Analizador Sintáctico

Ariana Bermúdez,Ximena Bolaños, Dylan Rodríguez

Instituto Tecnológico de Costa Rica

May 30, 2017

Análisis Sintáctico

Se hizo un analizador sintáctico con la ayuda de la herramienta de Bison, para el lenguaje C y que corre en C, este analizador trabaja en conjunto con Flex, para tomar los tokens que este le otorga y revisar con las gramáticas que les sean ingresadas.

Bison

Bison convierte de una gramática libre de contexto a un analizador sintáctico que emplea las tablas de Parsing LALR(1), siendo:

- L: Left algo
- A: ...
- L: ...
- R: rightmost
- (1): donde este uno significa que tiene como lookahead solo un símbolo.

Cabe destacar que Bison es compatible con Yacc. Sirve con C, C++ y Java.

Código

```
static int get_st_mode_bits ( const char * path , int * mode )
{
struct stat st ;
if ( lstat ( path , & st ) < 0 )
return - 1 ;
* mode = st . st_mode ;
return 0 ;
}
static char bad_path [ ] = "/bad-path/" ;
static struct strbuf * get_pathname ( void )
{
static struct strbuf pathname_array [ 4 ] = {
```

Código

```
STRBUF_INIT , STRBUF_INIT , STRBUF_INIT , STRBUF_INIT
} ;
static int index ;
struct strbuf * sb = & pathname_array [ index ] ;
index = ( index + 1 ) % ARRAY_SIZE ( pathname_array ) ;
strbuf_reset ( sb ) ;
return sb ;
}
static char * cleanup_path ( char * path )
{
if ( ! memcmp ( path , "./" , 2 ) ) {
path += 2 ;
while ( * path == '/' )
```

Código

```
path ++ ;
}
return path ;
}
static void strbuf_cleanup_path ( struct strbuf * sb )
{
char * path = cleanup_path ( sb -> buf ) ;
if ( path > sb -> buf )
strbuf_remove ( sb , 0 , path - sb -> buf ) ;
}
char * mksnpath ( char * buf , size_t n , const char * fmt , ... )
{
va_list args ;
```

Código

```
unsigned len ;
va_start ( args , fmt ) ;
len = vsnprintf ( buf , n , fmt , args ) ;
va_end ( args ) ;
if ( len >= n ) {
strlcpy ( buf , bad_path , n ) ;
return buf ;
}
return cleanup_path ( buf ) ;
}
static int dir_prefix ( const char * buf , const char * dir )
{
int len = strlen ( dir ) ;
```

Código

```
return ! strncmp ( buf , dir , len ) &&
( is_dir_sep ( buf [ len ] ) || buf [ len ] == '\0' ) ;
}
static int is_dir_file ( const char * buf , const char * dir , const char * file )
{
int len = strlen ( dir ) ;
if ( strncmp ( buf , dir , len ) || ! is_dir_sep ( buf [ len ] ) )
return 0 ;
while ( is_dir_sep ( buf [ len ] ) )
len ++ ;
return ! strcmp ( buf + len , file ) ;
}
static void replace_dir ( struct strbuf * buf , int len , const char * newdir )
```

Código

```
{
int newlen = strlen ( newdir ) ;
int need_sep = ( buf -> buf [ len ] && ! is_dir_sep ( buf -> buf [ len ] ) ) &&
! is_dir_sep ( newdir [ newlen - 1 ] ) ;
if ( need_sep )
len -- ;
strbuf_splice ( buf , 0 , len , newdir , newlen ) ;
if ( need_sep )
buf -> buf [ newlen ] = '/' ;
}
struct common_dir {
unsigned ignore_garbage : 1 ;
unsigned is_dir : 1 ;
```

Código

```
unsigned exclude : 1 ;
const char * dirname ;
} ;
static struct common_dir common_list [ ] = {
{ 0 , 1 , 0 , "branches" } ,
{ 0 , 1 , 0 , "hooks" } ,
{ 0 , 1 , 0 , "info" } ,
{ 0 , 0 , 1 , "info/sparse-checkout" } ,
{ 1 , 1 , 0 , "logs" } ,
{ 1 , 1 , 1 , "logs/HEAD" } ,
{ 0 , 1 , 1 , "logs/refs/bisect" } ,
{ 0 , 1 , 0 , "lost-found" } ,
{ 0 , 1 , 0 , "objects" } ,
```

Código

```
{ 0 , 1 , 0 , "refs" } ,
{ 0 , 1 , 1 , "refs/bisect" } ,
{ 0 , 1 , 0 , "remotes" } ,
{ 0 , 1 , 0 , "worktrees" } ,
{ 0 , 1 , 0 , "rr-cache" } ,
{ 0 , 1 , 0 , "svn" } ,
{ 0 , 0 , 0 , "config" } ,
{ 1 , 0 , 0 , "gc.pid" } ,
{ 0 , 0 , 0 , "packed-refs" } ,
{ 0 , 0 , 0 , "shallow" } ,
{ 0 , 0 , 0 , NULL }
} ;
struct trie {
```

Código

```
struct trie * children [ 256 ] ;
int len ;
char * contents ;
void * value ;
} ;
static struct trie * make_trie_node ( const char * key , void * value )
{
struct trie * new_node = xcalloc ( 1 , sizeof ( * new_node ) ) ;
new_node -> len = strlen ( key ) ;
if ( new_node -> len ) {
new_node -> contents = xmalloc ( new_node -> len ) ;
memcpy ( new_node -> contents , key , new_node -> len ) ;
}
```

Código

```
new_node -> value = value ;
return new_node ;
}
static void * add_to_trie ( struct trie * root , const char * key , void * value )
{
struct trie * child ;
void * old ;
int i ;
if ( ! * key ) {
old = root -> value ;
root -> value = value ;
return old ;
}
```

Código

```
for ( i = 0 ; i < root -> len ; i ++ ) {
if ( root -> contents [ i ] == key [ i ] )
continue ;
child = malloc ( sizeof ( * child ) ) ;
memcpy ( child -> children , root -> children , sizeof ( root -> children ) ) ;
child -> len = root -> len - i - 1 ;
if ( child -> len ) {
child -> contents = xstrndup ( root -> contents + i + 1 ,
child -> len ) ;
}
child -> value = root -> value ;
root -> value = NULL ;
root -> len = i ;
```

Código

```
memset ( root -> children , 0 , sizeof ( root -> children ) ) ;
root -> children [ ( unsigned char ) root -> contents [ i ] ] = child ;
root -> children [ ( unsigned char ) key [ i ] ] =
make_trie_node ( key + i + 1 , value ) ;
return NULL ;
}
if ( key [ i ] ) {
child = root -> children [ ( unsigned char ) key [ root -> len ] ] ;
if ( child ) {
return add_to_trie ( child , key + root -> len + 1 , value ) ;
} else {
child = make_trie_node ( key + root -> len + 1 , value ) ;
root -> children [ ( unsigned char ) key [ root -> len ] ] = child ;
```

Código

```
return NULL ;
}
}
old = root -> value ;
root -> value = value ;
return old ;
}
typedef int ( * match_fn ) ( const char * unmatched , void * data , void * baton ) ;
static int trie_find ( struct trie * root , const char * key , match_fn fn ,
void * baton )
{
int i ;
int result ;
```

Código

```
struct trie * child ;
if ( ! * key ) {
if ( root -> value && ! root -> len )
return fn ( key , root -> value , baton ) ;
else
return - 1 ;
}
for ( i = 0 ; i < root -> len ; i ++ ) {
if ( key [ i ] == '/' && key [ i + 1 ] == '/' ) {
key ++ ;
continue ;
}
if ( root -> contents [ i ] != key [ i ] )
```

Código

```
return - 1 ;
}
key += i ;
if ( ! * key )
return fn ( key , root -> value , baton ) ;
while ( key [ 0 ] == '/' && key [ 1 ] == '/' )
key ++ ;
child = root -> children [ ( unsigned char ) * key ] ;
if ( child )
result = trie_find ( child , key + 1 , fn , baton ) ;
else
result = - 1 ;
if ( result >= 0 || ( * key != '/' && * key != 0 ) )
```

Código

```
return result ;
if ( root -> value )
return fn ( key , root -> value , baton ) ;
else
return - 1 ;
}
static struct trie common_trie ;
static int common_trie_done_setup ;
static void init_common_trie ( void )
{
struct common_dir * p ;
if ( common_trie_done_setup )
return ;
```

Código

```
for ( p = common_list ; p -> dirname ; p ++ )
add_to_trie ( & common_trie , p -> dirname , p ) ;
common_trie_done_setup = 1 ;
}
static int check_common ( const char * unmatched , void * value , void * baton )
{
struct common_dir * dir = value ;
if ( ! dir )
return 0 ;
if ( dir -> is_dir && ( unmatched [ 0 ] == 0 || unmatched [ 0 ] == '/' ) )
return ! dir -> exclude ;
if ( ! dir -> is_dir && unmatched [ 0 ] == 0 )
return ! dir -> exclude ;
```

Código

```
return 0 ;
}
static void update_common_dir ( struct strbuf * buf , int git_dir_len ,
const char * common_dir )
{
char * base = buf -> buf + git_dir_len ;
init_common_trie ( ) ;
if ( ! common_dir )
common_dir = get_git_common_dir ( ) ;
if ( trie_find ( & common_trie , base , check_common , NULL ) > 0 )
replace_dir ( buf , git_dir_len , common_dir ) ;
}
void report_linked_checkout_garbage ( void )
```

Código

```
{
struct strbuf sb = STRBUF_INIT ;
const struct common_dir * p ;
int len ;
if ( ! git_common_dir_env )
return ;
strbuf_addf ( & sb , "%s/" , get_git_dir ( ) ) ;
len = sb . len ;
for ( p = common_list ; p -> dirname ; p ++ ) {
const char * path = p -> dirname ;
if ( p -> ignore_garbage )
continue ;
strbuf_setlen ( & sb , len ) ;
```

Código

```
strbuf_addstr ( & sb , path ) ;
if ( file_exists ( sb . buf ) )
report_garbage ( PACKDIR_FILE_GARBAGE , sb . buf ) ;
}
strbuf_release ( & sb ) ;
}
static void adjust_git_path ( struct strbuf * buf , int git_dir_len )
{
const char * base = buf -> buf + git_dir_len ;
if ( git_graft_env && is_dir_file ( base , "info" , "grafts" ) )
strbuf_splice ( buf , 0 , buf -> len ,
get_graft_file ( ) , strlen ( get_graft_file ( ) ) ) ;
else if ( git_index_env && ! strcmp ( base , "index" ) )
```

Código

```
strbuf_splice ( buf , 0 , buf -> len ,
get_index_file ( ) , strlen ( get_index_file ( ) ) ) ;
else if ( git_db_env && dir_prefix ( base , "objects" ) )
replace_dir ( buf , git_dir_len + 7 , get_object_directory ( ) ) ;
else if ( git_hooks_path && dir_prefix ( base , "hooks" ) )
replace_dir ( buf , git_dir_len + 5 , git_hooks_path ) ;
else if ( git_common_dir_env )
update_common_dir ( buf , git_dir_len , NULL ) ;
}
static void do_git_path ( const struct worktree * wt , struct strbuf * buf ,
const char * fmt , va_list args )
{
int gitdir_len ;
```

Código

```
strbuf_addstr ( buf , get_worktree_git_dir ( wt ) ) ;
if ( buf -> len && ! is_dir_sep ( buf -> buf [ buf -> len - 1 ] ) )
strbuf_addch ( buf , '/' ) ;
gitdir_len = buf -> len ;
strbuf_vaddf ( buf , fmt , args ) ;
adjust_git_path ( buf , gitdir_len ) ;
strbuf_cleanup_path ( buf ) ;
}
char * git_path_buf ( struct strbuf * buf , const char * fmt , ... )
{
va_list args ;
strbuf_reset ( buf ) ;
va_start ( args , fmt ) ;
```

Código

```
do_git_path ( NULL , buf , fmt , args ) ;
va_end ( args ) ;
return buf -> buf ;
}
void strbuf_git_path ( struct strbuf * sb , const char * fmt , ... )
{
va_list args ;
va_start ( args , fmt ) ;
do_git_path ( NULL , sb , fmt , args ) ;
va_end ( args ) ;
}
const char * git_path ( const char * fmt , ... )
{
```

Código

```
struct strbuf * pathname = get_pathname ( ) ;
va_list args ;
va_start ( args , fmt ) ;
do_git_path ( NULL , pathname , fmt , args ) ;
va_end ( args ) ;
return pathname -> buf ;
}
char * git_pathdup ( const char * fmt , ... )
{
struct strbuf path = STRBUF_INIT ;
va_list args ;
va_start ( args , fmt ) ;
do_git_path ( NULL , & path , fmt , args ) ;
```

Código

```
va_end ( args ) ;
return strbuf_detach ( & path , NULL ) ;
}
char * mkpathdup ( const char * fmt , ... )
{
struct strbuf sb = STRBUF_INIT ;
va_list args ;
va_start ( args , fmt ) ;
strbuf_vaddf ( & sb , fmt , args ) ;
va_end ( args ) ;
strbuf_cleanup_path ( & sb ) ;
return strbuf_detach ( & sb , NULL ) ;
}
```

Código

```
const char * mkpath ( const char * fmt , ... )
{
va_list args ;
struct strbuf * pathname = get_pathname ( ) ;
va_start ( args , fmt ) ;
strbuf_vaddf ( pathname , fmt , args ) ;
va_end ( args ) ;
return cleanup_path ( pathname -> buf ) ;
}
const char * worktree_git_path ( const struct worktree * wt , const char * fmt , ... )
{
struct strbuf * pathname = get_pathname ( ) ;
va_list args ;
```

Código

```
va_start ( args , fmt ) ;
do_git_path ( wt , pathname , fmt , args ) ;
va_end ( args ) ;
return pathname -> buf ;
}
static int do_submodule_path ( struct strbuf * buf , const char * path ,
const char * fmt , va_list args )
{
struct strbuf git_submodule_common_dir = STRBUF_INIT ;
struct strbuf git_submodule_dir = STRBUF_INIT ;
int ret ;
ret = submodule_to_gitdir ( & git_submodule_dir , path ) ;
if ( ret )
```

Código

```
goto cleanup ;
strbuf_complete ( & git_submodule_dir , '/' ) ;
strbuf_addbuf ( buf , & git_submodule_dir ) ;
strbuf_vaddf ( buf , fmt , args ) ;
if ( get_common_dir_noenv ( & git_submodule_common_dir , git_submodule_dir . buf ) )
update_common_dir ( buf , git_submodule_dir . len , git_submodule_common_dir . buf ) ;
strbuf_cleanup_path ( buf ) ;
cleanup :
strbuf_release ( & git_submodule_dir ) ;
strbuf_release ( & git_submodule_common_dir ) ;
return ret ;
}
char * git_pathdup_submodule ( const char * path , const char * fmt , ... )
```

Código

```
{
int err ;
va_list args ;
struct strbuf buf = STRBUF_INIT ;
va_start ( args , fmt ) ;
err = do_submodule_path ( & buf , path , fmt , args ) ;
va_end ( args ) ;
if ( err ) {
strbuf_release ( & buf ) ;
return NULL ;
}
return strbuf_detach ( & buf , NULL ) ;
}
```

Código

```
int strbuf_git_path_submodule ( struct strbuf * buf , const char * path ,
const char * fmt , ... )
{
int err ;
va_list args ;
va_start ( args , fmt ) ;
err = do_submodule_path ( buf , path , fmt , args ) ;
va_end ( args ) ;
return err ;
}
static void do_git_common_path ( struct strbuf * buf ,
const char * fmt ,
va_list args )
```

Código

```
{
strbuf_addstr ( buf , get_git_common_dir ( ) ) ;
if ( buf -> len && ! is_dir_sep ( buf -> buf [ buf -> len - 1 ] ) )
strbuf_addch ( buf , '/' ) ;
strbuf_vaddf ( buf , fmt , args ) ;
strbuf_cleanup_path ( buf ) ;
}
const char * git_common_path ( const char * fmt , ... )
{
struct strbuf * pathname = get_pathname ( ) ;
va_list args ;
va_start ( args , fmt ) ;
do_git_common_path ( pathname , fmt , args ) ;
```

Código

```
va_end ( args ) ;
return pathname -> buf ;
}
void strbuf_git_common_path ( struct strbuf * sb , const char * fmt , ... )
{
va_list args ;
va_start ( args , fmt ) ;
do_git_common_path ( sb , fmt , args ) ;
va_end ( args ) ;
}
int validate_headref ( const char * path )
{
struct stat st ;
```

Código

```
char * buf , buffer [ 256 ] ;
unsigned char sha1 [ 20 ] ;
int fd ;
ssize_t len ;
if ( lstat ( path , & st ) < 0 )
return - 1 ;
if ( S_ISLNK ( st . st_mode ) ) {
len = readlink ( path , buffer , sizeof ( buffer ) - 1 ) ;
if ( len >= 5 && ! memcmp ( "refs/" , buffer , 5 ) )
return 0 ;
return - 1 ;
}
fd = open ( path , O_RDONLY ) ;
```

Código

```
if ( fd < 0 )
return - 1 ;
len = read_in_full ( fd , buffer , sizeof ( buffer ) - 1 ) ;
close ( fd ) ;
if ( len < 4 )
return - 1 ;
if ( ! memcmp ( "ref:" , buffer , 4 ) ) {
buf = buffer + 4 ;
len -= 4 ;
while ( len && isspace ( * buf ) )
buf ++ , len -- ;
if ( len >= 5 && ! memcmp ( "refs/" , buf , 5 ) )
return 0 ;
```

Código

```
}
if ( ! get_sha1_hex ( buffer , sha1 ) )
return 0 ;
return - 1 ;
}
static struct passwd * getpw_str ( const char * username , size_t len )
{
struct passwd * pw ;
char * username_z = xmemdupz ( username , len ) ;
pw = getpwnam ( username_z ) ;
free ( username_z ) ;
return pw ;
}
```

Código

```
char * expand_user_path ( const char * path , int real_home )
{
struct strbuf user_path = STRBUF_INIT ;
const char * to_copy = path ;
if ( path == NULL )
goto return_null ;
if ( path [ 0 ] == '~' ) {
const char * first_slash = strchrnul ( path , '/' ) ;
const char * username = path + 1 ;
size_t username_len = first_slash - username ;
if ( username_len == 0 ) {
const char * home = getenv ( "HOME" ) ;
if ( ! home )
```

Código

```
goto return_null ;
if ( real_home )
strbuf_addstr ( & user_path , real_path ( home ) ) ;
else
```

Código con los Errores

```
static int get_st_mode_bits ( const char * path , int * mode )
{
struct stat st ;
if ( lstat ( path , & st ) < 0 )
return - 1 ;
* mode = st . st_mode ;
return 0 ;
}
static char bad_path [ ] = "/bad-path/" ;
static struct strbuf * get_pathname ( void )
{
static struct strbuf pathname_array [ 4 ] = {
```

Código con los Errores

```
STRBUF_INIT , STRBUF_INIT , STRBUF_INIT , STRBUF_INIT
} ;
static int index ;
struct strbuf * sb = & pathname_array [ index ] ;
index = ( index + 1 ) % ARRAY_SIZE ( pathname_array ) ;
strbuf_reset ( sb ) ;
return sb ;
}
static char * cleanup_path ( char * path )
{
if ( ! memcmp ( path , "./" , 2 ) ) {
path += 2 ;
while ( * path == '/' )
```

Código con los Errores

```
path ++ ;
}
return path ;
}
static void strbuf_cleanup_path ( struct strbuf * sb )
{
char * path = cleanup_path ( sb -> buf ) ;
if ( path > sb -> buf )
strbuf_remove ( sb , 0 , path - sb -> buf ) ;
}
char * mksnpath ( char * buf , size_t n , const char * fmt , ... )
{
va_list args ;
```

Código con los Errores

```
unsigned len ;
va_start ( args , fmt ) ;
len = vsnprintf ( buf , n , fmt , args ) ;
va_end ( args ) ;
if ( len >= n ) {
strlcpy ( buf , bad_path , n ) ;
return buf ;
}
return cleanup_path ( buf ) ;
}
static int dir_prefix ( const char * buf , const char * dir )
{
int len = strlen ( dir ) ;
```

Código con los Errores

```
return ! strncmp ( buf , dir , len ) &&
( is_dir_sep ( buf [ len ] ) || buf [ len ] == '\0' ) ;
}
static int is_dir_file ( const char * buf , const char * dir , const char * file )
{
int len = strlen ( dir ) ;
if ( strncmp ( buf , dir , len ) || ! is_dir_sep ( buf [ len ] ) )
return 0 ;
while ( is_dir_sep ( buf [ len ] ) )
len ++ ;
return ! strcmp ( buf + len , file ) ;
}
static void replace_dir ( struct strbuf * buf , int len , const char * newdir )
```

Código con los Errores

```
{
int newlen = strlen ( newdir ) ;
int need_sep = ( buf -> buf [ len ] && ! is_dir_sep ( buf -> buf [ len ] ) ) &&
! is_dir_sep ( newdir [ newlen - 1 ] ) ;
if ( need_sep )
len -- ;
strbuf_splice ( buf , 0 , len , newdir , newlen ) ;
if ( need_sep )
buf -> buf [ newlen ] = '/' ;
}
struct common_dir {
unsigned ignore_garbage : 1 ;
unsigned is_dir : 1 ;
```

Código con los Errores

```
unsigned exclude : 1 ;
const char * dirname ;
} ;
static struct common_dir common_list [ ] = {
{ 0 , 1 , 0 , "branches" } ,
{ 0 , 1 , 0 , "hooks" } ,
{ 0 , 1 , 0 , "info" } ,
{ 0 , 0 , 1 , "info/sparse-checkout" } ,
{ 1 , 1 , 0 , "logs" } ,
{ 1 , 1 , 1 , "logs/HEAD" } ,
{ 0 , 1 , 1 , "logs/refs/bisect" } ,
{ 0 , 1 , 0 , "lost-found" } ,
{ 0 , 1 , 0 , "objects" } ,
```

Código con los Errores

```
{ 0 , 1 , 0 , "refs" } ,
{ 0 , 1 , 1 , "refs/bisect" } ,
{ 0 , 1 , 0 , "remotes" } ,
{ 0 , 1 , 0 , "worktrees" } ,
{ 0 , 1 , 0 , "rr-cache" } ,
{ 0 , 1 , 0 , "svn" } ,
{ 0 , 0 , 0 , "config" } ,
{ 1 , 0 , 0 , "gc.pid" } ,
{ 0 , 0 , 0 , "packed-refs" } ,
{ 0 , 0 , 0 , "shallow" } ,
{ 0 , 0 , 0 , NULL }
} ;
struct trie {
```

Código con los Errores

```
struct trie * children [ 256 ] ;
int len ;
char * contents ;
void * value ;
} ;
static struct trie * make_trie_node ( const char * key , void * value )
{
struct trie * new_node = xcalloc ( 1 , sizeof ( * new_node ) ) ;
new_node -> len = strlen ( key ) ;
if ( new_node -> len ) {
new_node -> contents = xmalloc ( new_node -> len ) ;
memcpy ( new_node -> contents , key , new_node -> len ) ;
}
```

Código con los Errores

```
new_node -> value = value ;
return new_node ;
}
static void * add_to_trie ( struct trie * root , const char * key , void * value )
{
struct trie * child ;
void * old ;
int i ;
if ( ! * key ) {
old = root -> value ;
root -> value = value ;
return old ;
}
```

Código con los Errores

```
for ( i = 0 ; i < root -> len ; i ++ ) {
if ( root -> contents [ i ] == key [ i ] )
continue ;
child = malloc ( sizeof ( * child ) ) ;
memcpy ( child -> children , root -> children , sizeof ( root -> children ) ) ;
child -> len = root -> len - i - 1 ;
if ( child -> len ) {
child -> contents = xstrndup ( root -> contents + i + 1 ,
child -> len ) ;
}
child -> value = root -> value ;
root -> value = NULL ;
root -> len = i ;
```

Código con los Errores

```
memset ( root -> children , 0 , sizeof ( root -> children ) ) ;
root -> children [ ( unsigned char ) root -> contents [ i ] ] = child ;
root -> children [ ( unsigned char ) key [ i ] ] =
make_trie_node ( key + i + 1 , value ) ;
return NULL ;
}
if ( key [ i ] ) {
child = root -> children [ ( unsigned char ) key [ root -> len ] ] ;
if ( child ) {
return add_to_trie ( child , key + root -> len + 1 , value ) ;
} else {
child = make_trie_node ( key + root -> len + 1 , value ) ;
root -> children [ ( unsigned char ) key [ root -> len ] ] = child ;
```

Código con los Errores

```
return NULL ;
}
}
old = root -> value ;
root -> value = value ;
return old ;
}
typedef int ( * match_fn ) ( const char * unmatched , void * data , void * baton ) ;
static int trie_find ( struct trie * root , const char * key , match_fn fn ,
void * baton )
{
int i ;
int result ;
```

Código con los Errores

```
struct trie * child ;
if ( ! * key ) {
if ( root -> value && ! root -> len )
return fn ( key , root -> value , baton ) ;
else
return - 1 ;
}
for ( i = 0 ; i < root -> len ; i ++ ) {
if ( key [ i ] == '/' && key [ i + 1 ] == '/' ) {
key ++ ;
continue ;
}
if ( root -> contents [ i ] != key [ i ] )
```

Código con los Errores

```
return - 1 ;
}
key += i ;
if ( ! * key )
return fn ( key , root -> value , baton ) ;
while ( key [ 0 ] == '/' && key [ 1 ] == '/' )
key ++ ;
child = root -> children [ ( unsigned char ) * key ] ;
if ( child )
result = trie_find ( child , key + 1 , fn , baton ) ;
else
result = - 1 ;
if ( result >= 0 || ( * key != '/' && * key != 0 ) )
```

Código con los Errores

```
return result ;
if ( root -> value )
return fn ( key , root -> value , baton ) ;
else
return - 1 ;
}
static struct trie common_trie ;
static int common_trie_done_setup ;
static void init_common_trie ( void )
{
struct common_dir * p ;
if ( common_trie_done_setup )
return ;
```

Código con los Errores

```
for ( p = common_list ; p -> dirname ; p ++ )
add_to_trie ( & common_trie , p -> dirname , p ) ;
common_trie_done_setup = 1 ;
}
static int check_common ( const char * unmatched , void * value , void * baton )
{
struct common_dir * dir = value ;
if ( ! dir )
return 0 ;
if ( dir -> is_dir && ( unmatched [ 0 ] == 0 || unmatched [ 0 ] == '/' ) )
return ! dir -> exclude ;
if ( ! dir -> is_dir && unmatched [ 0 ] == 0 )
return ! dir -> exclude ;
```

Código con los Errores

```
return 0 ;
}
static void update_common_dir ( struct strbuf * buf , int git_dir_len ,
const char * common_dir )
{
char * base = buf -> buf + git_dir_len ;
init_common_trie ( ) ;
if ( ! common_dir )
common_dir = get_git_common_dir ( ) ;
if ( trie_find ( & common_trie , base , check_common , NULL ) > 0 )
replace_dir ( buf , git_dir_len , common_dir ) ;
}
void report_linked_checkout_garbage ( void )
```

Código con los Errores

```
{
struct strbuf sb = STRBUF_INIT ;
const struct common_dir * p ;
int len ;
if ( ! git_common_dir_env )
return ;
strbuf_addf ( & sb , "%s/" , get_git_dir ( ) ) ;
len = sb . len ;
for ( p = common_list ; p -> dirname ; p ++ ) {
const char * path = p -> dirname ;
if ( p -> ignore_garbage )
continue ;
strbuf_setlen ( & sb , len ) ;
```

Código con los Errores

```
strbuf_addstr ( & sb , path ) ;
if ( file_exists ( sb . buf ) )
report_garbage ( PACKDIR_FILE_GARBAGE , sb . buf ) ;
}
strbuf_release ( & sb ) ;
}
static void adjust_git_path ( struct strbuf * buf , int git_dir_len )
{
const char * base = buf -> buf + git_dir_len ;
if ( git_graft_env && is_dir_file ( base , "info" , "grafts" ) )
strbuf_splice ( buf , 0 , buf -> len ,
get_graft_file ( ) , strlen ( get_graft_file ( ) ) ) ;
else if ( git_index_env && ! strcmp ( base , "index" ) )
```

Código con los Errores

```
strbuf_splice ( buf , 0 , buf -> len ,
get_index_file ( ) , strlen ( get_index_file ( ) ) ) ;
else if ( git_db_env && dir_prefix ( base , "objects" ) )
replace_dir ( buf , git_dir_len + 7 , get_object_directory ( ) ) ;
else if ( git_hooks_path && dir_prefix ( base , "hooks" ) )
replace_dir ( buf , git_dir_len + 5 , git_hooks_path ) ;
else if ( git_common_dir_env )
update_common_dir ( buf , git_dir_len , NULL ) ;
}
static void do_git_path ( const struct worktree * wt , struct strbuf * buf ,
const char * fmt , va_list args )
{
int gitdir_len ;
```

Código con los Errores

```
strbuf_addstr ( buf , get_worktree_git_dir ( wt ) ) ;
if ( buf -> len && ! is_dir_sep ( buf -> buf [ buf -> len - 1 ] ) )
strbuf_addch ( buf , '/' ) ;
gitdir_len = buf -> len ;
strbuf_vaddf ( buf , fmt , args ) ;
adjust_git_path ( buf , gitdir_len ) ;
strbuf_cleanup_path ( buf ) ;
}
char * git_path_buf ( struct strbuf * buf , const char * fmt , ... )
{
va_list args ;
strbuf_reset ( buf ) ;
va_start ( args , fmt ) ;
```

Código con los Errores

```
do_git_path ( NULL , buf , fmt , args ) ;
va_end ( args ) ;
return buf -> buf ;
}
void strbuf_git_path ( struct strbuf * sb , const char * fmt , ... )
{
va_list args ;
va_start ( args , fmt ) ;
do_git_path ( NULL , sb , fmt , args ) ;
va_end ( args ) ;
}
const char * git_path ( const char * fmt , ... )
{
```

Código con los Errores

```
struct strbuf * pathname = get_pathname ( ) ;
va_list args ;
va_start ( args , fmt ) ;
do_git_path ( NULL , pathname , fmt , args ) ;
va_end ( args ) ;
return pathname -> buf ;
}
char * git_pathdup ( const char * fmt , ... )
{
struct strbuf path = STRBUF_INIT ;
va_list args ;
va_start ( args , fmt ) ;
do_git_path ( NULL , & path , fmt , args ) ;
```

Código con los Errores

```
va_end ( args ) ;
return strbuf_detach ( & path , NULL ) ;
}
char * mkpathdup ( const char * fmt , ... )
{
struct strbuf sb = STRBUF_INIT ;
va_list args ;
va_start ( args , fmt ) ;
strbuf_vaddf ( & sb , fmt , args ) ;
va_end ( args ) ;
strbuf_cleanup_path ( & sb ) ;
return strbuf_detach ( & sb , NULL ) ;
}
```

Código con los Errores

```
const char * mkpath ( const char * fmt , ... )
{
va_list args ;
struct strbuf * pathname = get_pathname ( ) ;
va_start ( args , fmt ) ;
strbuf_vaddf ( pathname , fmt , args ) ;
va_end ( args ) ;
return cleanup_path ( pathname -> buf ) ;
}
const char * worktree_git_path ( const struct worktree * wt , const char * fmt , ... )
{
struct strbuf * pathname = get_pathname ( ) ;
va_list args ;
```

Código con los Errores

```
va_start ( args , fmt ) ;
do_git_path ( wt , pathname , fmt , args ) ;
va_end ( args ) ;
return pathname -> buf ;
}
static int do_submodule_path ( struct strbuf * buf , const char * path ,
const char * fmt , va_list args )
{
struct strbuf git_submodule_common_dir = STRBUF_INIT ;
struct strbuf git_submodule_dir = STRBUF_INIT ;
int ret ;
ret = submodule_to_gitdir ( & git_submodule_dir , path ) ;
if ( ret )
```

Código con los Errores

```
goto cleanup ;
strbuf_complete ( & git_submodule_dir , '/' ) ;
strbuf_addbuf ( buf , & git_submodule_dir ) ;
strbuf_vaddf ( buf , fmt , args ) ;
if ( get_common_dir_noenv ( & git_submodule_common_dir , git_submodule_dir . buf ) )
update_common_dir ( buf , git_submodule_dir . len , git_submodule_common_dir . buf ) ;
strbuf_cleanup_path ( buf ) ;
cleanup :
strbuf_release ( & git_submodule_dir ) ;
strbuf_release ( & git_submodule_common_dir ) ;
return ret ;
}
char * git_pathdup_submodule ( const char * path , const char * fmt , ... )
```

Código con los Errores

```
{
int err ;
va_list args ;
struct strbuf buf = STRBUF_INIT ;
va_start ( args , fmt ) ;
err = do_submodule_path ( & buf , path , fmt , args ) ;
va_end ( args ) ;
if ( err ) {
strbuf_release ( & buf ) ;
return NULL ;
}
return strbuf_detach ( & buf , NULL ) ;
}
```

Código con los Errores

```
int strbuf_git_path_submodule ( struct strbuf * buf , const char * path ,
const char * fmt , ... )
{
int err ;
va_list args ;
va_start ( args , fmt ) ;
err = do_submodule_path ( buf , path , fmt , args ) ;
va_end ( args ) ;
return err ;
}
static void do_git_common_path ( struct strbuf * buf ,
const char * fmt ,
va_list args )
```

Código con los Errores

```
{
strbuf_addstr ( buf , get_git_common_dir ( ) ) ;
if ( buf -> len && ! is_dir_sep ( buf -> buf [ buf -> len - 1 ] ) )
strbuf_addch ( buf , '/' ) ;
strbuf_vaddf ( buf , fmt , args ) ;
strbuf_cleanup_path ( buf ) ;
}
const char * git_common_path ( const char * fmt , ... )
{
struct strbuf * pathname = get_pathname ( ) ;
va_list args ;
va_start ( args , fmt ) ;
do_git_common_path ( pathname , fmt , args ) ;
```

Código con los Errores

```
va_end ( args ) ;
return pathname -> buf ;
}
void strbuf_git_common_path ( struct strbuf * sb , const char * fmt , ... )
{
va_list args ;
va_start ( args , fmt ) ;
do_git_common_path ( sb , fmt , args ) ;
va_end ( args ) ;
}
int validate_headref ( const char * path )
{
struct stat st ;
```

Código con los Errores

```
char * buf , buffer [ 256 ] ;
unsigned char sha1 [ 20 ] ;
int fd ;
ssize_t len ;
if ( lstat ( path , & st ) < 0 )
return - 1 ;
if ( S_ISLNK ( st . st_mode ) ) {
len = readlink ( path , buffer , sizeof ( buffer ) - 1 ) ;
if ( len >= 5 && ! memcmp ( "refs/" , buffer , 5 ) )
return 0 ;
return - 1 ;
}
fd = open ( path , O_RDONLY ) ;
```

Código con los Errores

```
if ( fd < 0 )
return - 1 ;
len = read_in_full ( fd , buffer , sizeof ( buffer ) - 1 ) ;
close ( fd ) ;
if ( len < 4 )
return - 1 ;
if ( ! memcmp ( "ref:" , buffer , 4 ) ) {
buf = buffer + 4 ;
len -= 4 ;
while ( len && isspace ( * buf ) )
buf ++ , len -- ;
if ( len >= 5 && ! memcmp ( "refs/" , buf , 5 ) )
return 0 ;
```

Código con los Errores

```
}
if ( ! get_sha1_hex ( buffer , sha1 ) )
return 0 ;
return - 1 ;
}
static struct passwd * getpw_str ( const char * username , size_t len )
{
struct passwd * pw ;
char * username_z = xmemdupz ( username , len ) ;
pw = getpwnam ( username_z ) ;
free ( username_z ) ;
return pw ;
}
```

Código con los Errores

```
char * expand_user_path ( const char * path , int real_home )
{
struct strbuf user_path = STRBUF_INIT ;
const char * to_copy = path ;
if ( path == NULL )
goto return_null ;
if ( path [ 0 ] == '~' ) {
const char * first_slash = strchrnul ( path , '/' ) ;
const char * username = path + 1 ;
size_t username_len = first_slash - username ;
if ( username_len == 0 ) {
const char * home = getenv ( "HOME" ) ;
if ( ! home )
```

Código con los Errores

```
goto return_null ;
if ( real_home )
strbuf_addstr ( & user_path , real_path ( home ) ) ;
else
/*Pruebas/test01.c:472:39 syntax error, unexpected £end*/
```