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 void graph_padding_line ( struct git_graph * graph , struct strbuf * sb ) ;
static void graph_show_strbuf ( struct git_graph * graph ,
FILE * file ,
struct strbuf const * sb ) ;
struct column {
struct commit * commit ;
unsigned short color ;
} ;
enum graph_state {
GRAPH_PADDING ,
GRAPH_SKIP ,
GRAPH_PRE_COMMIT ,
```

Código

```
GRAPH_COMMIT ,
GRAPH_POST_MERGE ,
GRAPH_COLLAPSING
} ;
static void graph_show_line_prefix ( const struct diff_options * diffopt )
{
if ( ! diffopt || ! diffopt -> line_prefix )
return ;
fwrite ( diffopt -> line_prefix ,
sizeof ( char ) ,
diffopt -> line_prefix_length ,
diffopt -> file ) ;
}
```

Código

```
static const char * * column_colors ;
static unsigned short column_colors_max ;
static void parse_graph_colors_config ( struct argv_array * colors , const char * string )
{
const char * end , * start ;
start = string ;
end = string + strlen ( string ) ;
while ( start < end ) {
const char * comma = strchrnul ( start , ',' ) ;
char color [ COLOR_MAXLEN ] ;
if ( ! color_parse_mem ( start , comma - start , color ) )
argv_array_push ( colors , color ) ;
else
```

Código

```
warning ( _ ( "ignore invalid color '%.*s' in log.graphColors" ) ,
( int ) ( comma - start ) , start ) ;
start = comma + 1 ;
}
argv_array_push ( colors , GIT_COLOR_RESET ) ;
}
void graph_set_column_colors ( const char * * colors , unsigned short colors_max )
{
column_colors = colors ;
column_colors_max = colors_max ;
}
static const char * column_get_color_code ( unsigned short color )
{
```

Código

```
return column_colors [ color ] ;
}
static void strbuf_write_column ( struct strbuf * sb , const struct column * c ,
char col_char )
{
if ( c -> color < column_colors_max )
strbuf_addstr ( sb , column_get_color_code ( c -> color ) ) ;
strbuf_addch ( sb , col_char ) ;
if ( c -> color < column_colors_max )
strbuf_addstr ( sb , column_get_color_code ( column_colors_max ) ) ;
}
struct git_graph {
struct commit * commit ;
```

Código

```
struct rev_info * revs ;
int num_parents ;
int width ;
int expansion_row ;
enum graph_state state ;
enum graph_state prev_state ;
int commit_index ;
int prev_commit_index ;
int column_capacity ;
int num_columns ;
int num_new_columns ;
int mapping_size ;
struct column * columns ;
```

Código

```
struct column * new_columns ;
int * mapping ;
int * new_mapping ;
unsigned short default_column_color ;
} ;
static struct strbuf * diff_output_prefix_callback ( struct diff_options * opt , void * data )
{
struct git_graph * graph = data ;
static struct strbuf msgbuf = STRBUF_INIT ;
assert ( opt ) ;
strbuf_reset ( & msgbuf ) ;
if ( opt -> line_prefix )
strbuf_add ( & msgbuf , opt -> line_prefix ,
```

Código

```
opt -> line_prefix_length ) ;
if ( graph )
graph_padding_line ( graph , & msgbuf ) ;
return & msgbuf ;
}
static const struct diff_options * default_diffopt ;
void graph_setup_line_prefix ( struct diff_options * diffopt )
{
default_diffopt = diffopt ;
if ( diffopt && ! diffopt -> output_prefix )
diffopt -> output_prefix = diff_output_prefix_callback ;
}
struct git_graph * graph_init ( struct rev_info * opt )
```

Código

```
{
struct git_graph * graph = xmalloc ( sizeof ( struct git_graph ) ) ;
if ( ! column_colors ) {
char * string ;
if ( git_config_get_string ( "log.graphcolors" , & string ) ) {
graph_set_column_colors ( column_colors_ansi ,
column_colors_ansi_max ) ;
} else {
static struct argv_array custom_colors = ARGV_ARRAY_INIT ;
argv_array_clear ( & custom_colors ) ;
parse_graph_colors_config ( & custom_colors , string ) ;
free ( string ) ;
graph_set_column_colors ( custom_colors . argv ,
```

Código

```
custom_colors . argc - 1 ) ;
}
}
graph -> commit = NULL ;
graph -> revs = opt ;
graph -> num_parents = 0 ;
graph -> expansion_row = 0 ;
graph -> state = GRAPH_PADDING ;
graph -> prev_state = GRAPH_PADDING ;
graph -> commit_index = 0 ;
graph -> prev_commit_index = 0 ;
graph -> num_columns = 0 ;
graph -> num_new_columns = 0 ;
```

Código

```
graph -> mapping_size = 0 ;
graph -> default_column_color = column_colors_max - 1 ;
graph -> column_capacity = 30 ;
ALLOC_ARRAY ( graph -> columns , graph -> column_capacity ) ;
ALLOC_ARRAY ( graph -> new_columns , graph -> column_capacity ) ;
ALLOC_ARRAY ( graph -> mapping , 2 * graph -> column_capacity ) ;
ALLOC_ARRAY ( graph -> new_mapping , 2 * graph -> column_capacity ) ;
opt -> diffopt . output_prefix = diff_output_prefix_callback ;
opt -> diffopt . output_prefix_data = graph ;
return graph ;
}
static void graph_update_state ( struct git_graph * graph , enum graph_state s )
{
```

Código

```
graph -> prev_state = graph -> state ;
graph -> state = s ;
}
static void graph_ensure_capacity ( struct git_graph * graph , int num_columns )
{
if ( graph -> column_capacity >= num_columns )
return ;
do {
graph -> column_capacity *= 2 ;
} while ( graph -> column_capacity < num_columns ) ;
REALLOC_ARRAY ( graph -> columns , graph -> column_capacity ) ;
REALLOC_ARRAY ( graph -> new_columns , graph -> column_capacity ) ;
REALLOC_ARRAY ( graph -> mapping , graph -> column_capacity * 2 ) ;
```

Código

```
REALLOC_ARRAY ( graph -> new_mapping , graph -> column_capacity * 2 ) ;
}
static int graph_is_interesting ( struct git_graph * graph , struct commit * commit )
{
if ( graph -> revs && graph -> revs -> boundary ) {
if ( commit -> object . flags & CHILD_SHOWN )
return 1 ;
}
return get_commit_action ( graph -> revs , commit ) == commit_show ;
}
static struct commit_list * next_interesting_parent ( struct git_graph * graph ,
struct commit_list * orig )
{
```

Código

```
struct commit_list * list ;
if ( graph -> revs -> first_parent_only )
return NULL ;
for ( list = orig -> next ; list ; list = list -> next ) {
if ( graph_is_interesting ( graph , list -> item ) )
return list ;
}
return NULL ;
}
static struct commit_list * first_interesting_parent ( struct git_graph * graph )
{
struct commit_list * parents = graph -> commit -> parents ;
if ( ! parents )
```

Código

```
return NULL ;
if ( graph_is_interesting ( graph , parents -> item ) )
return parents ;
return next_interesting_parent ( graph , parents ) ;
}
static unsigned short graph_get_current_column_color ( const struct git_graph * graph )
{
if ( ! want_color ( graph -> revs -> diffopt . use_color ) )
return column_colors_max ;
return graph -> default_column_color ;
}
static void graph_increment_column_color ( struct git_graph * graph )
{
```

Código

```
graph -> default_column_color = ( graph -> default_column_color + 1 ) %
column_colors_max ;
}
static unsigned short graph_find_commit_color ( const struct git_graph * graph ,
const struct commit * commit )
{
int i ;
for ( i = 0 ; i < graph -> num_columns ; i ++ ) {
if ( graph -> columns [ i ] . commit == commit )
return graph -> columns [ i ] . color ;
}
return graph_get_current_column_color ( graph ) ;
}
```

Código

```
static void graph_insert_into_new_columns ( struct git_graph * graph ,
struct commit * commit ,
int * mapping_index )
{
int i ;
for ( i = 0 ; i < graph -> num_new_columns ; i ++ ) {
if ( graph -> new_columns [ i ] . commit == commit ) {
graph -> mapping [ * mapping_index ] = i ;
* mapping_index += 2 ;
return ;
}
}
graph -> new_columns [ graph -> num_new_columns ] . commit = commit ;
```

Código

```
graph -> new_columns [ graph -> num_new_columns ] . color = graph_find_commit_color ( graph , commit ) ;
graph -> mapping [ * mapping_index ] = graph -> num_new_columns ;
* mapping_index += 2 ;
graph -> num_new_columns ++ ;
}
static void graph_update_width ( struct git_graph * graph ,
int is_commit_in_existing_columns )
{
int max_cols = graph -> num_columns + graph -> num_parents ;
if ( graph -> num_parents < 1 )
max_cols ++ ;
if ( is_commit_in_existing_columns )
max_cols -- ;
```

Código

```
graph -> width = max_cols * 2 ;
}
static void graph_update_columns ( struct git_graph * graph )
{
struct commit_list * parent ;
int max_new_columns ;
int mapping_idx ;
int i , seen_this , is_commit_in_columns ;
SWAP ( graph -> columns , graph -> new_columns ) ;
graph -> num_columns = graph -> num_new_columns ;
graph -> num_new_columns = 0 ;
max_new_columns = graph -> num_columns + graph -> num_parents ;
graph_ensure_capacity ( graph , max_new_columns ) ;
```

Código

```
graph -> mapping_size = 2 * max_new_columns ;
for ( i = 0 ; i < graph -> mapping_size ; i ++ )
graph -> mapping [ i ] = - 1 ;
seen_this = 0 ;
mapping_idx = 0 ;
is_commit_in_columns = 1 ;
for ( i = 0 ; i <= graph -> num_columns ; i ++ ) {
struct commit * col_commit ;
if ( i == graph -> num_columns ) {
if ( seen_this )
break ;
is_commit_in_columns = 0 ;
col_commit = graph -> commit ;
```

Código

```
} else {
col_commit = graph -> columns [ i ] . commit ;
}
if ( col_commit == graph -> commit ) {
int old_mapping_idx = mapping_idx ;
seen_this = 1 ;
graph -> commit_index = i ;
for ( parent = first_interesting_parent ( graph ) ;
parent ;
parent = next_interesting_parent ( graph , parent ) ) {
if ( graph -> num_parents > 1 ||
! is_commit_in_columns ) {
graph_increment_column_color ( graph ) ;
```

Código

```
}
graph_insert_into_new_columns ( graph ,
parent -> item ,
& mapping_idx ) ;
}
if ( mapping_idx == old_mapping_idx )
mapping_idx += 2 ;
} else {
graph_insert_into_new_columns ( graph , col_commit ,
& mapping_idx ) ;
}
}
while ( graph -> mapping_size > 1 &&
```

Código

```
graph -> mapping [ graph -> mapping_size - 1 ] < 0 )
graph -> mapping_size -- ;
graph_update_width ( graph , is_commit_in_columns ) ;
}
void graph_update ( struct git_graph * graph , struct commit * commit )
{
struct commit_list * parent ;
graph -> commit = commit ;
graph -> num_parents = 0 ;
for ( parent = first_interesting_parent ( graph ) ;
parent ;
parent = next_interesting_parent ( graph , parent ) )
{
```

Código

```
graph -> num_parents ++ ;
}
graph -> prev_commit_index = graph -> commit_index ;
graph_update_columns ( graph ) ;
graph -> expansion_row = 0 ;
if ( graph -> state != GRAPH_PADDING )
graph -> state = GRAPH_SKIP ;
else if ( graph -> num_parents >= 3 &&
graph -> commit_index < ( graph -> num_columns - 1 ) )
graph -> state = GRAPH_PRE_COMMIT ;
else
graph -> state = GRAPH_COMMIT ;
}
```

Código

```
static int graph_is_mapping_correct ( struct git_graph * graph )
{
int i ;
for ( i = 0 ; i < graph -> mapping_size ; i ++ ) {
int target = graph -> mapping [ i ] ;
if ( target < 0 )
continue ;
if ( target == ( i / 2 ) )
continue ;
return 0 ;
}
return 1 ;
}
```

Código

```
static void graph_pad_horizontally ( struct git_graph * graph , struct strbuf * sb ,
int chars_written )
{
int extra ;
if ( chars_written >= graph -> width )
return ;
extra = graph -> width - chars_written ;
strbuf_addf ( sb , "%*s" , ( int ) extra , "" ) ;
}
static void graph_output_padding_line ( struct git_graph * graph ,
struct strbuf * sb )
{
int i ;
```

Código

```
if ( ! graph -> commit )
return ;
for ( i = 0 ; i < graph -> num_new_columns ; i ++ ) {
strbuf_write_column ( sb , & graph -> new_columns [ i ] , '|' ) ;
strbuf_addch ( sb , ' ' ) ;
}
graph_pad_horizontally ( graph , sb , graph -> num_new_columns * 2 ) ;
}
int graph_width ( struct git_graph * graph )
{
return graph -> width ;
}
static void graph_output_skip_line ( struct git_graph * graph , struct strbuf * sb )
```

Código

```
{
strbuf_addstr ( sb , "..." ) ;
graph_pad_horizontally ( graph , sb , 3 ) ;
if ( graph -> num_parents >= 3 &&
graph -> commit_index < ( graph -> num_columns - 1 ) )
graph_update_state ( graph , GRAPH_PRE_COMMIT ) ;
else
graph_update_state ( graph , GRAPH_COMMIT ) ;
}
static void graph_output_pre_commit_line ( struct git_graph * graph ,
struct strbuf * sb )
{
int num_expansion_rows ;
```

Código

```
int i , seen_this ;
int chars_written ;
assert ( graph -> num_parents >= 3 ) ;
num_expansion_rows = ( graph -> num_parents - 2 ) * 2 ;
assert ( 0 <= graph -> expansion_row &&
graph -> expansion_row < num_expansion_rows ) ;
seen_this = 0 ;
chars_written = 0 ;
for ( i = 0 ; i < graph -> num_columns ; i ++ ) {
struct column * col = & graph -> columns [ i ] ;
if ( col -> commit == graph -> commit ) {
seen_this = 1 ;
strbuf_write_column ( sb , col , '|' ) ;
```

Código

```
strbuf_addf ( sb , "%*s" , graph -> expansion_row , "" ) ;
chars_written += 1 + graph -> expansion_row ;
} else if ( seen_this && ( graph -> expansion_row == 0 ) ) {
if ( graph -> prev_state == GRAPH_POST_MERGE &&
graph -> prev_commit_index < i )
strbuf_write_column ( sb , col , '\\' ) ;
else
strbuf_write_column ( sb , col , '|' ) ;
chars_written ++ ;
} else if ( seen_this && ( graph -> expansion_row > 0 ) ) {
strbuf_write_column ( sb , col , '\\' ) ;
chars_written ++ ;
} else {
```

Código

```
strbuf_write_column ( sb , col , '|' ) ;
chars_written ++ ;
}
strbuf_addch ( sb , ' ' ) ;
chars_written ++ ;
}
graph_pad_horizontally ( graph , sb , chars_written ) ;
graph -> expansion_row ++ ;
if ( graph -> expansion_row >= num_expansion_rows )
graph_update_state ( graph , GRAPH_COMMIT ) ;
}
static void graph_output_commit_char ( struct git_graph * graph , struct strbuf * sb )
{
```

Código

```
if ( graph -> commit -> object . flags & BOUNDARY ) {
assert ( graph -> revs -> boundary ) ;
strbuf_addch ( sb , 'o' ) ;
return ;
}
strbuf_addstr ( sb , get_revision_mark ( graph -> revs , graph -> commit ) ) ;
}
static int graph_draw_octopus_merge ( struct git_graph * graph ,
struct strbuf * sb )
{
const int dashless_commits = 2 ;
int col_num , i ;
int num_dashes =
```

Código

```
( ( graph -> num_parents - dashless_commits ) * 2 ) - 1 ;
for ( i = 0 ; i < num_dashes ; i ++ ) {
col_num = ( i / 2 ) + dashless_commits + graph -> commit_index ;
strbuf_write_column ( sb , & graph -> new_columns [ col_num ] , '-' ) ;
}
col_num = ( i / 2 ) + dashless_commits + graph -> commit_index ;
strbuf_write_column ( sb , & graph -> new_columns [ col_num ] , '.' ) ;
return num_dashes + 1 ;
}
static void graph_output_commit_line ( struct git_graph * graph , struct strbuf * sb )
{
int seen_this = 0 ;
int i , chars_written ;
```

Código

```
seen_this = 0 ;
chars_written = 0 ;
for ( i = 0 ; i <= graph -> num_columns ; i ++ ) {
struct column * col = & graph -> columns [ i ] ;
struct commit * col_commit ;
if ( i == graph -> num_columns ) {
if ( seen_this )
break ;
col_commit = graph -> commit ;
} else {
col_commit = graph -> columns [ i ] . commit ;
}
if ( col_commit == graph -> commit ) {
```

Código

```
seen_this = 1 ;
graph_output_commit_char ( graph , sb ) ;
chars_written ++ ;
if ( graph -> num_parents > 2 )
chars_written += graph_draw_octopus_merge ( graph ,
sb ) ;
} else if ( seen_this && ( graph -> num_parents > 2 ) ) {
strbuf_write_column ( sb , col , '\\' ) ;
chars_written ++ ;
} else if ( seen_this && ( graph -> num_parents == 2 ) ) {
if ( graph -> prev_state == GRAPH_POST_MERGE &&
graph -> prev_commit_index < i )
strbuf_write_column ( sb , col , '\\' ) ;
```

Código

```
else
strbuf_write_column ( sb , col , '|' ) ;
chars_written ++ ;
} else {
strbuf_write_column ( sb , col , '|' ) ;
chars_written ++ ;
}
strbuf_addch ( sb , ' ' ) ;
chars_written ++ ;
}
graph_pad_horizontally ( graph , sb , chars_written ) ;
if ( graph -> num_parents > 1 )
graph_update_state ( graph , GRAPH_POST_MERGE ) ;
```

Código

```
else if ( graph_is_mapping_correct ( graph ) )
graph_update_state ( graph , GRAPH_PADDING ) ;
else
graph_update_state ( graph , GRAPH_COLLAPSING ) ;
}
static struct column * find_new_column_by_commit ( struct git_graph * graph ,
struct commit * commit )
{
int i ;
for ( i = 0 ; i < graph -> num_new_columns ; i ++ ) {
if ( graph -> new_columns [ i ] . commit == commit )
return & graph -> new_columns [ i ] ;
}
```

Código

```
return NULL ;
}
static void graph_output_post_merge_line ( struct git_graph * graph , struct strbuf * sb )
{
int seen_this = 0 ;
int i , j , chars_written ;
chars_written = 0 ;
for ( i = 0 ; i <= graph -> num_columns ; i ++ ) {
struct column * col = & graph -> columns [ i ] ;
struct commit * col_commit ;
if ( i == graph -> num_columns ) {
if ( seen_this )
break ;
```

Código

```
col_commit = graph -> commit ;
} else {
col_commit = col -> commit ;
}
if ( col_commit == graph -> commit ) {
struct commit_list * parents = NULL ;
struct column * par_column ;
seen_this = 1 ;
parents = first_interesting_parent ( graph ) ;
assert ( parents ) ;
par_column = find_new_column_by_commit ( graph , parents -> item ) ;
assert ( par_column ) ;
strbuf_write_column ( sb , par_column , '|' ) ;
```

Código

```
chars_written ++ ;
for ( j = 0 ; j < graph -> num_parents - 1 ; j ++ ) {
parents = next_interesting_parent ( graph , parents ) ;
assert ( parents ) ;
par_column = find_new_column_by_commit ( graph , parents -> item ) ;
assert ( par_column ) ;
strbuf_write_column ( sb , par_column , '\\' ) ;
strbuf_addch ( sb , ' ' ) ;
}
chars_written += j * 2 ;
} else if ( seen_this ) {
strbuf_write_column ( sb , col , '\\' ) ;
strbuf_addch ( sb , ' ' ) ;
```

Código

```
chars_written += 2 ;
} else {
strbuf_write_column ( sb , col , '|' ) ;
strbuf_addch ( sb , ' ' ) ;
chars_written += 2 ;
}
}
graph_pad_horizontally ( graph , sb , chars_written ) ;
if ( graph_is_mapping_correct ( graph ) )
graph_update_state ( graph , GRAPH_PADDING ) ;
else
graph_update_state ( graph , GRAPH_COLLAPSING ) ;
}
```

Código

```
static void graph_output_collapsing_line ( struct git_graph * graph , struct strbuf * sb )
{
int i ;
short used_horizontal = 0 ;
int horizontal_edge = - 1 ;
int horizontal_edge_target = - 1 ;
for ( i = 0 ; i < graph -> mapping_size ; i ++ )
graph -> new_mapping [ i ] = - 1 ;
for ( i = 0 ; i < graph -> mapping_size ; i ++ ) {
int target = graph -> mapping [ i ] ;
if ( target < 0 )
continue ;
assert ( target * 2 <= i ) ;
```

Código

```
if ( target * 2 == i ) {
assert ( graph -> new_mapping [ i ] == - 1 ) ;
graph -> new_mapping [ i ] = target ;
} else if ( graph -> new_mapping [ i - 1 ] < 0 ) {
graph -> new_mapping [ i - 1 ] = target ;
if ( horizontal_edge == - 1 ) {
int j ;
horizontal_edge = i ;
horizontal_edge_target = target ;
for ( j = ( target * 2 ) + 3 ; j < ( i - 2 ) ; j += 2 )
graph -> new_mapping [ j ] = target ;
}
} else if ( graph -> new_mapping [ i - 1 ] == target ) {
```

Código

```
} else {
assert ( graph -> new_mapping [ i - 1 ] > target ) ;
assert ( graph -> new_mapping [ i - 2 ] < 0 ) ;
assert ( graph -> new_mapping [ i - 3 ] == target ) ;
graph -> new_mapping [ i - 2 ] = target ;
if ( horizontal_edge == - 1 )
horizontal_edge = i ;
}
}
if ( graph -> new_mapping [ graph -> mapping_size - 1 ] < 0 )
graph -> mapping_size -- ;
for ( i = 0 ; i < graph -> mapping_size ; i ++ ) {
int target = graph -> new_mapping [ i ] ;
```

Código

```
if ( target < 0 )
strbuf_addch ( sb , ' ' ) ;
else if ( target * 2 == i )
strbuf_write_column ( sb , & graph -> new_columns [ target ] , '|' ) ;
else if ( target == horizontal_edge_target &&
i != horizontal_edge - 1 ) {
if ( i != ( target * 2 ) + 3 )
graph -> new_mapping [ i ] = - 1 ;
used_horizontal = 1 ;
strbuf_write_column ( sb , & graph -> new_columns [ target ] , '_' ) ;
} else {
if ( used_horizontal && i < horizontal_edge )
graph -> new_mapping [ i ] = - 1 ;
```

Código

```
strbuf_write_column ( sb , & graph -> new_columns [ target ] , '/' ) ;
}
}
graph_pad_horizontally ( graph , sb , graph -> mapping_size ) ;
SWAP ( graph -> mapping , graph -> new_mapping ) ;
if ( graph_is_mapping_correct ( graph ) )
graph_update_state ( graph , GRAPH_PADDING ) ;
}
int graph_next_line ( struct git_graph * graph , struct strbuf * sb )
{
switch ( graph -> state ) {
case GRAPH_PADDING :
graph_output_padding_line ( graph , sb ) ;
```

Código

```
return 0 ;
case GRAPH_SKIP :
graph_output_skip_line ( graph , sb ) ;
return 0 ;
case GRAPH_PRE_COMMIT :
graph_output_pre_commit_line ( graph , sb ) ;
return 0 ;
case GRAPH_COMMIT :
graph_output_commit_line ( graph , sb ) ;
return 1 ;
case GRAPH_POST_MERGE :
graph_output_post_merge_line ( graph , sb ) ;
return 0 ;
```

Código

```
case GRAPH_COLLAPSING :
graph_output_collapsing_line ( graph , sb ) ;
return 0 ;
}
assert ( 0 ) ;
return 0 ;
}
static void graph_padding_line ( struct git_graph * graph , struct strbuf * sb )
{
int i ;
int chars_written = 0 ;
if ( graph -> state != GRAPH_COMMIT ) {
graph_next_line ( graph , sb ) ;
```

Código

```
return ;
}
for ( i = 0 ; i < graph -> num_columns ; i ++ ) {
struct column * col = & graph -> columns [ i ] ;
strbuf_write_column ( sb , col , '|' ) ;
chars_written ++ ;
if ( col -> commit == graph -> commit && graph -> num_parents > 2 ) {
int len = ( graph -> num_parents - 2 ) * 2 ;
strbuf_addchars ( sb , ' ' , len ) ;
chars_written += len ;
} else {
strbuf_addch ( sb , ' ' ) ;
chars_written ++ ;
```

Código

```
}
}
graph_pad_horizontally ( graph , sb , chars_written ) ;
graph -> prev_state = GRAPH_PADDING ;
}
int graph_is_commit_finished ( struct git_graph const * graph )
{
return ( graph -> state == GRAPH_PADDING ) ;
}
void graph_show_commit ( struct git_graph * graph )
{
struct strbuf msgbuf = STRBUF_INIT ;
int shown_commit_line = 0 ;
```

Código

```
graph_show_line_prefix ( default_diffopt ) ;
if ( ! graph )
return ;
if ( graph_is_commit_finished ( graph ) ) {
graph_show_padding ( graph ) ;
shown_commit_line = 1 ;
}
while ( ! shown_commit_line && ! graph_is_commit_finished ( graph ) ) {
shown_commit_line = graph_next_line ( graph , & msgbuf ) ;
fwrite ( msgbuf . buf , sizeof ( char ) , msgbuf . len ,
graph -> revs -> diffopt . file ) ;
if ( ! shown_commit_line ) {
putc ( '\n' , graph -> revs -> diffopt . file ) ;
```

Código

```
graph_show_line_prefix ( & graph -> revs -> diffopt ) ;
}
strbuf_setlen ( & msgbuf , 0 ) ;
}
strbuf_release ( & msgbuf ) ;
}
void graph_show_oneline ( struct git_graph * graph )
{
struct strbuf msgbuf = STRBUF_INIT ;
graph_show_line_prefix ( default_diffopt ) ;
if ( ! graph )
return ;
graph_next_line ( graph , & msgbuf ) ;
```

Código

```
fwrite ( msgbuf . buf , sizeof ( char ) , msgbuf . len , graph -> revs -> diffopt . file ) ;
strbuf_release ( & msgbuf ) ;
}
void graph_show_padding ( struct git_graph * graph )
{
struct strbuf msgbuf = STRBUF_INIT ;
graph_show_line_prefix ( default_diffopt ) ;
if ( ! graph )
return ;
graph_padding_line ( graph , & msgbuf ) ;
fwrite ( msgbuf . buf , sizeof ( char ) , msgbuf . len , graph -> revs -> diffopt . file ) ;
strbuf_release ( & msgbuf ) ;
}
```

Código

```
int graph_show_remainder ( struct git_graph * graph )
{
struct strbuf msgbuf = STRBUF_INIT ;
int shown = 0 ;
graph_show_line_prefix ( default_diffopt ) ;
if ( ! graph )
return 0 ;
if ( graph_is_commit_finished ( graph ) )
return 0 ;
for ( ; ; ) {
graph_next_line ( graph , & msgbuf ) ;
fwrite ( msgbuf . buf , sizeof ( char ) , msgbuf . len ,
graph -> revs -> diffopt . file ) ;
```

Código

```
strbuf_setlen ( & msgbuf , 0 ) ;
shown = 1 ;
if ( ! graph_is_commit_finished ( graph ) ) {
putc ( '\n' , graph -> revs -> diffopt . file ) ;
graph_show_line_prefix ( & graph -> revs -> diffopt ) ;
} else {
break ;
}
}
strbuf_release ( & msgbuf ) ;
return shown ;
}
static void graph_show_strbuf ( struct git_graph * graph ,
```

Código

```
FILE * file ,
struct strbuf const * sb )
{
char * p ;
p = sb -> buf ;
while ( p ) {
size_t len ;
char * next_p = strchr ( p , '\n' ) ;
if ( next_p ) {
next_p ++ ;
len = next_p - p ;
} else {
len = ( sb -> buf + sb -> len ) - p ;
```

Código

```
}
fwrite ( p , sizeof ( char ) , len , file ) ;
if ( next_p && * next_p != '\0' )
graph_show_oneline ( graph ) ;
p = next_p ;
}
}
void graph_show_commit_msg ( struct git_graph * graph ,
FILE * file ,
struct strbuf const * sb )
{
int newline_terminated ;
graph_show_strbuf ( graph , file , sb ) ;
```

Código

```
if ( ! graph )
return ;
newline_terminated = ( sb -> len && sb -> buf [ sb -> len - 1 ] == '\n' ) ;
if ( ! graph_is_commit_finished ( graph ) ) {
if ( ! newline_terminated )
putc ( '\n' , file ) ;
graph_show_remainder ( graph ) ;
if ( newline_terminated )
putc ( '\n' , file ) ;
}
```