Holistic Revision Tree: A Better Version Control System for C Programs

Joseph Tessitore

1. Abstract

According to JetBrain's 2023 developer ecosystem survey, 87% of developers use Git to track revisions in their program [1]. While Git's model is powerful, it has a steep learning curve for new users, owing to its complexity. In this paper, we demonstrate the *Holistic Revision Tree* (HRT), an alternative version control system which takes advantage of the C preprocessor to encode multiple versions of a program in a single file. We will show how HRT makes you a better programmer by simplifying version control and sharing code with others.[1]

2. Background

While Git's distributed workflow is appealing to many developers [2], it must be acknowledged that a significant factor in Git's ongoing dominance is due to vendor-lock in. Popular software forges such as GitHub and integrated development environments offer built-in tooling for Git, and not for other version control systems. In addition, many developers are already familiar with Git, and aren't willing to spend the time needed to learn a new system. Because of this, many alternative version control systems are *forced* to offer Git compatibility. For any new version control system to challenge Git's tyrannical rule, it must take advantage of universal software features, and have a very shallow learning curve.

2.1. The C Preprocessor

The C preprocessor is a filter applied to C source code before they are compiled [3]. This is most commonly used to include *header files* in a source file with the syntax `#include "filename.h"`. More importantly for us, it can be used to conditionally compile code. The following example will print "TRUE" when compiled and executed, and the else branch will be excluded entirely from the compiled program:

```c
#include <stdio.h>

int main() {
#if 1 + 1 == 2
        printf("TRUE\n");
#else
        printf("FALSE\n");
#endif
}
```

Listing 1: Conditionally compiling code with math

This can be taken further by defining variables at compilation. For example, we can define the variable VERSION to be equal to 0 or 1 when compiling by using `gcc -DVERSION=0` or `gcc -DVERSION=1`, respectively. This definition will cause the respective branch to be compiled, while the other is filtered out. Using this technology, we can store multiple revisions of a program in the same file.

```c
#include <stdio.h>

int main() {
#if VERSION >= 1
        printf("Hello, Universe!\n");
#else
        printf("Hello, World!\n");
#endif
}
```

Listing 2: Conditionally compiling code with a variable

To allow users to compile without manually specifying a version, this block can be added to the top of the file:

```c
#ifndef VERSION
#define VERSION 1
#endif
```

Listing 3: A version header to specify the latest version

[1] Another banger by Fai– wait, wrong thing sorry

These lines will define `VERSION` to 1 (the latest version in our example) if it is not already defined. By updating the central line with each update, it can be assured that users will get the newest version.

By using these features of the C preprocessor, the entire revision history of a C file can be stored without any external metadata. While this is powerful, as will be elaborated later, we must accept that editing a program in this manner is very cumbersome. Thus, we must sacrifice a small amount of our method's ideological purity and actually write code instead of just thinking about it.

3. Introducing HRT

The *Holistic Revision Tree* methodology relies on two core files:

1. *The Tree* – A file containing every revision of the program merged into one file.
2. *The Work File* – A file containing one single revision of the program extracted from the tree, without *any* conditional compilation.

The program consists of two commands:

1. `checkout` - Extract a specific revision from the *tree file* into a *work file*.
2. `commit` - Combine a modified revision in a work file back into the tree as a new version.

These commands are named the same as they would be in Git, to aid in adoption.

4. Methodology

The program uses an *LL parser* [4] to parse the relevant preprocessor directives in the C programming language. Thanks to some public domain[2] code from nullprogram.com, the C string-handling code necessary was somewhat bearable[3] [5].

4.1. Checkout

When checking out a revision from the tree, the program evaluates any `#if` statements involving the `VERSION` variable[4] to decide whether or not to include its block of code[5].

4.2. Commit

The commit process first creates a work file from the tree of the latest version, which is determined by the version header (Listing 3). While doing this, the program creates a map[6] from line numbers in the *work file* to line numbers in the *tree file*.

It then calls `diff`, a standard UNIX command line utility to compare text files, to compare the provided and generated work-file. This produces a list of changes written by the developer to the work file. The output of `diff` is parsed [6], and the changes made to the work file are spliced into the tree file.

`diff` can output three different types of changes: Additions, deletions, and changes. The following code is generated for each of the three:

```
...
#if VERSION >= 5
/* Comment added in version 5 */
#endif
```

Listing 4: Code generated for a line addition in version 5

```
#if VERSION < 5
/* Comment removed in version 5 */
#endif
```

Listing 5: Code generated for a line deletion in version 5

```
#if VERSION < 5
/* Comment changed in version 5 */
#else
/* New comment :3 */
#endif
```

Listing 6: Code generated for a line change in version 5

It is important to note that while two lines of code might be next to each-other in a work file, they might be in different blocks in the tree. While

[2] A fact that we *definitely* checked before we based our program off of it

[3] also aided in memory allocation

[4] Assuming `VERSION` is on the left-hand side

[5] The current version is decided either through a command-line flag or through the version header (Listing 3)

[6] associative array

HRT doesn't need to handle this for addition like in Listing 4, it must be handled for removals. This is done by recursively descending through conditional-compilation blocks with overlapping line-ranges.

Extra care was required for changes to make sure the new code would not be included multiple times, or so that the wrong likes wouldn't be deleted. The explanation for this algorithm is omitted ~~because I don't fully understand how I fixed it~~ for brevity.

Finally, the program checks the version header shown in Listing 3, and increments it to show the latest version.

5. Usage

Upon obtaining HRT (see Section 7), it can be compiled as follows: `gcc -o hrt hrt.c`[7]. Intuitively, `hrt.c` itself is an HRT, and its versions can be selected as detailed below (Section 5.1).

To check out the latest version of a program stored in `tree.c` and save the work file in `work.c`, you can use `hrt checkout < tree.c > work.c`.[8] A specific version, such as version 3, can be checked out by running `hrt checkout -v 3 < tree.c > work.c`.

After editing your file in your favorite text editor, you can commit your changes by running `hrt commit tree.c < work.c > new_tree.c`, with `new_tree.c` intuitively containing the new version of the tree.[9].

5.1. Compiling a Specific Revision

`hrt` isn't needed to simply compile any version of a program. For example, to compile version 3, one can run `gcc -DVERSION=3 tree.c`. This provides the portability and simplicity that we all so desperately seek in our lives.

[7] Or whatever C compiler you prefer

[8] Standard input and output are used throughout this program because it makes it feel more UNIX-y

[9] Because of how shells work, you can't do `hrt commit tree.c < work.c > tree.c`. You can do `hrt commit tree.c < work.c > new_tree.c && mv new_tree.c tree.c` to accomplish the same thing

6. Results

One clear benefit of taking HRT is that it eschews the need for traditional Git forges such as GitHub, since our source tree can be hosted almost anywhere. This could be shared as a link on a web-site, stored in a Google Drive, or even provided in its entirety at the end of a paper. The latter is useful because it grants readers a deeper insight into the progression of a paper, rather than just its final state. It allows them to see how a program changed over time, including deleted functions, confused comments, and code that was never relevant at any point whatsoever. Changes are also simple to share by either sending a work file, or storing the tree on a shared filesystem that can be changed by all contributers.[10]

6.1. File Size

HRT also has significant size benefits. Table 1 shows a comparison between how much size the source repository for HRT takes as a git bundle [7], a tar archive containing each revision, and an HRT tree. It also includes the work files of the final revision in bold.

Method	**Bytes**
`tar` archive of revisions	450506
HRT tree	44389
Git bundle	25703
Final work file	24256
`tar.gz` archive of revisions	17517
HRT tree (gzipped)	8645
Final work file (gzipped)	5791

Table 1: Comparison of file sizes of hrt source code

As shown, the compressed HRT tree is the smallest way that every version can be stored.

[10] `hrt commit` as used above is non-atomic. Careful!

It's compression ratio can be calculated through $\frac{\text{size of all revisions}}{\text{size of HRT tree}}$, yielding 10.15x for an uncompressed tree, and 52.11x for a compressed tree. Other compression systems like Facebook's ZSTD, meanwhile, can only offer 5.5x on source-code [8]. Further research is needed to investigate HRT's clear potential as a general purpose compression algorithm.

7. Obtaining HRT

HRT can be obtained either by downloading the source tree from https://jonot.me/hrt.c, or by copying the source tree provided at the end of the paper.[11]

[11] Programmers in the UK must go through a waiting list to make sure that they *really* don't want to use Git instead, and that they're not just pretending.

8. Source Listing

```
#ifndef VERSION
#define VERSION 23
#endif
#include <assert.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#if VERSION >= 4
#include <unistd.h>
#include <sys/wait.h>
#if VERSION < 9
#include <errno.h>
#endif
#endif

#if VERSION >= 11
#if VERSION < 13
static int version = 1;
#else
static int version = 2;

#define max(x, y) (((x) > (y)) ? (x) : (y))
#define min(x, y) (((x) > (y)) ? (y) : (x))
#endif

#endif
/* Some code from nullprogram.com */
#define new(a, n, t) (t*)alloc(a, n, sizeof(t), _Alignof(t))

typedef struct {
  char *begin;
  char *end;
} Arena;

#if VERSION >= 4
#if VERSION < 5
/* Returns 3. Behavior if x == 4 is undefined */
int foo(int x) {
  if (x == 4) {
    return 5;
  }
  return 3;
}
#endif
#endif
#if VERSION < 5

#endif
void  *alloc(Arena  *a,  ptrdiff_t  count,  ptrdiff_t  size,
ptrdiff_t align) {
  ptrdiff_t pad = -(uintptr_t)a->begin & (align - 1);
  assert(count < (a->end - a->begin - pad) / size);
  void *r = a->begin + pad;
  a->begin += pad + count * size;
  return memset(r, 0, count * size);
}

#define S(s)
  (Str) { s, sizeof(s) - 1 }

typedef struct {
  char *data;
  ptrdiff_t len;
} Str;

Str span(char *begin, char *end) {
  Str r = {0};
  r.data = begin;
  r.len = begin ? end - begin : 0;
  return r;
}

_Bool equals(Str a, Str b) {
  if (a.len != b.len) {
    return 0;
  }
  if (a.len == 0) {
    return 1;
  }
  return !memcmp(a.data, b.data, a.len);
}

Str trimleft(Str s) {
  for (; s.len && *s.data <= ' '; s.data++, s.len--) {
  }
  return s;
}

Str trimright(Str s) {
  for (; s.len && s.data[s.len - 1] <= ' '; s.len--) {
  }
  return s;
}

Str substring(Str s, ptrdiff_t i) {
  if (i) {
    s.data += i;
    s.len -= i;
  }
  return s;
}

_Bool starts_with(Str s, Str prefix) {
  if (!prefix.len) {
    return 1;
  }
  if (s.len < prefix.len) {
    return 0;
  }
  s.len = prefix.len;
  return equals(s, prefix);
}

}

typedef struct {
  Str head;
  Str tail;
  _Bool ok;
} Cut;

Cut cut(Str s, char c) {
  Cut r = {0};
  if (!s.len) {
    return r;
  }
  char *begin = s.data;
  char *end = s.data + s.len;
  char *cut = begin;
  for (; cut < end && *cut != c; cut++) {
  }
  r.ok = cut < end;
  r.head = span(begin, cut);
  r.tail = span(cut + r.ok, end);
  return r;
}

#if VERSION >= 1
#if VERSION < 6
void printstr(Str s) {
  for (int i = 0; i < s.len; i++) {
    putchar(s.data[i]);
  }
#else
void fprintstr(FILE *f, Str s) {
  fwrite(s.data, sizeof(char), s.len, f);
#endif
}

#endif
Str read_file(Arena *a, FILE *f) {
  Str ret = {.data = a->begin, .len = a->end - a->begin};
  ret.len = fread(ret.data, 1, ret.len, f);
  a->begin += ret.len;
  return ret;
}

typedef enum {
  OP_LT = 0,
  OP_GT = 1,
  OP_LE = 2,
  OP_GE = 3,
  OP_EQ = 4,
  OP_NEQ = 5,
  OP_DEF = 6,
  OP_NDEF = 7,
} Operator;

typedef enum {
  LINE_TYPE_PLAIN = 0,
  LINE_TYPE_IF = 1,
  LINE_TYPE_ELIF = 2,
  LINE_TYPE_ELSE = 3,
  LINE_TYPE_ENDIF = 4,
} LineType;

typedef struct {
  Str variable;
  Operator operator;
  LineType line_type;
  int value;
  Str span;
#if VERSION >= 5
#if VERSION < 9
  int lineno;
#else
  int no;
#endif
#endif
} Line;

/* Parse a base 10 integer. Very basic routine, does not
handle things
   like negatives */
int parse_integer(Str s) {
  int ret = 0;
  for (int i = 0; i < s.len; i++) {
    ret *= 10;
    ret += s.data[i] - '0';
  }
  return ret;
}

Operator parse_operator(Str op) {
  const Str strings[] = {
      S("<"), S(">"), S("<="), S(">="), S("=="), S("!="),
  };
  for (size_t i = 0; i < sizeof(strings) / sizeof(Str); i+
+) {
    if (equals(op, strings[i])) {
      return i;
    }
  }
  /* DEADLINE I DONT HAVE TIME */
  return -1;
}

void parse_expression(Str expr, Line *out) {
  Cut c = cut(expr, ' ');
  Str variable = c.head;
  c = cut(c.tail, ' ');
  Str operator = c.head;
  Str value = c.tail;

  out->variable = variable;
  out->operator = parse_operator(operator);
  out->value = parse_integer(value);
}

Line parse_line(Str line) {
  Line ret = {0};
  ret.span = line;
  if (!line.len || line.data[0] != '#') {
    return ret;
  }
#if VERSION < 1
  Cut c = cut(line, ' ');
#else
  Cut c = cut(trimleft(line), ' ');
#endif
  const Str IFDEF = S("#ifdef");
  if (equals(c.head, IFDEF)) {
    ret.operator = OP_DEF;
    ret.variable = c.tail;
    ret.line_type = LINE_TYPE_IF;
    return ret;
  }
  const Str IFNDEF = S("#ifndef");
  if (equals(c.head, IFNDEF)) {
    ret.operator = OP_NDEF;
    ret.variable = c.tail;
    ret.line_type = LINE_TYPE_IF;
    return ret;
  }
  const Str IF = S("#if");
  if (equals(c.head, IF)) {
    ret.line_type = LINE_TYPE_IF;
    parse_expression(c.tail, &ret);
    return ret;
  }
  const Str ELIF = S("#elif");
  if (equals(c.head, ELIF)) {
    ret.line_type = LINE_TYPE_ELIF;
    parse_expression(c.tail, &ret);
    return ret;
  }
  const Str ELSE = S("#else");
  if (equals(c.head, ELSE)) {
    ret.line_type = LINE_TYPE_ELSE;
    return ret;
  }
  const Str ENDIF = S("#endif");
  if (equals(c.head, ENDIF)) {
    ret.line_type = LINE_TYPE_ENDIF;
    return ret;
  }
  return ret;
}

#if VERSION >= 1
typedef struct IfBlockParse IfBlockParse;

#if VERSION < 4
void print_line(const Line *line) {
  putchar('"');
  printstr(line->span);
  putchar('"');
}

#endif
typedef struct PlainBlockParse {
  Line line;
  struct PlainBlockParse *next;
} PlainBlockParse;

#if VERSION < 4
void print_plain_block(const PlainBlockParse *b) {
  if (b == NULL) {
    printf("nil");
  } else {
    putchar('(');
    print_line(&b->line);
    printf(" . ");
    print_plain_block(b->next);
    putchar(')');
  }
}

#endif
typedef struct {
  PlainBlockParse *plain_block;
  IfBlockParse* if_block;
} StatementBlockParse;

#if VERSION < 4
void print_if_block(const IfBlockParse *);

void print_statement_block(const StatementBlockParse *b) {
  if (b == NULL) {
    printf("nil");
  } else {
    putchar('(');
    print_plain_block(b->plain_block);
    printf(" . ");
    print_if_block(b->if_block);
    putchar(')');
  }
}

#endif
typedef struct ElIfBlockParse {
  Line line;
  StatementBlockParse *block;
  struct ElIfBlockParse *next;
} ElIfBlockParse;

#if VERSION < 4
void print_elif_block(const ElIfBlockParse *b) {
  if (b == NULL) {
    printf("nil");
  } else {
    printf("(elif ");
    print_line(&b->line);
```

```
    putchar(' ');
    print_statement_block(b->block);
    putchar(' ');
    print_elif_block(b->next);
    putchar(')');
  }
}

#endif
typedef struct ElseBlockParse {
#if VERSION >= 9
  Line line;
#endif
  StatementBlockParse *block;
} ElseBlockParse;

#if VERSION < 4
void print_else_block(const ElseBlockParse *b) {
  if (b == NULL) {
    printf("nil");
  } else {
    printf("(else ");
    print_statement_block(b->block);
    putchar(')');
  }
}

#endif
typedef struct IfBlockParse {
  Line line;
#if VERSION >= 2
  Line end_line;
#endif
  StatementBlockParse *statement_block;
  ElIfBlockParse *elif_block;
  ElseBlockParse *else_block;
  StatementBlockParse *next;
} IfBlockParse;

#if VERSION < 4
void print_if_block(const IfBlockParse *b) {
  if (b == NULL) {
    printf("nil");
  } else {
    printf("(if ");
    print_line(&b->line);
    putchar(' ');
    print_statement_block(b->statement_block);
    putchar(' ');
    print_elif_block(b->elif_block);
    putchar(' ');
    print_else_block(b->else_block);
    putchar(' ');
    print_statement_block(b->next);
    putchar(')');
  }
}

#endif
typedef struct {
  Str file;
#if VERSION >= 5
  int lineno;
#endif
} ParserState;

_Bool peek(ParserState *p, Line *out) {
  if (!p->file.len) {
    return 0;
  }
  Cut c = cut(p->file, '\n');
  *out = parse_line(c.head);
#if VERSION >= 5
#if VERSION < 9
  out->lineno = p->lineno + 1;
#else
  out->no = p->lineno + 1;
#endif
#endif
  return 1;
}

/* To be called after peek to advance the parser state */
void feed(ParserState *p) {
  Cut c = cut(p->file, '\n');
#if VERSION >= 5
  p->lineno++;
#endif
  p->file = c.tail;
}

PlainBlockParse *parse_plain_block(Arena *a, ParserState *p)
{
  Line next;
  if (peek(p, &next) && next.line_type == LINE_TYPE_PLAIN) {
#if VERSION < 5
    PlainBlockParse *ret = new(a, 1, PlainBlockParse);
#else
    PlainBlockParse *ret = new (a, 1, PlainBlockParse);
#endif
    feed(p);
    ret->line = next;
    ret->next = parse_plain_block(a, p);
    return ret;
  } else {
    return NULL;
  }
}

IfBlockParse *parse_if_block(Arena *, ParserState *);

StatementBlockParse     *parse_statement_block(Arena     *a,
ParserState *p) {
  StatementBlockParse *ret = new(a, 1, StatementBlockParse);
  ret->plain_block = parse_plain_block(a, p);
  ret->if_block = parse_if_block(a, p);
  return ret;
}

ElIfBlockParse *parse_elif_block(Arena *, ParserState *);
ElseBlockParse *parse_else_block(Arena *, ParserState *);

IfBlockParse *parse_if_block(Arena *a, ParserState *p) {
  /* (iSCTeS)? */
  Line next;
  if (peek(p, &next) && next.line_type == LINE_TYPE_IF) {
    IfBlockParse *ret = new(a, 1, IfBlockParse);
    feed(p);
    ret->line = next;
    ret->statement_block = parse_statement_block(a, p);
    ret->elif_block = parse_elif_block(a, p);
    ret->else_block = parse_else_block(a, p);
    if (peek(p, &next) && next.line_type == LINE_TYPE_ENDIF) {
      feed(p);
#if VERSION >= 2
      ret->end_line = next;
#endif
      ret->next = parse_statement_block(a, p);
      return ret;
    } else {
      /* This is a failed parse actually. TODO maybe do an
error here? */
      return NULL;
    }
  } else {
    return NULL;
  }
}

ElIfBlockParse *parse_elif_block(Arena *a, ParserState *p) {
  Line next;
  if (peek(p, &next) && next.line_type == LINE_TYPE_ELIF) {
    ElIfBlockParse *ret = new (a, 1, ElIfBlockParse);
    feed(p);
    ret->line = next;
    ret->block = parse_statement_block(a, p);
    ret->next = parse_elif_block(a, p);
    return ret;
  } else {
    return NULL;
  }
}

ElseBlockParse *parse_else_block(Arena *a, ParserState *p) {
  Line next;
  if (peek(p, &next) && next.line_type == LINE_TYPE_ELSE) {
    ElseBlockParse *ret = new (a, 1, ElseBlockParse);
#if VERSION >= 9
    ret->line = next;
#endif
    feed(p);
    ret->block = parse_statement_block(a, p);
    return ret;
  } else {
    return NULL;
  }
}

#endif
#if VERSION < 2
/* Returns the "turning number" for the line. If opening
something, 1. If closing, -1 */
int parse_turning(Str line) {
  Cut c = cut(line, ' ');
  const Str IF = S("#if");
  const Str IFDEF = S("#ifdef");
  const Str IFNDEF = S("#ifndef");
  const Str ENDIF = S("#endif");
  if (equals(c.head, IF)) {
    return 1;
  }
  if (equals(c.head, IFDEF)) {
    return 1;
#else
/* Returns true if the condition deals with the VERSION
variable */
_Bool condition_version(const Line *line) {
  const Str version_str = S("VERSION");
  if (!equals(line->variable, version_str)) {
    return 0;
  }
  /* Support that fallback in the file */
  if (line->operator == OP_DEF || line->operator== OP_NDEF) {
    return 0;
  }
  return 1;
}

#if VERSION < 9
_Bool evaluate_condition(const Line *line) {
#else
#if VERSION < 11
_Bool evaluate_condition(const Line *line, int version) {
#else
_Bool evaluate_condition(const Line *line) {
#endif
#endif
  /* Assumes that the variable is VERSION */
#if VERSION < 9
  const int version = 1;
#endif
  switch (line->operator) {
  case OP_LT:
    return version < line->value;
  case OP_GT:
    return version > line->value;
  case OP_LE:
    return version <= line->value;
  case OP_GE:
    return version >= line->value;
  case OP_EQ:
    return version == line->value;
  case OP_NEQ:
    return version != line->value;
  default:
    /* These shouldn't be handled here at all */
    return 0;
  }
}

#if VERSION < 3
void dump_statement_block(StatementBlockParse *b);
#else
#if VERSION < 6
void dump_statement_block(StatementBlockParse *, _Bool);
#else
#if VERSION < 9
void  dump_statement_block(FILE  *,  StatementBlockParse  *,
_Bool);
#else
#if VERSION < 15
/* Indexes are line numbers in the reduced tree file. Indexes
are values in the full tree */
#else
/* Indexes are line numbers in the reduced tree file. Values
are line numbers in the full tree */
#endif
static int *line_map;
static int current_line;

#if VERSION >= 15
int line_map_reverse(int full_line_number) {
  int *ptr = line_map + 1;
  while (1) {
    if (*ptr == full_line_number) {
      return ptr - line_map;
    }
    ptr++;
  }
}

#endif
#if VERSION < 10
void  dump_statement_block(FILE  *,  StatementBlockParse  *,
_Bool, int);
#else
static _Bool should_trim = 1;
#endif
#endif
#endif
#endif

#if VERSION < 3
void dump_plain_block(PlainBlockParse *b) {
#else
#if VERSION < 6
void dump_plain_block(PlainBlockParse *b, _Bool final) {
#else
#if VERSION < 10
void  dump_plain_block(FILE  *f,  PlainBlockParse  *b,  _Bool
final) {
#else
#if VERSION < 11
void  dump_statement_block(FILE  *,  StatementBlockParse  *,
int);
#else
void dump_statement_block(FILE *, StatementBlockParse *);
#endif

void dump_plain_block(FILE *f, PlainBlockParse *b) {
#endif
#endif
#endif
  if (b == NULL) {
    return;
  }
#if VERSION >= 5
#if VERSION < 6
  printf("%d: ", b->line.lineno);
#else
  fprintstr(f, b->line.span);
#endif
#endif
#if VERSION < 6
  printstr(b->line.span);
#endif
#if VERSION < 3
  putchar('\n');
  dump_plain_block(b->next);
#else
#if VERSION < 5
  if (!final || b->next != NULL) {
#else
#if VERSION < 9
  if (!final || b->next == NULL) {
#else
  fputc('\n', f);
#if VERSION < 10
  line_map[current_line++] = b->line.no;
#else
  if (should_trim) line_map[current_line++] = b->line.no;
  dump_plain_block(f, b->next);
#endif
#endif
#endif
#if VERSION < 6
    putchar('\n');
#else
#if VERSION < 9
    fputc('\n', f);
#endif
#endif
#if VERSION < 9
  }
#endif
```

```
#if VERSION < 6
  dump_plain_block(b->next, final);
#else
#if VERSION < 10
  dump_plain_block(f, b->next, final);
#endif
#endif
#endif
}

#if VERSION < 6
_Bool dump_elif_block(ElIfBlockParse *b) {
#else
#if VERSION < 9
_Bool dump_elif_block(FILE *f, ElIfBlockParse *b) {
#else
#if VERSION < 11
_Bool  dump_elif_block(FILE  *f,  ElIfBlockParse  *b,  int
version) {
#else
_Bool dump_elif_block(FILE *f, ElIfBlockParse *b) {
#endif
#endif
#endif
  if (b == NULL) {
    return 0;
#endif
  }
#if VERSION < 2
  if (equals(c.head, IFNDEF)) {
    return 1;
#else
#if VERSION < 10
  if (condition_version(&b->line)) {
#else
  if (should_trim && condition_version(&b->line)) {
#endif
#if VERSION < 9
    if (evaluate_condition(&b->line)) {
#else
#if VERSION < 11
    if (evaluate_condition(&b->line, version)) {
#else
    if (evaluate_condition(&b->line)) {
      dump_statement_block(f, b->block);
#endif
#if VERSION < 10
      dump_statement_block(f, b->block, 0, version);
#else
#if VERSION < 11
      dump_statement_block(f, b->block, version);
#endif
#endif
#endif
#if VERSION < 3
      dump_statement_block(b->block);
#else
#if VERSION < 6
      dump_statement_block(b->block, 0);
#else
#if VERSION < 9
      dump_statement_block(f, b->block, 0);
#endif
#endif
#endif
      return 1;
    } else {
#if VERSION < 6
      return dump_elif_block(b->next);
#else
#if VERSION < 9
      return dump_elif_block(f, b->next);
#else
#if VERSION < 11
      return dump_elif_block(f, b->next, version);
#else
      return dump_elif_block(f, b->next);
#endif
#endif
#endif
    }
  } else {
#if VERSION >= 10

#endif
#if VERSION >= 9
    /* This should dump the elif pragma, but I'm not using
that in my
       program so it's okay that it doesn't work at all */
#if VERSION >= 10
    fprintstr(f, b->line.span);
    fputc('\n', f);
#if VERSION < 11
    dump_statement_block(f, b->block, version);
    dump_elif_block(f, b->next, version);
#else
    dump_statement_block(f, b->block);
    dump_elif_block(f, b->next);
#endif
#endif
#endif
    return 0;
  }
}

#if VERSION < 6
void dump_else_block(ElseBlockParse *b) {
#else
#if VERSION < 9
void dump_else_block(FILE *f, ElseBlockParse *b) {
#else
#if VERSION < 11
void dump_else_block(FILE *f, ElseBlockParse *b, int version)
{
#else
void dump_else_block(FILE *f, ElseBlockParse *b) {
#endif
#endif
#endif
  if (b == NULL) {
    return;
  }
#if VERSION < 3
  dump_statement_block(b->block);
#else
#if VERSION < 6
  dump_statement_block(b->block, 0);
#else
#if VERSION < 9
  dump_statement_block(f, b->block, 0);
#else
#if VERSION < 10
  dump_statement_block(f, b->block, 0, version);
#else
  if (!should_trim) {
    fprintstr(f, b->line.span);
    fputc('\n', f);
  }
#if VERSION < 11
  dump_statement_block(f, b->block, version);
#else
  dump_statement_block(f, b->block);
#endif
#endif
#endif
#endif
#endif
}

#if VERSION < 6
void dump_if_block(IfBlockParse *b) {
#else
#if VERSION < 9
void dump_if_block(FILE *f, IfBlockParse *b) {
#else
#if VERSION < 11
void dump_if_block(FILE *f, IfBlockParse *b, int version) {
#else
void dump_if_block(FILE *f, IfBlockParse *b) {
#endif
#endif
#endif
  if (b == NULL) {
    return;
#endif
  }
#if VERSION < 2
  if (equals(c.head, ENDIF)) {
    return -1;
#else
#if VERSION < 10
  if (condition_version(&b->line)) {
#else
  if (should_trim && condition_version(&b->line)) {
#endif
#if VERSION < 9
    if (evaluate_condition(&b->line)) {
#else
#if VERSION < 11
    if (evaluate_condition(&b->line, version)) {
#else
    if (evaluate_condition(&b->line)) {
      dump_statement_block(f, b->statement_block);
    } else if (!dump_elif_block(f, b->elif_block)) {
      dump_else_block(f, b->else_block);
#endif
#if VERSION < 10
      dump_statement_block(f, b->statement_block, 0, version);
#else
#if VERSION < 11
      dump_statement_block(f, b->statement_block, version);
#endif
#endif
#if VERSION < 11
    } else if (!dump_elif_block(f, b->elif_block, version)) {
      dump_else_block(f, b->else_block, version);
#endif
#endif
#if VERSION < 3
      dump_statement_block(b->statement_block);
#else
#if VERSION < 6
      dump_statement_block(b->statement_block, 0);
#else
#if VERSION < 9
      dump_statement_block(f, b->statement_block, 0);
    } else if (!dump_elif_block(f, b->elif_block)) {
      dump_else_block(f, b->else_block);
#endif
#endif
#endif
#if VERSION < 6
    } else if (!dump_elif_block(b->elif_block)) {
      dump_else_block(b->else_block);
#endif
    }
  } else {
#if VERSION >= 5
#if VERSION < 6
    printf("%d: ", b->line.lineno);
#else
    fprintstr(f, b->line.span);
    fputc('\n', f);
#if VERSION < 9
    dump_statement_block(f, b->statement_block, 0);
#else
    line_map[current_line++] = b->line.no;
#if VERSION < 10
    dump_statement_block(f, b->statement_block, 0, version);
#else
#if VERSION < 11
    dump_statement_block(f, b->statement_block, version);
    dump_elif_block(f, b->elif_block, version);
    dump_else_block(f, b->else_block, version);
#else
    dump_statement_block(f, b->statement_block);
    dump_elif_block(f, b->elif_block);
    dump_else_block(f, b->else_block);
#endif
#endif
#endif
    fprintstr(f, b->end_line.span);
    fputc('\n', f);
#if VERSION >= 9
    line_map[current_line++] = b->end_line.no;
#endif
#endif
#endif
#if VERSION < 6
    printstr(b->line.span);
    putchar('\n');
#endif
#if VERSION < 3
    dump_statement_block(b->statement_block);
#else
#if VERSION < 6
    dump_statement_block(b->statement_block, 0);
#endif
#if VERSION >= 5
#if VERSION < 6
    printf("%d: ", b->end_line.lineno);
#endif
#endif
#endif
#if VERSION < 6
    printstr(b->end_line.span);
    putchar('\n');
#endif
  }
#if VERSION < 3
  dump_statement_block(b->next);
#else
#if VERSION < 6
  dump_statement_block(b->next, 1);
#else
#if VERSION < 9
  dump_statement_block(f, b->next, 1);
#else
#if VERSION < 10
  dump_statement_block(f, b->next, 1, version);
#else
#if VERSION < 11
  dump_statement_block(f, b->next, version);
#else
  dump_statement_block(f, b->next);
#endif
#endif
#endif
#endif
#endif
}

#if VERSION < 3
void dump_statement_block(StatementBlockParse *b) {
#else
#if VERSION < 6
void  dump_statement_block(StatementBlockParse  *b,  _Bool
final) {
#else
#if VERSION < 9
void dump_statement_block(FILE *f, StatementBlockParse *b,
_Bool final) {
#else
#if VERSION < 10
void dump_statement_block(FILE *f, StatementBlockParse *b,
_Bool final, int version) {
#else
#if VERSION < 11
void dump_statement_block(FILE *f, StatementBlockParse *b,
int version) {
#else
void dump_statement_block(FILE *f, StatementBlockParse *b) {
#endif
#endif
#endif
#endif
#endif
  if (b == NULL) {
    return;
#endif
  }
#if VERSION < 2
  return 0;
#else
#if VERSION < 3
  dump_plain_block(b->plain_block);
#else
#if VERSION < 6
  dump_plain_block(b->plain_block, final && b->if_block ==
NULL);
#else
#if VERSION < 10
  dump_plain_block(f, b->plain_block, final && b->if_block
== NULL);
#else
  dump_plain_block(f, b->plain_block);
#endif
#if VERSION < 9
  dump_if_block(f, b->if_block);
#else
#if VERSION < 11
  dump_if_block(f, b->if_block, version);
#else
  dump_if_block(f, b->if_block);
#endif
```

```c
#endif
#endif
#endif
#if VERSION < 6
  dump_if_block(b->if_block);
#endif
#endif
}

#if VERSION >= 8
Str read_from_popen(Arena *a, FILE *f) {
  Str ret = {0};
  ret.data = a->begin;
  while (!feof(f)) {
#if VERSION < 9
      ret.len += fread(ret.data + ret.len, sizeof(char), a->end - (ret.data + ret.len), f);
#else
    ret.len += fread(ret.data + ret.len, sizeof(char),
                     a->end - (ret.data + ret.len), f);
#endif
  }
  a->begin += ret.len;
  return ret;
}

#endif
#if VERSION >= 4
#if VERSION < 6
void diff() {
  /* We aren't ready yet */
  return;
  /* Child read, parent write */
  int p1[2];
  /* Parent read, child write */
  int p2[2];
  pipe(p1);
  pipe(p2);
  pid_t pid = fork();
  if (pid == 0) {
    /* Child */
    close(p1[1]);
    close(p2[0]);
    dup2(p1[0], 0);
    dup2(p2[1], 1);
    char *args[] = {"/usr/bin/env", "diff", "./test.c", "./main.c", NULL};
    if (execv(args[0], args)) {
      printf("Error #%d: %s\n", errno, strerror(errno));
    }
  } else {
    /* Parent */
    close(p1[0]);
    close(p2[1]);
    /* Wait for child to exit */
    int status;
    while (1) {
      wait(&status);
      if (WIFEXITED(status)) {
  break;
      }
#else
#if VERSION < 7
void diff(StatementBlockParse *b) {
#else
#if VERSION < 9
void diff(Arena *a, StatementBlockParse *b) {
#else
typedef struct {
  int begin;
  int end;
} DiffLineRange;

DiffLineRange parse_diff_range(Str range) {
  DiffLineRange ret = {0};
  Cut c = cut(range, ',');
  if (c.ok) {
    ret.begin = parse_integer(c.head);
    ret.end = parse_integer(c.tail) + 1; /* This might be wrong */
  } else {
    ret.begin = parse_integer(range);
    ret.end = ret.begin + 1;
  }
  return ret;
}

typedef struct {
  DiffLineRange tree;
  DiffLineRange work;
  char mode;
} DiffLine;

DiffLine parse_diff_line(Str line) {
  char mode = 'a';
  Cut c = cut(line, 'a');
  if (!c.ok) {
    mode = 'c';
    c = cut(line, 'c');
    if (!c.ok) {
      mode = 'd';
      c = cut(line, 'd');
    }
  }

  DiffLine ret = {0};
  ret.mode = mode;
  ret.tree = parse_diff_range(c.head);
  ret.work = parse_diff_range(c.tail);
  return ret;
}

#if VERSION < 18
Str skip_lines(Str str, DiffLine line) {
  int lines_to_skip;
  if (line.mode == 'a') {
    lines_to_skip = line.tree.end - line.tree.begin + 1;
  } else if (line.mode == 'c') {
      lines_to_skip = line.tree.end - line.tree.begin + line.work.end - line.work.begin + 2;
  } else {
    lines_to_skip = line.tree.end - line.tree.begin + 3;
  }
  Cut c = {0};
  c.tail = str;
  for (int i = 0; i < lines_to_skip; i++) {
    if (!c.tail.len) {
      break;
    }
#else
Str skip_lines(Str str) {
  Cut c = cut(str, '\n');
  while (c.tail.data[0] == '<' || c.tail.data[0] == '>' ||
         c.tail.data[0] == '-') {
#endif
    c = cut(c.tail, '\n');
  }
  return c.tail;
}

StatementBlockParse *parse_from_filename(Arena *a, const char *filename) {
  FILE *f = fopen(filename, "r");
  assert(f != NULL);
  ParserState p = {
    .file = read_file(a, f),
  };
  return parse_statement_block(a, &p);
}

#if VERSION >= 10
#if VERSION < 11
int first_line_if(const IfBlockParse *);

#endif
/* Returns the first line number of a statement block */
int first_line_statement(const StatementBlockParse *b) {
  if (b->plain_block) {
    return b->plain_block->line.no;
  }
  return b->if_block->line.no;
}

#if VERSION >= 12
int last_line_statement(const StatementBlockParse *b) {
  if (b->if_block) {
#if VERSION < 13
    if (b->if_block->next) {
#else
    if (b->if_block->next && (b->if_block->next->if_block || b->if_block->next->plain_block)) {
#endif
      return last_line_statement(b->if_block->next);
    }
    return b->if_block->end_line.no;
  }
  PlainBlockParse *ptr = b->plain_block;
  while (ptr->next) {
    ptr = ptr->next;
  }
  return ptr->line.no;
}

#endif
PlainBlockParse *get_line_range_descend(Arena *a, DiffLineRange range,
                                        const PlainBlockParse *b) {
  if (b == NULL || range.end - range.begin == 0) {
    return NULL;
  }
  /* Start cutting */
  if (b->line.no == range.begin) {
    PlainBlockParse *copy = new (a, 1, PlainBlockParse);
    copy->line = b->line;
#if VERSION >= 13
    copy->line.no = -1;
#endif
    DiffLineRange new_range = range;
    new_range.begin++;
    copy->next = get_line_range_descend(a, new_range, b->next);
    return copy;
  }
  return get_line_range_descend(a, range, b->next);
}

/* Copies the range of lines from the workfile referred to by the
 range.  Assumes that it doesn't contain any pragmas, which I don't
 need for the paper anyways. */
PlainBlockParse *get_line_range(Arena *a, DiffLineRange range, const StatementBlockParse *b) {
  /* First check if we need to go deeper into the if block */
  if (b->if_block) {
    if (range.begin > b->if_block->line.no) {
      if (range.begin >= first_line_statement(b->if_block->next)) {
  return get_line_range(a, range, b->if_block->next);
      }
    }
    /* TODO: This path isn't needed for the paper but wow */
    assert(0);
  }
  return get_line_range_descend(a, range, b->plain_block);
}

#endif
#if VERSION < 11
void diff(Arena *a, StatementBlockParse *tree) {
  const int version = 1;
#else
void splice(Arena *a, StatementBlockParse *tree, PlainBlockParse *dna,
            DiffLine range) {
  if (tree->if_block && range.tree.begin >= tree->if_block->line.no) {
#if VERSION < 21
    /* TODO: Parse into the tree */
#endif
    if (tree->if_block->next &&
#if VERSION >= 21
        (tree->if_block->next->if_block || tree->if_block->next->plain_block) &&
#endif
            range.tree.begin >= first_line_statement(tree->if_block->next)) {
      splice(a, tree->if_block->next, dna, range);
      return;
    }
    if (tree->if_block->else_block &&
            range.tree.begin > tree->if_block->else_block->line.no) {
          splice(a, tree->if_block->else_block->block, dna, range);
      return;
    }
    /* TODO: Elif is completely not handled here. Shouldn't matter, but IDK */
    splice(a, tree->if_block->statement_block, dna, range);
    return;
  }

  PlainBlockParse *prefix = new (a, 1, PlainBlockParse);
  PlainBlockParse *suffix = new (a, 1, PlainBlockParse);

  char *prefix_line_buf = new (a, 32, char);
  sprintf(prefix_line_buf, "#if VERSION >= %d", version + 1);

  prefix->line.span.data = prefix_line_buf;
  prefix->line.span.len = strlen(prefix->line.span.data);
  prefix->next = dna;
  suffix->line.span = S("#endif");
#endif

#if VERSION >= 11
  PlainBlockParse *ptr = tree->plain_block;
  while (ptr) {
    if (ptr->line.no == range.tree.begin) {
      break;
    }
    ptr = ptr->next;
  }
  PlainBlockParse *dna_ptr = dna;
  while (dna_ptr->next) {
    dna_ptr = dna_ptr->next;
  }
  suffix->next = ptr->next;
  ptr->next = prefix;
  dna_ptr->next = suffix;
}

#if VERSION >= 12
void splice_delete(Arena *a, StatementBlockParse *tree, DiffLineRange range) {
#if VERSION < 13
  assert(range.begin >= first_line_statement(tree) &&
         range.end <= last_line_statement(tree) + 1);
    if (tree->if_block && range.begin > tree->if_block->line.no) {
    if (tree->if_block->next &&
        range.begin > first_line_statement(tree->if_block->next)) {
      return splice_delete(a, tree->if_block->next, range);
    }
    if (tree->if_block->else_block &&
        range.begin > tree->if_block->else_block->line.no) {
      return splice_delete(a, tree->if_block->else_block->block, range);
#else
  if (tree == NULL) {
    return;
  }
  if (tree->plain_block) {
    PlainBlockParse *ptr = tree->plain_block;
    while (ptr->next) {
      ptr = ptr->next;
    }
    int plain_begin = tree->plain_block->line.no;
    int plain_end = ptr->line.no + 1;

    /* Check for overlap */
   if ((range.begin >= plain_begin && range.begin < plain_end) ||
       (range.end > plain_begin && range.end <= plain_end)) {
#if VERSION < 14
      printf("(%d, %d)[%d, %d]\n", range.begin, range.end, plain_begin,
             plain_end);
#endif
      ptr = tree->plain_block;
      PlainBlockParse *parent = NULL;
      while (ptr->line.no == -1 || ptr->line.no < range.begin) {
  parent = ptr;
  ptr = ptr->next;
      }

      PlainBlockParse *prefix = new (a, 1, PlainBlockParse);
      char *prefix_line_buf = new (a, 32, char);
      sprintf(prefix_line_buf, "#if VERSION < %d", version + 1);

      prefix->line.span.data = prefix_line_buf;
      prefix->line.span.len = strlen(prefix->line.span.data);
```

```
      prefix->next = ptr;
      if (parent == NULL) {
  tree->plain_block = prefix;
      } else {
        parent->next = prefix;
      }
      PlainBlockParse *back_ptr = ptr;
      while (back_ptr &&
             (back_ptr->line.no == -1 || back_ptr->line.no <
range.end)) {
  parent = back_ptr;
  back_ptr = back_ptr->next;
      }
      PlainBlockParse *suffix = new (a, 1, PlainBlockParse);
      suffix->line.span = S("#endif");
      suffix->next = back_ptr;
      parent->next = suffix;
#endif
    }
#if VERSION < 13
    /* TODO: Again, this doesn't handle elif */
    return splice_delete(a, tree->if_block->statement_block,
range);
#endif
  }
#if VERSION < 13
  PlainBlockParse *ptr = tree->plain_block;
  if (ptr->line.no == range.begin) {
    PlainBlockParse *back_ptr = ptr;
    while (back_ptr->line.no != range.end - 1) {
      back_ptr = back_ptr->next;
    }
    PlainBlockParse *prefix = new (a, 1, PlainBlockParse);
    char *prefix_line_buf = new (a, 32, char);
    sprintf(prefix_line_buf, "#if VERSION < %d", version + 1);
    prefix->line.span.data = prefix_line_buf;
    prefix->line.span.len = strlen(prefix->line.span.data);
    prefix->next = ptr;
    tree->plain_block = prefix;
    PlainBlockParse *suffix = new (a, 1, PlainBlockParse);
    suffix->line.span = S("#endif");
    suffix->next = back_ptr->next;
    back_ptr->next = suffix;
#else
  if (!tree->if_block) {
#endif
    return;
  }
#if VERSION < 13
  assert(0);
#else
  splice_delete(a, tree->if_block->statement_block, range);
  if (tree->if_block->else_block) {
      splice_delete(a,  tree->if_block->else_block->block,
range);
  }
  splice_delete(a, tree->if_block->next, range);
#endif
}

#if VERSION >= 15
#if VERSION < 22
void splice_change(Arena *a, StatementBlockParse *tree, const
StatementBlockParse  *work,  PlainBlockParse  *dna,  DiffLine
range) {
#else
#if VERSION < 23
void splice_change(Arena *a, StatementBlockParse *tree, const
StatementBlockParse *work, DiffLine range, _Bool delete) {
#else
static int splice_change_delete;
void splice_change(Arena *a, StatementBlockParse *tree, const
StatementBlockParse *work, DiffLine range) {
#endif
#endif
  if (tree == NULL) {
    return;
  }
  if (tree->plain_block) {
    PlainBlockParse *ptr = tree->plain_block;
    while (ptr->next) {
      ptr = ptr->next;
    }
    int plain_begin = tree->plain_block->line.no;
    int plain_end = ptr->line.no + 1;

    /* Check for overlap */
#if VERSION < 22
    if ((range.tree.begin >= plain_begin && range.tree.begin
< plain_end) ||
        (range.tree.end > plain_begin && range.tree.end <=
plain_end)) {
#else
    if (((range.tree.begin >= plain_begin && range.tree.begin
< plain_end) ||
     (range.tree.end  >  plain_begin  &&  range.tree.end  <=
plain_end)) || (plain_begin >= range.tree.begin && plain_end
< range.tree.end)) {
#endif
      ptr = tree->plain_block;
      PlainBlockParse *parent = NULL;
           while  (ptr->line.no  ==  -1  ||  ptr->line.no  <
range.tree.begin) {
  parent = ptr;
  ptr = ptr->next;
      }

      PlainBlockParse *prefix = new (a, 1, PlainBlockParse);
      char *prefix_line_buf = new (a, 32, char);
       sprintf(prefix_line_buf, "#if VERSION < %d", version
+ 1);

      prefix->line.span.data = prefix_line_buf;
      prefix->line.span.len = strlen(prefix->line.span.data);
      prefix->next = ptr;
      if (parent == NULL) {
  tree->plain_block = prefix;
      } else {
        parent->next = prefix;
      }
      PlainBlockParse *back_ptr = ptr;
      while (back_ptr &&
             (back_ptr->line.no == -1 || back_ptr->line.no <
range.tree.end)) {
  parent = back_ptr;
  back_ptr = back_ptr->next;
      }
      PlainBlockParse *interlude = new (a, 1, PlainBlockParse);
      interlude->line.span = S("#else");
      parent->next = interlude;
      PlainBlockParse *suffix = new (a, 1, PlainBlockParse);
      suffix->line.span = S("#endif");
      suffix->next = back_ptr;
#if VERSION >= 22
#if VERSION < 23
      if (delete) {
  parent->next = suffix;
      }
#else

#endif
#endif
      DiffLineRange range_to_get = {
#if VERSION < 19
        .begin     =     line_map_reverse(ptr->line.no)     -
line_map_reverse(range.tree.begin) + range.work.begin,
     .end   =   line_map_reverse(parent->line.no)   +   1   -
line_map_reverse(range.tree.begin) + range.work.begin};
#else
          .begin = line_map_reverse(ptr->line.no) -
                                line_map_reverse(range.tree.begin)
+ range.work.begin,
      };
#if VERSION < 23
      range_to_get.end = range_to_get.begin + (range.work.end
- range.work.begin);
      if (range_to_get.end - range_to_get.begin >
          range.work.end - range.work.begin) {
      }
#else

      range_to_get.end =
                    range_to_get.begin  +  (range.work.end  -
range.work.begin);
        if (range_to_get.begin  >  splice_change_delete  &&
range_to_get.end <= range.work.end) {
  splice_change_delete = range_to_get.end;
          PlainBlockParse  *fresh_dna  =  get_line_range(a,
range_to_get, work);
  if (fresh_dna) {
    interlude->next = fresh_dna;
    while (fresh_dna->next) {
      fresh_dna = fresh_dna->next;
    }
    fresh_dna->next = suffix;
        } else {
    parent->next = suffix;
#endif
#endif
#if VERSION < 23
          PlainBlockParse  *fresh_dna  =  get_line_range(a,
range_to_get, work);
#endif
#if VERSION < 22
      interlude->next = fresh_dna;
      while (fresh_dna->next) {
  fresh_dna = fresh_dna->next;
#else
#if VERSION < 23
      if (fresh_dna) {
  if (!delete &&range_to_get.end >= range.work.end) {
    delete = 1;
  }
  interlude->next = fresh_dna;
  while (fresh_dna->next) {
    fresh_dna = fresh_dna->next;
#endif
  }
#if VERSION < 23
  fresh_dna->next = suffix;
#else
      } else {
  parent->next = suffix;
#endif
#endif
      }
#if VERSION < 22
      fresh_dna->next = suffix;
#endif
    }
  }
  if (!tree->if_block) {
    return;
  }
#if VERSION < 22
   splice_change(a,  tree->if_block->statement_block,  work,
dna, range);
#else
#if VERSION < 23
    splice_change(a,  tree->if_block->statement_block,  work,
range, delete);
#else
  if (evaluate_condition(&tree->if_block->line)) {
    splice_change(a, tree->if_block->statement_block, work,
range);
  } else {
    if (tree->if_block->else_block) {
        splice_change(a,  tree->if_block->else_block->block,
work, range);
    }
#endif
#endif
#if VERSION < 23
  if (tree->if_block->else_block) {
#endif
#if VERSION < 22
    splice_change(a, tree->if_block->else_block->block, work,
dna, range);
#else
#if VERSION < 23
    splice_change(a, tree->if_block->else_block->block, work,
range, delete);
#endif
#endif
  }
#if VERSION < 22
  splice_change(a, tree->if_block->next, work, dna, range);
#else
#if VERSION < 23
    splice_change(a,  tree->if_block->next,  work,  range,
delete);
#else
  splice_change(a, tree->if_block->next, work, range);
#endif
#endif
}

#if VERSION >= 17
void update_version_tree(StatementBlockParse *);

#endif
#endif
#endif
void diff(Arena *a, StatementBlockParse *tree) {
#endif
#endif
#endif
  char tree_file_name[] = "/tmp/tree.cXXXXXX";
  int fileno = mkstemp(tree_file_name);
  FILE *tree_file = fdopen(fileno, "w");
#if VERSION < 9
  dump_statement_block(tree_file, b, 0);
#else

#if VERSION < 10
  line_map = new (a, 1024 * 1024, int);
#endif
  current_line = 1;
#if VERSION < 10
  dump_statement_block(tree_file, tree, 0, version);
#else
#if VERSION < 11
  dump_statement_block(tree_file, tree, version);
#else
  dump_statement_block(tree_file, tree);
#endif
#endif
#endif
  fclose(tree_file);

  char work_file_name[] = "/tmp/work.cXXXXXX";
  fileno = mkstemp(work_file_name);
  FILE *work_file = fdopen(fileno, "w");
  /* Assuming C file under 1mb for the paper */
#if VERSION < 7
  char *buf = malloc(1024 * 1024);
#else
  char *buf = new (a, 1024 * 1024, char);
#endif
   unsigned long fssize = fread(buf, sizeof(char), 1024 *
1024, stdin);
  fwrite(buf, sizeof(char), fssize, work_file);
#if VERSION < 8
  free(buf);
#endif
  fclose(work_file);
#if VERSION >= 9

#endif
  /* Give the buffer some extra space */
  char cmdbuf[256];
      sprintf(cmdbuf,   "diff   %s   %s",   tree_file_name,
work_file_name);
#if VERSION < 8
  FILE *diff_output = popen(cmdbuf, "r");
  for (;;) {
    char outbuf[4096];
     unsigned long r = fread(outbuf, sizeof(char), 4096,
diff_output);
    if (r == 0) {
      break;
#else
  FILE *diff_out = popen(cmdbuf, "r");
  Str diff_output = read_from_popen(a, diff_out);
#if VERSION < 9
  fprintstr(stdout, diff_output);
#else

     StatementBlockParse  *work  =  parse_from_filename(a,
work_file_name);

  while (diff_output.len) {
    Cut c = cut(diff_output, '\n');
    DiffLine d = parse_diff_line(c.head);
#if VERSION < 10
      printf("%c %d %d - %d %d\n", d.mode, d.tree.begin,
d.tree.end, d.work.begin,
          d.work.end);
#else
#if VERSION < 11
       printf("%c  TREE  %d  %d  -  WORK  %d  %d\n",  d.mode,
d.tree.begin, d.tree.end,
           d.work.begin, d.work.end);

    PlainBlockParse *dr = get_line_range(a, d.work, work);
```

```c
    should_trim = 0;
    dump_plain_block(stderr, dr);
#endif
#endif
#if VERSION < 12
    if (d.mode != 'a') {
      fprintf(stderr, "Diff mode currently not supported");
      return;
    }
#endif
#if VERSION >= 11
    d.tree.begin = line_map[d.tree.begin];
#if VERSION < 12
    d.tree.end = line_map[d.tree.end];
#else
#if VERSION < 22
    d.tree.end = line_map[d.tree.end -  1] + 1;
#else
    d.tree.end = line_map[d.tree.end - 1] + 1;
    should_trim = 0;
#endif
#endif
#if VERSION < 15
    fprintf(stderr, "[%c] TREE %d %d - WORK %d %d\n", d.mode,
d.tree.begin, d.tree.end,
            d.work.begin, d.work.end);
#endif
#if VERSION < 12

    PlainBlockParse *dna = get_line_range(a, d.work, work);
    should_trim = 0;

    splice(a, tree, dna, d);
    dump_statement_block(stdout, tree);
#else
    if (d.mode == 'a') {
      PlainBlockParse *dna = get_line_range(a, d.work, work);
#if VERSION < 17
      should_trim = 0;

#endif
      splice(a, tree, dna, d);
#if VERSION < 17
      dump_statement_block(stdout, tree);
#endif
    } else if (d.mode == 'd') {
      splice_delete(a, tree, d.tree);
#if VERSION >= 15
#if VERSION < 17
      should_trim = 0;
      dump_statement_block(stdout, tree);
#endif
    } else if (d.mode == 'c') {
#if VERSION < 22
      PlainBlockParse *dna = get_line_range(a, d.work, work);
      splice_change(a, tree, work, dna, d);
#else
#if VERSION < 23
      splice_change(a, tree, work, d, 0);
#else
      splice_change_delete = -1;
      splice_change(a, tree, work, d);
#endif
#endif
#endif
#if VERSION < 17
      should_trim = 0;
      dump_statement_block(stdout, tree);
#endif
    } else {
      fprintf(stderr, "Diff mode currently not supported\n");
      return;
    }
#if VERSION >= 17
    should_trim = 0;
#endif
#endif
#endif
#if VERSION < 12

#endif
#if VERSION < 18
    diff_output = skip_lines(diff_output, d);
#else
    diff_output = skip_lines(diff_output);
#endif
  }
#if VERSION >= 17
  update_version_tree(tree);
  dump_statement_block(stdout, tree);
#endif
#endif
#endif
#endif
#if VERSION < 8
    }
#endif
#if VERSION < 5
    char buf[200];
#else
#if VERSION < 6
    char buf[2000];
#else
#if VERSION < 8
    fwrite(outbuf, sizeof(char), r, stdout);
#endif
#endif
#endif
#if VERSION < 6
    size_t s = read(p2[0], &buf, 200);
    sleep(1);
    buf[s] = '\0';
    printf("Read %ld: %s\n", s, buf);
#endif
#if VERSION < 8
  }
#endif
}

#endif
#if VERSION < 5
int main() {
#else
void print_usage(const char *name) {
    fprintf(stderr, "Usage: %s checkout\n%s commit <tree-
file>\n", name, name);
}

#if VERSION >= 16
/* Returns the version number, or -1 if none is given */
int get_version_args(int argc, char *argv[]) {
  for (int i = 1; i < argc - 1; i++) {
    if (!strcmp(argv[i], "-v")) {
      return atoi(argv[i + 1]);
    }
  }
  return -1;
}

/* Returns the version number from the block, or -1 if none
is given */
int get_version_tree(StatementBlockParse *b) {
  if (!b) {
    return -1;
  }
  if (b->plain_block) {
    return -1;
  }
  if (!b->if_block) {
    return -1;
  }
  if (!equals(b->if_block->line.variable, S("VERSION"))) {
    return -1;
  }
  if (b->if_block->line.operator != OP_NDEF) {
    return -1;
  }
    Str  line  =  b->if_block->statement_block->plain_block-
>line.span;
  /* Use offset to get the version number alone */
  line.data += 16;
  line.len -= 16;
  return parse_integer(line);
#if VERSION >= 17
}

static char version_tree_buf[32];

void update_version_tree(StatementBlockParse *b) {
  /* TODO: Add version tree if it doesn't exist */
   sprintf(version_tree_buf, "#define VERSION %d", version
+ 1);
  b->if_block->statement_block->plain_block->line.span.data
= version_tree_buf;
  b->if_block->statement_block->plain_block->line.span.len =
strlen(version_tree_buf);
#endif
}

#endif
int main(int argc, char *argv[]) {
  if (argc < 2) {
    print_usage(argv[0]);
    return 1;
  }
#endif
  /* Allocate a gigabyte for the whole program */
  ptrdiff_t buf_size = 1024 * 1024 * 1024;
  void *buf = malloc(buf_size);
  Arena a = {
#if VERSION < 5
      .begin = buf,
      .end = buf + buf_size,
  };
  Str file_contents = read_file(&a, stdin);
#else
#if VERSION < 10
    .begin = buf,
    .end = buf + buf_size,
#else
      .begin = buf,
      .end = buf + buf_size,
#endif
#endif
#if VERSION < 1
  Cut c = {0};
  c.tail = file_contents;
  while (c.tail.len) {
    c = cut(c.tail, '\n');
    Str line = trimleft(c.head);
    Line l = parse_line(line);
    printf("%d\n", l.line_type);
  }
#else
#if VERSION < 5
  ParserState p = {
    .file = file_contents,
#endif
  };
#if VERSION >= 10
  line_map = new (&a, 1024 * 1024, int);
#endif
#if VERSION < 5
  StatementBlockParse *s = parse_statement_block(&a, &p);
#else
  if (!strcmp("checkout", argv[1])) {
    Str file_contents = read_file(&a, stdin);
    ParserState p = {
      .file = file_contents,
    };
    StatementBlockParse *s = parse_statement_block(&a, &p);
#if VERSION < 6
    dump_statement_block(s, 0);
#else
#if VERSION < 9
    dump_statement_block(stdout, s, 0);
#else
#if VERSION < 16
    /* TODO: Get the version from the program */
#else
#if VERSION < 20
    version = get_version_tree(s);
#else
    version = get_version_args(argc, argv);
    if (version == -1) {
      version = get_version_tree(s);
    }
#endif
#endif
#if VERSION < 11
    const int version = 1;
#else
    dump_statement_block(stdout, s);
#endif
#if VERSION < 10
    dump_statement_block(stdout, s, 0, version);
#else
#if VERSION < 11
    dump_statement_block(stdout, s, version);
#endif
#endif
#endif
#endif
  } else if (!strcmp("commit", argv[1])) {
    if (argc < 3) {
      print_usage(argv[0]);
      goto end;
    }
    FILE *tree_file = fopen(argv[2], "r");
    Str file_contents = read_file(&a, tree_file);
#if VERSION >= 6
    ParserState p = {
        .file = file_contents,
    };
    StatementBlockParse *s = parse_statement_block(&a, &p);
#if VERSION >= 16
    version = get_version_tree(s);
#endif
#if VERSION < 7
    diff(s);
#else
    diff(&a, s);
#endif
#endif
    fclose(tree_file);
  } else {
    print_usage(argv[0]);
  }
 end:
#endif
#if VERSION < 2
  print_statement_block(s);
#else
#if VERSION < 3
  dump_statement_block(s);
#else
#if VERSION < 5
  dump_statement_block(s, 0);
#endif
#endif
#endif
#if VERSION < 5
  putchar('\n');
#endif
#endif
  free(buf);
#if VERSION >= 4
#if VERSION < 5
  diff();
#else
  return 0;
#endif
#endif
}
```

Bibliography

[1] JetBrains, “Team Tools - The State of Developer Ecosystem in 2023 Infographic.” [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/team-tools/

[2] Linus Torvalds, “Git.” [Online]. Available: https://git-scm.org/

[3] cppreference.com, “Preprocessor.” [Online]. Available: https://en.cppreference.com/w/c/preprocessor

[4] Wikipedia, “LL parser.” [Online]. Available: https://en.wikipedia.org/wiki/LL_parser

[5] Chris Wellons, “Robust Wavefront OBJ model parsing in C.” [Online]. Available: https://nullprogram.com/blog/2025/03/02/

[6] OpenBSD, [Online]. Available: https://man.openbsd.org/diff

[7] “Git - git-bundle Documentation.” [Online]. Available: https://git-scm.com/docs/git-bundle

[8] Matt Mahoney, “Silesia Open Source Compression Benchmark.” [Online]. Available: http://mattmahoney.net/dc/silesia.html