N87-28306

p.21

MANUAL PAGES FOR SAGA SOFTWARE TOOLS

Carol S. Beckman

George Beshers

David Hammerslag

Peter A.Kirslis

Hal Render

Robert Terwilliger

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, IL

July, 1985

NAME

dfbase—set the base version for finding differences between SAGA parse trees

SYNOPSIS

dfbase <saga directory>

DESCRIPTION

Dfbase sets the base version for dfdiff to use. The saga directory contains the files created by epos. The modified fields in the current files are cleared and the parse tree is copied to the base version. The parse tree may not become the base version if it contains errors or parse suspension points.

DIAGNOSTICS

Error messages are (hopefully) self-explanatory.

FILES

In the saga directory for which dfbase is invoked: dfbaseparse parse tree for base version dfbasestr string file for base version dfdebug debugging output saga1p parse tree for the version being edited saga1s string file for the version being edited

SEE ALSO

dfdiff, dfundo

IDENTIFICATION

Carol Beckman

BUGS

Dfbase will change in the near future with little notice.

NAME

dfdiff—display differences between SAGA parse trees

SYNOPSIS

dfdiff <saga directory> [<root or range>] [<context>] [<version>]

DESCRIPTION

Find the differences between the current version of the parse tree and an older, base, version.

The <root or range> argument tells which differences to print. If the argument is an integer, it is taken as the root (nodeindex) of a subtree. If the argument is two integers separated by a colon, it is taken as the beginning and ending locations (nodeindices) of the range in which to find differences. Only differences in the selected part of the parse tree are printed. If no argument is given, all the differences in the tree are printed.

The <context> argument tells how many lines of context to print around each difference. <context> is an integer. A partial line adjacent to a difference counts as one line. If no argument is given, one is used.

The <version> argument is used to select the version of the difference command. <version> is an integer. Currently only one version is available. This version is used if no <version> argument is given.

Dfdiff operates in screen mode or line mode. In line mode the differences will all print with no further input from the user.

In screen mode, the differences are displayed one at a time. If a difference cannot fit on one screen, the old and new parts of the difference each get half the space. The text can be scrolled so that all the difference can be viewed. Control-L scrolls the parts forward, while control-H scrolls back. The old and new parts can be scrolled individually by prefixing the command with control-O or control-N for the old and new parts, respectively. So control-O control-L scrolls just the old part forward. Control-N control-H scrolls just the new part back. Moving from one difference to the next is accomplished with control-J and control-K. Control-J moves to the next difference. Control-K moves back one difference. The default action is to move to the next difference. So if any other key is hit, the next difference is displayed.

DIAGNOSTICS

Error messages are (hopefully) self-explanatory.

FILES

In the saga directory for which dfdiff is invoked: dfbaseparse parse tree for base version dfbasestr string file for base version dfdiffinfo information for the differences found dfdebug debugging output sagalp parse tree for the version being edited sagals string file for the version being edited

SEE ALSO

dfbase, dfundo

IDENTIFICATION

Carol Beckman

BUGS

When called as a filter command from the SAGA editor, the first screen display is not always correct. This affects further screen displays since only the new text is plotted and the replotter assumes the first screen was properly displayed. This might be fixed now.

The field in the parse tree which is supposed to indicate whether a change has been made since the last time dfdiff was executed does not get set by all changes. Thus dfdiff may not display the new changes since it will reuse the old information, on the false assumption that it is current.

If dfundo is used to undo differences, but these differences are not actually undone, dfdiff will not display the undone differences unless the parse tree is modified.

NAME

dfundo—generate commands for undoing differences between SAGA parse trees

SYNOPSIS

dfundo <saga directory> <diff#> ... <diff#>

DESCRIPTION

Dfundo generates the commands needed to undo a difference. Dfdiff must have been executed after any changes to the parse tree and before dfundo is invoked. The <diff#>s are the numbers given by dfdiff of the differences which are to be undone. One or more <diff#>s may be given for one invocation of dfundo.

DIAGNOSTICS

Error messages are (hopefully) self-explanatory.

FILES

In the saga directory for which dfundo is invoked: dfbaseparse parse tree for base version dfbasestr string file for base version dfdiffinfo information for the differences found dfdebug debugging output sagalp parse tree for the version being edited sagals string file for the version being edited #dfundoexec file of commands to execute to undo differences

SEE ALSO

dfdiff, dfbase

IDENTIFICATION

Carol Beckman

BUGS

The commands generated by dfundo cannot be executed by epos with an exec command. It seems that epos interprets the text for insertions as commands. The range syntax needed for the deletions is not implemented.

Dfundo will report that a difference has been undone already even if the file of commands has not be executed unless some change is made to the parse tree and dfdiff is executed again.

NAME

epos — language–oriented editor based on an LR(1) parser.

SYNOPSIS

epos [–l] [–P<parse–tables>] [–cdiimprstvx] <parse–tree> [<parse–tree>]

DESCRIPTION

Epos is an editor for languages based on formal BNF style grammars and LR(1) parsers. An editor can be produced for any language for which such a description exists. The editor provides both text–oriented commands and additional structure–oriented commands, which are based on the structure of the parse tree produced by the editor.

The editor incorporates an LR(1) style parser to perform syntactic and optional semantic analysis of the program being edited. Each time the user completes an insertion or modification, the parse tree is incrementally updated with the new information. The user of the editor is provided with additional analysis during the editing process, and presented with immediate feedback about the correctness of the input.

The amount of semantic analysis performed (and whether any at all occurs) is dependent both on the parser–generating system used to produce the editor, and the type of semantic analysis defined in the input grammar file.

The editor is screen–oriented, using the *termcap* facility to adapt itself for a particular terminal; a line mode is also provided. The SAGA editor user manual provides a description of editor commands. Information about the run–time environment of the editor, and its command line options and arguments is presented here.

The command line options are:

–l Invoke the editor in line mode instead of screen mode.

–P Specifies an alternate file (–P<parse–tables>) from which to load the parse tables to be used.

Since the editor is still an experimental prototype, a number of the available debugging options are listed below to aid the individuals managing the implementation. These options can be activated either by command line flags or the *on* and *off* commands of the editor. Users might find them useful in formulating bug reports. The command line options for debugging are:

–b Turn on paging system debugging. Same as the "on db" editor command. If specified twice, also enables detailed debugging.

–c Turn on command interpreter debugging. Same as "on dc".

–i Turn on input and editor initialization debugging. Same as "on di". If specified twice, also enables detailed debugging.

–m Turn on make (incremantal recompilation) system debugging. Same as "on dm".

–p Turn on parser debugging. Same as "on dp".

–r Turn on parser initialization and recovery debugging. Same as "on dr".

–s Turn on debugging of the semantic analysis phase of the parse. Same as "on ds".

–t Turn on debugging of the parse tables (used in the editor's language dependent module only). Same as "on dt".

–x Turn on debugging of the lexical analysis phase of the parse. Same as "on dl".

FILES

saga/bin/epos:
cshell script to invoke the editor,
saga/obj/editor/<language>.mystro/epos:

the actual editor process,

saga/obj/editor/<language>.mystro/parse.tables:
: the binary parse tables,

saga/obj/editor/<language>.mystro/help.index:
: index to on-line help file,

saga/obj/editor/<language>.mystro/epos.help:
: on-line help file,

saga/src/editor/lib/epos.cmds:
: user-defined commands for all editors,

saga/src/editor/lib/epos.<language>cmds:
: user-defined commands for this language,

<current-directory>/.epos.<language>cmds:
: the user's private user-defined commands for this language.

SEE ALSO

scat(1), dfbase(1), dfdiff(1), dfundo(1), rulecount(1).

AUTHOR

Peter A. Kirslis, Dept. Computer Science, Univ. Illinois — Urbana, 1304 W. Springfield Ave., Urbana, Illinois, 61801. Written 1982, revised and extended 1983, 1984, 1985.

BUGS

The editor is still an experimental prototype. Some bugs still exist in the parser, although most problems will be found in the screen-mode command interpreter. If a parse tree file is garbled by the editor, its text representation can usually be recovered with the *scat(1)* command.

The second parse tree argument to the editor specifies an alternate parse tree to be accessed read-only. Use of the alternate file is restricted to line mode, since the screen mode interpreter does not yet provide any support for accessing it.

Multi-line comments are not yet supported in the editor. The lexical analyzer does recognize them and store them properly, but the command interpreters and screen display do not yet handle them properly.

NAME

Make – maintain program groups

SYNOPSIS

Make [–f makefile] [option] ... file ...

DESCRIPTION

Make executes commands in *makefile* to update one or more target *names. Name* is typically a program. If no –f option is present, 'makefile' and 'Makefile' are tried in order. If *makefile* is '–', the standard input is taken. More than one –f option may appear

Make updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated list of targets, then a colon, then a list of prerequisite files. Text following a semicolon, and all following lines that begin with a tab, are shell commands to be executed to update the target. If a name appears on the left of more than one **:** then it depends on all of the names on the right of the colon on those lines, but only one command sequence may be specified for it. If a name appears on the left of a colon exclamation mark **:!** then it depends on exactly one of the files on the right of the colon exclamation mark. The file choosen is the first one (left to right) that exists, or the last one if none of them exists. If a name appears on the left of a colon question mark **:?** then it depends on all the files on the right of the colon question mark *if* they exist. If a name appears on the left of a colon exclamation question mark **:!?** then it depends on no more than one of the files on the right, if no file on the right exists, then it behaves like a **:?** . If a name appears on a line with a double colon **::** then the command sequence following that line is performed only if the name is out of date with respect to the names to the right of the double colon, and is not affected by other double colon lines on which that name may appear.

Three special forms of a name are recognized. A name like *a*(*b*) means the file named *b* stored in the archive named *a*. A name like *a*((*b*)) means the file stored in archive *a* containing the entry point *b*. Also a name like *a,v*(*b*) refers to the RCS file of *a* with revision *b*. The revision may contain symbolic names as defined in RCS. If the revision refers to a branch then the last member of that branch is the revision chosen. Note: Using the modified *ci* command with –l or –u options the modification dates of a revision and the working file are equal, i.e., neither one is considered to be out of date with the other.

Sharp and newline surround comments.

The following makefile says that 'pgm' depends on two files 'a.o' and 'b.o', and that they in turn depend on '.c' files and a common file 'incl'.

```
pgm: a.o b.o
        cc a.o b.o –lm –o pgm
a.o: incl a.c
        cc –c a.c
b.o: incl b.c
        cc –c b.c
```

Makefile entries of the form

```
string1 = string2
```

are macro definitions. Subsequent appearances of $(*string1*) or ${*string1*} are replaced by *string2*. If *string1* is a single character, the parentheses or braces are optional.

The value of a macro may be edited before being replaced in the input stream. The syntax is ${string1:modifier} where *modifier* specifies the edit to be made. If an edit fails a default value is returned and a warning is sent to stderr. The modifiers are:

–a Which returns the archive file. Thus dir1/archive(member) becomes dir1/archive. If no (exists then the argument is returned.

–e Which returns the extension if one exists or .junk otherwise. Thus ../dir1/root.e1.e2 becomes .e2.

–h Which returns the head of the path name if a / exists in the argument, otherwise it returns a '.' (current directory). Special case, if the path is the root name / then that is returned. Thus dir1/dir2/name becomes dir1/dir2.

–m Which returns the member of an archive if a (exists, otherwise it returns its argument.

–R –R/.E/ The first case returns the "local" root of the path name, i.e., all the directories and the extension are discarded. The second case appends the new extension to the former result. Thus dir1/dir2/name.e becomes name.

–r –r/.E/ This version retains the directories. In the example dir1/dir2/name is returned.

–t Which returns the tail of the path name if a / exists or its argument otherwise.

–s Which implements the Unix ed command s/pattern/replace/. If the pattern match fails the argument is returned.

All of the modifiers work on lists of names by processing each name individually, i.e., the strings are broken into lists of names based on space delimiters and each name is modified separately.

For each rule four special variables are set, $@, $*, $<, and $?. The special macro $@ stands for the full target name, $* stands for the target name with the suffix deleted. Both of these variables may be used in the prerequisites list and the commands in conjunction with the editing operations explained above. The macro $< lists the prerequisites that exist on the line with the commands, and $? lists all the prerequisites that are out of date. The special variables can be used with the modifiers discussed above.

Shell meta characters can occur in both target and prerequisite file names. When used in target file names the pattern is used to find the rules associated with an actual target name. When a match occurs the $@ and $* variables are set to the actual target name, and the prerequisites are processed. If a prerequisite contains a meta character the corresponding directory is searched and any file which matches becomes an actual prerequisite. The standard glob(1) patterns have been extended with the ** pattern which is like * but capable of matching a sequence of directories when used in the target name.

Make can infer prerequisites for files for which the *Makefile* gives no explicit commands. For example, a '.c' file may be inferred as prerequisite for a '.o' file and be compiled to produce the '.o' file. Thus the preceding example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o –lm –o pgm
a.o b.o: incl
```

Prerequisites are inferred from a list of optional rules. Optional rules are distinguished by a **:?** between the targets and dependent files. The optional rules only apply if the dependent file(s) exists, and only one optional rule applies for a particular target. Thus order is significant; the commands associated the first target pattern that matches target name and for which there exists a dependent file are the commands used. For example, the rule for making optimized '.o' files from '.c' files is

```
*.o :? $*.c
cc –c –O –o $@ $*.c
```

Notice the use of a shell meta character in the target file name, and the special macro $* to specify the exact prerequisite desired.

Certain macros are used by the default inference rules to communicate optional arguments to any resulting compilations. In particular, 'CFLAGS' is used for *cc*(1) options, 'FFLAGS' for *f77*(1) options, 'PFLAGS' for *pc*(1) options, and 'LFLAGS' and 'YFLAGS' for *lex* and *yacc*(1) options. In addition, the macro 'MFLAGS' is filled in with the initial command line options supplied to *make*. This simplifies maintaining a hierarchy of makefiles as one may then invoke *make* on makefiles in subdirectories and pass along useful options such as –k.

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the special target '.SILENT' is in *makefile,* or the first character of the command is '@'.

Commands returning nonzero status (see *intro*(1)) cause *make* to terminate unless the special target '.IGNORE' is in *makefile* or the command begins with <tab><hyphen>.

Interrupt and quit cause the target to be deleted unless the target is a directory or depends on the special name '.PRECIOUS'. All files ending in ,v or having the form ,v() are assumed to be precious.

Other options:

–i	Equivalent to the special entry '.IGNORE:'.
–k	When a command returns nonzero status, abandon work on the current entry, but continue on branches that do not depend on the current entry.
–n	Trace and print, but do not execute the commands needed to update the targets.
–t	Touch, i.e. update the modified date of targets, without executing any commands.
–r	The predefined macros and default rules are not processed which saves processing time, and protects the user from hidden intertactions. The special entry '.NORULES:' is equivalent.
–s	Equivalent to the special entry '.SILENT:'.
–q	Test the prerequisites of a (single) target, and return a 0 status if the target is up to date and –1 status if it needs to be remade.
–Q	For recursive calls to make asking for the special status reports of –q. Notice that a positive status indicates an error in the child make.

The most common use of make is in maintaining large programs. In the following example all the *.p* files are stored in the directory *../src* and all the *.h* are stored in the directory *../hdr* and the objects are going to be placed in this directory.

```
SrcDir   = ../src
Srcs     = program.p module1.p module2. module3.p
Objs     = ${Srcs:r,.o,}
program : ${Objs}
         ${PC} ${PFLAGS} ${Objs} -o program
${Objs} : ${SrcDir}/$*.p
         ${PC} ${PFLAGS} -c $<
${Objs} : ../hdr/*.h
```

Notice that the object names were generated with the modifier *r*. The second rule should be considered a *foreach* object file generate the specified prerequisite and Pascal compile. The third rule specifies that all the objects are dependent on all the headers.

We present two examples of using make to maintain RCS files. (Macros as defined above).

```
Rev      = working
```

```
        RcsFiles = ${Srcs:s,.*,RCS/&,v(${Rev}),}
        All      : ${RcsFiles}
        ${RcsFiles} : $*.p
                ci -u${Rev} $<
```

After you are done editing the working files this make script automatically discovers which files were actually touched, and checks them in. Note the use of a symbolic revision name.

```
        program : ${Objs}
                ${PC} ${PFLAGS} ${Objs} -o program
        ${Objs} :? $*.p
                ${PC} ${PFLAGS} -c $<
        ${Objs} :? ${SrcDir}/$*.p
                ${PC} ${PFLAGS} -c $<
        **.p : ${SrcDir}/RCS/${@:t},v(working)
                ${CO} -r${Rev} $@ $<
```

This example searches two directories for the Pascal sources, first the current directory, and then the SrcDir. However both sets of sources are dependent on the same RCS files.

An example of archive maintainance is

```
        SRCDIR=         ../src
        INCLUDE = /usr/include
        SRCS=open.c  close.c  creat.c
        archive.a:      ${SRCS:s,^).c$,system.o(1.o),}
                        ar rv archive.a ${?:m}
                        rm ${?:m}
                        ranlib archive.a
        archive.a: ${INCLUDE}/system.h
        archive.a(*.o):? ${@:m}
                        echo Using ${@:m}
        *.o:      ${*:s,.*,${SRCDIR}/&.c,}
                        ${CC} ${CFLAGS} $<
        archive.a(*.o):?        ${${@:m}:s,).o,${SRCDIR}/1.c,}
                        ${CC} ${CFLAGS} $<
        Maketd:
                        Maketd -mMakefile -Asystem.o -s${SRCDIR} ${SRCS}
```

Notice that the **ar** command is executed once with all the **.o** files which are out of date, avoiding some overhead.

The macro ${MAKE} is recognized as the current make command, and treated specially. It is called with ${MFLAGS} as arguments, and also called when the –n option is in effect. When Make is called from Make a return code is requested and examined to see if the target was remade.

FILES

makefile, Makefile

SEE ALSO

sh(1), touch(1), f77(1), pc(1), Maketd(1)

BUGS

Some commands return nonzero status inappropriately. Use –i to overcome the difficulty.
Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across newlines in *make.*

NAME

rulecount — a SAGA parse tree analyzer

SYNOPSIS

rulecount [options] countfile [sagafile ...]

DESCRIPTION

Rulecount is a program which counts the uses of production rules in a SAGA parse tree. A report is produced on the standard output giving the indices of the rules found and their corresponding multiplicity. Various options may be invoked to produce different reports. The counts are stored in the file given as the countfile on the command line, and these counts can be accumulated over several runs of the program. This allows one, for example, to run the program with a test suite for a given set of editor files and determine whether all rules have been used or, if not, which ones have not. Each *sagafile* is a directory produced by a SAGA language-oriented editor, and from 0 to 32 files may be given on the command line. If no *sagafile* is given, the *countfile* is analyzed and a summary report is produced for the values stored in it.

Rulecount first performs a traversal on the SAGA parse tree file from an input SAGA editor directory, saving the counts of the rules used in the *countfile*, either creating a new file if one does not exist, or adding the counts to the *countfile* if one does exist. The program performs a traversal on each SAGA parse tree file on the command line, accumulating the results in the countfile. On completion of all the traversals, a summary report is produced for the accumulated counts, including the counts which existed, if any did, in the *countfile* when the program was run. Various options can be used to control the analysis and the report produced:

–o*N* inform *rulecount* of the index, *N*, of the origin rule of the grammar which the particular SAGA editor used in producing the parse tree file.

–r*N* inform *rulecount* of the index, *N*, of the maximum rule of the grammar which the particular SAGA editor used in producing the parse tree file.

–r*N* include in the output report only those rules which occurred *N* or more times in the input file. This defaults to 1 if this option is not used.

–i generate a report for each SAGA file in addition to the summary report which is always produced. This allows one to see which files used which rules. A few additional statistics are included in the individual reports, such as a count of the nodes and their types as found in each SAGA file, as well as the maximum depth reached in the traversal stack. This last value may be used to gauge the depth of the parse tree.

–p print the percentage of the grammar rules used in a particular parse tree. To use this option, the –o and –r options must also be used (for obvious reasons). If the –i option is on, the percentage used by each parse tree as well as the total percentage covered by all are reported.

–z display only those rules which have not been used (have a count of zero). It is recommended that the –r and –o options be turned on when using this, so that the program knows what the upper and lower bounds of the grammar rules are. Otherwise, it only gives those rules which lie between the current minimum and maximum rules found.

–t trace the traversal of the SAGA parse trees. This is primarily a debugging option, and is recommended only as a last resort, as it produces scads of output (a single line for each node of a parse tree).

–h display the usage line and the list of available options for the program. This information is stored in the file 'help.rulecount' in the saga/src directory containing the program source.

DIAGNOSTICS

Errors in the arguments to rulecount are flagged, and conditions which violate the integrity of the report are also checked, such as the occurrence of a rule whose index is greater than that given in the -r option. Most of these errors cause the program to halt immediately. As intermediate counts are written out to the countfile after each parse tree has been traversed, the contents of the countfile may be corrupted by spurious input. Some attempts have been made to indicate where the error occurred, thoUGH these may not always be sufficient for full debugging.

FILE MODES

The user must have read/write permission on the countfile and read permission on the SAGA file(s) on the command line.

FILES

~saga/bin/rulecount -- the executable program file ~saga/src/utilities/rulecount — the source directory ~saga/lib/help.rulecount -- the help file

IDENTIFICATION

The author of this program was Hal Render, currently working for the University of Illinois. All problems and suggestions for improvement should be addressed to him. His current address is:

Hal Render
222 Digital Computer Lab
University of Illinois
1304 W. Springfield
Urbana, Illinois 61801
(217) 333-7937

BUGS

The program does not currently check to see if the input SAGA files come from the same editor or even the same language. The user must take care not to mix files from different editors or languages, if he/she wishes an accurate report on the parse tree files. This program has not been tested very rigorously, and is thus subject to error. If any problems are found, please contact Hal Render.

NAME

scat -- catenate and print the text from SAGA parse tree directories.

SYNOPSIS

scat <parse-tree-directory> [<parse-tree-directory> ...]

DESCRIPTION

Scat produces the source text representation of a SAGA parse tree on standard output. If more than one parse tree is specified, the output will contain the text from each tree, in the order that the arguments were supplied. Scat operates by traversing only the frontier of the parse tree, so it may be used to extract the text from parse trees containing discontinuities (suspension points and errors). It also can recover the text from parse trees whose internal structure has been scrambled, as long as the frontier is intact (which is usually the case when a parser bug in the editor occurs).

SEE ALSO

epos(1)

AUTHOR

Peter A. Kirslis, Dept. Computer Science, Univ. Illinois -- Urbana, 1304 W. Springfield Ave., Urbana, Illinois, 61801. Written February, 1985.

NAME
sem_create - create a semaphore

SYNOPSIS
~saga/bin/sem_create semaphore_name

DESCRIPTION
sem_create creates a semaphore to control interprocess communication. The semaphore is implemented with a file. To create a semaphore, execute sem_create and provide a name for a semaphore. The name of the semaphore should have the suffix **.sem.** sem_create creates a file named semaphore_name.

DIAGNOSTICS
sem_create will print an error message if more than one argument is given or if the argument does not end with .sem.

SEE ALSO
sem_intro(1), sem_destroy(1), sem_p(1), and sem_v(1). A C interface is described in sem_C_int(2).

IDENTIFICATION
Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

sem_destroy - destroy a semaphore

SYNOPSIS

~saga/bin/sem_destroy semaphore_name

DESCRIPTION

sem_destroy destroys a semaphore. To destroy a semaphore, execute sem_destroy with the semaphore name as the only argument. The name of the semaphore should have the suffix **.sem.**

DIAGNOSTICS

sem_destroy will print an error message if more than one argument is given or if the argument does not end with .sem.

SEE ALSO

sem_intro(1), sem_create(1), sem_p(1), and sem_v(1). A C interface is described in sem_C_int(2).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

sem_p - perform a P operation on a semaphore

SYNOPSIS

~saga/bin/sem_p semaphore_name

DESCRIPTION

sem_p performs a P operation on a semaphore. If a P operation has already been performed on the semaphore, the new P operation will block. The name of the semaphore should have the suffix **.sem.** The P operation is performed in the following manner. An flock is performed on the file that represents the semaphore (the file is created by sem_create). If a P operation has already been performed, the flock will block. The process now attempting the P will remain blocked until the process holding the flock is killed.

When the flock succeeds, a new process is forked to hold the flock. The PID of the new process is written in the semaphore file and the process goes to sleep. The corresponding V operation reads the PID from the semaphore file and kills the process holding the flock allowing the next process to perform its P operation.

DIAGNOSTICS

sem_p will print an error message if more than one argument is given or if the argument does not end with .sem.

SEE ALSO

sem_intro(1), sem_create(1), sem_destroy(1), and sem_v(1). A C interface is described in sem_C_int(2).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

ted, browse, peg – a family of prototype tree structure editors

SYNOPSIS

ted [<filename>]
browse [<filename>]
peg [<filename>]

DESCRIPTION

These are a family of closely related editors for editing unrestricted trees. Each of these editors is unique, although they share a common editor core and common editing features. Each editor consists of the (slightly tailored) editor core, and packages of external programs that operate on the tree constucted by the editor. The basic paradigm of ted editing is: the user constructs or modifies trees using the editor, then from within the editor, invokes external programs to certify that the tree maintains its desired properties. The user is encouraged to create his own external programs to suit his particular needs.

DIAGNOSTICS

Ted–based editors are chocked full of self–explanatory error messages.

FILES

.tedrc	ted initialization file (lisp commands)

SEE ALSO

Since the ted editors are prototypes, they are rapidly changing; however the most comprehensive document is "Ted: a Tree Editor with Applications for Theorem Proving", by David Hammerslag. The uiucdcs local notesfile "ted" is a good source for up–to–date (tho less comprehensive) information.

IDENTIFICATION

David Hammerslag uiucdcs!hammer

BUGS

Being prototypes these editors are problably loaded with bugs.

There is very little hard documentation on any of the editors except ted.

NAME

sem_create - create a semaphore to control access to a file

SYNOPSIS

#include "~saga/src/sem_C_int/sem_C_int.h" #include "~saga/src/msc/msc.h"

int rtrn ;

int sem_create(file_name,semaphore,argc,argv) char file_name[] ; char semaphore[] ; int argc ; char *argv ;

cc * ~saga/src/sem_C_int/sem_C_int.o ~saga/src/sem_C_int/msc.o

DESCRIPTION

sem_create creates a semaphore to control access to a file. The semaphore controls access to **file_name. semaphore** receives the name of the semaphore when sem_create is done. The name of the semaphore is **file_name** with **.sem** concatenated to the end. **sem_create** executes the system program ~saga/bin/sem_create to create the semaphore. **semaphore** is the name of the file used for the semaphore. In other words, this function executes the command "sem_create semaphore".

DIAGNOSTICS

rtrn gets the return code from the system call to execute sem_create.

SEE ALSO

sem_create(1), sem_destroy(2), sem_p(2), sem_v(2).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

C interface to semaphore routines.

SYNOPSIS

#include "~saga/src/sem_C_int/sem_C_int.h" #include "~saga/src/msc/msc.h"

int rtrn ;

int sem_destroy(semaphore,argc,argv) char semaphore[] ; int argc ; char *argv ;

cc * ~saga/src/sem_C_int/sem_C_int.o ~saga/src/sem_C_int/msc.o

DESCRIPTION

sem_destroy destroys the semaphore created by The argument **semaphore** is the name of the semaphore created when sem_create(2) was called.

DIAGNOSTICS

rtrn contains the return code from the system call.

SEE ALSO

sem_C_int(2), sem_create(2), sem_intro(1), sem_create(1), sem_destroy(1).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

sem_p - perform a P operation on a semaphore

SYNOPSIS

```
#include "~saga/src/sem_C_int/sem_C_int.h" #include "~saga/src/msc/msc.h"

int rtrn ;

int sem_p(semaphore,argc,argv) char semaphore[] ; int argc ; char *argv ;

cc * ~saga/src/sem_C_int/sem_C_int.o ~saga/src/sem_C_int/msc.o
```

DESCRIPTION

sem_p performs a P operation on semaphore. The function really executes the command "sem_p semaphore". A V operation can be performed on the semaphore by calling sem_v(2). **semaphore** is the name of the semaphore created by calling sem_create(2).

DIAGNOSTICS

rtrn contains the return code from the call to system.

SEE ALSO

sem_C_int(2), sem_v(2), sem_intro(1), sem_p(1), sem_v(1).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

sem_v - perform a V operation on a semaphore

SYNOPSIS

```
#include "~saga/src/sem_C_int/sem_C_int.h" #include "~saga/src/msc/msc.h"

int rtrn ;

int sem_v(semaphore,argc,argv) char semaphore[] ; int argc ; char *argv ;

cc * ~saga/src/sem_C_int/sem_C_int.o ~saga/src/sem_C_int/msc.o
```

DESCRIPTION

sem_v performs a V operation on semaphore. The function really executes the command "sem_v semaphore". A P operation can be performed on the semaphore by calling sem_p(2). **semaphore** is the name of the semaphore created when sem_create(2) was called.

DIAGNOSTICS

rtrn contains the return code from the call to system.

SEE ALSO

sem_C_int(2), sem_p(2), sem_intro(1), sem_p(1), sem_v(1).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

Pascal to System interface.

SYNOPSIS

#include "/mntb/3/srg/saga/include/system.h" pc * saga/lib/system/system/system.o

DESCRIPTION

The purpose of these routines is to provide a standard interface from Pascal (the pc compiler) to the Unix system. The idea is that the SYS library should be the only thing which needs to be altered to port the Pascal portion of SAGA to System 5 or Xenix (I know, fat chance). There are two essential differences between the Pascal and C versions of the system calls. First strings in Pascal are passed as "systring", and converted to the C NULL terminated format internally. Second pointers in Pascal must be typed. If the **value** of a pointer is required then the "ord()" of that pointer returns an integer which agrees with the type **address** defined in system.h. Sadly, there is not a well defined mechanism for going the other way. An undiscriminated variant record is necessary to convert pointers to integers. Further, the **size** of a record must be calculated by calling a "Delta" function with two var parameters which are successive array elements. The function must be written in C and should define the arguements as integers. For Example:

```
function DeltaMyType(var lo, hi : MyType) : integer  ;
               external  ;


int
DeltaMyType(lo, hi)
int       lo, hi  ;
{
          return(hi - lo)  ;
}
```

There are some other special types. The Unix file system sets permission codes for files. In the header files these parameters are always called **mode.** The constants **OtherExec, OtherWrite, .., GroupExec, .., OwnWrite,** can be added together to form the desired permission code. The SYSaccess function has the testmode argument, which takes a sum of the **AccessExist, AccessExec, AccessWrite,** and **AccessRead** constants. The SYSlseek function uses the **SeekAbsolute, SeekRelative,** and **SeekFromEnd** constants (not added together). Finally, the SYSopen function uses the constants **OpenReadOnly, OpenWriteOnly, OpenReadWrite, OpenNoDelay, OpenAppend, OpenCreat, OpenTrunc,** and **OpenExcl.**

Normally the parameters of each SYS procedure correspond to the parameters of the C function. The acceptions are the memory allocation routines, which return the pointer as a var parameter rather than as a function result. Note: these procedures also had to be integrated into the Pascal runtime environment, care should be taken when rewriting.

DIAGNOSTICS

Generally, error returns are the same as for C. SYSerror can be used to obtain a text description of each error, providing there are no intervening SYS calls.

FILES

$

SEE ALSO

Associated C functions, and section 2 introduction.

IDENTIFICATION

George McA Beshers, UIUC DCL Urbana Ill. 61801.

BUGS

The systring type is currently limited to 126 characters which is somewhat small.

NAME

AllocPermid

SYNOPSIS

```
AllocPermid(
                        name:          systring) : sypermidindex;
```

DESCRIPTION

This procedure allocates a permanent id for SAGA string and symbol tables. For this routine to work the environment variable SAGA_INDEX_FILE must be set the pathname of a writeable file. The file is maintained in a format similiar to /etc/passwd. Specifically, the permanent id, colon, and the full path name. Unfortunately, AllocPermid is no smarter than **csh,** i.e., it is fooled by symbolic links.

In practice this function need only be called when a new file is created. If the full path name equals one already in the table, that permanent id is returned. Currently, the table size is 1k, the goal being support SAGA (editor, olorin, filters, ...) under SAGA. Another way to think of this is that the SAGA_INDEX_FILE is a view of the SAGA system.

If an error occures a message is printed. Index 1024 is the error return.

DIAGNOSTICS

getwd failed.
Unix United not supported (path starts with /../).
getenv failed (SAGA_INDEX_FILE is not set).
SAGA Index File open failed.

FILES

File specified by SAGA_INDEX_FILE.

SEE ALSO

String.3, Richards Thesis.

IDENTIFICATION

Beshers, George. beshers@uiucdcs.

BUGS

Perhaps one should be the error return. One is a valid permanent id, thus the editor would keep working in an improper environment.

NAME

String Manager – String table management for SAGA.

SYNOPSIS

```
****** String Table Routines ******

createstringtable(
              name:           systring;
              permid:         sypermidindex;
              mode:           integer;
          var rootcontext:    contexttag;
          var error:          boolean);

openstringtable(
              name:           systring;
          var permid:         sypermidindex;
          var rootcontext:    contexttag;
          var error:          boolean);

closestringtable(
              rootcontext:    contexttag;
          var error:          boolean);


flushstringtable(
              rootcontext:    contexttag;
          var error:          boolean);

geterrorflags(
          var errorflags:     errorset);

geterrtext(
              errortype:              syerrorkind;
          var errtxt:         systring);

initstringmanager;


****** String Manipulation Routines ******

insertstring(
              name:           systring;
              context:        contexttag;
          var newstring:      stringtag;
          var found:          boolean;
          var error:          boolean);

retrievestring(
              string:         stringtag;
          var name:           systring;
          var error:          boolean);

locatestring(
            name:         systring;
            context:      contexttag;
        var string:       stringtag;
        var found:        boolean;
        var error:        boolean);

retrievestringlength(
            string:       stringtag;
        var error:        boolean) : integer;

deletestring(        *Not Active*
            string:       stringtag;
        var error:        boolean);

comparestring(
            strtg1:       stringtag;
            strtg2:       stringtag;
        var error:        boolean) : sycompareresult;

comparestringbystring(
            str1:         systring;
            strtg2:       stringtag;
        var error:        boolean) : sycompareresult;

getstringtype(
            string:       stringtag;
        var stringtype:   integer;
        var error:        boolean);

setstringtype(
            string:       stringtag;
            stringtype:   integer;
        var error:        boolean);

gettagfrag(
            string:       stringtag) : sytagfragment;

buildtag(
            permid:       sypermidindex;
            tagfrag:      sytagfragment) : stringtag;

sycompareresult = (strlt, streq, strgt) ;


****** Systring Utility Routines ******

makestring(
            s :      charbuf  ;
        var sy:    systring)  ;

concatsystring(
        var result:       systring;
```

```
                first:          systring;
                second:         systring) ;

    int2string(
                i : integer;
            var result:         systring;

    wrsystr(
            var out :           text;
            s : systring) ;
```

DESCRIPTION

These routines constitute the SAGA string manager. The **initstringmanager** routine must be called first since it is the Pascal "solution" to compile time initialization.

The **openstringtable, createstringtable, flushstringtable,** and **closestringtable** procedures provide the file system level access to a string table. The file system procedures append ".str" to the name provided and attempt the operation implied by their name. You can not open or creat the same file (by path name) twice, or two files with the same permanent id. All four of the operations can fail due to file system access failure.

The concept of "contexttag" pertains more to the symbol manager than the string manager, and is used here for compatability. The context tags actually used may be either the root context returned by the **createstringtable** and **openstringtable,** or any other active context for the symbol table with the same permanent id. The permanent id is used to distinguish between different string tables. It is encoded in both "contexttags" and "stringttags" so that a tag uniquely identifiers a particular string throughout the system. The mechanism for assigning permanent ids is described in AllocPermid.

The string manager deals with **systring(s)** which are a record with the following fields:

```
start:  1..126;
count:  0..126;
chars:  array [1..126] of char;
```

Thus if the chars contains "This is a test", with start=4 and count=5 then the string equals "s is ". The procedures makestring, concatsystring, int2systring, and wrsystr are auxilary routines to help manipulate systrings. Note: makestring('testing 1 2 3', s) works fine, but trailing spaces are lost. **Wrsystr** writes the string to the specified file.

The **insertstring** is the only way to put strings into the symbol table. The inserted string's tag is returned in new string. NOTE: if the string exists **found** is set, and NO error is generated, contrary to earlier versions. The **retrievestring** routine is the inverse. It is of course an error to try to retrieve a string associated with an un-opened string table, or a string which doesn't exist. The **retrievestringlength** is faster than retrievestring, used mostly by the editor for screen refresh. The **deletestring** procedure exists, but is disabled because it is not possible to inhibit copying of editor pointers. The **getstringtype** and **setstringtype** permit an integer to be stored with each string for classification purposes (reserved words, function/procedure/variable classification ...).

The **geterrorflags** and **geterrortext** routines are used by both the string and symbol table managers. They should be called whenever the "error" parameter is set upon procedure return.

The **gettagfrag** and **buildtag** routines provide support for optimizations used by the editor. The sytagfragment is a 2 byte quantity, and the stringtag is 4 bytes. This saves some space in the parse tree node.

DIAGNOSTICS

FILES

name.str

SEE ALSO

symbol(3), AllocPermid(3)

IDENTIFICATION

beshers@uiucdcs

BUGS

126 is too small.