# 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

```
enum crlf_action {
CRLF_UNDEFINED ,
CRLF_BINARY ,
CRLF_TEXT ,
CRLF_TEXT_INPUT ,
CRLF_TEXT_CRLF ,
CRLF_AUTO ,
CRLF_AUTO_INPUT ,
CRLF_AUTO_CRLF
} ;
struct text_stat {
unsigned nul , lonecr , lonelf , crlf ;
```

# Código

```
unsigned printable , nonprintable ;
} ;
static void gather_stats ( const char * buf , unsigned long size , struct text_stat * stats )
{
unsigned long i ;
memset ( stats , 0 , sizeof ( * stats ) ) ;
for ( i = 0 ; i < size ; i ++ ) {
unsigned char c = buf [ i ] ;
if ( c == '\r' ) {
if ( i + 1 < size && buf [ i + 1 ] == '\n' ) {
stats -> crlf ++ ;
i ++ ;
} else
```

# Código

```
stats -> lonecr ++ ;
continue ;
}
if ( c == '\n' ) {
stats -> lonelf ++ ;
continue ;
}
if ( c == 127 )
stats -> nonprintable ++ ;
else if ( c < 32 ) {
switch ( c ) {
case '\b' : case '\t' : case '\033' : case '\014' :
stats -> printable ++ ;
```

# Código

```
break ;
case 0 :
stats -> nul ++ ;
default :
stats -> nonprintable ++ ;
}
}
else
stats -> printable ++ ;
}
if ( size >= 1 && buf [ size - 1 ] == '\032' )
stats -> nonprintable -- ;
}
```

# Código

```
static int convert_is_binary ( unsigned long size , const struct text_stat * stats )
{
if ( stats -> lonecr )
return 1 ;
if ( stats -> nul )
return 1 ;
if ( ( stats -> printable >> 7 ) < stats -> nonprintable )
return 1 ;
return 0 ;
}
static unsigned int gather_convert_stats ( const char * data , unsigned long size )
{
struct text_stat stats ;
```

# Código

```
int ret = 0 ;
if ( ! data || ! size )
return 0 ;
gather_stats ( data , size , & stats ) ;
if ( convert_is_binary ( size , & stats ) )
ret |= 0x4 ;
if ( stats . crlf )
ret |= 0x2 ;
if ( stats . lonelf )
ret |= CONVERT_STAT_BITS_TXT_LF ;
return ret ;
}
static const char * gather_convert_stats_ascii ( const char * data , unsigned long size )
```

# Código

```
{
unsigned int convert_stats = gather_convert_stats ( data , size ) ;
if ( convert_stats & 0x4 )
return "-text" ;
switch ( convert_stats ) {
case CONVERT_STAT_BITS_TXT_LF :
return "lf" ;
case 0x2 :
return "crlf" ;
case CONVERT_STAT_BITS_TXT_LF | 0x2 :
return "mixed" ;
default :
return "none" ;
```

# Código

```
}
}
const char * get_cached_convert_stats_ascii ( const char * path )
{
const char * ret ;
unsigned long sz ;
void * data = read_blob_data_from_cache ( path , & sz ) ;
ret = gather_convert_stats_ascii ( data , sz ) ;
free ( data ) ;
return ret ;
}
const char * get_wt_convert_stats_ascii ( const char * path )
{
```

# Código

```
const char * ret = "" ;
struct strbuf sb = STRBUF_INIT ;
if ( strbuf_read_file ( & sb , path , 0 ) >= 0 )
ret = gather_convert_stats_ascii ( sb . buf , sb . len ) ;
strbuf_release ( & sb ) ;
return ret ;
}
static int text_eol_is_crlf ( void )
{
if ( auto_crlf == AUTO_CRLF_TRUE )
return 1 ;
else if ( auto_crlf == AUTO_CRLF_INPUT )
return 0 ;
```

# Código

```
if ( core_eol == EOL_CRLF )
return 1 ;
if ( core_eol == EOL_UNSET && EOL_NATIVE == EOL_CRLF )
return 1 ;
return 0 ;
}
static enum eol output_eol ( enum crlf_action crlf_action )
{
switch ( crlf_action ) {
case CRLF_BINARY :
return EOL_UNSET ;
case CRLF_TEXT_CRLF :
return EOL_CRLF ;
```

# Código

```
case CRLF_TEXT_INPUT :
return EOL_LF ;
case CRLF_UNDEFINED :
case CRLF_AUTO_CRLF :
return EOL_CRLF ;
case CRLF_AUTO_INPUT :
return EOL_LF ;
case CRLF_TEXT :
case CRLF_AUTO :
return text_eol_is_crlf ( ) ? EOL_CRLF : EOL_LF ;
}
warning ( "Illegal crlf_action %d\n" , ( int ) crlf_action ) ;
return core_eol ;
```

# Código

```
}
static void check_safe_crlf ( const char * path , enum crlf_action crlf_action ,
struct text_stat * old_stats , struct text_stat * new_stats ,
enum safe_crlf checksafe )
{
if ( old_stats -> crlf && ! new_stats -> crlf ) {
if ( checksafe == SAFE_CRLF_WARN )
warning ( _ ( "CRLF will be replaced by LF in %s.\n"
"The file will have its original line"
" endings in your working directory." ) , path ) ;
else
die ( _ ( "CRLF would be replaced by LF in %s." ) , path ) ;
} else if ( old_stats -> lonelf && ! new_stats -> lonelf ) {
```

# Código

```
if ( checksafe == SAFE_CRLF_WARN )
warning ( _ ( "LF will be replaced by CRLF in %s.\n"
"The file will have its original line"
" endings in your working directory." ) , path ) ;
else
die ( _ ( "LF would be replaced by CRLF in %s" ) , path ) ;
}
}
static int has_cr_in_index ( const char * path )
{
unsigned long sz ;
void * data ;
int has_cr ;
```

# Código

```
data = read_blob_data_from_cache ( path , & sz ) ;
if ( ! data )
return 0 ;
has_cr = memchr ( data , '\r' , sz ) != NULL ;
free ( data ) ;
return has_cr ;
}
static int will_convert_lf_to_crlf ( size_t len , struct text_stat * stats ,
enum crlf_action crlf_action )
{
if ( output_eol ( crlf_action ) != EOL_CRLF )
return 0 ;
if ( ! stats -> lonelf )
```

# Código

```
return 0 ;
if ( crlf_action == CRLF_AUTO || crlf_action == CRLF_AUTO_INPUT || crlf_action == CRLF_AUTO_CRLF ) {
if ( stats -> lonecr || stats -> crlf )
return 0 ;
if ( convert_is_binary ( len , stats ) )
return 0 ;
}
return 1 ;
}
static int crlf_to_git ( const char * path , const char * src , size_t len ,
struct strbuf * buf ,
enum crlf_action crlf_action , enum safe_crlf checksafe )
{
```

# Código

```
struct text_stat stats ;
char * dst ;
int convert_crlf_into_lf ;
if ( crlf_action == CRLF_BINARY ||
( src && ! len ) )
return 0 ;
if ( ! buf && ! src )
return 1 ;
gather_stats ( src , len , & stats ) ;
convert_crlf_into_lf = ! ! stats . crlf ;
if ( crlf_action == CRLF_AUTO || crlf_action == CRLF_AUTO_INPUT || crlf_action == CRLF_AUTO_CRLF ) {
if ( convert_is_binary ( len , & stats ) )
return 0 ;
```

# Código

```
if ( ( checksafe != SAFE_CRLF_RENORMALIZE ) && has_cr_in_index ( path ) )
convert_crlf_into_lf = 0 ;
}
if ( ( checksafe == SAFE_CRLF_WARN ||
( checksafe == SAFE_CRLF_FAIL ) ) && len ) {
struct text_stat new_stats ;
memcpy ( & new_stats , & stats , sizeof ( new_stats ) ) ;
if ( convert_crlf_into_lf ) {
new_stats . lonelf += new_stats . crlf ;
new_stats . crlf = 0 ;
}
if ( will_convert_lf_to_crlf ( len , & new_stats , crlf_action ) ) {
new_stats . crlf += new_stats . lonelf ;
```

# Código

```
new_stats . lonelf = 0 ;
}
check_safe_crlf ( path , crlf_action , & stats , & new_stats , checksafe ) ;
}
if ( ! convert_crlf_into_lf )
return 0 ;
if ( ! buf )
return 1 ;
if ( strbuf_avail ( buf ) + buf -> len < len )
strbuf_grow ( buf , len - buf -> len ) ;
dst = buf -> buf ;
if ( crlf_action == CRLF_AUTO || crlf_action == CRLF_AUTO_INPUT || crlf_action == CRLF_AUTO_CRLF ) {
do {
```

# Código

```
unsigned char c = * src ++ ;
if ( c != '\r' )
* dst ++ = c ;
} while ( -- len ) ;
} else {
do {
unsigned char c = * src ++ ;
if ( ! ( c == '\r' && ( 1 < len && * src == '\n' ) ) )
* dst ++ = c ;
} while ( -- len ) ;
}
strbuf_setlen ( buf , dst - buf -> buf ) ;
return 1 ;
```

# Código

```
}
static int crlf_to_worktree ( const char * path , const char * src , size_t len ,
struct strbuf * buf , enum crlf_action crlf_action )
{
char * to_free = NULL ;
struct text_stat stats ;
if ( ! len || output_eol ( crlf_action ) != EOL_CRLF )
return 0 ;
gather_stats ( src , len , & stats ) ;
if ( ! will_convert_lf_to_crlf ( len , & stats , crlf_action ) )
return 0 ;
if ( src == buf -> buf )
to_free = strbuf_detach ( buf , NULL ) ;
```

# Código

```
strbuf_grow ( buf , len + stats . lonelf ) ;
for ( ; ; ) {
const char * nl = memchr ( src , '\n' , len ) ;
if ( ! nl )
break ;
if ( nl > src && nl [ - 1 ] == '\r' ) {
strbuf_add ( buf , src , nl + 1 - src ) ;
} else {
strbuf_add ( buf , src , nl - src ) ;
strbuf_addstr ( buf , "\r\n" ) ;
}
len -= nl + 1 - src ;
src = nl + 1 ;
```

# Código

```
}
strbuf_add ( buf , src , len ) ;
free ( to_free ) ;
return 1 ;
}
struct filter_params {
const char * src ;
unsigned long size ;
int fd ;
const char * cmd ;
const char * path ;
} ;
static int filter_buffer_or_fd ( int in , int out , void * data )
```

# Código

```
{
struct child_process child_process = CHILD_PROCESS_INIT ;
struct filter_params * params = ( struct filter_params * ) data ;
int write_err , status ;
const char * argv [ ] = { NULL , NULL } ;
struct strbuf cmd = STRBUF_INIT ;
struct strbuf path = STRBUF_INIT ;
struct strbuf_expand_dict_entry dict [ ] = {
{ "f" , NULL , } ,
{ NULL , NULL , } ,
} ;
sq_quote_buf ( & path , params -> path ) ;
dict [ 0 ] . value = path . buf ;
```

# Código

```
strbuf_expand ( & cmd , params -> cmd , strbuf_expand_dict_cb , & dict ) ;
strbuf_release ( & path ) ;
argv [ 0 ] = cmd . buf ;
child_process . argv = argv ;
child_process . use_shell = 1 ;
child_process . in = - 1 ;
child_process . out = out ;
if ( start_command ( & child_process ) )
return error ( "cannot fork to run external filter '%s'" , params -> cmd ) ;
sigchain_push ( SIGPIPE , SIG_IGN ) ;
if ( params -> src ) {
write_err = ( write_in_full ( child_process . in ,
params -> src , params -> size ) < 0 ) ;
```

# Código

```
if ( errno == EPIPE )
write_err = 0 ;
} else {
write_err = copy_fd ( params -> fd , child_process . in ) ;
if ( write_err == COPY_WRITE_ERROR && errno == EPIPE )
write_err = 0 ;
}
if ( close ( child_process . in ) )
write_err = 1 ;
if ( write_err )
error ( "cannot feed the input to external filter '%s'" , params -> cmd ) ;
sigchain_pop ( SIGPIPE ) ;
status = finish_command ( & child_process ) ;
```

# Código

```
if ( status )
error ( "external filter '%s' failed %d" , params -> cmd , status ) ;
strbuf_release ( & cmd ) ;
return ( write_err || status ) ;
}
static int apply_single_file_filter ( const char * path , const char * src , size_t len , int fd ,
struct strbuf * dst , const char * cmd )
{
int err = 0 ;
struct strbuf nbuf = STRBUF_INIT ;
struct async async ;
struct filter_params params ;
memset ( & async , 0 , sizeof ( async ) ) ;
```

# Código

```
async . proc = filter_buffer_or_fd ;
async . data = & params ;
async . out = - 1 ;
params . src = src ;
params . size = len ;
params . fd = fd ;
params . cmd = cmd ;
params . path = path ;
fflush ( NULL ) ;
if ( start_async ( & async ) )
return 0 ;
if ( strbuf_read ( & nbuf , async . out , len ) < 0 ) {
err = error ( "read from external filter '%s' failed" , cmd ) ;
```

# Código

```
}
if ( close ( async . out ) ) {
err = error ( "read from external filter '%s' failed" , cmd ) ;
}
if ( finish_async ( & async ) ) {
err = error ( "external filter '%s' failed" , cmd ) ;
}
if ( ! err ) {
strbuf_swap ( dst , & nbuf ) ;
}
strbuf_release ( & nbuf ) ;
return ! err ;
}
```

# Código

```
struct cmd2process {
struct subprocess_entry subprocess ;
unsigned int supported_capabilities ;
} ;
static int subprocess_map_initialized ;
static struct hashmap subprocess_map ;
static int start_multi_file_filter_fn ( struct subprocess_entry * subprocess )
{
int err ;
struct cmd2process * entry = ( struct cmd2process * ) subprocess ;
struct string_list cap_list = STRING_LIST_INIT_NODUP ;
char * cap_buf ;
const char * cap_name ;
```

# Código

```
struct child_process * process = & subprocess -> process ;
const char * cmd = subprocess -> cmd ;
sigchain_push ( SIGPIPE , SIG_IGN ) ;
err = packet_writel ( process -> in , "git-filter-client" , "version=2" , NULL ) ;
if ( err )
goto done ;
err = strcmp ( packet_read_line ( process -> out , NULL ) , "git-filter-server" ) ;
if ( err ) {
error ( "external filter '%s' does not support filter protocol version 2" , cmd ) ;
goto done ;
}
err = strcmp ( packet_read_line ( process -> out , NULL ) , "version=2" ) ;
if ( err )
```

# Código

```
goto done ;
err = packet_read_line ( process -> out , NULL ) != NULL ;
if ( err )
goto done ;
err = packet_writel ( process -> in , "capability=clean" , "capability=smudge" , NULL ) ;
for ( ; ; ) {
cap_buf = packet_read_line ( process -> out , NULL ) ;
if ( ! cap_buf )
break ;
string_list_split_in_place ( & cap_list , cap_buf , '=' , 1 ) ;
if ( cap_list . nr != 2 || strcmp ( cap_list . items [ 0 ] . string , "capability" ) )
continue ;
cap_name = cap_list . items [ 1 ] . string ;
```

# Código

```
if ( ! strcmp ( cap_name , "clean" ) ) {
entry -> supported_capabilities |= ( 1u << 0 ) ;
} else if ( ! strcmp ( cap_name , "smudge" ) ) {
entry -> supported_capabilities |= ( 1u << 1 ) ;
} else {
warning (
"external filter '%s' requested unsupported filter capability '%s'" ,
cmd , cap_name
) ;
}
string_list_clear ( & cap_list , 0 ) ;
}
done :
```

# Código

```
sigchain_pop ( SIGPIPE ) ;
return err ;
}
static int apply_multi_file_filter ( const char * path , const char * src , size_t len ,
int fd , struct strbuf * dst , const char * cmd ,
const unsigned int wanted_capability )
{
int err ;
struct cmd2process * entry ;
struct child_process * process ;
struct strbuf nbuf = STRBUF_INIT ;
struct strbuf filter_status = STRBUF_INIT ;
const char * filter_type ;
```

# Código

```
if ( ! subprocess_map_initialized ) {
subprocess_map_initialized = 1 ;
hashmap_init ( & subprocess_map , ( hashmap_cmp_fn ) cmd2process_cmp , 0 ) ;
entry = NULL ;
} else {
entry = ( struct cmd2process * ) subprocess_find_entry ( & subprocess_map , cmd ) ;
}
fflush ( NULL ) ;
if ( ! entry ) {
entry = xmalloc ( sizeof ( * entry ) ) ;
entry -> supported_capabilities = 0 ;
if ( subprocess_start ( & subprocess_map , & entry -> subprocess , cmd , start_multi_file_filter_fn ) ) {
free ( entry ) ;
```

# Código

```
return 0 ;
}
}
process = & entry -> subprocess . process ;
if ( ! ( wanted_capability & entry -> supported_capabilities ) )
return 0 ;
if ( ( 1u << 0 ) & wanted_capability )
filter_type = "clean" ;
else if ( ( 1u << 1 ) & wanted_capability )
filter_type = "smudge" ;
else
die ( "unexpected filter type" ) ;
sigchain_push ( SIGPIPE , SIG_IGN ) ;
```

# Código

```
assert ( strlen ( filter_type ) < LARGE_PACKET_DATA_MAX - strlen ( "command=\n" ) ) ;
err = packet_write_fmt_gently ( process -> in , "command=%s\n" , filter_type ) ;
if ( err )
goto done ;
err = strlen ( path ) > LARGE_PACKET_DATA_MAX - strlen ( "pathname=\n" ) ;
if ( err ) {
error ( "path name too long for external filter" ) ;
goto done ;
}
err = packet_write_fmt_gently ( process -> in , "pathname=%s\n" , path ) ;
if ( err )
goto done ;
err = packet_flush_gently ( process -> in ) ;
```

# Código

```
if ( err )
goto done ;
if ( fd >= 0 )
err = write_packetized_from_fd ( fd , process -> in ) ;
else
err = write_packetized_from_buf ( src , len , process -> in ) ;
if ( err )
goto done ;
err = subprocess_read_status ( process -> out , & filter_status ) ;
if ( err )
goto done ;
err = strcmp ( filter_status . buf , "success" ) ;
if ( err )
```

# Código

```
goto done ;
err = read_packetized_to_strbuf ( process -> out , & nbuf ) < 0 ;
if ( err )
goto done ;
err = subprocess_read_status ( process -> out , & filter_status ) ;
if ( err )
goto done ;
err = strcmp ( filter_status . buf , "success" ) ;
done :
sigchain_pop ( SIGPIPE ) ;
if ( err ) {
if ( ! strcmp ( filter_status . buf , "error" ) ) {
} else if ( ! strcmp ( filter_status . buf , "abort" ) ) {
```

# Código

```
entry -> supported_capabilities &= ~ wanted_capability ;
} else {
error ( "external filter '%s' failed" , cmd ) ;
subprocess_stop ( & subprocess_map , & entry -> subprocess ) ;
free ( entry ) ;
}
} else {
strbuf_swap ( dst , & nbuf ) ;
}
strbuf_release ( & nbuf ) ;
return ! err ;
}
static struct convert_driver {
```

# Código

```
const char * name ;
struct convert_driver * next ;
const char * smudge ;
const char * clean ;
const char * process ;
int required ;
} * user_convert , * * user_convert_tail ;
static int apply_filter ( const char * path , const char * src , size_t len ,
int fd , struct strbuf * dst , struct convert_driver * drv ,
const unsigned int wanted_capability )
{
const char * cmd = NULL ;
if ( ! drv )
```

# Código

```
return 0 ;
if ( ! dst )
return 1 ;
if ( ( ( 1u << 0 ) & wanted_capability ) && ! drv -> process && drv -> clean )
cmd = drv -> clean ;
else if ( ( ( 1u << 1 ) & wanted_capability ) && ! drv -> process && drv -> smudge )
cmd = drv -> smudge ;
if ( cmd && * cmd )
return apply_single_file_filter ( path , src , len , fd , dst , cmd ) ;
else if ( drv -> process && * drv -> process )
return apply_multi_file_filter ( path , src , len , fd , dst , drv -> process , wanted_capability ) ;
return 0 ;
}
```

# Código

```
static int read_convert_config ( const char * var , const char * value , void * cb )
{
const char * key , * name ;
int namelen ;
struct convert_driver * drv ;
if ( parse_config_key ( var , "filter" , & name , & namelen , & key ) < 0 || ! name )
return 0 ;
for ( drv = user_convert ; drv ; drv = drv -> next )
if ( ! strncmp ( drv -> name , name , namelen ) && ! drv -> name [ namelen ] )
break ;
if ( ! drv ) {
drv = xcalloc ( 1 , sizeof ( struct convert_driver ) ) ;
drv -> name = xmemdupz ( name , namelen ) ;
```

# Código

```
* user_convert_tail = drv ;
user_convert_tail = & ( drv -> next ) ;
}
if ( ! strcmp ( "smudge" , key ) )
return git_config_string ( & drv -> smudge , var , value ) ;
if ( ! strcmp ( "clean" , key ) )
return git_config_string ( & drv -> clean , var , value ) ;
if ( ! strcmp ( "process" , key ) )
return git_config_string ( & drv -> process , var , value ) ;
if ( ! strcmp ( "required" , key ) ) {
drv -> required = git_config_bool ( var , value ) ;
return 0 ;
}
```

# Código

```
return 0 ;
}
static int count_ident ( const char * cp , unsigned long size )
{
int cnt = 0 ;
char ch ;
while ( size ) {
ch = * cp ++ ;
size -- ;
if ( ch != '$' )
continue ;
if ( size < 3 )
break ;
```

# Código

```
if ( memcmp ( "Id" , cp , 2 ) )
continue ;
ch = cp [ 2 ] ;
cp += 3 ;
size -= 3 ;
if ( ch == '$' )
cnt ++ ;
if ( ch != ':' )
continue ;
while ( size ) {
ch = * cp ++ ;
size -- ;
if ( ch == '$' ) {
```

# Código

```
cnt ++ ;
break ;
}
if ( ch == '\n' )
break ;
}
}
return cnt ;
}
static int ident_to_git ( const char * path , const char * src , size_t len ,
struct strbuf * buf , int ident )
{
char * dst , * dollar ;
```

# Código

```
if ( ! ident || ( src && ! count_ident ( src , len ) ) )
return 0 ;
if ( ! buf )
return 1 ;
if ( strbuf_avail ( buf ) + buf -> len < len )
strbuf_grow ( buf , len - buf -> len ) ;
dst = buf -> buf ;
for ( ; ; ) {
dollar = memchr ( src , '$' , len ) ;
if ( ! dollar )
break ;
memmove ( dst , src , dollar + 1 - src ) ;
dst += dollar + 1 - src ;
```

# Código

```
len -= dollar + 1 - src ;
src = dollar + 1 ;
if ( len > 3 && ! memcmp ( src , "Id:" , 3 ) ) {
dollar = memchr ( src + 3 , '$' , len - 3 ) ;
if ( ! dollar )
break ;
if ( memchr ( src + 3 , '\n' , dollar - src - 3 ) ) {
continue ;
}
memcpy ( dst , "Id$" , 3 ) ;
dst += 3 ;
len -= dollar + 1 - src ;
src = dollar + 1 ;
```

# Código

```
}
}
memmove ( dst , src , len ) ;
strbuf_setlen ( buf , dst + len - buf -> buf ) ;
return 1 ;
}
static int ident_to_worktree ( const char * path , const char * src , size_t len ,
struct strbuf * buf , int ident )
{
unsigned char sha1 [ 20 ] ;
char * to_free = NULL , * dollar , * spc ;
int cnt ;
if ( ! ident )
```

# Código

```
return 0 ;
cnt = count_ident ( src , len ) ;
if ( ! cnt )
return 0 ;
if ( src == buf -> buf )
to_free = strbuf_detach ( buf , NULL ) ;
hash_sha1_file ( src , len , "blob" , sha1 ) ;
strbuf_grow ( buf , len + cnt * 43 ) ;
for ( ; ; ) {
dollar = memchr ( src , '$' , len ) ;
if ( ! dollar )
break ;
strbuf_add ( buf , src , dollar + 1 - src ) ;
```

# Código

```
len -= dollar + 1 - src ;
src = dollar + 1 ;
if ( len < 3 || memcmp ( "Id" , src , 2 ) )
continue ;
if ( src [ 2 ] == '$' ) {
src += 3 ;
len -= 3 ;
} else if ( src [ 2 ] == ':' ) {
dollar = memchr ( src + 3 , '$' , len - 3 ) ;
if ( ! dollar ) {
break ;
}
if ( memchr ( src + 3 , '\n' , dollar - src - 3 ) ) {
```

# Código

```
continue ;
}
spc = memchr ( src + 4 , ' ' , dollar - src - 4 ) ;
if ( spc && spc < dollar - 1 ) {
continue ;
}
len -= dollar + 1 - src ;
src = dollar + 1 ;
} else {
continue ;
}
strbuf_addstr ( buf , "Id: " ) ;
strbuf_add ( buf , sha1_to_hex ( sha1 ) , 40 ) ;
```

# Código

```
strbuf_addstr ( buf , " $" ) ;
}
strbuf_add ( buf , src , len ) ;
free ( to_free ) ;
return 1 ;
}
static enum crlf_action git_path_check_crlf ( struct attr_check_item * check )
{
const char * value = check -> value ;
if ( ATTR_TRUE ( value ) )
return CRLF_TEXT ;
else if ( ATTR_FALSE ( value ) )
return CRLF_BINARY ;
```

# Código

```
else if ( ATTR_UNSET ( value ) )
;
else if ( ! strcmp ( value , "input" ) )
return CRLF_TEXT_INPUT ;
else if ( ! strcmp ( value , "auto" ) )
return CRLF_AUTO ;
return CRLF_UNDEFINED ;
}
static enum eol git_path_check_eol ( struct attr_check_item * check )
{
const char * value = check -> value ;
if ( ATTR_UNSET ( value ) )
;
```

# Código

```
else if ( ! strcmp ( value , "lf" ) )
return EOL_LF ;
else if ( ! strcmp ( value , "crlf" ) )
return EOL_CRLF ;
return EOL_UNSET ;
}
static struct convert_driver * git_path_check_convert ( struct attr_check_item * check )
{
const char * value = check -> value ;
struct convert_driver * drv ;
if ( ATTR_TRUE ( value ) || ATTR_FALSE ( value ) || ATTR_UNSET ( value ) )
return NULL ;
for ( drv = user_convert ; drv ; drv = drv -> next )
```

# Código

```
if ( ! strcmp ( value , drv -> name ) )
return drv ;
return NULL ;
}
static int git_path_check_ident ( struct attr_check_item * check )
{
const char * value = check -> value ;
return ! ! ATTR_TRUE ( value ) ;
}
struct conv_attrs {
struct convert_driver * drv ;
enum crlf_action attr_action ;
enum crlf_action crlf_action ;
```

# Código

```
int ident ;
} ;
static void convert_attrs ( struct conv_attrs * ca , const char * path )
{
static struct attr_check * check ;
if ( ! check ) {
check = attr_check_initl ( "crlf" , "ident" , "filter" ,
"eol" , "text" , NULL ) ;
user_convert_tail = & user_convert ;
git_config ( read_convert_config , NULL ) ;
}
if ( ! git_check_attr ( path , check ) ) {
struct attr_check_item * ccheck = check -> items ;
```

# Código

```
ca -> crlf_action = git_path_check_crlf ( ccheck + 4 ) ;
if ( ca -> crlf_action == CRLF_UNDEFINED )
ca -> crlf_action = git_path_check_crlf ( ccheck + 0 ) ;
ca -> attr_action = ca -> crlf_action ;
ca -> ident = git_path_check_ident ( ccheck + 1 ) ;
ca -> drv = git_path_check_convert ( ccheck + 2 ) ;
if ( ca -> crlf_action != CRLF_BINARY ) {
enum eol eol_attr = git_path_check_eol ( ccheck + 3 ) ;
if ( ca -> crlf_action == CRLF_AUTO && eol_attr == EOL_LF )
ca -> crlf_action = CRLF_AUTO_INPUT ;
else if ( ca -> crlf_action == CRLF_AUTO && eol_attr == EOL_CRLF )
ca -> crlf_action = CRLF_AUTO_CRLF ;
else if ( eol_attr == EOL_LF )
```

# Código

```
ca -> crlf_action = CRLF_TEXT_INPUT ;
else if ( eol_attr == EOL_CRLF )
ca -> crlf_action = CRLF_TEXT_CRLF ;
}
ca -> attr_action = ca -> crlf_action ;
} else {
ca -> drv = NULL ;
ca -> crlf_action = CRLF_UNDEFINED ;
ca -> ident = 0 ;
}
if ( ca -> crlf_action == CRLF_TEXT )
ca -> crlf_action = text_eol_is_crlf ( ) ? CRLF_TEXT_CRLF : CRLF_TEXT_INPUT ;
if ( ca -> crlf_action == CRLF_UNDEFINED && auto_crlf == AUTO_CRLF_FALSE )
```

# Código

```
ca -> crlf_action = CRLF_BINARY ;
if ( ca -> crlf_action == CRLF_UNDEFINED && auto_crlf == AUTO_CRLF_TRUE )
ca -> crlf_action = CRLF_AUTO_CRLF ;
if ( ca -> crlf_action == CRLF_UNDEFINED && auto_crlf == AUTO_CRLF_INPUT )
ca -> crlf_action = CRLF_AUTO_INPUT ;
}
int would_convert_to_git_filter_fd ( const char * path )
{
struct conv_attrs ca ;
convert_attrs ( & ca , path ) ;
if ( ! ca . drv )
return 0 ;
if ( ! ca . drv -> required )
```

# Código

```
return 0 ;
return apply_filter ( path , NULL , 0 , - 1 , NULL , ca . drv , ( 1u << 0 ) ) ;
}
const char * get_convert_attr_ascii ( const char * path )
{
struct conv_attrs ca ;
convert_attrs ( & ca , path ) ;
switch ( ca . attr_action ) {
case CRLF_UNDEFINED :
return "" ;
case CRLF_BINARY :
return "-text" ;
case CRLF_TEXT :
```

# Código

```
return "text" ;
case CRLF_TEXT_INPUT :
return "text eol=lf" ;
case CRLF_TEXT_CRLF :
return "text eol=crlf" ;
case CRLF_AUTO :
return "text=auto" ;
case CRLF_AUTO_CRLF :
return "text=auto eol=crlf" ;
case CRLF_AUTO_INPUT :
return "text=auto eol=lf" ;
}
return "" ;
```

# Código

```
}
int convert_to_git ( const char * path , const char * src , size_t len ,
struct strbuf * dst , enum safe_crlf checksafe )
{
int ret = 0 ;
struct conv_attrs ca ;
convert_attrs ( & ca , path ) ;
ret |= apply_filter ( path , src , len , - 1 , dst , ca . drv , ( 1u << 0 ) ) ;
if ( ! ret && ca . drv && ca . drv -> required )
die ( "%s: clean filter '%s' failed" , path , ca . drv -> name ) ;
if ( ret && dst ) {
src = dst -> buf ;
len = dst -> len ;
```

# Código

```
}
ret |= crlf_to_git ( path , src , len , dst , ca . crlf_action , checksafe ) ;
if ( ret && dst ) {
src = dst -> buf ;
len = dst -> len ;
}
return ret | ident_to_git ( path , src , len , dst , ca . ident ) ;
}
void convert_to_git_filter_fd ( const char * path , int fd , struct strbuf * dst ,
enum safe_crlf checksafe )
{
struct conv_attrs ca ;
convert_attrs ( & ca , path ) ;
```

# Código

```
assert ( ca . drv ) ;
assert ( ca . drv -> clean || ca . drv -> process ) ;
if ( ! apply_filter ( path , NULL , 0 , fd , dst , ca . drv , ( 1u << 0 ) ) )
die ( "%s: clean filter '%s' failed" , path , ca . drv -> name ) ;
crlf_to_git ( path , dst -> buf , dst -> len , dst , ca . crlf_action , checksafe ) ;
ident_to_git ( path , dst -> buf , dst -> len , dst , ca . ident ) ;
}
static int convert_to_working_tree_internal ( const char * path , const char * src ,
size_t len , struct strbuf * dst ,
int normalizing )
{
int ret = 0 , ret_filter = 0 ;
struct conv_attrs ca ;
```

# Código

```
convert_attrs ( & ca , path ) ;
ret |= ident_to_worktree ( path , src , len , dst , ca . ident ) ;
if ( ret ) {
src = dst -> buf ;
len = dst -> len ;
}
if ( ( ca . drv && ( ca . drv -> smudge || ca . drv -> process ) ) || ! normalizing ) {
ret |= crlf_to_worktree ( path , src , len , dst , ca . crlf_action ) ;
if ( ret ) {
src = dst -> buf ;
len = dst -> len ;
}
}
```

# Código

```
ret_filter = apply_filter ( path , src , len , - 1 , dst , ca . drv , ( 1u << 1 ) ) ;
if ( ! ret_filter && ca . drv && ca . drv -> required )
die ( "%s: smudge filter %s failed" , path , ca . drv -> name ) ;
return ret | ret_filter ;
}
int convert_to_working_tree ( const char * path , const char * src , size_t len , struct strbuf * dst )
{
return convert_to_working_tree_internal ( path , src , len , dst , 0 ) ;
}
int renormalize_buffer ( const char * path , const char * src , size_t len , struct strbuf * dst )
{
int ret = convert_to_working_tree_internal ( path , src , len , dst , 1 ) ;
if ( ret ) {
```

# Código

```
src = dst -> buf ;
len = dst -> len ;
}
return ret | convert_to_git ( path , src , len , dst , SAFE_CRLF_RENORMALIZE ) ;
}
typedef int ( * filter_fn ) ( struct stream_filter * ,
const char * input , size_t * isize_p ,
char * output , size_t * osize_p ) ;
typedef void ( * free_fn ) ( struct stream_filter * ) ;
struct stream_filter_vtbl {
filter_fn filter ;
free_fn free ;
} ;
```

# Código

```
struct stream_filter {
struct stream_filter_vtbl * vtbl ;
} ;
static int null_filter_fn ( struct stream_filter * filter ,
const char * input , size_t * isize_p ,
char * output , size_t * osize_p )
{
size_t count ;
if ( ! input )
return 0 ;
count = * isize_p ;
if ( * osize_p < count )
count = * osize_p ;
```

# Código

```
if ( count ) {
memmove ( output , input , count ) ;
* isize_p -= count ;
* osize_p -= count ;
}
return 0 ;
}
static void null_free_fn ( struct stream_filter * filter )
{
;
}
static struct stream_filter_vtbl null_vtbl = {
null_filter_fn ,
```

# Código

```
null_free_fn ,
} ;
static struct stream_filter null_filter_singleton = {
& null_vtbl ,
} ;
int is_null_stream_filter ( struct stream_filter * filter )
{
return filter == & null_filter_singleton ;
}
struct lf_to_crlf_filter {
struct stream_filter filter ;
unsigned has_held : 1 ;
char held ;
```

# Código

```
} ;
static int lf_to_crlf_filter_fn ( struct stream_filter * filter ,
const char * input , size_t * isize_p ,
char * output , size_t * osize_p )
{
size_t count , o = 0 ;
struct lf_to_crlf_filter * lf_to_crlf = ( struct lf_to_crlf_filter * ) filter ;
if ( lf_to_crlf -> has_held && ( lf_to_crlf -> held != '\r' || ! input ) ) {
output [ o ++ ] = lf_to_crlf -> held ;
lf_to_crlf -> has_held = 0 ;
}
if ( ! input ) {
* osize_p -= o ;
```

# Código

```
return 0 ;
}
count = * isize_p ;
if ( count || lf_to_crlf -> has_held ) {
size_t i ;
int was_cr = 0 ;
if ( lf_to_crlf -> has_held ) {
was_cr = 1 ;
lf_to_crlf -> has_held = 0 ;
}
for ( i = 0 ; o < * osize_p && i < count ; i ++ ) {
char ch = input [ i ] ;
if ( ch == '\n' ) {
```

# Código

```
output [ o ++ ] = '\r' ;
} else if ( was_cr ) {
output [ o ++ ] = '\r' ;
}
if ( * osize_p <= o ) {
lf_to_crlf -> has_held = 1 ;
lf_to_crlf -> held = ch ;
continue ;
}
if ( ch == '\r' ) {
was_cr = 1 ;
continue ;
}
```

# Código

```
was_cr = 0 ;
output [ o ++ ] = ch ;
}
* osize_p -= o ;
* isize_p -= i ;
if ( ! lf_to_crlf -> has_held && was_cr ) {
lf_to_crlf -> has_held = 1 ;
lf_to_crlf -> held = '\r' ;
}
}
return 0 ;
}
static void lf_to_crlf_free_fn ( struct stream_filter * filter )
```

# Código

```
{
free ( filter ) ;
}
static struct stream_filter_vtbl lf_to_crlf_vtbl = {
lf_to_crlf_filter_fn ,
lf_to_crlf_free_fn ,
} ;
static struct stream_filter * lf_to_crlf_filter ( void )
{
struct lf_to_crlf_filter * lf_to_crlf = xcalloc ( 1 , sizeof ( * lf_to_crlf ) ) ;
lf_to_crlf -> filter . vtbl = & lf_to_crlf_vtbl ;
return ( struct stream_filter * ) lf_to_crlf ;
}
```

# Código

```
struct cascade_filter {
struct stream_filter filter ;
struct stream_filter * one ;
struct stream_filter * two ;
char buf [ 1024 ] ;
int end , ptr ;
} ;
static int cascade_filter_fn ( struct stream_filter * filter ,
const char * input , size_t * isize_p ,
char * output , size_t * osize_p )
{
struct cascade_filter * cas = ( struct cascade_filter * ) filter ;
size_t filled = 0 ;
```

# Código

```
size_t sz = * osize_p ;
size_t to_feed , remaining ;
while ( filled < sz ) {
remaining = sz - filled ;
if ( cas -> ptr < cas -> end ) {
to_feed = cas -> end - cas -> ptr ;
if ( stream_filter ( cas -> two ,
cas -> buf + cas -> ptr , & to_feed ,
output + filled , & remaining ) )
return - 1 ;
cas -> ptr += ( cas -> end - cas -> ptr ) - to_feed ;
filled = sz - remaining ;
continue ;
```

# Código

```
}
to_feed = input ? * isize_p : 0 ;
if ( input && ! to_feed )
break ;
remaining = sizeof ( cas -> buf ) ;
if ( stream_filter ( cas -> one ,
input , & to_feed ,
cas -> buf , & remaining ) )
return - 1 ;
cas -> end = sizeof ( cas -> buf ) - remaining ;
cas -> ptr = 0 ;
if ( input ) {
size_t fed = * isize_p - to_feed ;
```

# Código

```
* isize_p -= fed ;
input += fed ;
}
if ( input || cas -> end )
continue ;
to_feed = 0 ;
remaining = sz - filled ;
if ( stream_filter ( cas -> two ,
NULL , & to_feed ,
output + filled , & remaining ) )
return - 1 ;
if ( remaining == ( sz - filled ) )
break ;
```

# Código

```
filled = sz - remaining ;
}
* osize_p -= filled ;
return 0 ;
}
static void cascade_free_fn ( struct stream_filter * filter )
{
struct cascade_filter * cas = ( struct cascade_filter * ) filter ;
free_stream_filter ( cas -> one ) ;
free_stream_filter ( cas -> two ) ;
free ( filter ) ;
}
static struct stream_filter_vtbl cascade_vtbl = {
```

# Código

```
cascade_filter_fn ,
cascade_free_fn ,
} ;
static struct stream_filter * cascade_filter ( struct stream_filter * one ,
struct stream_filter * two )
{
struct cascade_filter * cascade ;
if ( ! one || is_null_stream_filter ( one ) )
return two ;
if ( ! two || is_null_stream_filter ( two ) )
return one ;
cascade = xmalloc ( sizeof ( * cascade ) ) ;
cascade -> one = one ;
```

# Código

```
cascade -> two = two ;
cascade -> end = cascade -> ptr = 0 ;
cascade -> filter . vtbl = & cascade_vtbl ;
return ( struct stream_filter * ) cascade ;
}
struct ident_filter {
struct stream_filter filter ;
struct strbuf left ;
int state ;
char ident [ 45 ] ;
} ;
static int is_foreign_ident ( const char * str )
{
```

# Código

```
int i ;
if ( ! skip_prefix ( str , "$Id: " , & str ) )
return 0 ;
for ( i = 0 ; str [ i ] ; i ++ ) {
if ( isspace ( str [ i ] ) && str [ i + 1 ] != '$' )
return 1 ;
}
return 0 ;
}
static void ident_drain ( struct ident_filter * ident , char * * output_p , size_t * osize_p )
{
size_t to_drain = ident -> left . len ;
if ( * osize_p < to_drain )
```

# Código

```
to_drain = * osize_p ;
if ( to_drain ) {
memcpy ( * output_p , ident -> left . buf , to_drain ) ;
strbuf_remove ( & ident -> left , 0 , to_drain ) ;
* output_p += to_drain ;
* osize_p -= to_drain ;
}
if ( ! ident -> left . len )
ident -> state = 0 ;
}
static int ident_filter_fn ( struct stream_filter * filter ,
const char * input , size_t * isize_p ,
char * output , size_t * osize_p )
```

# Código

```
{
struct ident_filter * ident = ( struct ident_filter * ) filter ;
static const char head [ ] = "$Id" ;
if ( ! input ) {
switch ( ident -> state ) {
default :
strbuf_add ( & ident -> left , head , ident -> state ) ;
case ( - 2 ) :
case ( - 1 ) :
ident_drain ( ident , & output , osize_p ) ;
}
return 0 ;
}
```

# Código

```
while ( * isize_p || ( ident -> state == ( - 1 ) ) ) {
int ch ;
if ( ident -> state == ( - 1 ) ) {
ident_drain ( ident , & output , osize_p ) ;
if ( ! * osize_p )
break ;
continue ;
}
ch = * ( input ++ ) ;
( * isize_p ) -- ;
if ( ident -> state == ( - 2 ) ) {
strbuf_addch ( & ident -> left , ch ) ;
if ( ch != '\n' && ch != '$' )
```

# Código

```
continue ;
if ( ch == '$' && ! is_foreign_ident ( ident -> left . buf ) ) {
strbuf_setlen ( & ident -> left , sizeof ( head ) - 1 ) ;
strbuf_addstr ( & ident -> left , ident -> ident ) ;
}
ident -> state = ( - 1 ) ;
continue ;
}
if ( ident -> state < sizeof ( head ) &&
head [ ident -> state ] == ch ) {
ident -> state ++ ;
continue ;
}
```

# Código

```
if ( ident -> state )
strbuf_add ( & ident -> left , head , ident -> state ) ;
if ( ident -> state == sizeof ( head ) - 1 ) {
if ( ch != ':' && ch != '$' ) {
strbuf_addch ( & ident -> left , ch ) ;
ident -> state = 0 ;
continue ;
}
if ( ch == ':' ) {
strbuf_addch ( & ident -> left , ch ) ;
ident -> state = ( - 2 ) ;
} else {
strbuf_addstr ( & ident -> left , ident -> ident ) ;
```

# Código

```
ident -> state = ( - 1 ) ;
}
continue ;
}
strbuf_addch ( & ident -> left , ch ) ;
ident -> state = ( - 1 ) ;
}
return 0 ;
}
static void ident_free_fn ( struct stream_filter * filter )
{
struct ident_filter * ident = ( struct ident_filter * ) filter ;
strbuf_release ( & ident -> left ) ;
```

# Código

```
free ( filter ) ;
}
static struct stream_filter_vtbl ident_vtbl = {
ident_filter_fn ,
ident_free_fn ,
} ;
static struct stream_filter * ident_filter ( const unsigned char * sha1 )
{
struct ident_filter * ident = xmalloc ( sizeof ( * ident ) ) ;
xsnprintf ( ident -> ident , sizeof ( ident -> ident ) ,
": %s $" , sha1_to_hex ( sha1 ) ) ;
strbuf_init ( & ident -> left , 0 ) ;
ident -> filter . vtbl = & ident_vtbl ;
```

# Código

```
ident -> state = 0 ;
return ( struct stream_filter * ) ident ;
}
struct stream_filter * get_stream_filter ( const char * path , const unsigned char * sha1 )
{
struct conv_attrs ca ;
struct stream_filter * filter = NULL ;
convert_attrs ( & ca , path ) ;
if ( ca . drv && ( ca . drv -> process || ca . drv -> smudge || ca . drv -> clean ) )
return NULL ;
if ( ca . crlf_action == CRLF_AUTO || ca . crlf_action == CRLF_AUTO_CRLF )
return NULL ;
if ( ca . ident )
```

# Código

```
filter = ident_filter ( sha1 ) ;
if ( output_eol ( ca . crlf_action ) == EOL_CRLF )
filter = cascade_filter ( filter , lf_to_crlf_filter ( ) ) ;
else
filter = cascade_filter ( filter , & null_filter_singleton ) ;
return filter ;
}
void free_stream_filter ( struct stream_filter * filter )
{
filter -> vtbl -> free ( filter ) ;
}
int stream_filter ( struct stream_filter * filter ,
const char * input , size_t * isize_p ,
```

# Código

```
char * output , size_t * osize_p )
{
return filter -> vtbl -> filter ( filter , input , isize_p , output , osize_p ) ;
```