Viewers pop up menu or a "delete window" option in the window manager.

2.2 The Commander

The Commander is the viewer with a directory name (initially "/Cedar/Commands %"), as its banner, and is where commands are typed. (The banner is light blue on color displays, reverse video on monochrome displays). You can create additional such viewers by clicking on the Cmd button at the very top of the screen.

Since I’m assuming that you’re seated in front of a SPARCstation, a hard copy of this document in hand, and have managed to bring up a Cedar world, let’s type some commands to the Commander. Let’s list the current directory. Type the following to your Commander (the Commander prompt is a bold-faced percent sign **%**), and hit RETURN:

```
% cd
  /yourhomedirectory/
% ls
/yourhomedirectory/
.              1536 05-May-90 15:09:45 PDT
..              512 07-May-90 09:47:33 PDT
.cshrc          903 04-May-90 13:47:37 PDT
.emacs           71 08-Mar-90 09:43:57 PST
.login           38 07-Feb-90 12:50:07 PST
.logout          56 27-Mar-90 14:29:51 PST
```

...

This is the Cedar-style directory listing. You can stop the listing by clicking with the left mouse button on the STOP! button at the top of the Commander viewer.

You can use the * wildcard in file names:

% ls *.mesa

You can find out the usage of a command with the ? command:

% ? ls
List {switch | pattern}*
Lists files matching pattern.
-a print attachments
-b brief format
-d date sort
-f full name print
-h highest version
-k keep print
-n narrow print
-o one line
-p prefixes only
-s size sort
-t file type print
-x exact level match
-u unattached files only
-z 0-length files only
-> sort decreasing
-< sort increasing

You can change directory in the hierarchical file system with the cd command.

Cd with no arguments returns you to your home directory.

Given that you have a hard copy of this document in hand, let's try something more ambitious: open a viewer on this document so you can view it on-line. Type the following to your Commander, and hit RETURN:

% open /Cedar/CedarDoc/CedarPrimer.tioga
Created Viewer: /Cedar/CedarDoc/CedarPrimer.tioga

A viewer will be created, displaying the contents of CedarPrimer.tioga.

Errors caught by the Commander

We're getting a bit ahead of ourselves here, but while you play around with the Commander, you might run some program that bombs out. The Commander tries to catch any uncaught errors. The Commander asks you to do something. What do you do?

*** Uncaught ERROR or SIGNAL: unrecognized error
*** Do you want to try to debug this? (y, n, s <flags>, or ?)

Typing ? is the help command:

'y' to REJECT the signal and land in the system debugger

'n' to get back to top level

’s <flags>’ to invoke the StackTrace command (prints an abbreviated call stack for your viewing pleasure)

If you respond with **n** the commander will abort the errant thread. This is generally ok, although occasionally this can leave your world in a bad state. If you respond with **s** you get a stack trace that might give you some insight into the problem. <This probably doesn’t work in the PPCR based world released on CDROM). After getting the stack trace you are given the prompt again and can abort the thread with an **n** reply.

Scrolling

Position the cursor in the Commander viewer and move it to the left until you nearly touch the black border. A gray, vertical, scroll bar should appear. You should notice two things: first, the cursor changes shape to a double-headed arrow, and second, there’s a dark gray glob. That gray glob indicates your relative position in this viewer. Now experiment with the mouse buttons. LEFT-click means to depress the left mouse button, and so on, for the other two buttons:

LEFT-click: Scroll to top of viewer

RIGHT-click: Top of viewer to click point

MIDDLE-click: Scroll to absolute position.

Selecting/copying

Rather than typing the same pieces of text over and over again, you can use the selection capability to copy pieces from elsewhere in the Commander viewer. Imagine that I’d like to view my .tioga file, and I’d already used the ‘‘ls’’ command to list the files in my home directory. It’s a pain to type the whole string again. Let’s say that you know there’s a document on how to use Cedar, but don’t quite remember the name. You know that the name starts wtih ‘‘Cedar’’, followed by the .tioga extension. * is a wildcard that matches anything.

% ls /Cedar/CedarDoc/Cedar*.tioga
/Cedar/CedarDoc/
CedarDocDoc.tioga!7 5444 14-May-92 12:20:20 PDT
...
%

Type ‘‘Openr ’’ with the space to your Commander. The ‘‘OpenR’’ command says to look in the Cedar system release for the file in question. The release is maintained as a version map that lists the directory and version number of each item in the release. Move the cursor to the file name listed and do the following:

1. Hold down the SHIFT key, and MIDDLE-click on ‘‘CedarDocDoc.’’ A gray underline appears under the word. This indicates that you’re performing a copy operation.
2. Keeping the SHIFT key depressed, RIGHT-click and drag the mouse cursor arrow rightward until the gray underline also underscores the .tioga extension.
3. Now release the SHIFT key. You should now see the following at your Commander:

   % OpenR CedarDocDoc.tioga
4. The flashing cursor should be positioned at the end of the line. You can hit RETURN if you like, but this example is meant to illustrate how to select and copy a piece of text from one place to another. If you do hit RETURN a new viewer will pop up. You can close the viewer with a RIGHT-CLICK over the menu bar. You can destroy the viewer with a LEFT-CLICK over the `Destroy` button, which you can find underneath the

banner. To abrogate the command, hit the DELETE key.

% OpenR CedarDocDoc.tioga -- *<DEL>*

If you select the wrong thing, that's okay; just hit the DELETE key before releasing the SHIFT key; that will unselect your selection.

Commander buttons

Here's a brief explanation of the buttons at the top of each Commander, placed there by the Commander mechanism (others can be created by your profile).

Find button. This button appears to the row of buttons just below the blue banner of the Commander. Think of this command as a postfix operation, that is, you select the argument first and then click on this button, unlike typing commands. Select some text with the mouse and then search for the text within the Commander viewer (you can also search for text within other viewers, but we won't get into that). In this command, the direction of search and capitalization are determined by the mouse buttons and SHIFT keys. LEFT-clicking searches forward from the current selection point; RIGHT-clicking searches backwards; MIDDLE-clicking does a wrap-around search by first searching forward to the end of the document then searching from the beginning until the entire document has been searched. By default, the search matches capitalization. Clicking with the SHIFT key down invokes a 'caseless' search in which capitalization does not matter.

STOP! button. Attempts to stop the current Commander process running in the Commander viewer. It raises the signal ABORTED in the Commander process, which usually manifests itself with the printed string ‘‘ -- Aborted’’, along with a new Commander prompt. This may not always work because, in Cedar, it's up to the process to check whether a user has requested an abort. But it works for the most important commands.

Viewer buttons

Since the Commander is itself a viewer, it has buttons in common with other viewers. You can find these buttons beneath the light blue banner of the viewer by moving your mouse cursor on top of it.

~~Destroy~~ button. Destroy the viewer. This is an example of a guarded button: click it once and the guard goes away. Click again at least .1 second later and not more than 4 seconds later and it does its action. The Reset and Store buttons in Tioga viewers are other examples of guarded buttons.

Adjust button. Changes the size of the window. Click on this button and move the mouse cursor around to see its effects.

Top button. Moves the viewer in question to the top of the screen, moving other viewers further down.

Grow button. Allows a viewer to take up the entire side of the screen, in effect, ‘‘growing’’ to fill the whole side. Other viewers will be turned iconic. Clicking on this button again, in the full grown viewer, restores the viewers to the positions they were in before you clicked Grow. Growing/shrinking windows is a fairly common operation, so a shorthand was devised: MIDDLE-click in the banner selects the Grow button by default.

Close button. RIGHT-click in the banner selects the Close button by default. This is much easier than finding the button.

For more information on the Commander, please consult <70>.

2.3 User profile

Many components of Cedar permit the user to customize system behavior along certain predefined dimensions via a mechanism called the *User Profile*. Whenever the user boots his Cedar world, his user profile is consulted to obtain the value for these parameters. The user profile goes under the name `.cedar.profile`, it is stored in your home directory. To just run Cedar from the CDROM you don't need a user profile, but if you may find that you wish to customize the way Cedar starts up.

To customize your Cedar environment to your liking, copy CedarNoviceUser.profile from the Cedar directory by typing the following commands to your Commander Viewer:

% cd
% copy .cedar.profile _ /Cedar/CedarDoc/CedarNoviceUser.profile

(_ will display on the screen as a left-pointing arrow. The keypad 4 key will also work. Note the spaces surrounding the _) Having copied the user profile, you need to modify it. Create a viewer.

% open .cedar.profile
Created Viewer: .cedar.profile

Select anywhere at the beginning of the viewer. Find the key on the right hand side of the keyboard, labelled either ''6'' or ''R12.'' This is the so-called NEXT key. Punch it once; you'll notice that it looks for placeholder markers that look like this ▶stuff in between◀, changes to reverse video, and positions the cursor at the beginning. Just for fun, select the word ''Select'' at the beginning of this paragraph. Now NEXT to see what happens. Alas, it doesn't wrap around, so to go backward, hold down the SHIFT key and depress NEXT.

1. Type the following depending on what's between the placeholders.
    a. ▶YourHomeDirectory◀. Type /net/palain/rosa/boki, for example, or /net/palain/palain/welch, for another (but don't use those; Oki and Welch won't be very happy if you mess with their directories)
    b. ▶YourName◀. Type your user name in lowercase, say, boki or nichols, at the cursor.
2. Depress NEXT again to get to the next occurrence where you type type your name.
3. Go back to step 1. Continue this procedure until you reach the end of the file.
4. Click on Save button in that viewer to save the changes.

Now let's look at the profile more carefully. Notice that each entry in this profile is of the form <key>: <value>, where key is typically a string (case does not matter) and value is either TRUE or FALSE; an INT; or a sequence of strings. For example,

Tioga.TryVersionMap: TRUE

The Tioga editor looks in your profile to see what options are set. In this case, TryVersionMap tells the editor to use the version map mechanism to help it find files.

The above is a particularly simple example of setting an entry to either TRUE or FALSE. A more complex example is the **BootCommands** entry, which the Commander looks at when initializing itself after booting. BootCommands is run once, when you boot your Cedar world.

Note that a double quote at the beginning and end of this entry encloses the set of commands.

CommandTool.BootCommands: "

BootCommands are run once, when you boot a Viewers world. First we get the aliases and prefixes defined in the NoViewersBootCommands.

CommandsFromProfile CommandTool.NoViewersBootCommands

Require commands are used to load packages and define commands.
Require <world> <component> <resource>
The <world> corresponds to a prefix map entry and it defines the main software package.
The <component> refers to a package in the world, i.e. Imager, Tioga.
The <resource> refers to a "resource.require" file found with the package.
In effect, Require means
CD /<world>/<component> ; Source <resource>.require

Require Cedar IconHacks IconHacks
Require Cedar TiogaExecCommands TiogaExecCommands
Require Cedar TiogaExecCommands TiogaExecViewerCommands

Term starts up a vt100-like terminal emulator for UNIX hacking.
Term
Clock

EditTool is a useful adjuct to tioga. It lets you search/replace, change fonts, etc. It has some silly default settings, so we correct them.
EditTool
DoTiogaOps SaveSelectionA MatchCase MatchLiterally LeaveInitCap SubstituteInSel MatchAnywhere EqualLooksTest DoReplace TypeName \"code\" StyleName \"Cedar\" PropName \"Comment\" RestoreSelectionA

TiogaLineNumberButton adds a tioga button that turns on line numbering.
TiogaLineNumberButton
KeyboardScan turns on an incremental search mode in Tioga.
KeyboardScan

TiogaDWIM turns on a "do-what-I-mean" mode.
TiogaDWIM

Viewers-oriented aliases
Alias myprofile open /tilde/▶YourName◀/.cedar.profile
Alias profile(user) open /tilde/user/.cedar.profile
...
CommandsFromProfile CommandTool.PerCommandTool
"

The **PerCommandTool** entry contains commands that are run for each new Commander viewer. Mostly these create new buttons for the Commander. For example,

```
CommandTool.PerCommandTool: "
    CreateButton Open Open $FileNameSelection$
    CreateButton New New $FileNameSelection$
    ...
    "
```

creates two new buttons in the Commander viewer, one called Open, the other called New.

If you change your profile type ‘‘ProfileChanged’’ to your Commander. This action causes the new user profile values to take effect.

Much of this may be mysterious until you’ve been around for a while, but your profile can make the Cedar environment much more pleasant to work in.

2.4 Static Buttons

These are the buttons at the very top of the screen, from right to left:

X11Viewers: Invokes a pop-up menu with operations including the very important "Exit Cedar". To get rid of the pop-up click on "Dismiss".

New: Create a blank Tioga viewer bearing the path name in the banner of the home directory.

Open: Open a file by its full path name, given a mouse-selected text.

Cmd: Create a new Commander viewer

2.5 Some of the most common/useful Commands

Cd *directory*
: Change directory, as in Unix.

Push *directory*
: Change to a new directory, but remember the previous one on a stack.

Pop
: Pop the directory stack, returning to the previous one.

List *args*
: File listing program, gives file date and size. (Also known as ‘‘LS’’.)

Open *file*
: Open a file for reading/editing (defaults to current working directory, followed by /R/ if no path name is specified).

FindR *file*
: Find out where a system file lives in the released version of software.

OpenR *file*
: Open a system file from the released version of software (/R/<filename>).

Copy *destination _ source*
: Copy *source* file to *destination*. If *destination* is a directory (i.e., ends in ‘‘/’’), then *source* can be a list of files. (_ will be displayed as a left-pointing arrow. The keypad 4 key also works.)

Rename *destination _ source*
: Like copy but just changes the name of a file.

DelVer *file-pattern*
Delete all but the most recent version of some files.

? *command-name*
Print a usage message regarding the command.

Bringover *DF-file*
Copy into the local directory all the source, mob, and object files specified in the DF file.

Smodel *DF-file*
Save your files in a stable place.

GetFromRelease
A command to give in response to the Compiler complaint that it can't find the files it needs.

Run *file*
Loads the C object file into the system. The global default for loaded code is optimized.

MakeDo -df *DF-File*
Rebuild an entire package, first determining dependencies. Creates optimized object files (sun4)

MakeDo *sun4/foo.c2c.o*
Generate object code for SPARC for the implementation file named *foo.*

Implementor *command-name*
Print the name of the procedure that implements the named command.

Mkdir *dirname*
Unix-like shell command to create a new directory in the current directory.

Rmdir *dirname*
Unix-like shell command to delete a directory from the current directory. You must first delete all contained files.

Plumb *machine-name*
If no argument, then defaults to local host machine. Opens a typescript window that emulates the Unix shell of *machine-name.* This is a very useful command, especially if you need to drop down to Unix occassionally.

2.6 When Cedar dies unexpectedly

X11Viewers shows its main menu when you try to delete its X window. This menu has a rescues sub-menu with an entry to save your edits. These menus are most often still alive even when viewers are wedged.

2.7 Short introduction to editing using Tioga

In the world outside PARC, document production systems are usually de-coupled from text editors. One normally takes the text that one wants to include in a document, wraps it in mysterious commands understood by a document processor, and feeds it to that processor. Programmers may think this is neat; after all, one can do anything with a sufficiently powerful programming language (remember, Turing machines supply a sufficiently powerful programming language too!). However, document processors of this sort frequently define semantically complex languages, and one soon discovers that a great deal of time goes into the edit/compile/debug cycle.

Tioga is the editor within Cedar, and it is generally free of imbedded commands to a document processor. It shows formatted text on the screen, or WYSIWIG (‘‘What You See Is What You Get’’). When printing a Tioga document, WYSINAWIP (‘‘What You See Is Not Always What Is Printed’’). See TiogaDoc.tioga <29> for details.

Nodes

A Tioga document is a tree of nodes of text (or graphics). Think of a node as a selectable, meaningful unit, defined by the user. For example, in this document, this paragraph you are now reading is a node. The next paragraph is also a node. The bold-faced heading ‘‘2.9 Short introduction ...’’ is the *parent* node of these two paragraph nodes, which are its *children*. The tree structure of a document usually manifests itself visually as nesting; thus, the two children nodes are nested (indented) from the parent node.

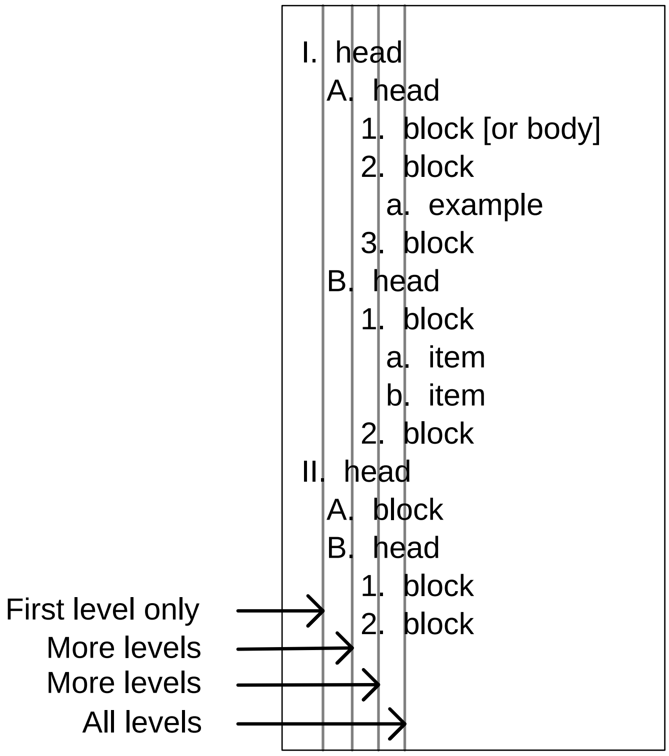
Nodes are separated by special node break rather than the usual carriage return. A CTRL-RETURN creates a node break. At the node break, you can create another meaningful unit of stuff, at the same depth of the tree as the previous node; in other words, a sibling node. CTRL-I both inserts a new node and nests it, making a child node. CTRL-SHIFT-I both inserts a new node and unnests it; think of this command as creating a new node at the same level in the tree as the parent, that is, create a new node at level i-1, if you’re at level i.

The previous commands created new nodes. There are commands to change the nesting level of already existing nodes. You must first select a node, using the mouse, and then invoke the command. For example, move the mouse cursor over this paragraph (anywhere will do), and LEFT-click twice in rapid succession. The entire paragraph is underlined, indicating selection. Next, type CTRL-N, to nest the node. To unnest it, type CTRL-SHIFT-N. Then, to ensure that you don’t accidentally change the contents of this document out on the file server (of course, not possible on the CDROM), click the ~~Reset~~ button at the top of the viewer twice in rapid succession; it will reset the contents.

Perhaps you’ve already noticed that you can select individual characters with a single LEFT-click with the mouse, words with a single MIDDLE-click, and individual nodes with a DOUBLE-click with the mouse.

Structuring a document as tree of nodes is quite useful because it makes it easy to hunt through a long document, provided it is well-structured. Why? Because you can search through it at different levels. To view this document at different levels, click on the Levels button in this viewer’s banner, which brings up another line of buttons: FirstLevelOnly, MoreLevels, FewerLevels, and AllLevels. With these you can decide just how much detail of a document you want to view at any given moment. (To see the following illustration, type ‘‘Artwork on’’ to your Commander.)

viewing a document at many levels



If you clicked FirstLevelOnly, then only the head nodes (designated here by I. and II.) would be displayed. If you clicked MoreLevels, then you'd be able to view the next level of nodes. Clicking on AllLevels would show you the entire document.

Format

In Tioga editing, we name nodes, which will in turn give them formats. In Cedar, we will rely on the *EditTool* to help us do this. If you don't already have an EditTool running, type ''EditTool'' to your Commander. An icon in the shape of a piece of paper, bearing the words ''EditTool'' and a pencil scribbling on the paper, will appear. MIDDLE-click to open a viewer.

Particular types of documents will tend to make use of certain groups of nodes. For example, on a title page you'll tend to find or use the following nodes:

title, subtitle, authors, boilerplate, abstract, copyrightNotice

On pages of text or prose, you'll tend to find/use these nodes, among others:

head, body, block, item, indent, example, center, note

Click on the Format button in your EditTool. Double LEFT-click on the line beginning with the word ''Particular.'' Having selected that node, click on the Get button in your EditTool. The name of the current format of that selected node will appear in the area labelled ''Format name.'' You might try selecting various other nodes just to see what format they're in.

Each node *format* contains its specific recipe for the structure of how that node will look. Title nodes have larger, bold text, with a specific amount of space allotted above and below each line. A body node is one type of node used for paragraphs. Its ''recipe'' (or structure) regulates that the node will have a specific font type and size, indentation of the first word, and predetermined spacing above and below lines at the beginning, middle, and end.

When you apply a Format to a node, that node will take on a structure specific to the type of node you have named it. It would be accurate to say that when formatting a document with Tioga editing tools, you are adding a structure or foundation to the node. This structure (formatting) will determine how the node will look.

Looks

Looks are a way of making changes to the appearance of text within a node, without changing the structure of the node. You would add *looks* to add italics, boldness, or underlining; or to change a word to a superscript, subscript, or larger or smaller size. Different looks can be added to different words within a node, and multiple looks can be added to one selection.

Once you've selected the words to which you want to add Looks, you do so by either using the EditTool or by pressing the Looks key (key R9 on the righthand keypad) and the appropriate lower case code letter on the keyboard. For a good summary of the Looks available, consult SampleSheet.tioga <67>.

Style

Appearance is influenced by the format of the node, which determines things such as vertical and horizontal spacing. The characters of the text within a node can have looks that control various aspects of their appearance such as font and size. The document refers to the names of looks and formats, but not to any specific interpretation of them. A *style* is a collection of interpretations (or ‘‘complementary flavors’’, if you will) for looks and formats. This document is composed with the default Cedar style.

Styles can be shared by many documents. For example, in the style for this document there are definitions of formats for titles, headings, and standard paragraphs, and there are definitions of looks for emphasis and for small caps. Rather than specifying all the details for the formats and looks, the document refers to formats and looks by name (like ‘‘title’’, ‘‘block’’, ‘‘abstract’’). These names are defined in the style, so it's easy to change the definitions in the style and modify the appearance uniformly throughout the document.

Properties

Adding a *property* is a way of making changes to <u>specific</u> characters or nodes in a way that goes beyond the boundaries of what's included in a particular *style*. Most often, you'll hear about Postfix properties. For example, you could use Postfix properties to add background color to a node, to change the color of the text itself, to place boxes around words without using a graphics program, or change the size of margins or indents. Find a Postfix wizard if you'd like further details.

Dealing with Editor Bugs

All text editors have bugs. The most common -- perhaps we mean ‘‘the least rare’’ -- source of editing disasters in Tioga is problems with monitor (or ‘‘viewer’’) locks. Unfortunately, this class of problem usually makes further progress in any part of Cedar impossible.

In X11Viewers: The "delete window" from a window manager shows a pop up menu which allows selection of a "save all edits" command..

Non-Textual Editors

We humans often convey information more effectively in ways other than the written word. Cedar likewise provides tools for manipulating representations of information other than text, such as 2-D illustrations, mathematical expressions and voice.

Gargoyle [45] is an interactive 2D illustrator for creating color pictures. It includes novel features to aid the user in precise geometric placement of objects in the scene. Refer to [46] for an on-line tutorial. I used Gargoyle to draw all the pictures in this document.

CaminoReal[44] is an environment for two kinds of manipulations of mathematical expressions: (1) interactive, syntax-directed, two-dimensional, WYSIWYG editing and (2) algebraic manipulation.

Because Tioga provides a rich common ground, the above heterogeneous data (image and voice) can be integrated into a single source (namely a Tioga document) for later browsing. The other editors, of course, recognize Tioga's role as chief editor and each provides a facility to insert (‘‘Stuff’’) its data into a Tioga document.

2.8 Exiting Cedar

Click on the X11Viewers static button to bring up a pop-up menu. Move the mouse cursor to the words ‘‘Exit Cedar’’ and click on it. Your X window containing the Cedar world will vanish.

3. Printing

3.1 Interpress and Postscript

Interpress [51] is a Xerox standard that permits every printing client to use every printer. Cedar is most adept at converting its online document formats (.tioga, .gargoyle, etc.) to Interpress. Outside of Xerox, however, the world has standardized on Postscript printing and Interpress printers are uncommon, so a further conversion of the Interpress file to Postscript will likely be required. There may be some confusion about fonts in this latter conversion, but the result is usually a readable, though seldom beautiful, printed page.

Converting to Interpress. The following commands produce Postscript from tioga.

% TiogaToInterpress foo.interpress _ foo.tioga
TiogaToInterpress is a program that converts Tioga documents to Interpress format.
% IPtoPS foo.ps _ foo.interpress
Converts interpress to Postscript.

Use the Unix lpr command to print the resulting file on your favorite Postscript printer.

4. Programming

4.1 Naming conventions

The prevailing local philosophy about naming is perhaps different from elsewhere. We have our share of alphabet soup, (that is, acronyms of varying degrees of cuteness and artificiality, like PARC, FTP, IFS, PUP). But we are trying to avoid worsening this situation. To this worthy end, Grapevine (mail) servers are named after wines, Dorados are named after capital ships, some Suns are named after places in E. E. ‘‘Doc’’ Smith science fiction novels, and so on. As these conventions don’t meet with universal approval, you will doubtless provoke interesting discussions if you advance your own views on naming to almost anyone wandering in the corridors.

While we are on the topic of names, let’s discuss for a moment the local customs for constructing single identifiers out of multiple word phrases. Suppose you’d like to name a variable ‘‘name several words long.’’ In some environments, a special character that isn’t a letter but acts like a letter is used as a word separator, leading to names such as

‘‘name!several!words!long’’ or ‘‘name_several_words_long.’’

No such character is in common use locally, however. Instead, we shift between upper and lower case to show word boundaries, leading to the name

‘‘NameSeveralWordsLong.’’

Some people think this looks terribly ugly. We won’t express our opinions in this document, but once again exhort you to espouse your views in the corridors.

As a general rule, case is significant for identifiers in the Cedar programming language, but not significant in file names or mail names. Thus, the Cedar identifiers ‘‘REF’’, ‘‘Ref’’, and ‘‘ref’’ are quite distinct, but the file names ‘‘BriefingBlurb.tioga’’ and ‘‘briefingblurb.tioga’’ are equivalent, as are the mail names ‘‘BOki.PARC’’ and ‘‘boki.parc.’’ In Cedar, there is a further convention that the case of the first letter of an identifier distinguishes fancy objects, such as procedures and types, from simple ones, such as integers and reals. Thus, the identifier name ‘‘ProcWithFiveWordName’’ begins with an upper case ‘‘P’’, but the name ‘‘integerWithFiveWordName’’ begins with a lower case ‘‘i.’’ The latter form looks strange to most people when they first see it. When you first tasted an olive, you probably didn’t like it. Now, you probably do. Give these capitalization conventions the same chance that you would an olive. Caveat: In Cedar, case does not matter in a file name if the file you’re interested in happens to be in a versioned, case-insensitive directory like the Cedar system release, so LoganBerryImpl.mesa and loganberryimpl.mesa are the same file; in a Unix-like directory, case *does* matter. These distinctions will become clearer further ahead.

These capitalization conventions don’t work too well when acronyms and normal words appear together in one identifier. Suppose, for example, that we wanted an identifier named ‘‘FTP version number.’’ Since it’s an integer, probably ‘‘ftpVersionNumber’’ is best.

4.2 Cedar programming language

Cedar

In 1978, CSL began to consider what programming environment to use on the emerging D-machines, such as the Dorado. CSL chose to build a new programming environment, based on the Mesa language, that would be the basis for most of our programming during the following years. That new environment is named ‘‘Cedar.’’ We refer the interested reader to [40] for an introductory ‘‘tour’’ through the Cedar programming environment and to [<42>] for an in-depth description of the overall structure of Cedar and its organization.

Cedar documentation is always in flux; indeed, it might be said that Cedar as a whole is in a constant state of flux. Much of the documentation for the current release is accessible through a ‘‘.df’’ file named Documentation.df <25>.

Cedar’s predecessor, Mesa, was a strongly typed language designed and built locally. Although Mesa programs look a lot like PASCAL programs when viewed in the small, Mesa provides and enforces a modularization concept that allows large programs to be built from smaller pieces. These pieces are compiled separately, but the strong type checking of Mesa is enforced between different modules. The basic idea is to structure a system by determining which functions are supplied by particular portions of the system to other portions. This supply of functionality is called an ‘‘interface;’’ it is manifested in a Mesa source file called an ‘‘interface module.’’ An interface module may define types and procedures that act on values of those types. Only the procedure headers go into the interface module, not the procedure bodies. This makes sense, since all the interface module has to do is to give the compiler enough information to type-check programs that use the interface.

An interface is implemented by an ‘‘implementation module,’’ which contains the actual procedure code. An implementation module is said to ‘‘export’’ the interface that it is implementing; it may also ‘‘import’’ other interfaces that it needs to do its job, interfaces that some other program will implement.

In simple systems, each interface is exported by exactly one module. In such a system, it is clear who should be supplying which services to whom. In fact, in these simple cases, the *binding*, that is, the resolution of imports and exports, can be done on the fly by the loader. But in more complex cases, there might be several different modules in the system that can supply the same service under somewhat different conditions, or with somewhat different performance. Then, the job of describing exactly which modules supply which services to which other modules can become subtle. A small language was devised to describe these cases. A program reads such a description, called a *config*, and builds a runable system by filling imports requested from exports according to the config.

The programming language underlying Cedar is essentially Mesa plus garbage collection. Now, garbage collection changes one’s programming style in large systems. Without garbage collection, one must enforce some conventions about who owns the storage. When I call you and pass you a string argument, we must agree whether I am just letting you look at my string, or I am actually turning over ownership of the string to you, because it’s the owner’s responsibility to reclaim the string’s storage and prevent a storage leak. With garbage collection, most problems like this go away: the garbage collector owns all storage; it gets reclaimed when it is no longer needed, and not before. But there is a price for this convenience: the garbage collector takes time to do its work. In addition, all programmers must follow certain rules about using pointers so as not to confuse the garbage collector about what is garbage and what is not.

Cedar is really two programming languages: a restricted subset called the *safe language*, and the unrestricted full language. Programmers who stick to the safe language can rest secure in the confidence that nothing they write can confuse the garbage collector. Their bugs will not risk bringing down the entire environment around them in a rubble of bits. Those who choose to veer outside of the safe language should know what they are doing.

Programs in the programming language underlying Cedar look a lot like Mesa programs, but, on a deeper level, they aren’t really Mesa programs at all. To avoid confusion, we decided to use the name ‘‘Cedar’’ to describe the Cedar programming language, as well as the environment built on top of it.

Cedar, PCedar or ‘‘Portable Cedar’’

Neither the Cedar language nor the environment was originally intended to be portable, and for many years ran only on D-machines at PARC and a few other locations in Xerox. We recently re-implemented the language to make it portable across many different architectures. Our strategy was, first, to use machine-dependent C code as an intermediate language; second, to create a language-independent layer known as the Portable Common Runtime; and third, to write a relatively large amount of Cedar-specific runtime code in a subset of Cedar itself. By treating C as an intermediate code we are able to achieve reasonably fast compilation, very good eventual machine code, and all with relatively small programmer effort. Because Cedar is a much richer language than C, there were numerous issues to resolve in performing an efficient translation and in providing reasonable debugging.

Reference [<52>] is a paper presented at SIGPLAN ’89 Conference on Programming Language Design and Implementation. It discusses how Cedar was re-implemented to make it portable across many different architectures, in particular, the strategy for the compiler and runtime, and the manner of making connections to other languages and the Unix^{TM} operating system.

Mimosa. The Mimosa compiler translates the Cedar language into machine dependent C. There is a front end, compiling into a simple intermediate form, and a back end, translating from the intermediate form to C code. In the future, back ends may generate other machine codes, other assembler languages, or other versions of C.

Although treating C as merely an intermediate language has significant advantages, the decision to keep program maintenance in Cedar was made independently on the merits of the Cedar language. We were not willing to do a one-time translation from Cedar to C (even readable C). For example, such things as compiler-generated runtime checks should not be exposed to the programmer for maintenance.

Everything in the front end was affected by the retargeting, and some files were completely rewritten. The use of an intermediate code is completely new. The old code generator was changed to produce the intermediate code, in 16 files and over 8,000 lines of code. A few other files are completely new, for about 30% of the source lines. Since substantial changes have been made to other parts of the compiler, it would be reasonable to estimate that over 50% of the source lines have received major change (not counting the lines that were completely discarded). We estimate that about 2 man-years overall were spent in redoing the front-end.

The back-end generates machine dependent C code from the intermediate form. It was written completely from scratch for the port, consists of 10,000 lines of Cedar in 24 modules, and took about 6 man-months to write.

Portable Common Runtime. The Cedar runtime environment, to which the generated C code is targeted, is written mostly in Cedar, the rest is written in C, except for a small amount of assembler code.

The environment is built in layers. The lowest layer is akin to an operating system, and provides dynamic loading, threads support, and storage management (including garbage collection). This is about 20,000 lines of C code and less than a 100 lines of assembler (either SPARC and Motorola 68020 at the moment). This layer is not specific to Cedar, and is in fact intended to provide a language-independent base for high level languages. We call it the Portable Common Runtime (PCR), which we describe elsewhere [71]. Below we describe the PCR only where its functionality is particularly important to implementing Cedar features.

Support for the remaining features of Cedar is provided by the penultimate layer, called CedarCore, which contains 400 lines of C code and 10,000 lines of Cedar. From CedarCore on, the full Cedar language is supported. The final support layer, BasicCedar, while not necessary for the language itself, contains services that are considered essential for most Cedar applications. For example it includes several kinds of hash table mechanisms and a general-purpose stream I/O package. Cedar lightweight processes, interface binding, and exception handling are handled by the runtime system.

4.3 Portable File System (PFS)

File system access by all Cedar packages is PFS, the new Portable File System interface. PFS provides a uniform, extensible mechanism allowing client programs to use files in any of a number of different kinds of file systems. For example, clients can currently view UNIX™ file systems in either ‘‘Unix™’’ style, i.e., versionless and case-sensitive names, or in ‘‘Cedar’’ style, i.e. versioned and case-insensitive names. We dub the Unix view ‘‘-ux’’ (pronounced ‘‘ucks’’) and the Cedar view ‘‘-vux’’ (pronounced ‘‘vucks’’). There is also a view that uses the Cedar version map as its directory. A view of NS files is in development, as is a DF view. To ease porting of Cedar clients and to allow sharing of source, a Cedar FS compatibility package was also provided.

This work builds on work done in Cedar7.0 for accessing files on many different kinds of file servers, including Sun NFSs, where essentially the same problems had to be solved: a common interface to different, but essentially similar file systems; a registration mechanism to make the solution extensible to new file systems; and a simulation of Cedar-style names using the Unix™ file system. See <53> for details.

PFS introduces the *prefix map* concept to Cedar. Prefix maps introduce a level of name indirection to all file system accesses using PFS. This indirection is used to make client programs less dependent on the actual location of files. Because prefix maps allow replacing an arbitrary prefix of a path by another path, and because this substitution is done uniformly on all paths used by PFS, prefix maps have additional uses in customizing clients’ views of the file system.

Type the following command ‘‘pmp’’ (short for PrefixMapPrint) to your commander. The current default prefix maps will be printed out. It is easiest to think of these prefix maps as top level directories in the Unix style.

```
% pmp
 /cedar10.1    -vux:/project/cedar10.1/release
 /cedar        -vux:/project/cedar10.1/release
 ...
 /             -ux:/
 /r/           -VerMapA:/Source/
```

-Vux view

The following doesn’t apply to Cedar as it appears on the CDROM. The /cedar and /cedar10.1 prefixes are mapped by the CDROM prefix map to ordinary, case sensitive, versionless directories using the -ux:/ prefix.

For example, note that /cedar10.1 and /cedar are both prefixes of a somewhat longer pathname, called /project/cedar10.1/release and is to be viewed as versioned, case-insensitive names, indicated by ‘‘-vux:’’ -- think of -vux as "versioned unix." If I type the ‘‘ls’’ command in the /cedar/top directory, all file names together with version numbers are printed out. A portion of this list looks like the following:

```
% push /cedar/top
/cedar/top/
% ls
/cedar/top/
  Adobe.df!3          12309 21-Apr-92 16:42:08 PDT
  Adobe.df!4          12309 29-Apr-92 14:41:17 PDT
  Adobe.df!5          12336 12-May-92 14:35:27 PDT
  AIS.df!1             3619 10-Mar-92 11:59:50 PST
  Args.df!1            1092 22-Feb-92 10:59:17 PST
  ArpaTransport.df!1   2039 21-Feb-92 23:29:43 PST
```

To those of you who cut your teeth on Unix™, version numbers are probably alien to you. The semantics of files created and modified in a vux view is *immutability*. A file can never be modified in place or overwritten, so you need never worry about clobberting an old version of a file as you make changes. The only way to modify an existing file is to create a new version with the same contents and modify that new version. Reading a file defaults to the latest version, so if you type

```
% open /Cedar/Top/Adobe.df
  Created Viewer: /Cedar/Top/Adobe.df!5
```

to your Commander without specifying a version number, a viewer is opened displaying the latest version; to read a specific version you must supply that number using ‘‘!2’’ for example, if you want version number 2. The obvious disadvantage is that all those versions accumulate and clutter your disk space.

And now for a word on Cedar version maps. To achieve the same effect as the previous command without having to specify the particular subdirectory, like /Top, which you might not even know anyway, you can go through Cedar version maps. These maps represent a kind of latest snapshot of all the software packages making up Cedar. Provided that a package is in the version maps, then you will always get the latest version of the file; otherwise, you must explicitly specify the path. The following accomplishes the same thing as the previous command. In this case, I happen know that Adobe.df is in the /Cedar directory and that there are version maps for /Cedar.

```
% openr Adobe.df
  Created Viewer: /R/Adobe.df!5
```

A word of **warning**: Packages in the Cedar release are, unfortunately, not read-only. It is possible for you inadvertently to modify a file and then Save it. Please be careful!!!

-Ux view

Think of this as the Unix view of the file system. In fact, it is the default view for your root directory ‘‘/’’ and thus for your home directory. In the -ux view, PFS maintains only two versions: the current version and the previous version. This previous version is distinguished by a tilde, ‘‘~’’, appended to the file name. If I modify the current version and write it back, the old previous version vanishes and the old current version becomes the previous version. If you create a subdirectory, it defaults to -ux. Case is important!

Adding a prefix to the map.

Suppose you want to define your own -vux view of some existing directory of yours, say, /YourHomeDirectory/nothing/zippo, then type the following to your Commander:

% pma /zippo -vux:/YourHomeDirectory/nothing/zippo

‘‘pma’’ is short for ‘‘PrefixMapAdd’’ and is the command for adding new entries. There are two arguments: the first one, /zippo, is the prefix directory you wish to define; the second one is the translation, where -vux says you want a versioned view, after the colon, is the path name (ending in a directory name) for which /zippo is the abbreviation.

Then type ‘‘pmp’’ to your Commander, and you’ll see /zippo listed as a prefix map entry with the -vux view. To view this directory as versioned, you must always refer to it via the prefix name, *not* the full path name. In fact, we now have *both* an -ux and a -vux view of the same subdirectory. Any file created in one is visible from the other view.

```
% push /YourHomeDirectory/nothing/zippo
/YourHomeDirectory/nothing/zippo/
% ls
/YourHomeDirectory/nothing/zippo/
.                     512 05-May-90 14:47:07 PDT
..                    512 05-May-90 14:45:17 PDT
.~case~Hello.world      0 05-May-90 14:46:09 PDT
Hello.world            17 05-May-90 14:47:07 PDT
hello.world.~1~         5 05-May-90 14:46:09 PDT
hello.world.~2~        11 05-May-90 14:58:21 PDT
Hello.world~            5 05-May-90 14:45:05 PDT
-- 6 files, 1051 total bytes
```

Here we’re listing the files in the -ux view, and we’re also seeing the files that were created under the -vux view (.~case~Hello.world, hello.world.~1~, and hello.world.~2~). Perhaps the only puzzling file is the .~case~ that precedes the file name Hello.world. It’s there to make the file name case-insensitive.

```
% push /zippo
/zippo/
% ls
Hello.world            17 05-May-90 14:47:07 PDT
Hello.world!1           5 05-May-90 14:46:09 PDT
Hello.world!2          11 05-May-90 14:58:21 PDT
Hello.world~            5 05-May-90 14:45:05 PDT
-- 3 files, 27 total bytes
```

It’s not clear that you really want to mix your -ux and -vux views of the same directory. I think it’s best to maintain the separation.

Removing a prefix from the map

Removing a prefix is easy. Suppose you want to remove the prefix /zippo. The following does the trick:

% pma /zippo

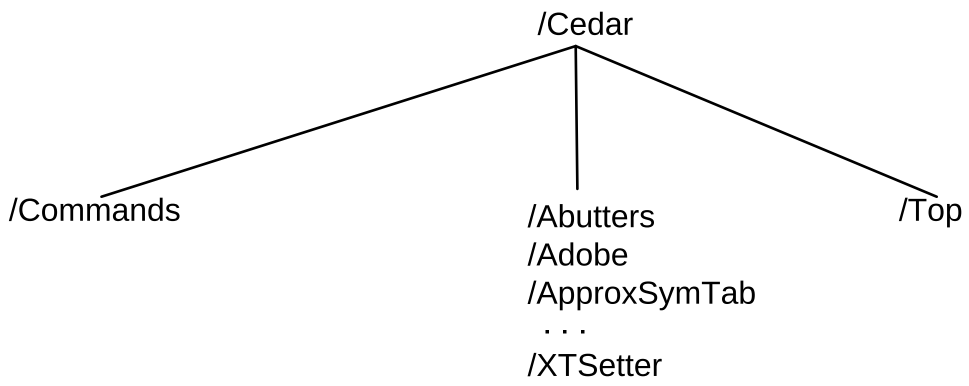
%

Note, however, that the prefix is removed only from the map; the filesystem on disk is unaffected.

4.4 Name space

The Cedar name space of interest to most users essentially consists of the aforementioned Prefix maps that serve as a kind of location-transparent shorthand for file servers. In this section, we discuss how the various name spaces are organized.

/Cedar consists of the following major sub-directories: /Commands, /Top, and all the software packages, from /Abutters to /XTSetter. The illustration below shows this same organization.



Top. This subdirectory contains for each software package the DF file that describes that software package. For example, for the software package LoganBerry, there is one DF file: LoganBerry.df. We defer explanation of DF files to the next section.

Commands. Discussed in the next section.

Software packages. Each package has its own subdirectory in the hierarchy. Thus XTSetter has /XTSetter and LoganBerry has /LoganBerry. Each package subdirectory contains the source files (.mesa, .config), symbol table files (.mob), intermediate code files (.c2c.c), and a subdirectory, sun4. The sun4 directory, contains, as you might expect, all the (optimized) C object files targeted for the SPARC architecture.

4.5 DF Files

Bringover. In the days Before The Change (that is before Cedar on Unix), the Cedar File System was organized into a local file system managed by a user's Dorado worksation and a remote file system, managed by an IFS file server such as Ivy or Indigo: the file server was the truth! If a user wished to modify a software package, he copied the necessary bits from the file server to his local file system, munged with them locally, and then stored them out at the file server, presumably in some consistent state. Actually, that's an oversimplification, if not an outright lie. What really happens is that you create *attachments* to remote files rather than copying them to the local disk. The local disk is just a cache. The attached files may then be referenced by their short file names rather than the full path name including the server and directory names. If attached files were subsequently modified, new versions were automatically created on the local disk. The remote versions would later be updated to reflect these ‘‘dirty’’ files in the cache. The local file system was truly distinct from the remote file server although the software did much to make things transparent. This induced a certain mode of operation called *Keeping your bags packed*. If you spent weeks working on modifying some package without putting out intermediate changes to the file server, you were taking big risks because all manner of nasty things could happen in the meantime to destroy the data on the local disk. Your local copy is indeed only a cached copy of the data on the file server. Dorados themselves were never backed up, but IFSs were.

In the days After The Change, we now use NFS servers (and later, even AFS) to store our files, and the distinction between local and remote files goes away; however, I hasten to add that our SPARCstation's do have 200 Megabytes of local disk space, some of which stores Unix itself and some of which is devoted to swap space, but no one seriously uses the local disk. I note you can get *much* better performance if you use it! Rumor has it that the DF view will provide something equivalent to Dorado Cedar attachments. One pleasant advantage of NFS is that it allows you to always login to your home directory on an NFS server, whatever machine you happen to be on, and thus share the same file system subtree; however, you pay a cost in performance and in network traffic. Viewed through versioned PFS, the files making up the latest Cedar release on /Cedar are immutable, that is, you cannot modify a version in place but must create a new version. One still copies *all* these bits, but instead of to your local disk, to some subdirectory in your file system space.

The Bringover command retrieves to some subdirectory you've specified the set of files described by a particular DF suite. Typically, the DF files specify the Cedar release directory (/Cedar) on which to look for a file, or in some cases, the old Cedar7.0 release (/Cedar7.0). These directories are just hints, in the sense that Bringover will always verify by checking the create date that it is getting you the correct version of the specified file. If Bringover can't find the correct version on the specified directory, it will issue a sprightly error message. Consider our SimpleExample.df file example. Imagine that you've organized your file system so that development takes place in /tilde/boki/Cedar/Development/ subdirectory. Try typing the following to your Commander:

% mkdir SimpleExample
% cd SimpleExample
/tilde/boki/Cedar/Development/SimpleExample/
% mkdir sun4
% BringOver /Cedar10.1/Top/SimpleExample.df
it types a lot of stuff ...
% open ReverseName.mesa
% open Calculate.mesa

On line 1, we must explicitly make a subdirectory called *SimpleExample* in the Development subdirectory we're already in. Then we change directory to that newly created directory. Since the Bringover command, in copying the object files, does not create subdirectories sun4 and sun4-o3, you must explicitly do that yourself. Next, do the bringover. Finally, open some viewers on the two source files making up SimpleExample. You're welcome to stare at the Cedar code on your screen, but we're not here going to explain anything about it.

Smodel. Suppose we modify a local copy of a file described in a DF file; now we want to put the new version back on the Cedar release directory, say, along with a new DF file that describes the new version. This is performed by the *SModel* command (short for Simple Modelling). SModel considers each file described in the DF file, and looks to see if it has been edited it (that is, it looks to see if the create date of that file in my copy of the file is now different from the create date stored in the old DF file). If so, SModel stores the new version onto the /Cedar directory. After doing this for each file, SModel writes a new version of the DF file itself, filling in all of the create dates correctly to describe the new version of the entire ensemble. If the DF file describes itself, as most DF files do, SModel is smart enough to make sure that the new version of the DF file is stored out to the /Cedar release directory as well. SModel also has lots of bells and whistles, but let's not go into them.

A real example. Let's consider a real example. I've extracted the simplest Cedar program from the Cedar examples usually inflicted on summer interns and permanent hires alike and constructed a DF file that packages them up. Without showing you the source code for the programs themselves, here's what the various DF files look like (excluding the Sun4O3 df). It also includes some comment lines (each line is preceded by two hypens --) provided by Brent.

SimpleExample.df. The first few comment lines give the name of the DF file, copyright year, and a line for each person who last modified the file. ‘‘Exports [Cedar10.1]<Top>’’ means that the following files are stored in the Cedar Top directory, in this case, ‘‘SimpleExample.df’’, followed by the create date ‘‘14-May-92 15:15:34 PDT’’; note that [Cedar10.1]<Top> and /Cedar10.1/Top are different notations for the same thing; you will see them used interchangeably.

Includes: (Don't occur in this df file)‘‘Include’’ merely means that the specified file (usually a DF file) is included in the Suite. Think of the ‘‘Include’’ line as syntactic inclusion, similar to macro-expansion, in which the DF-suite references other DF files. Bringover invokes itself recursively on the included file at the point it encounters the Include statement.

```
-- SimpleExample.df
-- Copyright © 1992 by Xerox Corporation.  All rights reserved.
-- DFPort: Christian Jacobi, May 14, 1992 3:01 pm PDT
--
-- Simple example programs useful for introductions to Cedar only
--

Exports [Cedar10.1]<Top>
  SimpleExample.df                   14-May-92 15:15:34 PDT

Directory [Cedar10.1]<Commands>
  Calculate.command!1                14-May-92 14:11:48 PDT
  ReverseName.command!1              14-May-92 14:11:15 PDT

Directory [Cedar10.1]<SimpleExample>

  SimpleExampleDoc.tioga!2           14-May-92 15:14:55 PDT

  Calculate.require!1                02-May-90 12:39:04 PDT
  Calculate.mob!1                    14-May-92 15:00:14 PDT
  Calculate.mesa!1                   14-May-92 15:00:07 PDT
  Calculate.c2c.c!1                  14-May-92 15:00:15 PDT
  +sun4>Calculate.c2c.o!1            14-May-92 15:00:39 PDT

  ReverseName.require!1              02-May-90 10:24:08 PDT
  ReverseName.mob!1                  14-May-92 14:22:28 PDT
  ReverseName.mesa!1                 14-May-92 14:22:19 PDT
  ReverseName.c2c.c!1                14-May-92 14:22:28 PDT
```

```
 +sun4>ReverseName.c2c.o!1                    14-May-92 14:22:36 PDT

Imports [Cedar10.1]<Top>BasicPackages.df Of ~=
 Using [Commander.mob]

Imports [Cedar10.1]<Top>Basics.df Of ~=
 Using [Process.mob]

Imports [Cedar10.1]<Top>IO.df Of ~=
 Using [IO.mob]

Imports [Cedar10.1]<Top>Rope.df Of ~=
 Using [Rope.mob]

Imports [Cedar10.1]<Top>SystemNames.df Of ~=
 Using [SystemNames.mob]

Imports [Cedar10.1]<Top>ViewerIO.df Of ~=
 Using [ViewerIO.mob]
```

As you can see, if we only discussed DF files as descriptions of ensembles of files, then mere mortals could figure out DF files without unduly straining their gray cells. But there's more. In fact, these ensembles are often components of larger ensembles. It's complicated.

DF files grew up over time in response to several needs. As they became more popular, features were added one by one to make them more useful in these varying contexts. The resulting system as a whole is hard to grok, particularly in PCedar2.0, but we hope that Cedar10.1 is a little bit simpler again.

4.6 Reading programs

A Sample Program

About the Cedar code you will see, I won't explain syntax or semantics very much. A tutorial on Cedar programming is beyond the scope of this primer.

After the listing of the ReverseName.mesa module, there is a set of notes to help in reading it. The notes are keyed to the programs by the numbers in parentheses to the *right* of some lines, e.g., as ‘‘--***(note 3.1)***’’. You will also find references to lines in the program from the notes, as, e.g., ‘‘***[see line 1.1]***’’. The corresponding lines in the programs are marked as, e.g., ***[1.1]***. When Cedar identifiers appear in the notes, they are displayed in italics to make reading easier; e.g., *ReverseName*, *CommandProc*. You should read this example for understanding and use the notes as references, rather than the other way around. Consult the on-line documentation *Cedar Language Overview* <57>, which details, somewhat cryptically and hastily, the additional changes that produced Cedar. Finally, see *Language Change Summary* <58> for the latest word on the subtle changes made to the Cedar language definition intended to foster portability of code from one target architecture to another.

ReverseName.mesa --***(note 1.1)***

Brian Oki, May 7, 1990 8:23 pm PDT

```
DIRECTORY
    Commander USING [CommandProc, Register],
    IO USING [PutF, rope, STREAM],
    Rope USING [Cat, Equal, Fetch, Find, FromChar, Length, ROPE, Substr],
    UserCredentials USING [Get];

ReverseName: CEDAR PROGRAM
    IMPORTS  Commander, IO, Rope, UserCredentials
    ~ BEGIN              --(note 1.3)

    ROPE: TYPE = Rope.ROPE;                              --(note 1.4)

    ReverseName: Commander.CommandProc ~ {
              --(note 1.5)
        [cmd: Commander.Handle] RETURNS [result: REF ANY _ NIL, msg: ROPE _ NIL]

            Reverses the user's login name and prints it out in the CommandTool window.
            userName: ROPE _ UserCredentials.Get[].name;              --(note 1.6)
            out: IO.STREAM _ cmd.out; -- cmd is an arg to ReverseName.
            backwordsName: ROPE _ NIL;
            Remove anything after the period in user name, e.g., ".PA", and check for the name
              Weiser.
            dotPos: INT = userName.Find["."];                         --(note 1.7)
            IF dotPos # -1 THEN
                userName _ userName.Substr[0, dotPos];
            IF userName.Equal[s2: "Weiser", case: FALSE] THEN
                out.PutF["Hi, Mark!\n"];
            Now reverse the name and concatenate them in reverse order.
            FOR i: INT DECREASING IN [0..userName.Length[]) DO
                --[1.1] (note 1.8)
                backwordsName _ backwordsName.Cat[Rope.FromChar[userName.Fetch[i]]]
                ENDLOOP;
            out.PutF["Your user name backwards is: %g.\n", IO.rope[backwordsName]]; --(Note
                        1.9)
            };

        Start code registers a ReverseName command, which must be invoked for this program to
          do anything:

        --(note 1.16)
        Commander.Register[key: "ReverseName", proc: ReverseName, doc: "Reverses your
                    user name"];
END.
```

Program notes

(1.1) These stylized comments give the name of the module, and who last edited it. See the document CedarProgramStyle.tioga for a list of the stylistic conventions recommended for Cedar programmers.

(1.3) *ReverseName* imports four interfaces, *Commander, IO, Rope, and UserCredentials* because it needs to call procedures defined in those interfaces. You've seen the Commander's interactive face before. IO provides abstract STREAM operations, and Rope provides immutable string values with lots of operations for creating new ones out of old ones. The student of Cedar would do well to study Rope.mesa, RopeImpl.mesa, IO.mesa and IOImpl.mesa.

(1.4) Since ROPEs are so heavily used in the program, an unqualified version of the type name is generated simply by equating it with the type in the *Rope* interface. This is a commonly used means for making one or a small set of names from interfaces available as simple identifiers in a module.

(1.5) The argument and returns lists given here as comments show the meanging of the type *Commander.CommandProc*; they were inserted semi-automatically using Tioga's ‘‘Expand Abbreviation’’ command to assist in reading the program. *ReverseName* has this type so that it can be registered with the *Commander* as a command that a user can invoke by typing a simple identifier (see note 1.18).

(1.6) *UserCredentials.Get* has the type

PROC RETURNS [name, password: ROPE]

userName is initialized to the value of the ‘‘*name*’’ return value from *UserCredentials.Get* by qualifying the call with ‘‘*.name*’’.

(1.7) The notation

userName.Find["."]

is interpreted by the compiler as follows: Look in the interface where the type of *userName* is defined (*Rope* in this case) and look for the procedure *Find,* whose first parameter should be a ROPE. Generate code to call that procedure, inserting *userName* at the head of its argument list. Thus *userName.Find*["."] is an alternate way of saying *Rope.Find*[*userName,* "."]. The *userName.Find* form is called ‘‘object-style notation’’, and the *Rope.Find* form is called ‘‘procedure-oriented notation’’.

Note also that *dotPos* is supposed to be constant over the scope of its declaration, so it is initialized with ‘‘=’’ (as opposed to ‘‘_’’) to prevent any subsequent assignment to it.

This ‘‘object-style notation’’ might be handy for some well known data types. However usage of ‘‘object-style notation’’ for your own data types makes the program nearly unreadable for casual readers and is strongly discouraged.

(1.8) *Find*, *Fetch*, *Equal*, *Substr*, *Cat,* and *FromChar* are all procedures from the *Rope* interface. The program uses object-style notation for those calls whose first argument is a nameable object, e.g., *userName*, and procedure-oriented notation otherwise. [line 1.1] is a good example of both. Here it is again, spread out to exhibit the two forms:

```
backwordsName _
    backwordsName.Cat[               -- object-style notation
        Rope.FromChar[               -- procedure-oriented notation
            userName.Fetch[i]        -- object-style notation
            ]
        ];
```

(1.9) *IO.PutF* is the standard way of providing formatted output to a stream (much like

FORTRAN output with FORMAT). Its first argument is an IO.STREAM. The second is a ROPE with embedded Fortran-like formatting commands where variables are to be output. The ‘‘%g’’ format is the most useful one; it will handle any sort of variable: INTEGER, CHARACTER, ROPE, etc., in a general default format. The third through last arguments are values to be output, surrounded by calls to inline *IO* procedures that tell *PutF* the type of the argument: *int*[*answer*], for example. For details, see the description of *IO* in IODoc.tioga.

(1.16) Here, at the end of the module, is the code that is executed when the module is started. You can create an instance of a module and start it using the "Run" command in *Commander*. The loader creates instances of programs when loading configurations. If a component of a configuration is not STARTed explicitly, then it will be started automatically (as the result of a trap) the first time one of its procedures is called. See the Mesa Language Reference Manual for more information.

In this case, the start code for the module consists of one call on the procedure *Commander.Register*, which registers the command ‘‘ReverseName’’ with the Commander, so that you can invoke the procedures *ReverseName* by typing its command name.

These procedure calls also illustrate the ability to specify the association between a procedure’s formal parameters and the arguments using keyword notation. Generally, keyword notation is preferred over positional for all but simple one- or two-argument calls, and it is definitely better for two-argument calls if the types of the arguments are the same. Either

```
Copy[to: arg1, from: arg2]
```

or

```
Copy[from: arg2, to: arg1]
```

is preferable to

```
Copy[arg1, arg2]
```

4.7 Creating programs using Tioga

This section shows you the rudiments of using Tioga to create Cedar programs, such as the ones you’ve seen in the previous section. In fact, we’ll reconstruct the framework of the ReverseName.mesa Cedar module.

A Cedar module is usually one of three kinds: A definitions module, which contains type definitions and assorted other things needed by the compiler to type-check uses of the module; an implementation module, which typically implements some interface, and is said, in CSL parlance to ‘‘export’’ that interface; and configuration modules, which bind together object files into one, loadable file. Not all implementation modules need implement an interface; in fact, ReverseName.mesa stands alone.

Our module consists of three parts:

1. Comment consisting of the module name, and the person who last changed the module.
2. DIRECTORY, which says what other modules are used by this one, and in particular, states explicitly *what* symbols or procedures are used.
3. Module name together with IMPORTS and EXPORTS clauses.

To construct a similar format, let's go through the following steps:

1. Type the following to your Commander

   % new ReverseName.mesa
   /tilde/boki/InternCzar/SimpleExample/ReverseName.mesa already exists! Use the -d switch to create a new viewer anyway.
   Oops! That name already exists in the current directory, so let's try another name, say HelloWorld.mesa
   % new HelloWorld.mesa
   Created Viewer: /tilde/boki/InternCzar/SimpleExample/HelloWorld.mesa

2. A new, empty viewer should appear on your screen. Move the mouse cursor into that window. Type the name ''HelloWorld.mesa''. MIDDLE-click twice on that name to select the name, and type CTRL-\ to turn it into a *comment* node, in italics. Type CTRL-RETURN twice (to create new nodes) and CTRL-SHIFT-\ to undo the comment mode.

3. Type ''dir'' (without) italics in the viewer.

4. Type CTRL-e. ''dir'' will be magically replaced with the following:

   DIRECTORY
   ▶Interface◀ USING [▶Item◀],
   ▶Interface◀ USING [▶Item◀];

This lists the interfaces that this program uses. CTRL-e is the abbreviation expander command to help making coding and editing Cedar programs much easier. See /Cedar/EssentialStyles/cedar.abbreviations for a complete list.

5. Reposition the mouse cursor to the beginning, and hit the NEXT key. Type ''Commander'', hit the NEXT key again, and type ''CommandProc, Register''. Hit NEXT again and type ''IO''; hit NEXT, and type ''PutRope''. Delete the semi-colon, replace it with a comma, and type CTRL-RETURN. Now type ''Rope USING [ROPE];'' don't worry that it's not in small capitals; the prettyprinter will take care of that.

6. Type CTRL-RETURN twice to create new nodes, followed by CTRL-SHIFT-n to unnest the node. Now we're at the top level.

7. Type ''prog'' (without quotes) to the viewer, and then type CTRL-e. It replaces the word prog with

```
▶Name◀: CEDAR PROGRAM
    IMPORTS ▶ImportsList◀
    EXPORTS ▶ExportsList◀
~ BEGIN

    ▶Body◀

END.
```

8. Position the mouse just in front of ▶▶Name◀ and type the NEXT key (remember key R15). Now type ''HelloWorld'' without the quotes.

9. Type the NEXT key again, and type ''Commander, IO''.

10. Delete the EXPORTS clause since we're not exporting an interface. Hold down the CTRL key, position the mouse cursor on the line, and fast double MIDDLE-click to delete the line. Release the CTRL key.

11. Hit NEXT and position the cursor at the placeholder labelled ''Body''. Type ''proc''

and type CTRL-e. This expands to a procedure skeleton.

12. Type ‘‘PrintHelloWorld’’. This is the name of the procedure. Note that procedure names must start with a capital letter.
13. MIDDLE-click on the word ‘‘PROC’’. Then RIGHT-click, which turns the selected stuff to reverse video; drag the mouse until you reach the RETURNS clause’s right square bracket:]. This mode is called pending delete. Now type ‘‘Commander.CommandProc’’. This references the interface contained in the Commander module, in particular, CommandProc.
14. Hit NEXT, which takes you to the body of the procedure. Type the following two lines:

```
ROPE: TYPE = Rope.ROPE;
IO.PutRope[cmd.out, "Hello world!\n"];
```

15. A curly bracket ‘‘}’’ delimits a procedure definition. Position your cursor after the statement separator semi-colon. Type CTRL-RETURN twice, followed by CTRL-SHIFT-n to unnest the node to the same level as the procedure definition.
16. Finally, type the line

```
Commander.Register[key: "PrintHelloWorld", proc: PrintHelloWorld, doc: "Prints
          Hello world"];
```

17. Type CTRL-d. This selects the contents of the viewer. Now type CTRL-m, which invokes the prettyprinter to format Cedar code.
18. Click on the <u>Save</u> button. Notice that your name will appear (as a nested node) just below the commented file name. You are marked as the last person to modify this file.

You have now constructed a legitimate program in Cedar. You may not understand all the details of the syntax and semantics of the language constructs, but at least you’ve used Tioga to construct the program. Try running MakeDo to compile the program; you might even try running it. Good luck!

The Cedar programming community follows established programming conventions. They cannot be hard and fast rules, and there can be good and compelling reasons for not following some convention in a particular instance. Nevertheless, to the extent that we all *usually* follow them, it will help us in reading (and therefore in modifying) one anothers’ programs. See *Stylizing Cedar Programs* <77>. Here are some of the rules.

a. Capitalize the first letter of a name.
b. Capitalize the first letter of each embedded word.
c. Case shift should be used to distinguish identifiers.
d. All module source file names have the extension ‘‘.mesa’’
e. A DEFINITIONS source file does not need any standard suffix on its name.
f. A PROGRAM or MONITOR moduel name should have the suffix ‘‘Impl’’.
g. A CONFIGURATION file name should have the extension ‘‘.config’’

Prettyprinting Cedar programs. This depends on proper capitalization of keywords (all capitals), procedure names (begin with capital), and variables (lower case). The following operations help you to achieve this:

CTRL-SHIFT-c	all capitals
CTRL-c	all lower case
CTRL-c-c	capitalize just the first character

5. Additional Documentation

5.1 Cedar Documentation

To view these documents, type Openr *name* to your commander.

CedarProgramStyle.tioga: Discusses the conventions that have been generally agreed upon by the Cedar programming community and are approximations to current practice. Consequently, they cannot be hard and fast rules, and there can be good and compelling reasons for not following some convention in a particular instance. Nevertheless, to the extent that we all *usually* follow them, it will help us in reading (and therefore in modifying) one anothers' programs.

MimosaDoc.tioga: Mimosa is a reworked version of the ancient Cedar/Mesa compiler. The intention is to eventually arrive at a compiler base that can have multiple targets. The current target is restricted to 32-bit words, byte addressing, big-endian bit and byte order (bit/byte 0 is most significant within a word), and code generation for the C language (specifically for Sun-3 and Sun-4).

ExperiencesCreatingAPortableCedar.tioga: It discusses how Cedar was re-implemented to make it portable across many different architectures, in particular, the strategy for the compiler and runtime, and the manner of making connections to other languages and the UNIX operating system.

PFSDoc.tioga: Preliminary documentation. Mostly details the latest news. It describes the standard facilities available for file access from Cedar. These facilities are available through the interfaces PFS and PFSNames. PFS and PFSNames contain facilities of interest to all users.

CirioDoc.tioga: Cirio is a debugger intended to support the debugging of systems written in multiple programming languages and running on multiple target machines. Currently Cirio understands the Cedar and C programming languages and understands about programs run on D-machines or in PCR-based systems. The current version is still very much a prototype, with much of its functionality missing.

RawViewersDoc.tioga: The RawViewersImpl module starts up Viewers using just the low level Sun drivers for the keyboard, mouse, and frame buffers. Useful if you do not wish to use X.

X11ViewersDoc.tioga: Analogous to RawViewersDoc for X11Viewers.

UserProfileDoc.tioga: Tells you how to customize system behavior using the User Profile.

References

Reference numbers in [square brackets] are for conventional hardcopy documents. Many of them are Xerox reports published in blue and white covers; the CSL blue-and-whites are available on bookshelves in the CSL Alcove. Reference numbers in <angle brackets> are for on-line documents. If they are included on the CDROM you can open them with "openr filename".

<*n*>: The generic form for a reference to an on-line document.

[*n*]: The generic form for a reference to a hardcopy document.

[1]: **Sunset New Western Garden Book.** Lane Publishing Company, Menlo Park, CA, 1979. The definitive document on Western gardening for non-botanists; 1200 plant identification drawings; comprehensive Western plant encyclopedia; zoned for all Western climates; plant selection guide in color.

[2]: Warnock, John E. **The Display of Characters Using Gray Level Sample Arrays**. blue-and-white report CSL-80-6.

[3]: Lyon, Richard F. **The Optical Mouse, and an Architectural Methodology for Smart Digital Sensors**. blue-and-white report VLSI-81-1.

[4]: **The Ethernet Local Network: Three Reports**. blue-and-white report CSL-80-2.

[5]: Shoch, John F., Yogen K. Dalal, Ronald C. Crane, and David D. Redell. **Evolution of the Ethernet Local Computer Network**. blue-and-white report OPD-T8102.

<6>: *no reference*.

[7]: Boggs, David R., John F. Shoch, Edward A. Taft, and Robert M. Metcalfe. **Pup: An Internetwork Architecture**. blue-and-white report CSL-79-10.

[8]: **Internet Transport Protocols**. Xerox System Integration Standard report XSIS 028112, December 1981.

[9]: **Courier: The Remote Procedure Call Protocol**. Xerox System Integration Standard report XSIS 038112, December 1981.

[10]: Thacker, Charles P., Edward M. McCreight, Butler. W. Lampson, Robert F. Sproull, and David R. Boggs. **Alto: A personal computer**. blue-and-white report CSL-79-11.

<11>: [Indigo]<AltoDocs>AltoHardware.press. Everything that you need to know to write your own Alto microcode.

[12]: **The Dorado: A High-Performance Personal Computer; Three Papers.** blue-and-white report CSL-81-1.

~~<13>: [Indigo]<DoradoDocs>DoradoBooting.press. Describes how to boot a Dorado, and how to configure it to boot in various ways.~~

[14]: Myer, T. H. and Barnaby, J. R. **TENEX Executive Language Manual for Users.** Available from Arpa Network Information Center as NIC 16874, but in the relatively unlikely event that you need one, borrow one from a Tenex wizard.

<15>: ~~[Indigo]<AltoDocs>BCPL.press~~ The reference manual for the BCPL programming language.

<16>: ~~[Indigo]<AltoDocs>OS.press~~. The programmer's reference manual for the Alto Operating System, including detailed information on the services provided and the interface requirements.

<17>: ~~[Indigo]<AltoDocs>Packages.press~~. A catalogue giving documentation for the various BCPL packages that other hacker's have made available.

[18]: **Mesa Language Manual, Version 11.0**, March 1984. Published by the Systems

Development Department of **OSD**.

[19]: James G. Mitchell, William Maybury, and Richard Sweet. **Mesa Language Manual, Version 5.0.** blue-and-white report CSL-79-3. A cross between a tutorial and a reference manual, though much closer to the latter than the former.

[20]: Morris, J. H. **The Elements of Mesa Style.** Xerox PARC Internal Report, June 1976. Somewhat out of date (since Mesa has changed under it), but a readable introduction to some useful program structuring techniques in Mesa.

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

[22]: **Interlisp Reference Manual**. Published in a blue and white cover, although not officially a blue-and-white. October, 1983.

[23]: The Interlisp-D Group. **Papers on Interlisp-D**. blue-and-white report CIS-5 (also given the number SSL-80-4), Revised version, July 1981.

[24]: Deutsch, L. Peter, and Edward A. Taft, editors. **Requirements for an Experimental Programming Environment**. blue-and-white report CSL-80-10.

<25>: ~~[CedarChest7.0]<Top>Documentation.df~~. Hardcopies are entitled **The Cedar Manual**.

[26]: **Alto User's Handbook**. Internal report, published in a black cover. The version of September, 1979 is identical to the version of November, 1978 except for the date on the cover and title page. Includes sections on Bravo, Laurel, FTP, Draw, Markup, and Neptune

<27>: ~~[Indigo]<AltoDocs>SubSystems.press~~. Documentation on individual Alto subsystems, collected in a single file. Individual systems are documented on [Indigo]<AltoDocs>systemname.TTY, and these files are sometimes more recent than SubSystems.press.

[28]: Jerome, Suzan. **Bravo Course Outline.** Internal report, published in a red cover. Oriented to non-programmers.

<29>: **The Tioga editor**. [Cedar10.1]<Tioga>TiogaDoc.tioga.

~~<30>: [Indigo]<PrintingDocs>PressFormat.press. Describes the Press print file format.~~

~~<31>: [Indigo]<PrintingDocs>PressOps.press. Describes the Press printing program.~~

~~<32>: [Indigo]<PrintingDocs>PDPrintOps.press. Describes the PDPrint printing program.~~

[33]: Birrell, Andrew D., Roy Levin, Roger M. Needham, and Michael D. Schroeder. **Grapevine: Two Papers and a Report.** blue-and-white report CSL-83-12.

~~<34>: [Ivy]<Laurel>Maintain.press. Documentation for the teletype version of Maintain, the version that is used from within Laurel or Tajo.~~

[35]: Brotz, Douglas K. **The Laurel Manual.** blue-and-white report CSL-81-6.

[36]: Schimidt, Eric Emerson. **Controlling Large Software Development in a Distributed Environment.** blue-and-white report CSL-82-7.

<37>: [Cedar10.1]<DF>DFDoc.tioga. The reference manual for the use of DF files.

~~<38>: [CedarChest7.0]<Documentation>CSL-NotebookDoc.tioga~~

~~*also stored as* [Indigo]<CSL-Notebook>Docs>CSL-NotebookDoc.tioga~~

~~<39>: [CedarChest7.0]<Documentation>HowToUseAPublicCedarMachine.tioga.~~

[40]: Teitelman, Warren. **The Cedar Programming Environment: A Midterm Report and Examination.** blue-and-white report CSL-83-11.

[41]: Brown, Mark R., Karen Kolling, and Edward A. Taft. **The Alpine File System.** blue-and-white report CSL-84-4.

[<42>]: Swinehart, Daniel, Polle Zellweger, Rick Beach, Robert Hagmann. **A Structural View of the Cedar Programming Environment.** blue-and-white report CSL-86-1.

available online as [CedarChest7.0]<Documentation>StructureOfCedar.tioga.

<43>: Hagmann, Robert B. **A Labelless File System For Cedar.**

~~[Indigo]<CSL-Notebook>Docs>86CSLN-0047.press~~

<44>: Arnon, Dennis, and Carl Waldspurger. **CaminoReal.**

[CedarChest7.0]<Documentation>CaminoRealDoc.tioga.

[45]: Bier, Eric and Ken Pier. **The Gargoyle Reference Manual.**

[CedarChest7.0]<Documentation>GargoyleDoc.tioga.

<46>: Bier, Eric and Ken Pier. **Gargoyle Tutorial.**

[CedarChest7.0]<Documentation>GargoyleTutorialDoc.tioga.

[47]: Ades, Stephen, and Daniel Swinehart. **Voice Annotation and Editing in a Workstation Environment.** blue-and-white report CSL-86-3.

[48]: Donahue, James, and Willie-Sue Orr. **Walnut: Storing Electronic Mail in a Database.** blue-and-white report CSL-85-9.

<49>: Kent, Jack, and Doug Terry. **Wallaby.**

[CedarChest7.0]<Documentation>WallabyDoc.tioga.

<50>: Frailong, Jean-Marc. **XTSetter.**

[Cedar10.1]<XTSetter>XTSetterDoc.tioga.

[51]: Bhushan, Abbay, and Michael Plass. **The Interpress Page and Document Description Language.** IEEE Computer, Vol 19, Number 6, June 1986, pp. 72-77.

[<52>]: Atkinson, Russ, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser. **Experiences Creating a Portable Cedar**. Proceedings of the SIGPLAN '89 conference on Programming Language Design and Implmentation, held in Portland, Oregon, June 21-23, 1989, pp. 322-329.

available online as [PCedar2.0]<Documentation>ExperiencesCreatingAPortableCedar.tioga

<53>: Hauser, Carl. **PFS.** [Cedar10.1]<PFS>PFSDoc.tioga

<54>: Sturgis, Howard, and Marvin Theimer. **The Cirio Debugger: a multi-language, multi-target-world debugger.** [PCedar2.0]<Documentation>CirioDoc.tioga

<55>: Pier, Ken. **Life On SPARCstation**. Available on-line as [Cedar10.1]<CedarDoc>LifeOnSPARCstation.tioga

<56>: Pelegri-Llopart, Eduardo. **Life in PCedar**. Available on-line as [PCedar2.0]<OldDocumentation>LifeInPCedar.tioga

<57>: **Cedar Language Overview**. [Cedar10.1]<CedarDoc>LanguageOverviewDoc.tioga

<58>: **Cedar/Mesa Language Changes**. [Cedar10.1]<MimosaOnly>LanguageChangesSummary.tioga

<59>: Johnson, Sharon, and John Larson. **Hitchhikers Guide to External Mail: Using the PARC ARPA Internet Mail Gateway**. [CedarChest7.0]<Documentation>ExternalMail.tioga

<60>: **Peanut**. [PCedar2.0]<Documentation>PeanutDoc.tioga

<61>: **BlackCherry**. [PCedar2.0]<Documentation>BlackCherryDoc.tioga

~~<62>: Tammer, Laine. **Doggy Paddling in PCedar**. /Indigo/CSL-Notebook/Entries/90CSLN-0003/IntroPCedarUse.tioga~~

<63>: Tammer, Laine. **Tioga Editing in PCedar**. /Indigo/CSL-Notebook/Entries/90CSLN-0003/TiogaEditingInPCedar.tioga

<64>: Tammer, Laine. **Using the Commander**. /Indigo/CSL-Notebook/Entries/90CSLN-0003/UsingTheCommander.tioga

<65>: **Introduction to Cedar**. /CedarChest7.0/Documentation/Introduction.tioga

<66>: Jacobi, Christian**.** **The X Window System Version 11**. /Cedar10.1/X11/X11Doc.tioga and /Cedar10.1/X11Viewers/X11ViewersDoc.tioga

<67>: **Cedar Document Style**. /CedarChest7.0/Documentation/SampleSheet.tioga

<68>: Plass, Michael, et al. **RawViewers: PCedar Viewers without X**. /Cedar10.1/RawViewers/RawViewersDoc.tioga

<69>: Atkinson, Russ, et al. **PCRDOC: Portable Common Runtime**.

<70>: Plass, Michael. **The Cedar Commander**. /Cedar10.1/Commander/CommanderDoc.tioga

[71]: Weiser, Mark, Alan Demers, and Carl Hauser. **The Portable Common Runtime Approach to Interoperability**. Proceedings of the 12th ACM Symposium on Operating Systems Principles, pp. 114-122. Special issue of SIGOPS Operating Systems Review, volume 23, number 5.

~~<72>: Pelegri-Llopart, Eduardo, and Ken Pier. **DF Suites in PCedar**. /PCedar2.0/Documentation/DFSuitesDoc.tioga~~

<73>: **A Glossary of Terms, Subsystems, Directories, and Files**. /Cedar7.0/Documentation/Glossary.tioga

<74>: Irish, Wesley. **Maintain**. /Cedar10.1/Maintain/MaintainDoc.tioga

<75>: Spreitzer, Mike. **MakeDo**. /Cedar10.1/MakeDo/MakeDoDoc.tioga

<76>: Nichols, David. **Cedar Essentials**. /PCedar2.0/Documentation/CedarEssentials.tioga

<77>: **Stylizing Cedar Programs**. /Cedar10.1/CedarDoc/CedarProgramStyle.tioga