1. Introduction. This is the `GOTANGLE` program by Alexander Sychev, based on `CTANGLE` by Silvio Levy and Donald E. Knuth.

The "banner line" defined here should be changed whenever `GOTANGLE` is modified.

```
⟨ Constants 1 ⟩ ≡
  banner = "This␣is␣GOTANGLE␣(Version␣0.82)\n"
```

See also sections 4, 100, 106, 111, 125, 129, 131, 144, and 147.

This code is used in section 2.

2.

```
package main
import(
  ⟨ Import packages 13 ⟩
)
const(
  ⟨ Constants 1 ⟩
)
⟨ Typedef declarations 92 ⟩
⟨ Global variables 93 ⟩
```

3. `GOTANGLE` has a fairly straightforward outline. It operates in two phases: first it reads the source file, saving the Go code in compressed form; then it shuffles and outputs the code.

```
func main(){
  common_init()
  ⟨ Set initial values 99 ⟩
  if show_banner() {
    fmt.Print(banner)      /* print a "banner line" */
  }
  phase_one()      /* read all the user's text and compress it into tok_mem */
  phase_two()      /* output the contents of the compressed tables */
  os.Exit(wrap_up())      /* and exit gracefully */
}
```

4.

```
⟨ Constants 1 ⟩ +≡
  max_texts = 2500      /* number of replacement texts, must be less than 10240 */
```

5. The next few sections contain stuff from the file `gocommon.w` that must be included in both `gotangle.w` and `goweave.w`.

6. Introduction in common code. Next few sections contain code common to both `GOTANGLE` and `GOWEAVE`, which roughly concerns the following problems: character uniformity, input routines, error handling and parsing of command line.

```
const(
  ⟨ Common constants 10 ⟩
)
⟨ Definitions that should agree with GOTANGLE and GOWEAVE 12 ⟩
⟨ Other definitions 7 ⟩
```

7. `GOWEAVE` operates in three phases: First it inputs the source file and stores cross-reference data, then it inputs the source once again and produces the TEX output file, and finally it sorts and outputs the index. Similarly, `GOTANGLE` operates in two phases. The global variable *phase* tells which phase we are in.

```
⟨ Other definitions 7 ⟩ ≡
  var phase int      /* which phase are we in? */
```

See also section 18.

This code is used in section 6.

8. There's an initialization procedure that gets both `GOTANGLE` and `GOWEAVE` off to a good start. We will fill in the details of this procedure later.

```
func common_init(){
  ⟨ Initialize pointers 44 ⟩
  ⟨ Set the default options common to GOTANGLE and GOWEAVE 82 ⟩
  ⟨ Scan arguments and open output files 89 ⟩
}
```

9. A few character pairs are encoded internally as single characters, using the definitions below. These definitions are consistent with an extension of ASCII code originally developed at MIT and explained in Appendix C of *The TEXbook*; thus, users who have such a character set can type things like ≠ and ∧ instead of `!=` and `&&`. (However, their files will not be too portable until more people adopt the extended code.). Actually, for `GOWEB` these codes is not significant, because `GOWEB` operates with UTF8 encoded sources.

10. ⟨ Common constants 10 ⟩ ≡

```
  and_and rune = °4      /* '&&'; corresponds to MIT's ∧ */
  lt_lt rune = °20       /* '<<'; corresponds to MIT's ⊂ */
  gt_gt rune = °21       /* '>>'; corresponds to MIT's ⊃ */
  plus_plus rune = °200  /* '++'; corresponds to MIT's ↑ */
  minus_minus rune = °201  /* '--'; corresponds to MIT's ↓ */
  col_eq rune = °207     /* ':=' */
  not_eq rune = °32      /* '!='; corresponds to MIT's ≠ */
  lt_eq rune = °34       /* '<='; corresponds to MIT's ≤ */
  gt_eq rune = °35       /* '>='; corresponds to MIT's ≥ */
  eq_eq rune = °36       /* '=='; corresponds to MIT's ≡ */
  or_or rune = °37       /* '||'; corresponds to MIT's ∨ */
  dot_dot_dot rune = °202  /* '...' */
  begin_comment rune = '\t'  /* tab marks will not appear */
  and_not rune = °10     /*'&^';*/
  direct rune = °203     /*'<-';*/
  begin_short_comment rune = °31  /* short comment */
```

See also sections 31, 42, 55, 63, and 65.

This code is used in section 6.

11. Input routines. The lowest level of input to the GOWEB programs is performed by *input_ln*, which must be told which file to read from. The return value of *input_ln* is nil if the read is successful and not nil otherwise (generally this means the file has ended). The *buffer* always contains whole string without ending newlines.

12. ⟨ Definitions that should agree with GOTANGLE and GOWEAVE 12 ⟩ ≡

```
var buffer []rune      /* where each line of input goes */
var loc int = 0        /* points to the next character to be read from the buffer */
var section_text []rune      /* name being sought for */
var id []rune      /* slice pointed to the current identifier */
```

See also sections 17, 32, 40, 43, 68, 81, and 88.

This code is used in section 6.

13. ⟨ Import packages 13 ⟩ ≡

```
"io"
"bytes"
```

See also sections 16, 20, 27, and 34.

This code is used in section 2.

14.

```
    /* copies a line into buffer or returns error */
func input_ln(fp *bufio.Reader) error{
  var prefix bool
  var err error
  var buf []byte
  var b []byte
  buffer = nil
  for buf, prefix, err = fp.ReadLine(); err ≡ nil ∧ prefix; b, prefix, err = fp.ReadLine() {
    buf = append(buf, b...)
  }
  if len(buf)⟩0 {
    buffer = bytes.Runes(buf)
  }
  if err ≡ io.EOF ∧ len(buffer) ≠ 0 {
    return nil
  }
  if err ≡ nil ∧ len(buffer) ≡ 0 {
    buffer = append(buffer, '␣')
  }
  return err
}
```

15. Now comes the problem of deciding which file to read from next. Recall that the actual text that GOWEB should process comes from two *bufio.Reader*: a *file*[0], which can contain possibly nested include commands @i, and a *change_file*, which might also contain includes. The *file*[0] together with the currently open include files form a stack *file*, whose names are stored in a parallel stack *file_name*. The boolean *changing* tells whether or not we're reading from the *change_file*.

The line number of each open file is also kept for error reporting and for the benefit of GOTANGLE.

16. ⟨ Import packages 13 ⟩ +≡

```
"bufio"
```

17. ⟨ Definitions that should agree with `GOTANGLE` and `GOWEAVE` 12 ⟩ +≡

```
var include_depth int      /* current level of nesting */
var file []*bufio.Reader   /* stack of non-change files */
var change_file *bufio.Reader   /* change file */
var file_name []string
    /* stack of non-change file names */
var change_file_name string = "/dev/null"   /* name of change file */
var alt_file_name string   /* alternate name to try */
var line []int   /* number of current line in the stacked files */
var change_line int   /* number of current line in change file */
var change_depth int   /* where @y originated during a change */
var input_has_ended bool   /* if there is no more input */
var changing bool   /* if the current line is from change_file */
```

18. When *changing* ≡ **false**, the next line of *change_file* is kept in *change_buffer*, for purposes of comparison with the next line of *file*[*include_depth*]. After the change file has been completely input, we set *change_limit* = 0, so that no further matches will be made.

⟨ Other definitions 7 ⟩ +≡

```
var change_buffer []rune   /* next line of change_file */
```

19. Procedure *prime_the_change_buffer* sets *change_buffer* in preparation for the next matching operation. Since blank lines in the change file are not used for matching, we have (*change_limit* ≡ 0 ∧ ¬*changing*) if and only if the change file is exhausted. This procedure is called only when *changing* is true; hence error messages will be reported correctly.

```
func prime_the_change_buffer(){
  change_buffer = nil
  ⟨ Skip over comment lines in the change file; return if end of file 21 ⟩
  ⟨ Skip to the next nonblank line; return if end of file 22 ⟩
  ⟨ Move buffer to change_buffer 23 ⟩
}
```

20. ⟨ Import packages 13 ⟩ +≡

```
"unicode"
```

21. While looking for a line that begins with `@x` in the change file, we allow lines that begin with `@`, as long as they don't begin with `@y`, `@z`, or `@i` (which would probably mean that the change file is fouled up).

```
⟨ Skip over comment lines in the change file; return if end of file 21 ⟩ ≡
  for true {
    change_line++
    if err := input_ln(change_file); err ≠ nil {
      return
    }
    if len(buffer)⟨2 {
      continue
    }
    if buffer[0] ≠ '@' {
      continue
    }
    if unicode.IsUpper(buffer[1]) {
      buffer[1] = unicode.ToLower(buffer[1])
    }
    if buffer[1] ≡ 'x' {
      break
    }
    if buffer[1] ≡ 'y' ∨ buffer[1] ≡ 'z' ∨ buffer[1] ≡ 'i' {
      loc = 2
      err_print("!␣Missing␣@x␣in␣change␣file")
    }
  }
```

This code is used in section 19.

22. Here we are looking at lines following the `@x`.

```
⟨ Skip to the next nonblank line; return if end of file 22 ⟩ ≡
  for true {
    change_line++
    if err := input_ln(change_file); err ≠ nil {
      err_print("!␣Change␣file␣ended␣after␣@x")
      return
    }
    if len(buffer) ≠ 0 {
      break
    }
  }
```

This code is used in section 19.

23.

```
⟨ Move buffer to change_buffer 23 ⟩ ≡
  {
    change_buffer = buffer
    buffer = nil
  }
```

This code is used in sections 19 and 26.

24. The following procedure is used to see if the next change entry should go into effect; it is called only when *changing* is false. The idea is to test whether or not the current contents of *buffer* matches the current contents of *change_buffer*. If not, there's nothing more to do; but if so, a change is called for: All of the text down to the `@y` is supposed to match. An error message is issued if any discrepancy is found. Then the procedure prepares to read the next line from *change_file*.

When a match is found, the current section is marked as changed unless the first line after the `@x` and after the `@y` both start with either `'@*'` or `'@␣'` (possibly preceded by whitespace).

This procedure is called only when the current line is nonempty.

```
func if_section_start_make_pending(b bool){
  for loc = 0; loc⟨len(buffer) ∧ unicode.IsSpace(buffer[loc]); loc++ {}
  if len(buffer) ≥ 2 ∧ buffer[0] ≡ '@' ∧ (unicode.IsSpace(buffer[1]) ∨ buffer[1] ≡ '*') {
    change_pending = b
  }
}
```

25. We need a function to compare buffers of runes. It behaves like the classic *strcmp* function: it returns -1, 0 or 1 if a left buffer is less, equal or more of a right buffer.

```
func compare_runes(l []rune, r []rune) int{
  i := 0
  for ; i⟨len(l) ∧ i⟨len(r) ∧ l[i] ≡ r[i]; i++ {}
  if i ≡ len(r) {
    if i ≡ len(l) {
      return 0
    } else {
      return −1
    }
  } else {
    if i ≡ len(l) {
      return 1
    } else if l[i]⟨r[i] {
      return −1
    } else {
      return 1
    }
  }
  return 0
}
```

26.

```
    /* switches to change_file if the buffers match */
  func check_change(){
    n := 0      /* the number of discrepancies found */
    if compare_runes(buffer, change_buffer) ≠ 0 {
      return
    }
    change_pending = false
    if ¬changed_section[section_count] {
      if_section_start_make_pending(true)
      if ¬change_pending {
        changed_section[section_count] = true
      }
    }
    for true {
      changing = true
      print_where = true
      change_line++
      if err := input_ln(change_file); err ≠ nil {
        err_print("! Change file ended before @y")
        change_buffer = nil
        changing = false
        return
      }
      if len(buffer)⟩1 ∧ buffer[0] ≡ '@' {
        var xyz_code rune
        if unicode.IsUpper(buffer[1]) {
          xyz_code = unicode.ToLower(buffer[1])
        } else {
          xyz_code = buffer[1]
        }
        ⟨ If the current line starts with @y, report any discrepancies and return 28 ⟩
      }
      ⟨ Move buffer to change_buffer 23 ⟩
      changing = false
      line[include_depth]++
      for input_ln(file[include_depth]) ≠ nil {      /* pop the stack or quit */
        if include_depth ≡ 0 {
          err_print("! GOWEB file ended during a change")
          input_has_ended = true
          return
        }
        include_depth--
        line[include_depth]++
      }
      if compare_runes(buffer, change_buffer) ≠ 0 {
        n++
      }
    }
  }
```

27. ⟨ Import packages 13 ⟩ +≡

```
"fmt"
```

28. ⟨ If the current line starts with `@y`, report any discrepancies and **return** 28 ⟩ ≡

```
if xyz_code ≡ 'x' ∨ xyz_code ≡ 'z' {
    loc = 2
    err_print("! Where is the matching @y?")
} else if xyz_code ≡ 'y' {
    if n⟩0 {
        loc = 2
        err_print("! Hmm... %d of the preceding lines failed to match", n)
    }
    change_depth = include_depth
    return
}
```

This code is used in section 26.

29. The *reset_input* procedure, which gets `GOWEB` ready to read the user's `GOWEB` input, is used at the beginning of phase one of `GOTANGLE`, phases one and two of `GOWEAVE`.

```
func reset_input(){
    loc = 0
    file = file[:0]
    ⟨ Open input files 30 ⟩
    include_depth = 0
    line = line[:0]
    line = append(line, 0)
    change_line = 0
    change_depth = include_depth
    changing = true
    prime_the_change_buffer()
    changing = ¬changing
    loc = 0
    input_has_ended = false
}
```

30. The following code opens the input files.

```
⟨ Open input files 30 ⟩ ≡
  if wf, err := os.Open(file_name[0]); err ≠ nil {
    file_name[0] = alt_file_name
    if wf, err = os.Open(file_name[0]); err ≠ nil {
      fatal("! Cannot open input file ", file_name[0])
    } else {
      file = append(file, bufio.NewReader(wf))
    }
  } else {
    file = append(file, bufio.NewReader(wf))
  }
  if cf, err := os.Open(change_file_name); err ≠ nil {
    fatal("! Cannot open change file ", change_file_name)
  } else {
    change_file = bufio.NewReader(cf)
  }
```

This code is used in section 29.

31. The *get_line* procedure is called when $loc \geq \mathbf{len}(buffer)$; it puts the next line of merged input into the buffer and updates the other variables appropriately.

This procedure returns $\neg input\_has\_ended$ because we often want to check the value of that variable after calling the procedure.

If we've just changed from the $file[include\_depth]$ to the *change_file*, or if the $file[include\_depth]$ has changed, we tell GOTANGLE to print this information in the Go file by means of the *print_where* flag.

```
⟨ Common constants 10 ⟩ +≡
  max_sections = 2000      /* number of identifiers, strings, section names; must be less than 10240 */
```

32.

```
⟨ Definitions that should agree with GOTANGLE and GOWEAVE 12 ⟩ +≡
  var section_count int32      /* the current section number */
  var changed_section [max_sections]bool      /* is the section changed? */
  var change_pending bool
    /* if the current change is not yet recorded in changed_section[section_count] */
  var print_where bool = false      /* should GOTANGLE print line and file info? */
```

33.

```
func get_line() bool {      /* inputs the next line */
restart : if changing ∧ include_depth ≡ change_depth{⟨ Read from change_file and maybe turn off
    changing 37 ⟩}
if ¬changing ∨ include_depth⟩change_depth {
  ⟨ Read from file[include_depth] and maybe turn on changing 36 ⟩
  if changing ∧ include_depth ≡ change_depth {
    goto restart
  }
}
if input_has_ended {
  return false
}
loc = 0
if len(buffer) ≥ 2 ∧ buffer[0] ≡ '@' ∧ (buffer[1] ≡ 'i' ∨ buffer[1] ≡ 'I') {
  loc = 2
  for loc⟨len(buffer) ∧ unicode.IsSpace(buffer[loc]) {
    loc++
  }
  if loc ≥ len(buffer) {
    err_print("! Include file name not given")
    goto restart
  }
  include_depth++      /* push input stack */
  ⟨ Try to open include file, abort push if unsuccessful, go to restart 35 ⟩
}
return true
}
```

34. When an `@i` line is found in the *file*[*include_depth*], we must temporarily stop reading it and start reading from the named include file. The `@i` line should give a complete file name with or without double quotes. If the environment variable `GOWEBINPUTS` is set `GOWEB` will look for include files in the colon-separated directories thus named, if it cannot find them in the current directory. The remainder of the `@i` line after the file name is ignored.

```
⟨ Import packages 13 ⟩ +≡
  "os"
  "strings"
```

35. ⟨ Try to open include file, abort push if unsuccessful, go to *restart* 35 ⟩ ≡

```
{
  l := loc
  if buffer[loc] ≡ '"' {
    loc++
    l++
    for loc⟨len(buffer) ∧ buffer[loc] ≠ '"' {
      loc++
    }
  } else {
    for loc⟨len(buffer) ∧ ¬unicode.IsSpace(buffer[loc]) {
      loc++
    }
  }
  file_name = append(file_name, string(buffer[l:loc]))
  if f, err := os.Open(file_name[include_depth]); err ≡ nil {
    file = append(file, bufio.NewReader(f))
    line = append(line, 0)
    print_where = true
    goto restart      /* success */
  }
  temp_file_name := os.Getenv("GOWEBINPUTS")
  if len(temp_file_name) ≠ 0 {
    for _, fn := range strings.Split(temp_file_name, ":") {
      file_name[include_depth] = fn + "/" + file_name[include_depth]
      if f, err := os.Open(file_name[include_depth]); err ≡ nil {
        file = append(file, bufio.NewReader(f))
        line = append(line, 0)
        print_where = true
        goto restart      /* success */
      }
    }
  }
  file_name = file_name[:include_depth]
  file = file[:include_depth]
  line = line[:include_depth]
  include_depth--
  err_print("! Cannot open include file")
  goto restart
}
```

This code is used in section 33.

36. ⟨ Read from *file*[*include_depth*] and maybe turn on *changing* 36 ⟩ ≡

```
{
  line[include_depth]++
  for input_ln(file[include_depth]) ≠ nil {      /* pop the stack or quit */
    print_where = true
    if include_depth ≡ 0 {
      input_has_ended = true
      break
    } else {
      file[include_depth] = nil
      file_name = file_name[:include_depth]
      file = file[:include_depth]
      line = line[:include_depth]
      include_depth--
      if changing ∧ include_depth ≡ change_depth {
        break
      }
      line[include_depth]++
    }
  }
  if ¬changing ∧ ¬input_has_ended {
    if len(buffer) ≡ len(change_buffer) {
      if buffer[0] ≡ change_buffer[0] {
        if len(change_buffer)⟩0 {
          check_change()
        }
      }
    }
  }
}
```

This code is used in section 33.

37. ⟨ Read from *change_file* and maybe turn off *changing* 37 ⟩ ≡

```
{
  change_line++
  if input_ln(change_file) ≠ nil {
    err_print("!␣Change␣file␣ended␣without␣@z")
    buffer = append(buffer, []rune("@z")...)
  }
  if len(buffer)⟩0 {      /* check if the change has ended */
    if change_pending {
      if_section_start_make_pending(false)
      if change_pending {
        changed_section[section_count] = true
        change_pending = false
      }
    }
    if len(buffer) ≥ 2 ∧ buffer[0] ≡ '@' {
      if unicode.IsUpper(buffer[1]) {
        buffer[1] = unicode.ToLower(buffer[1])
      }
      if buffer[1] ≡ 'x' ∨ buffer[1] ≡ 'y' {
        loc = 2
        err_print("!␣Where␣is␣the␣matching␣@z?")
      } else if buffer[1] ≡ 'z' {
        prime_the_change_buffer()
        changing = ¬changing
        print_where = true
      }
    }
  }
}
```

This code is used in section 33.

38. At the end of the program, we will tell the user if the change file had a line that didn't match any relevant line in *file*[0].

```
func check_complete(){
  if len(change_buffer)⟩0 {      /* changing is false */
    buffer = change_buffer
    change_buffer = nil
    changing = true
    change_depth = include_depth
    loc = 0
    err_print("!␣Change␣file␣entry␣did␣not␣match")
  }
}
```

39. Storage of names and strings. Both `GOWEAVE` and `GOTANGLE` store identifiers, section names and other strings in a large array *name_dir*, whose elements are structures of type *name_info*, containing a slice of runes with text information and other data.

40. ⟨ Definitions that should agree with `GOTANGLE` and `GOWEAVE` 12 ⟩ +≡

```
type name_info struct{
  name []rune
  ⟨ More elements of name_info structure 41 ⟩
}    /* contains information about an identifier or section name */
type name_index int    /* index into array of name_infos */
var name_dir []name_info    /* information about names */
var name_root int32
```

41. The names of identifiers are found by computing a hash address h and then looking at strings of bytes signified by the indexes $name\_dir[hash[h]]$, $name\_dir[hash[h]].llink$, $name\_dir[name\_dir[hash[h]].llink].llink$, ..., until either finding the desired name or encountering -1.

⟨ More elements of *name_info* structure 41 ⟩ ≡

```
llink int32
```

See also sections 50 and 94.

This code is used in section 40.

42. The hash table itself consists of *hash_size* indexes, and is updated by the *id_lookup* procedure, which finds a given identifier and returns the appropriate index. The matching is done by the function *names_match*, which is slightly different in `GOWEAVE` and `GOTANGLE`. If there is no match for the identifier, it is inserted into the table.

⟨ Common constants 10 ⟩ +≡

```
hash_size = 353    /* should be prime */
```

43. ⟨ Definitions that should agree with `GOTANGLE` and `GOWEAVE` 12 ⟩ +≡

```
var hash [hash_size]int32    /* heads of hash lists */
var h int32    /* index into hash-head array */
```

44. ⟨ Initialize pointers 44 ⟩ ≡

```
for i, _ := range hash {
  hash[i] = -1
}
```

See also section 51.

This code is used in section 8.

45. Here is the main procedure for finding identifiers:

```
    /* looks up a string in the identifier table */
func id_lookup(
  id []rune,    /* string with id */
  t int32    /* the ilk; used by GOWEAVE only */) int32{
  ⟨ Compute the hash code h 46 ⟩
  ⟨ Compute the name location p 47 ⟩
  if p ≡ -1 {
    ⟨ Enter a new name into the table at position p 49 ⟩
  }
  return p
}
```

46. A simple hash code is used: If the sequence of character codes is $c_1 c_2 \ldots c_n$, its hash value will be

$$(2^{n-1}c_1 + 2^{n-2}c_2 + \cdots + c_n) \bmod hash\_size.$$

⟨ Compute the hash code h 46 ⟩ ≡
```
h := id[0]
for i := 1; i⟨len(id); i++ {
  h = (h + h + id[i]) % hash_size
}
```
This code is used in section 45.

47. If the identifier is new, it will be placed in the end of *name_dir*, otherwise p will point to its existing location.

⟨ Compute the name location p 47 ⟩ ≡
```
p := hash[h]
for p ≠ −1 ∧ ¬names_match(p, id, t) {
  p = name_dir[p].llink
}
if p ≡ −1 {
  p := int32(len(name_dir))      /* the current identifier is new */
  name_dir = append(name_dir, name_info{})
  name_dir[p].llink = −1
  init_node(p)
  name_dir[p].llink = hash[h]
  hash[h] = p      /* insert p at beginning of hash list */
}
```
This code is used in section 45.

48. The information associated with a new identifier must be initialized in a slightly different way in `GOWEAVE` than in `GOTANGLE`; both should implement the `Initialization of a new identifier` section.

49. ⟨ Enter a new name into the table at position p 49 ⟩ ≡
```
p = int32(len(name_dir) − 1)
name_dir[p].name = append(name_dir[p].name, id ...)
⟨ Initialization of a new identifier 97 ⟩
```
This code is used in section 45.

50. The names of sections are stored in *name_dir* together with the identifier names, but a hash table is not used for them because `GOTANGLE` needs to be able to recognize a section name when given a prefix of that name. A conventional binary search tree is used to retrieve section names, with fields called *llink* and *rlink* . The root of this tree is stored in *name_root*.

⟨ More elements of *name_info* structure 41 ⟩ +≡
```
ispref bool      /* prefix flag */
rlink int32      /* right link in binary search tree for section names */
```

51. ⟨ Initialize pointers 44 ⟩ +≡
```
name_root = −1      /* the binary search tree starts out with nothing in it */
```

52. If p is a *name_dir* index variable, as we have seen, *name_dir*$[p]$.*name* is the area where the name corresponding to p is stored. However, if p refers to a section name, the name may need to be stored in chunks, because it may "grow": a prefix of the section name may be encountered before the full name. Furthermore we need to know the length of the shortest prefix of the name that was ever encountered.

We solve this problem by inserting **int32** at *name_dir*$[p]$.*name*, representing the length of the shortest prefix, when p is a section name. Furthermore, the *ispref* field will be true if p is a prefix. In the latter case, the name pointer $p+1$ will allow us to access additional chunks of the name: The second chunk will begin at the name pointer *name_dir*$[p+1]$.*llink*, and if it too is a prefix (ending with blank) its *llink* will point to additional chunks in the same way. Null links are represented by -1.

```
func get_section_name(p int32) (dest []rune, complete bool){
  q := p + 1
  for p ≠ −1 {
    dest = append(dest, name_dir[p].name[1:]...)
    if name_dir[p].ispref {
      p = name_dir[q].llink
      q = p
    } else {
      p = −1
      q = −2
    }
  }
  complete = true
  if q ≠ −2 {
    complete = false      /* complete name not yet known */
  }
  return
}
```

53.

```
func sprint_section_name(p int32) string{
  s, c := get_section_name(p)
  str := string(s)
  if ¬c {
    str += "..."      /* complete name not yet known */
  }
  return str
}
```

54.

```
func print_prefix_name(p int32) (str string){
  l := name_dir[p].name[0]
  str = fmt.Sprint(string(name_dir[p].name[1:]))
  if int(l)⟨len(name_dir[p].name) {
    str += "..."
  }
  return
}
```

55. When we compare two section names, we'll need a function to looking for prefixes and extensions too.

```
⟨ Common constants 10 ⟩ +≡
  less = 0      /* the first name is lexicographically less than the second */
  equal = 1     /* the first name is equal to the second */
  greater = 2   /* the first name is lexicographically greater than the second */
  prefix = 3    /* the first name is a proper prefix of the second */
  extension = 4 /* the first name is a proper extension of the second */
```

56.

```
    /* fuller comparison than strcmp */
  func web_strcmp(
    j []rune,     /* first string */
    k []rune      /* second string */) int{
    i := 0
    for ; i⟨len(j) ∧ i⟨len(k) ∧ j[i] ≡ k[i]; i++ {}
    if i ≡ len(k) {
      if i ≡ len(j) {
        return equal
      } else {
        return extension
      }
    } else {
      if i ≡ len(j) {
        return prefix
      } else if j[i]⟨k[i] {
        return less
      } else {
        return greater
      }
    }
    return equal
  }
```

57. Adding a section name to the tree is straightforward if we know its parent and whether it's the *rlink* or *llink* of the parent. As a special case, when the name is the first section being added, we set the "parent" to −1. When a section name is created, it has only one chunk, which however may be just a prefix; the full name will hopefully be unveiled later. Obviously, prefix length starts out as the length of the first chunk, though it may decrease later.

The information associated with a new node must be initialized differently in `GOWEAVE` and `GOTANGLE`; hence the *init_node* procedure, which is defined differently in `goweave.w` and `gotangle.w`.

```
    /* install a new node in the tree */
func add_section_name(
  par int32,      /* parent of new node */
  c int,      /* right or left? */
  name []rune,      /* section name */
  ispref bool      /* are we adding a prefix or a full name? */) int32{
  p := int32(len(name_dir))      /* new node */
  name_dir = append(name_dir, name_info{})
  name_dir[p].llink = -1
  init_node(p)
  if ispref {
    name_dir = append(name_dir, name_info{})
    name_dir[p + 1].llink = -1
    init_node(p + 1)
  }
  name_dir[p].ispref = ispref
  name_dir[p].name = append(name_dir[p].name, int32(len(name)))      /* length of section name */
  name_dir[p].name = append(name_dir[p].name, name...)
  name_dir[p].llink = -1
  name_dir[p].rlink = -1
  init_node(p)
  if par == -1 {
    name_root = p
  } else {
    if c == less {
      name_dir[par].llink = p
    } else {
      name_dir[par].rlink = p
    }
  }
  return p
}
```

58.

```
func extend_section_name(
  p int32,      /* index name to be extended */
  text []rune,      /* extension text */
  ispref bool      /* are we adding a prefix or a full name? */){
  q := p + 1
  for name_dir[q].llink ≠ −1 {
    q = name_dir[q].llink
  }
  np := int32(len(name_dir))
  name_dir[q].llink = np
  name_dir = append(name_dir, name_info{})
  name_dir[np].llink = −1
  init_node(np)
  name_dir[np].name = append(name_dir[np].name, int32(len(text)))
    /* length of section name */
  name_dir[np].name = append(name_dir[np].name, text...)
  name_dir[np].ispref = ispref
}
```

59. The *section_lookup* procedure is supposed to find a section name that matches a new name, installing the new name if it doesn't match an existing one. A "match" means that the new name exactly equals or is a prefix or extension of a name in the tree.

```
    /* find or install section name in tree */
func section_lookup(
  name []rune,      /* new name */
  ispref bool      /* is the new name a prefix or a full name? */) int32{
  c := less      /* comparison between two names*/
  p := name_root      /* current node of the search tree */
  var q int32 = −1      /* another place to look in the tree */
  var r int32 = −1      /* where a match has been found */
  var par int32 = −1      /* parent of p, if r is NULL; otherwise parent of r */
  name_len := len(name)
  ⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 60 ⟩
  ⟨ If no match found, add new name to tree 61 ⟩
  ⟨ If one match found, check for compatibility and return match 62 ⟩
  return −1
}
```

60. A legal new name matches an existing section name if and only if it matches the shortest prefix of that section name. Therefore we can limit our search for matches to shortest prefixes, which eliminates the need for chunk-chasing at this stage.

```
⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 60 ⟩ ≡
  for p ≠ −1 {      /* compare shortest prefix of p with new name */
    c = web_strcmp(name, name_dir[p].name[1:])
    if c ≡ less ∨ c ≡ greater {      /* new name does not match p */
      if r ≡ −1 {      /* no previous matches have been found */
        par = p
      }
      if c ≡ less {
        p = name_dir[p].llink
      } else {
        p = name_dir[p].rlink
      }
    } else {      /* new name matches p */
      if r ≠ −1 {      /* and also r: illegal */
        err_print("! Ambiguous prefix: matches <%s>\n and <%s>", print_prefix_name(p),
            print_prefix_name(r))
        return 0      /* the unsection */
      }
      r = p      /* remember match */
      p = name_dir[p].llink      /* try another */
      q = name_dir[r].rlink      /* we'll get back here if the new p doesn't match */
    }
    if p ≡ −1 {
      p = q
      q = −1      /* q held the other branch of r */
    }
  }
```

This code is used in section 59.

61. ⟨ If no match found, add new name to tree 61 ⟩ ≡

```
  if r ≡ −1 {      /* no matches were found */
    return add_section_name(par, c, name, ispref)
  }
```

This code is used in section 59.

62. Although error messages are given in anomalous cases, we do return the unique best match when a discrepancy is found, because users often change a title in one place while forgetting to change it elsewhere.

```
⟨ If one match found, check for compatibility and return match 62 ⟩ ≡
  first, cmp := section_name_cmp(name, r)
  switch cmp {
      /* compare all of r with new name */
    case prefix:
      if ¬ispref {
        err_print("! New name is a prefix of <%s>", sprint_section_name(r))
      } else if name_len⟨int(name_dir[r].name[0]) {
        name_dir[r].name[0] = int32(len(name) − first)
      }
      fallthrough
    case equal:
      return r
    case extension:
      if ¬ispref ∨ first⟨len(name) {
        extend_section_name(r, name[first:], ispref)
      }
      return r
    case bad_extension:
      err_print("! New name extends <%s>", sprint_section_name(r))
      return r
    default:      /* no match: illegal */
      err_print("! Section name incompatible with <%s>,\n which abbreviates <%s>",
          print_prefix_name(r), sprint_section_name(r))
      return r
  }
```

This code is used in section 59.

63. The return codes of *section_name_cmp*, which compares a string with the full name of a section, are those of *web_strcmp* plus *bad_extension*, used when the string is an extension of a supposedly already complete section name. This function has a side effect when the comparison string is an extension: It advances the address of the first character of the string by an amount equal to the length of the known part of the section name.

The name `@<foo...@>` should be an acceptable "abbreviation" for `@<foo@>`. If such an abbreviation comes after the complete name, there's no trouble recognizing it. If it comes before the complete name, we simply append a null chunk. This logic requires us to regard `@<foo...@>` as an "extension" of itself.

```
⟨ Common constants 10 ⟩ +≡
  bad_extension = 5
```

64.

```
func section_name_cmp(
  name []rune,      /* comparison string */
  r int32      /* section name being compared */) (int, int){
  q := r + 1      /* access to subsequent chunks */
  var ispref bool      /* is chunk r a prefix? */
  first := 0
  for true {
    if name_dir[r].ispref {
      ispref = true
      q = name_dir[q].llink
    } else {
      ispref = false
      q = −1
    }
    c := web_strcmp(name, name_dir[r].name[1:])
    switch c {
      case equal:
        if q ≡ −1 {
          if ispref {
            return first + len(name_dir[r].name[1:]), extension      /* null extension */
          } else {
            return first, equal
          }
        } else {
          if compare_runes(name_dir[q].name, name_dir[q + 1].name) ≡ 0 {
            return first, equal
          } else {
            return first, prefix
          }
        }
      case extension:
        if ¬ispref {
          return first, bad_extension
        }
        first += len(name_dir[r].name[1:])
        if q ≠ −1 {
          name = name[len(name_dir[r].name[1:]):]
          r = q
          continue
        }
        return first, extension
      default:
        return first, c
    }
  }
  return −2, −1
}
```

65. Reporting errors to the user. A global variable called *history* will contain one of four values at the end of every run: *spotless* means that no unusual messages were printed; *harmless_message* means that a message of possible interest was printed but no serious errors were detected; *error_message* means that at least one error was found; *fatal_message* means that the program terminated abnormally. The value of *history* does not influence the behavior of the program; it is simply computed for the convenience of systems that might want to use such information.

```
⟨ Common constants 10 ⟩ +≡
  spotless = 0      /* history value for normal jobs */
  harmless_message = 1      /* history value when non-serious info was printed */
  error_message = 2      /* history value when an error was noted */
  fatal_message = 3      /* history value when we had to stop prematurely */
```

66.

```
func mark_harmless(){
  if history ≡ spotless {
    history = harmless_message
  }
}
```

67.

```
func mark_error(){
  history = error_message
}
```

68.

```
⟨ Definitions that should agree with GOTANGLE and GOWEAVE 12 ⟩ +≡
  var history int = spotless      /* indicates how bad this run was */
```

69. The command '*err_print*(`"!␣Error␣message"`)' will report a syntax error to the user, by printing the error message at the beginning of a new line and then giving an indication of where the error was spotted in the source file. Note that no period follows the error message, since the error routine will automatically supply a period. A newline is automatically supplied if the string begins with `"!"`.

```
      /* prints '.' and location of error message */
func err_print(s string, a ...interface{}){
  var l int      /* pointers into buffer */
  if len(s)⟩0 ∧ s[0] ≡ '!' {
    fmt.Fprintf(os.Stdout, "\n\n" + s, a...)
  } else {
    fmt.Fprintf(os.Stdout, "\n" + s, a...)
  }
  if len(file)⟩0 ∧ file[0] ≠ nil {
    ⟨ Print error location based on input buffer 71 ⟩
  }
  os.Stdout.Sync()
  mark_error()
}
```

70. The command '*warn_print*(`"!␣Warning␣message"`)' will report a warning to the user, by printing the warning message at the beginning of a new line. A newline is automatically supplied if the string begins with `"!"`.

```
func warn_print(s string, a ...interface{}){
  if len(s)⟩0 ∧ s[0] ≡ '!' {
    fmt.Fprintf(os.Stdout, "\n\n" + s, a...)
  } else {
    fmt.Fprintf(os.Stdout, "\n" + s, a...)
  }
  os.Stdout.Sync()
  mark_harmless()
}
```

71. The error locations can be indicated by using the global variables *loc*, *line*[*include_depth*], *file_name*[*include_depth*] and *changing*, which tell respectively the first unlooked-at position in *buffer*, the current line number, the current file, and whether the current line is from *change_file* or *file*[*include_depth*]. This routine should be modified on systems whose standard text editor has special line-numbering conventions.

⟨ Print error location based on input buffer 71 ⟩ ≡

```
{
  if changing ∧ include_depth ≡ change_depth {
    fmt.Printf(".␣(change␣file␣%s:%d)\n", change_file_name, change_line)
  } else if include_depth ≡ 0 ∧ len(line)⟩0 {
    fmt.Printf(".␣(%s:%d)\n", file_name[include_depth], line[include_depth])
  } else if len(line)⟩include_depth {
    fmt.Printf(".␣(include␣file␣%s:%d)\n", file_name[include_depth], line[include_depth])
  }
  l = len(buffer)
  if loc⟨l {
    l = loc
  }
  if l⟩0 {
    for k := 0; k⟨l; k++ {
      if buffer[k] ≡ '\t' {
        fmt.Print("␣")
      } else {
        fmt.Printf("%c", buffer[k])      // print the characters already read
      }
    }
    fmt.Println()
    fmt.Printf("%*c", l, '␣')
  }
  fmt.Println(string(buffer[l:]))
  if len(buffer)⟩0 ∧ buffer[len(buffer) − 1] ≡ '|' {
    fmt.Print("|")      /* end of Go text in section names */
  }
  fmt.Print("␣")      /* to separate the message from future asterisks */
}
```

This code is used in section 69.

72. When no recovery from some error has been provided, we have to wrap up and quit as graciously as possible. This is done by calling the function *wrap_up* at the end of the code.

`GOTANGLE` and `GOWEAVE` have their own notions about how to print the job statistics.

73. Some implementations may wish to pass the *history* value to the operating system so that it can be used to govern whether or not other programs are started. Here, for instance, we pass the operating system a status of 0 if and only if only harmless messages were printed.

```
func wrap_up() int{
  fmt.Print("\n")
  if show_stats() {
    print_stats()     /* print statistics about memory usage */
  }
  ⟨ Print the job history 74 ⟩
  if history⟩harmless_message {
    return 1
  }
  return 0
}
```

74. ⟨ Print the job *history* 74 ⟩ ≡

```
switch history {
  case spotless:
    if show_happiness() {
      fmt.Printf("(No␣errors␣were␣found.)\n")
    }
  case harmless_message:
    fmt.Printf("(Did␣you␣see␣the␣warning␣message␣above?)\n")
  case error_message:
    fmt.Printf("(Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong.)\n")
  case fatal_message:
    fmt.Printf("(That␣was␣a␣fatal␣error,␣my␣friend.)\n")
}    /* there are no other cases */
```

This code is used in section 73.

75. When there is no way to recover from an error, the *fatal* subroutine is invoked.

The two parameters to *fatal* are strings that are essentially concatenated to print the final error message.

```
func fatal(s string,t string){
  if len(s) ≠ 0 {
    fmt.Print(s)
  }
  err_print(t)
  history = fatal_message
  os.Exit(wrap_up())
}
```

76. Command line arguments. The user calls `GOWEAVE` and `GOTANGLE` with arguments on the command line. These are either file names or flags to be turned off (beginning with `"-"`) or flags to be turned on (beginning with `"+"`). The following functions are for communicating the user's desires to the rest of the program. The various file name variables contain strings with the names of those files. Most of the 128 flags are undefined but available for future extensions.

77.

```
func show_banner() bool{
  return flags['b']     /* should the banner line be printed? */
}
```

78.

```
func show_progress() bool{
  return flags['p']     /* should progress reports be printed? */
}
```

79.

```
func show_stats() bool{
  return flags['s']     /* should statistics be printed at end of run? */
}
```

80.

```
func show_happiness() bool{
  return flags['h']     /* should lack of errors be announced? */
}
```

81. ⟨ Definitions that should agree with `GOTANGLE` and `GOWEAVE` 12 ⟩ +≡

```
var go_file_name string      /* name of go_file */
var tex_file_name string     /* name of tex_file */
var idx_file_name string     /* name of idx_file */
var scn_file_name string     /* name of scn_file */
var flags [128]bool          /* an option for each 7-bit code */
```

82. The *flags* will be initially zero. Some of them are set to 1 before scanning the arguments; if additional flags are 1 by default they should be set before calling *common_init*.

⟨ Set the default options common to `GOTANGLE` and `GOWEAVE` 82 ⟩ ≡

```
flags['b'] = true
flags['h'] = true
flags['p'] = true
```

This code is used in section 8.

83. We now must look at the command line arguments and set the file names accordingly. At least one file name must be present: the `GOWEB` file. It may have an extension, or it may omit the extension to get `".w"` or `".web"` added. The TEX output file name is formed by replacing the `GOWEB` file name extension by `".tex"`, and the Go file name by replacing the extension by `".go"`, after removing the directory name (if any).

If there is a second file name present among the arguments, it is the change file, again either with an extension or without one to get `".ch"`. An omitted change file argument means that `"/dev/null"` should be used, when no changes are desired.

If there's a third file name, it will be the output file.

```
func scan_args() { dot_pos := −1 /* position of '.' in the argument */
name_pos := 0 /* file name beginning, sans directory */
found_web := false
found_change := false
found_out := false
    /* have these names been seen? */
flag_change := false
for i := 1;
i⟨len(os.Args);
i ++ { arg := os.Args[i]
if (arg[0] ≡ '-' ∨ arg[0] ≡ '+') ∧ len(arg)⟩1 {⟨Handle flag argument 87⟩} else { name_pos = 0
dot_pos = −1
for j := 0; j⟨len(arg); j++ {
  if arg[j] ≡ '.' {
    dot_pos = j
  } else if arg[j] ≡ '/' {
    dot_pos = −1
    name_pos = j + 1
  }
}
if ¬found_web{⟨Make file_name[0], tex_file_name, and go_file_name 84⟩} else if ¬found_change{⟨Make
      change_file_name from fname 85⟩} else if ¬found_out{⟨Override tex_file_name and
      go_file_name 86⟩} else {
  ⟨Print usage error message and quit 166⟩
}
}
}
if ¬found_web{⟨Print usage error message and quit 166⟩}
}
```

84. We use all of *arg* for the *file_name*[0] if there is a `'.'` in it, otherwise we add `".w"`. If this file can't be opened, we prepare an *alt_file_name* by adding `"web"` after the dot. The other file names come from adding other things after the dot. We must check that there is enough room in *file_name*[0] and the other arrays for the argument.

```
⟨ Make file_name[0], tex_file_name, and go_file_name 84 ⟩ ≡
  {
    if dot_pos ≡ −1 {
      file_name = append(file_name, fmt.Sprintf("%s.w", arg))
    } else {
      file_name = append(file_name, arg)
      arg = arg[:dot_pos]      /* string now ends where the dot was */
    }
    alt_file_name = fmt.Sprintf("%s.web", arg)
    tex_file_name = fmt.Sprintf("%s.tex", arg[name_pos:])      /* strip off directory name */
    idx_file_name = fmt.Sprintf("%s.idx", arg[name_pos:])
    scn_file_name = fmt.Sprintf("%s.scn", arg[name_pos:])
    go_file_name = fmt.Sprintf("%s.go", arg[name_pos:])
    found_web = true
  }
```

This code is used in section 83.

85.

```
⟨ Make change_file_name from fname 85 ⟩ ≡
  {
    if arg[0] ≡ '-' {
      found_change = true
    } else {
      if dot_pos ≡ −2 {
        change_file_name = fmt.Sprintf("%s.ch", arg)
      } else {
        change_file_name = arg
      }
      found_change = true
    }
  }
```

This code is used in section 83.

86. ⟨ Override *tex_file_name* and *go_file_name* 86 ⟩ ≡

```
{
  if dot_pos ≡ −1 {
    tex_file_name = fmt.Sprintf("%s.tex", arg)
    idx_file_name = fmt.Sprintf("%s.idx", arg)
    scn_file_name = fmt.Sprintf("%s.scn", arg)
    go_file_name = fmt.Sprintf("%s.go", arg)
  } else {
    tex_file_name = arg
    go_file_name = arg
    if flags['x'] {      /* indexes will be generated */
      dot_pos = −1
      idx_file_name = fmt.Sprintf("%s.idx", arg)
      scn_file_name = fmt.Sprintf("%s.scn", arg)
    }
  }
  found_out = true
}
```

This code is used in section 83.

87. ⟨ Handle flag argument 87 ⟩ ≡

```
{
  if arg[0] ≡ '-' {
    flag_change = false
  } else {
    flag_change = true
  }
  for i := 1; i⟨len(arg); i++ {
    flags[arg[i]] = flag_change
  }
}
```

This code is used in section 83.

88. Output. Here is the code that opens the output file:

```
⟨ Definitions that should agree with GOTANGLE and GOWEAVE 12 ⟩ +≡
  var go_file io.WriteCloser      /* where output of GOTANGLE goes */
  var tex_file io.WriteCloser     /* where output of GOWEAVE goes */
  var idx_file io.WriteCloser     /* where index from GOWEAVE goes */
  var scn_file io.WriteCloser     /* where list of sections from GOWEAVE goes */
  var active_file io.WriteCloser  /* currently active file for GOWEAVE output */
```

89.

```
⟨ Scan arguments and open output files 89 ⟩ ≡
  scan_args()
  ⟨ Try to open output file 165 ⟩
```

This code is used in section 8.

90. *xisxdigit* checks for hexdecimal digits, that is, one of 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F.

```
func xisxdigit(r rune) bool{
  if unicode.IsDigit(r) {
    return true
  }
  if ¬unicode.IsLetter(r) {
    return false
  }
  r = unicode.ToLower(r)
  if r ≥ 'a' ∧ r ≤ 'f' {
    return true
  }
  return false
}
```

91. The following code assigns values to the combinations `++`, `--`, `->`, `>=`, `<=`, `==`, `<<`, `>>`, `!=`, and `&&`, ... The compound assignment operators (e.g., `+=`) are treated as separate tokens.

```
⟨ Compress two-symbol operator 91 ⟩ ≡
  switch c {
    case '/':
      if nc ≡ '*' {
        l := loc
        loc++
        if l ≤ len(buffer) {
          return begin_comment
        }
      } else if nc ≡ '/' {
        l := loc
        loc++
        if l ≤ len(buffer) {
          return begin_short_comment
        }
      }
    case '+':
      if nc ≡ '+' {
        l := loc
        loc++
        if l ≤ len(buffer) {
          return plus_plus
        }
      }
    case '-':
      if nc ≡ '-' {
        l := loc
        loc++
        if l ≤ len(buffer) {
          return minus_minus
        }
      }
    case '.':
      if nc ≡ '.' ∧ loc + 1⟨len(buffer) ∧ buffer[loc + 1] ≡ '.' {
        loc++
        l := loc
        loc++
        if l ≤ len(buffer) {
          return dot_dot_dot
        }
      }
    case '=':
      if nc ≡ '=' {
        l := loc
        loc++
        if l ≤ len(buffer) {
          return eq_eq
        }
      }
    case '>':
```

```
    if nc ≡ '=' {
      l := loc
      loc++
      if l ≤ len(buffer) {
        return gt_eq
      }
    } else if nc ≡ '>' {
      l := loc
      loc++
      if l ≤ len(buffer) {
        return gt_gt
      }
    }
  case '<':
    if nc ≡ '<' {
      l := loc
      loc++
      if l ≤ len(buffer) {
        return lt_lt
      }
    } else if nc ≡ '-' {
      l := loc
      loc++
      if l ≤ len(buffer) {
        return direct
      }
    } else if nc ≡ '=' {
      l := loc
      loc++
      if l ≤ len(buffer) {
        return lt_eq
      }
    }
  case '&':
    if nc ≡ '&' {
      l := loc
      loc++
      if l ≤ len(buffer) {
        return and_and
      }
    } else if nc ≡ '^' {
      l := loc
      loc++
      if l ≤ len(buffer) {
        return and_not
      }
    }
  case '|':
    if nc ≡ '|' {
      l := loc
      loc++
      if l ≤ len(buffer) {
```

```
            return or_or
          }
        }
      case '!':
        if nc ≡ '=' {
          l := loc
          loc++
          if l ≤ len(buffer) {
            return not_eq
          }
        }
      case ':':
        if nc ≡ '=' {
          l := loc
          loc++
          if l ≤ len(buffer) {
            return col_eq
          }
        }
    }
```

This code is used in section 133.

92. Data structures exclusive to GOTANGLE. A *text* is a structure containing a token into *token*, and an integer *text_link*, which, as we shall see later, is used to connect pieces of text that have the same name. All the *text* are stored in the array *text_info*.

```
⟨ Typedef declarations 92 ⟩ ≡
  type text struct{
    token []rune      /* pointer into tok_mem */
    text_link int32      /* relates replacement texts */
  }
```

See also section 101.

This code is used in section 2.

93.
```
⟨ Global variables 93 ⟩ ≡
  var text_info []text
  var tok_mem []rune
```

See also sections 98, 102, 107, 112, 115, 126, 132, and 145.

This code is used in section 2.

94. If p is an index of a section name, $p.equiv$ is an index of its replacement text, an element of the array *text_info*.

```
⟨ More elements of name_info structure 41 ⟩ +≡
  equiv int32      /* info corresponding to names */
```

95. Here's the procedure that decides whether a name *id* equals the identifier pointed to by p:

```
func names_match(
    p int32,      /* points to the proposed match */
    id []rune,      /* the identifier */
    t int32
) bool{
    if len(name_dir[p].name) ≠ len(id) {
        return false
    }
    return compare_runes(id, name_dir[p].name) ≡ 0
}
```

96. The common lookup routine refers to separate routines *init_node* when the data structure grows.

```
func init_node(node int32){
    name_dir[node].equiv = −1
}
```

97. Actually GOTANGLE haven't got any specific code for initialization a new identifier, so we declare an empty corresponding section.

```
⟨ Initialization of a new identifier 97 ⟩ ≡
```

This code is used in section 49.

98. Tokens. Replacement texts, which represent Go code in a compressed format, appear in *tok_mem* as mentioned above. The codes in these texts are called 'tokens'..

If p is an index of a replacement text, $p.token$ contains code of that text. If $text\_info[p].text\_link \equiv 0$, this is the replacement text for a macro, otherwise it is the replacement text for a section. In the latter case $text\_info[p].text\_link$ is either equal to max_texts, which means that there is no further text for this section, or $text\_info[p].text\_link$ points to a continuation of this replacement text; such links are created when several sections have Go texts with the same name, and they also tie together all the Go texts of unnamed sections. The replacement text pointer for the first unnamed section appears in $text\_info[0].text\_link$, and the most recent such pointer is $last\_unnamed$.

```
⟨ Global variables 93 ⟩ +≡
  var last_unnamed int32      /* most recent replacement text of unnamed section */
```

99.
```
⟨ Set initial values 99 ⟩ ≡
  last_unnamed = 0
  text_info = append(text_info, text{})
  text_info[0].text_link = 0
```
See also section 127.

This code is used in section 3.

100. GOTANGLE operates with UTF8 encoding texts and represents a text in 4-bytes unicode code points internally. If the first byte of a token is less than $unicode.UpperLower$, this is usual character. Otherwise if it is equal $unicode.UpperLower + °211$, the next rune is a section number; if it is equal $unicode.UpperLower + °212$, the next rune is an identifier index; if it is equal $unicode.UpperLower + °214$, the next element is an line number; if it is equal $unicode.UpperLower + °311$ and the next rune is equal $unicode.UpperLower + °215$, it is a macro definition, otherwise it is an index of section in which the current replacement text appears.

Some of the 7-bit codes will not be present, however, so we can use them for special purposes. The following symbolic names are used:

join denotes the concatenation of adjacent items with no space or line breaks allowed between them (the `@&` operation of GOWEB).

strs denotes the beginning or end of a string, verbatim construction or numerical constant.

```
⟨ Constants 1 ⟩ +≡
  strs = °2      /* takes the place of extended ASCII α */
  join = °177      /* takes the place of ASCII delete */
```

101. Stacks for output. The output process uses a stack to keep track of what is going on at different "levels" as the sections are being written out. Entries on this stack have five parts:
byte_field is a slice of the next token on a particular level;
name_field is an index of the name corresponding to a particular level;
repl_field is an index of the replacement text currently being read at a particular level;
section_field is the section number, or zero if this is a macro.

The current values of these five quantities are referred to quite frequently, so they are stored in a separate place instead of in the *stack* array. We call the current values *cur_state.byte_field*, *cur_state.name_field*, *cur_state.repl_field*, and *cur_state.section_field*.

```
⟨ Typedef declarations 92 ⟩ +≡
  type output_state struct{
    byte_field []rune      /* present location within replacement text */
    name_field int32       /* byte_start index for text being output */
    repl_field int32       /* token index for text being output */
    section_field int32    /* section number or zero if not a section */
  }
```

102.
```
⟨ Global variables 93 ⟩ +≡
  var cur_state output_state
    /* cur_state.byte_field, cur_state.name_field,cur_state.repl_field, and cur_state.section_field */
  var stack []output_state      /* info for non-current levels */
```

103. To get the output process started, we will perform the following initialization steps. We may assume that *text_info*[0].*text_link* is nonzero, since it points to the Go text in the first unnamed section that generates code; if there are no such sections, there is nothing to output, and an error message will have been generated before we do any of the initialization.

```
⟨ Initialize the output stacks 103 ⟩ ≡
  cur_state.name_field = 0
  cur_state.repl_field = text_info[0].text_link
  cur_state.byte_field = text_info[cur_state.repl_field].token
  cur_state.section_field = 0
  stack = append(stack, output_state{})
```

This code is used in section 117.

104. When the replacement text for name p is to be inserted into the output, the following subroutine is called to save the old level of output and get the new one going.

```
      /* suspends the current level */
  func push_level(p int32){
    stack = append(stack, cur_state)
    cur_state.name_field = p
    cur_state.repl_field = name_dir[p].equiv
    cur_state.byte_field = text_info[cur_state.repl_field].token
    cur_state.section_field = 0
  }
```

105. When we come to the end of a replacement text, the *pop_level* subroutine does the right thing: It either moves to the continuation of this replacement text or returns the state to the most recently stacked level.

```
    /* do this when cur_state.byte_field reaches end */
  func pop_level(){
    if text_info[cur_state.repl_field].text_link⟨max_texts {    /* link to a continuation */
      cur_state.repl_field = text_info[cur_state.repl_field].text_link    /* stay on the same level */
      cur_state.byte_field = text_info[cur_state.repl_field].token
      return
    }
    if len(stack)⟩0 {
      cur_state = stack[len(stack) − 1]
      stack = stack[:len(stack) − 1]
    }
  }
```

106. The heart of the output procedure is the function *get_output*, which produces the next token of output and sends it on to the lower-level function *out_char*. The main purpose of *get_output* is to handle the necessary stacking and unstacking. It sends the value *section_number* if the next output begins or ends the replacement text of some section, in which case *cur_val* is that section's number (if beginning) or the negative of that value (if ending). (A section number of 0 indicates not the beginning or ending of a section, but a **//line** command.) And it sends the value *identifier* if the next output is an identifier, in which case *cur_val* points to that identifier name.

```
⟨ Constants 1 ⟩ +≡
  section_number = °211    /* code returned by get_output for section numbers */
  identifier = °212    /* code returned by get_output for identifiers */
```

107.
```
⟨ Global variables 93 ⟩ +≡
  var cur_val rune    /* additional information corresponding to output token */
```

108.

```
    /* sends next token to out_char */
  func get_output(){
    restart:
    if len(stack) ≡ 0 {
      return
    }
    if len(cur_state.byte_field) ≡ 0 {
      cur_val = −cur_state.section_field      /* cast needed because of sign extension */
      pop_level()
      if cur_val ≡ 0 {
        goto restart
      }
      out_char(section_number)
      return
    }
    a := cur_state.byte_field[0]
    cur_state.byte_field = cur_state.byte_field[1:]
    if out_state ≡ verbatim ∧ a ≠ strs ∧ a ≠ constant ∧ a ≠ comment ∧ a ≠ '\n' {
      fmt.Fprintf(go_file, "%c", a)
    } else if a⟨unicode.UpperLower {
      out_char(a)
    } else {
      c := cur_state.byte_field[0]
      cur_state.byte_field = cur_state.byte_field[1:]
      switch a % unicode.UpperLower {
        case identifier:
          cur_val = c
          out_char(identifier)
        case section_name:
          ⟨ Expand section c, goto restart 109 ⟩
        case line_number:
          cur_val = c
          out_char(line_number)
        case section_number:
          cur_val = c
          if cur_val⟩0 {
            cur_state.section_field = cur_val
          }
          out_char(section_number)
      }
    }
  }
```

109. The user may have forgotten to give any Go text for a section name, or the Go text may have been associated with a different name by mistake.

```
⟨ Expand section c, goto restart 109 ⟩ ≡
  {
    if name_dir[c].equiv ≠ −1 {
      push_level(c)
    } else if a ≠ 0 {
      err_print("!␣Not␣present:␣<%s>", sprint_section_name(c))
    }
    goto restart
  }
```

This code is used in section 108.

110. Producing the output. The *get_output* routine above handles most of the complexity of output generation, but there are two further considerations that have a nontrivial effect on GOTANGLE's algorithms.

111. We want to make sure that the output has spaces and line breaks in the right places (e.g., not in the middle of a string or a constant or an identifier, not at a '@&' position where quantities are being joined together).

The output process can be in one of following states:

num_or_id means that the last item in the buffer is a number or identifier, hence a blank space or line break must be inserted if the next item is also a number or identifier.

unbreakable means that the last item in the buffer was followed by the @& operation that inhibits spaces between it and the next item.

verbatim means we're copying only character tokens, and that they are to be output exactly as stored. This is the case during strs, verbatim constructions and numerical constants.

post_slash means we've just output a slash.

normal means none of the above.

```
⟨ Constants 1 ⟩ +≡
  normal = 0      /* non-unusual state */
  num_or_id = 1      /* state associated with numbers and identifiers */
  post_slash = 2      /* state following a / */
  unbreakable = 3      /* state associated with @& */
  verbatim = 4      /* state in the middle of a string */
```

112.
```
⟨ Global variables 93 ⟩ +≡
  var out_state rune      /* current status of partial output */
```

113. Here is a routine that is invoked when we want to output the current line. During the output process, *line*[*include_depth*] equals the number of the next line to be output.

```
      /* writes one line to output file */
  func flush_buffer(){
    fmt.Fprintln(go_file)
    if line[include_depth] % 100 ≡ 0 ∧ show_progress() {
      fmt.Print(".")
      if line[include_depth] % 500 ≡ 0 {
        fmt.Printf("%d", line[include_depth])
      }
      os.Stdout.Sync()      /* progress report */
    }
    line[include_depth]++
  }
```

114. If a section name is introduced in at least one place by @(instead of @<, we treat it as the name of a file. All these special sections are saved on a stack, *output_files*. We write them out after we've done the unnamed section.

115.
```
⟨ Global variables 93 ⟩ +≡
  var output_files []int32
  var cur_section_name_char rune      /* is it '<' or '(' */
  var output_file_name string      /* name of the file */
```

116. ⟨ If it's not there, add *cur_section_name* to the output file stack, or complain we're out of room 116 ⟩ ≡

```
{
  an_output_file := 0
  for ; an_output_file⟨len(output_files); an_output_file++ {
    if output_files[an_output_file] ≡ cur_section_name {
      break
    }
  }
  if an_output_file ≡ len(output_files) {
    output_files = append(output_files, cur_section_name)
  }
}
```

This code is used in section 139.

117. The big output switch. Here then is the routine that does the output.

```
func phase_two(){
  line[include_depth] = 1
  ⟨ Initialize the output stacks 103 ⟩
  if text_info[0].text_link ≡ 0 ∧ len(output_files) ≡ 0 {
    warn_print("! No program text was specified.")
  } else {
    if len(output_files) ≡ 0 {
      if show_progress() {
        fmt.Printf("\nWriting the output file (%s):", go_file_name)
      }
    } else {
      if show_progress() {
        fmt.Printf("\nWriting the output files: (%s)", go_file_name)
        os.Stdout.Sync()
      }
      if text_info[0].text_link ≡ 0 {
        goto writeloop
      }
    }
    for len(stack)⟩0 {
      get_output()
    }
    flush_buffer()
    writeloop:
    ⟨ Write all the named output files 118 ⟩
    if show_happiness() {
      fmt.Print("\nDone.")
    }
  }
}
```

118. To write the named output files, we proceed as for the unnamed section. The only subtlety is that we have to open each one.

```
⟨ Write all the named output files 118 ⟩ ≡
  for an_output_file := len(output_files); an_output_file⟩0; {
    an_output_file--
    output_file_name = string(sprint_section_name(output_files[an_output_file]))
    if f, err := os.OpenFile(output_file_name, os.O_WRONLY | os.O_CREATE | os.O_TRUNC, °666);
      err ≠ nil {
      fatal("! Cannot open output file:", output_file_name)
    } else {
      go_file.Close()
      go_file = f
    }
    fmt.Printf("\n(%s)", output_file_name)
    os.Stdout.Sync()
    line[include_depth] = 1
    stack = append(stack, output_state{})
    cur_state.name_field = output_files[an_output_file]
    cur_state.repl_field = name_dir[cur_state.name_field].equiv
    cur_state.byte_field = text_info[cur_state.repl_field].token
    for len(stack)⟩0 {
      get_output()
    }
    flush_buffer()
  }
```

This code is used in section 117.

119. A many-way switch is used to send the output. Note that this function is not called if *out_state* ≡ *verbatim*, except perhaps with arguments `'\n'` (protect the newline), **string** (end the string), or *constant* (end the constant).

```
func out_char(cur_char rune){
  switch cur_char {
    case '\n':
      flush_buffer()
      if out_state ≠ verbatim {
        out_state = normal
      }
      ⟨ Case of an identifier 121 ⟩
      ⟨ Case of a section number 122 ⟩
      ⟨ Case of a line number 123 ⟩
      ⟨ Cases like != 120 ⟩
    case '=', '>':
      fmt.Fprintf(go_file, "%c␣", cur_char)
      out_state = normal
    case join:
      out_state = unbreakable
    case constant:
      switch out_state {
        case verbatim:
          out_state = num_or_id
        case num_or_id:
          fmt.Fprint(go_file, "␣")
          fallthrough
        default:
          out_state = verbatim
      }
    case strs:
      if out_state ≡ verbatim {
        out_state = normal
      } else {
        out_state = verbatim
      }
    case comment:
      if out_state ≡ verbatim {
        out_state = normal
      } else {
        out_state = verbatim
      }
    case '/':
      fmt.Fprint(go_file, "/")
      out_state = post_slash
    case '*':
      if out_state ≡ post_slash {
        fmt.Fprint(go_file, "␣")
      }
      fallthrough
    default:
      fmt.Fprintf(go_file, "%c", cur_char)
      out_state = normal
```

```
    }
  }
```

120. ⟨ Cases like `!=` 120 ⟩ ≡

```
case plus_plus:
  fmt.Fprint(go_file, "++")
  out_state = normal
case minus_minus:
  fmt.Fprint(go_file, "--")
  out_state = normal
case gt_gt:
  fmt.Fprint(go_file, ">>")
  out_state = normal
case eq_eq:
  fmt.Fprint(go_file, "==")
  out_state = normal
case lt_lt:
  fmt.Fprint(go_file, "<<")
  out_state = normal
case gt_eq:
  fmt.Fprint(go_file, ">=")
  out_state = normal
case lt_eq:
  fmt.Fprint(go_file, "<=")
  out_state = normal
case not_eq:
  fmt.Fprint(go_file, "!=")
  out_state = normal
case and_and:
  fmt.Fprint(go_file, "&&")
  out_state = normal
case or_or:
  fmt.Fprint(go_file, "||")
  out_state = normal
case dot_dot_dot:
  fmt.Fprint(go_file, "...")
  out_state = normal
case direct:
  fmt.Fprint(go_file, "<-")
  out_state = normal
case and_not:
  fmt.Fprint(go_file, "&^")
  out_state = normal
case col_eq:
  fmt.Fprint(go_file, ":=")
  out_state = normal
```

This code is used in section 119.

121. ⟨ Case of an identifier 121 ⟩ ≡

```
case identifier:
    if out_state ≡ num_or_id {
        fmt.Fprint(go_file, "␣")
    }
    fmt.Fprintf(go_file, "%s", string(name_dir[cur_val].name))
    out_state = num_or_id
```

This code is used in section 119.

122. ⟨ Case of a section number 122 ⟩ ≡

```
case section_number:
    if cur_val⟩0 {
        fmt.Fprintf(go_file, "\n\n/*%d:*/\n\n", cur_val)
    } else if cur_val⟨0 {
        fmt.Fprintf(go_file, "\n\n/*:%d*/\n\n", −cur_val)
    }
```

This code is used in section 119.

123. ⟨ Case of a line number 123 ⟩ ≡

```
case line_number:
    fmt.Fprint(go_file, "\n//line␣")
    line := cur_val
    cur_val = cur_state.byte_field[0]
    cur_state.byte_field = cur_state.byte_field[1:]
    for _, v := range name_dir[cur_val].name {
        if v ≡ '\\' ∨ v ≡ '"' {
            fmt.Fprint(go_file, "\\")
        }
        fmt.Fprintf(go_file, "%c", v)
    }
    fmt.Fprintf(go_file, ":%d\n", line)
```

This code is used in section 119.

124. Introduction to the input phase. We have now seen that `GOTANGLE` will be able to output the full Go program, if we can only get that program into the byte memory in the proper format. The input process is something like the output process in reverse, since we compress the text as we read it in and we expand it as we write it out.

There are three main input routines. The most interesting is the one that gets the next token of a Go text; the other two are used to scan rapidly past TEX text in the `GOWEB` source code. One of the latter routines will jump to the next token that starts with '`@`', and the other skips to the end of a Go comment.

125. Control codes in `GOWEB` begin with '`@`', and the next character identifies the code. Some of these are of interest only to `GOWEAVE`, so `GOTANGLE` ignores them; the others are converted by `GOTANGLE` into internal code numbers by the *ccode* table below. The ordering of these internal code numbers has been chosen to simplify the program logic; larger numbers are given to the control codes that denote more significant milestones.

```
⟨ Constants 1 ⟩ +≡
  ignore rune = 0      /* control code of no interest to GOTANGLE */
  ord rune = °302      /* control code for '@'' */
  control_text rune = °303      /* control code for '@t', '@^', etc. */
  format_code rune = °306      /* control code for '@f' */
  definition rune = °307      /* control code for '@d' */
  begin_code rune = °310      /* control code for '@c' */
  section_name rune = °311      /* control code for '@<' */
  new_section rune = °312      /* control code for '@␣' and '@*' */
```

126.
```
⟨ Global variables 93 ⟩ +≡
  var ccode [256]rune      /* meaning of a char following @ */
```

127. ⟨ Set initial values 99 ⟩ +≡

```
{
  for c := 0; c<len(ccode); c++ {
    ccode[c] = ignore
  }
  ccode[' '] = new_section
  ccode['\t'] = new_section
  ccode['\n'] = new_section
  ccode['\v'] = new_section
  ccode['\r'] = new_section
  ccode['\f'] = new_section
  ccode['*'] = new_section
  ccode['@'] = '@'
  ccode['='] = strs
  ccode['d'] = definition
  ccode['D'] = definition
  ccode['f'] = format_code
  ccode['F'] = format_code
  ccode['s'] = format_code
  ccode['S'] = format_code
  ccode['c'] = begin_code
  ccode['C'] = begin_code
  ccode['p'] = begin_code
  ccode['P'] = begin_code
  ccode['^'] = control_text
  ccode[':'] = control_text
  ccode['.'] = control_text
  ccode['t'] = control_text
  ccode['T'] = control_text
  ccode['r'] = control_text
  ccode['R'] = control_text
  ccode['q'] = control_text
  ccode['Q'] = control_text
  ccode['&'] = join
  ccode['<'] = section_name
  ccode['('] = section_name
  ccode['\''] = ord
}
```

128. The *skip_ahead* procedure reads through the input at fairly high speed until finding the next non-ignorable control code, which it returns.

```
    /* skip to next control code */
func skip_ahead() rune{
  for true {
    if loc ≥ len(buffer) ∧ ¬get_line() {
      return new_section
    }
    for loc⟨len(buffer) ∧ buffer[loc] ≠ '@' {
      loc++
    }
    if loc⟨len(buffer) {
      loc++
      c := new_section
      if loc⟨len(buffer) ∧ buffer[loc]⟨int32(len(ccode)) {
        c = ccode[buffer[loc]]
      }
      loc++
      if c ≠ ignore ∨ (loc ≤ len(buffer) ∧ buffer[loc − 1] ≡ '>') {
        return c
      }
    }
  }
  return 0
}
```

129. The *copy_comment* procedure reads through the input at somewhat high speed in order to pass over comments. If the comment is introduced by `/*`, *copy_comment* proceeds until finding the end-comment token `*/` or a newline; in the latter case the newline will be added to the comment. On the other hand, if the comment is introduced by `//` (i.e., if it is a Go "short comment"), it always is simply delimited by the next newline. The boolean argument *is_long_comment* distinguishes between the two types of comments.

If *copy_comment* comes to the end of the section, it prints an error message. No comment, long or short, is allowed to contain '`@␣`' or '`@*`'.

```
⟨ Constants 1 ⟩ +≡
  comment = °213
```

130.

```
func copy_comment(is_long_comment bool) rune{
  section_text = section_text[0:0]
  for true {
    if loc ≥ len(buffer) {
      if ¬is_long_comment {
        break
      }
      section_text = append(section_text, '\n')
      if ¬get_line() {
        err_print("! Input ended in mid-comment")
        break
      }
    }
    c := buffer[loc]
    if is_long_comment ∧ c ≡ '*' ∧ loc + 1⟨len(buffer) ∧ buffer[loc + 1] ≡ '/' {
      section_text = append(section_text, '*', '/')
      loc += 2
      break
    }
    if c ≡ '@' {
      if loc + 1⟨len(buffer) ∧ buffer[loc + 1]⟨int32(len(ccode)) ∧ ccode[buffer[loc + 1]] ≡ new_section {
        err_print("! Section name ended in mid-comment")
        break
      } else {
        loc++
      }
    }
    section_text = append(section_text, c)
    loc++
  }
  id = section_text
  return comment
}
```

131. Inputting the next token.

```
⟨ Constants 1 ⟩ +≡
  constant = °3
```

132.

```
⟨ Global variables 93 ⟩ +≡
  var cur_section_name int32      /* name of section just scanned */
  var no_where bool      /* suppress print_where? */
```

133. As one might expect, *get_next* consists mostly of a big switch that branches to the various special cases that can arise.

```
      /* produces the next input token */
  func get_next() rune{
    for true {
      if loc ≥ len(buffer) {
        if ¬get_line() {
          return new_section
        } else if print_where ∧ ¬no_where {
          print_where = false
          ⟨ Insert the line number into tok_mem 148 ⟩
        } else {
          return '\n'
        }
      }
      c := buffer[loc]
      var nc rune = '␣'
      if loc + 1⟨len(buffer) {
        nc = buffer[loc + 1]
      }
      if c ≡ '/' ∧ (nc ≡ '*' ∨ nc ≡ '/') {
        return copy_comment(nc ≡ '*')
      }
      loc++
      if unicode.IsDigit(c) ∨ c ≡ '.' {
        ⟨ Get a constant 135 ⟩
      } else if c ≡ '\'' ∨ c ≡ '"' ∨ c ≡ '‘' {
        ⟨ Get a string 136 ⟩
      } else if unicode.IsLetter(c) ∨ c ≡ '_' {
        ⟨ Get an identifier 134 ⟩
      } else if c ≡ '@' {
        ⟨ Get control code and possible section name 137 ⟩
      } else if unicode.IsSpace(c) {
        continue      /* ignore spaces and tabs */
      }
      mistake:
      ⟨ Compress two-symbol operator 91 ⟩
      return c
    }
    return 0
  }
```

134. ⟨ Get an identifier 134 ⟩ ≡

```
{
  loc−−
  id_first := loc
  for loc⟨len(buffer) ∧
    (unicode.IsLetter(buffer[loc]) ∨
    unicode.IsDigit(buffer[loc]) ∨
    buffer[loc] ≡ '_' ∨
    buffer[loc] ≡ '$') {
    loc++
  }
  id = buffer[id_first:loc]
  return identifier
}
```

This code is used in section 133.

135. ⟨ Get a constant 135 ⟩ ≡

```
{
  id_first := loc − 1
  if buffer[id_first] ≡ '.' ∧ (loc ≥ len(buffer) ∨ ¬unicode.IsDigit(buffer[loc])) {
    goto mistake      /* not a constant */
  }
  if buffer[id_first] ≡ '0' {
    if loc⟨len(buffer) ∧ (buffer[loc] ≡ 'x' ∨ buffer[loc] ≡ 'X') {      /* hex constant */
      loc++
      for loc⟨len(buffer) ∧ xisxdigit(buffer[loc]) {
        loc++
      }
      goto found
    }
  }
  for loc⟨len(buffer) ∧ unicode.IsDigit(buffer[loc]) {
    loc++
  }
  if loc⟨len(buffer) ∧ buffer[loc] ≡ '.' {
    loc++
    for loc⟨len(buffer) ∧ unicode.IsDigit(buffer[loc]) {
      loc++
    }
  }
  if loc⟨len(buffer) ∧ (buffer[loc] ≡ 'e' ∨ buffer[loc] ≡ 'E') {      /* float constant */
    loc++
    if loc⟨len(buffer) ∧ (buffer[loc] ≡ '+' ∨ buffer[loc] ≡ '-') {
      loc++
    }
    for loc⟨len(buffer) ∧ unicode.IsDigit(buffer[loc]) {
      loc++
    }
  }
  found:
  for loc⟨len(buffer) ∧
    (buffer[loc] ≡ 'u' ∨ buffer[loc] ≡ 'U' ∨
    buffer[loc] ≡ 'l' ∨ buffer[loc] ≡ 'L' ∨
    buffer[loc] ≡ 'f' ∨ buffer[loc] ≡ 'F') {
    loc++
  }
  id = buffer[id_first:loc]
  return constant
}
```

This code is used in section 133.

136. Go strs and character constants, delimited by double and single quotes, respectively, can contain newlines or instances of their own delimiters if they are protected by a backslash.

```
⟨ Get a string 136 ⟩ ≡
  {
    delim := c      /* what started the string */
    section_text = section_text[0:0]
    section_text = append(section_text, delim)
    for true {
      if loc ≥ len(buffer) {
        if ¬get_line() {
          err_print("! Input ended in middle of string")
          loc = 0
          break
        } else {
          section_text = append(section_text, '\n')
        }
      }
      l := loc
      loc++
      if c = buffer[l]; c ≡ delim {
        section_text = append(section_text, c)
        break
      }
      if c ≡ '\\' {
        if loc ≥ len(buffer) {
          continue
        }
        section_text = append(section_text, '\\')
        c = buffer[loc]
        loc++
      }
      section_text = append(section_text, c)
    }
    id = section_text
    return strs
  }
```

This code is used in section 133.

137. After an `@` sign has been scanned, the next character tells us whether there is more work to do.

```
⟨ Get control code and possible section name 137 ⟩ ≡
  {
    c = ccode[nc]
    loc++
    switch c {
      case ignore:
        continue
      case control_text:
        for c = skip_ahead(); c ≡ '@'; c = skip_ahead() {}
            /* only @@ and @> are expected */
        if buffer[loc − 1] ≠ '>' {
          err_print("! Double @ should be used in control text")
        }
        continue
      case section_name:
        cur_section_name_char = buffer[loc − 1]
        ⟨ Scan the section name and make cur_section_name point to it 139 ⟩
      case strs:
        ⟨ Scan a verbatim string 143 ⟩
      case ord:
        ⟨ Scan an ASCII constant 138 ⟩
      default:
        return c
    }
  }
```

This code is cited in section 156.

This code is used in section 133.

138. After scanning a valid ASCII constant that follows `@'`, this code plows ahead until it finds the next single quote. (Special care is taken if the quote is part of the constant.) Anything after a valid ASCII constant is ignored; thus, `@'\nopq'` gives the same result as `@'\n'`.

```
⟨ Scan an ASCII constant 138 ⟩ ≡
  if buffer[loc] ≡ '\\' {
    loc++
    if buffer[loc] ≡ '\'' {
      loc++
    }
  }
  for buffer[loc] ≠ '\'' {
    if buffer[loc] ≡ '@' {
      if buffer[loc + 1] ≠ '@' {
        err_print("! Double @ should be used in ASCII constant")
      } else {
        loc++
      }
    }
    loc++
    if loc ≥ len(buffer) {
      err_print("! String didn't end")
      loc = len(buffer) − 1
      break
    }
  }
  loc++
  return ord
```

This code is used in section 137.

139. ⟨ Scan the section name and make *cur_section_name* point to it 139 ⟩ ≡

```
{
  section_text = section_text[0:0]
  ⟨ Put section name into section_text 141 ⟩
  if len(section_text)⟩3 ∧ compare_runes(section_text[len(section_text) − 3:], []rune("...")) ≡ 0 {
    cur_section_name = section_lookup(section_text[0:len(section_text) − 3], true)
      /* 1 means is a prefix */
  } else {
    cur_section_name = section_lookup(section_text, false)
  }
  if cur_section_name_char ≡ '(' {
    ⟨ If it's not there, add cur_section_name to the output file stack, or complain we're out of room 116 ⟩
  }
  return section_name
}
```

This code is used in section 137.

140. Section names are placed into the *section_text* array with consecutive spaces, tabs, and carriage-returns replaced by single spaces. There will be no spaces at the beginning or the end.

141. ⟨ Put section name into *section_text* 141 ⟩ ≡

```
for true {
  if loc ≥ len(buffer) {
    if ¬get_line() {
      err_print("!␣Input␣ended␣in␣section␣name")
      loc = 1
      break
    }
    if len(section_text)⟩0 {
      section_text = append(section_text, '␣')
    }
  }
  c = buffer[loc]
  ⟨ If end of name or erroneous nesting, break 142 ⟩
  loc++
  if unicode.IsSpace(c) {
    c = '␣'
    if len(section_text)⟩0 ∧ section_text[len(section_text) − 1] ≡ '␣' {
      section_text = section_text[:len(section_text) − 1]
    }
  }
  section_text = append(section_text, c)
}
```

This code is used in section 139.

142. ⟨ If end of name or erroneous nesting, break 142 ⟩ ≡

```
if c ≡ '@' {
  if loc + 1 ≥ len(buffer) {
    err_print("!␣Section␣name␣didn't␣end")
    break
  }
  c = buffer[loc + 1]
  if (c ≡ '>') {
    loc += 2
    break
  }
  cc := ignore
  if c⟨int32(len(ccode)) {
    cc = ccode[c]
  }
  if cc ≡ new_section {
    err_print("!␣Section␣name␣didn't␣end")
    break
  }
  if cc ≡ section_name {
    err_print("!␣Nesting␣of␣section␣names␣not␣allowed")
    break
  }
  section_text = append(section_text, '@')
  loc++      /* now c ≡ buffer[loc] again */
}
```

This code is used in section 141.

143. At the present point in the program we have $buffer[loc-1] \equiv strs$; we set *id_first* to the beginning of the string itself, and *loc* to the position just after the ending delimiter.

```
⟨ Scan a verbatim string 143 ⟩ ≡
  {
    id_first := loc
    loc++
    for loc⟨len(buffer) {
      if buffer[loc] ≠ '@' {
        loc++
        continue
      }
      loc++
      if loc ≡ len(buffer) {
        break
      }
      if buffer[loc] ≡ '>' {
        break
      }
    }
    if loc ≥ len(buffer) {
      err_print("! Verbatim string didn't end")
    }
    id = buffer[id_first:loc - 1]
    loc += 1
    return strs
  }
```

This code is used in section 137.

144. Scanning a macro definition. The rules for generating the replacement texts corresponding to macros and Go texts of a section are almost identical; the only differences are that

a) Section names are not allowed in macros; in fact, the appearance of a section name terminates such macros and denotes the name of the current section.
b) The symbols `@d` and `@f` and `@c` are not allowed after section names, while they terminate macro definitions.

Therefore there is a single procedure *scan_repl* whose parameter t specifies either *macro* or *section_name*. After *scan_repl* has acted, *cur_text* will point to the replacement text just generated, and *next_control* will contain the control code that terminated the activity.

```
⟨ Constants 1 ⟩ +≡
  macro = 0
```

145.
```
⟨ Global variables 93 ⟩ +≡
  var cur_text int32      /* replacement text formed by scan_repl */
  var next_control rune
```

146.
```
      /* creates a replacement text */
  func scan_repl(t rune) { var a int32      /* the current token */
  if t ≡ section_name{⟨ Insert the line number into tok_mem 148 ⟩}
  for true {
    a = get_next()
    switch a {
      ⟨ In cases that a is a non-char token (identifier, section_name, etc.), either process it and change a
          to a byte that should be stored, or continue if a should be ignored, or goto done if a signals
          the end of this replacement text 149 ⟩
      case ')':
        tok_mem = append(tok_mem, a)
        if t ≡ macro {
          tok_mem = append(tok_mem, '␣')
        }
      default:
        tok_mem = append(tok_mem, a)      /* store a in tok_mem */
    }
  }
  done:
  next_control = a
  cur_text = int32(len(text_info))
  text_info = append(text_info, text{})
  text_info[cur_text].token = tok_mem
  tok_mem = nil
  }
```

147. Here is the code for the line number: a first element is equal to *unicode.UpperLower* plus *line_number*; then the numeric line number; then a pointer to the file name.

```
⟨ Constants 1 ⟩ +≡
  line_number = °214
```

148. ⟨ Insert the line number into *tok_mem* 148 ⟩ ≡

```
tok_mem = append(tok_mem, unicode.UpperLower + line_number)
if changing {
  id = []rune(change_file_name)
} else {
  id = []rune(file_name[include_depth])
}
if changing {
  tok_mem = append(tok_mem, rune(change_line))
} else {
  tok_mem = append(tok_mem, rune(line[include_depth]))
}
{
  a := id_lookup(id, 0)
  tok_mem = append(tok_mem, a)
}
```

This code is used in sections 133, 146, and 149.

149. ⟨ In cases that *a* is a non-*char* token (*identifier*, *section_name*, etc.), either process it and change *a* to a byte that should be stored, or **continue** if *a* should be ignored, or **goto** *done* if *a* signals the end of this replacement text 149 ⟩ ≡

```
case identifier:
  a = id_lookup(id, 0)
  tok_mem = append(tok_mem, unicode.UpperLower + identifier)
  tok_mem = append(tok_mem, a)
case section_name:
  if t ≠ section_name {
    goto done
  } else {
    ⟨ Was an '@' missed here? 150 ⟩
    tok_mem = append(tok_mem, unicode.UpperLower + section_name)
    a = cur_section_name
    tok_mem = append(tok_mem, a)
    ⟨ Insert the line number into tok_mem 148 ⟩
  }
case constant, strs:
  ⟨ Copy a string or verbatim construction or numerical constant 151 ⟩
case comment:
  ⟨ Copy a comment 152 ⟩
case ord:
  ⟨ Copy an ASCII constant 153 ⟩
case definition, format_code, begin_code:
  if t ≠ section_name {
    goto done
  } else {
    err_print("! @d, @f and @c are ignored in Go text")
    continue
  }
case new_section:
  goto done
```

This code is used in section 146.

150. ⟨ Was an '@' missed here? 150 ⟩ ≡

```
{
  try_loc := loc
  for try_loc⟨len(buffer) ∧ buffer[try_loc] ≡ '␣' {
    try_loc++
  }
  if try_loc⟨len(buffer) ∧ buffer[try_loc] ≡ '+' {
    try_loc++
  }
  for try_loc⟨len(buffer) ∧ buffer[try_loc] ≡ '␣' {
    try_loc++
  }
  if try_loc⟨len(buffer) ∧ buffer[try_loc] ≡ '=' {
    err_print("!␣Missing␣'@␣'␣before␣a␣named␣section")
  }
    /* user who isn't defining a section should put newline after the name, as explained in the manual
    */
}
```

This code is used in section 149.

151. ⟨ Copy a string or verbatim construction or numerical constant 151 ⟩ ≡

```
tok_mem = append(tok_mem, a)    /* string or constant */
for i := 0; i⟨len(id); {    /* simplify @@ pairs */
  if id[i] ≡ '@' {
    if id[i + 1] ≡ '@' {
      i++
    } else {
      err_print("!␣Double␣@␣should␣be␣used␣in␣string")
    }
  }
  tok_mem = append(tok_mem, id[i])
  i++
}
tok_mem = append(tok_mem, a)
```

This code is used in section 149.

152. Comments are copied as is except some symbols

⟨ Copy a comment 152 ⟩ ≡

```
tok_mem = append(tok_mem, a)    /* comment */
for i := 0; i⟨len(id); {
  if id[i] ≡ '|' {
    i++
    continue
  }
  tok_mem = append(tok_mem, id[i])
  i++
}
tok_mem = append(tok_mem, a)    /* comment */
```

This code is used in section 149.

153. ⟨ Copy an ASCII constant 153 ⟩ ≡

```
{
  c := id[0]
  if c ≡ '\\' {
    id = id[1:]
    c = id[0]
    if c ≥ '0' ∧ c ≤ '7' {
      c -= '0'
      if id[1] ≥ '0' ∧ id[1] ≤ '7' {
        id = id[1:]
        c = 8 * c + id[0] - '0'
        if id[1] ≥ '0' ∧ id[1] ≤ '7' ∧ c⟨32 {
          id = id[1:]
          c = 8 * c + id[0] - '0'
        }
      }
    } else {
      switch c {
        case 't':
          c = '\t'
        case 'n':
          c = '\n'
        case 'b':
          c = '\b'
        case 'f':
          c = '\f'
        case 'v':
          c = '\v'
        case 'r':
          c = '\r'
        case 'a':
          c = '\a'
        case '?':
          c = '?'
        case 'x':
          if unicode.IsDigit(id[1]) {
            id = id[1:]
            c = id[0] - '0'
          } else if xisxdigit(id[1]) ∧ unicode.IsLower(id[1]) {
            id = id[1:]
            c = unicode.ToUpper(id[0]) - 'A' + 10
          }
          if unicode.IsDigit(id[1]) {
            id = id[1:]
            c = 16 * c + id[0] - '0'
          } else if xisxdigit(id[1]) ∧ unicode.IsLower(id[1]) {
            id = id[1:]
            c = 16 * c + unicode.ToUpper(id[0]) - 'A' + 10
          }
        case '\\':
          c = '\\'
        case '\'':
```

```
            c = '\''
          case '"':
            c = '"'
          default:
            err_print("! Unrecognized escape sequence")
        }
      }
    }
        /* at this point c should have been converted to its ASCII code number */
    tok_mem = append(tok_mem, constant)
    if c ≥ 100 {
      tok_mem = append(tok_mem, '0' + c/100)
    }
    if c ≥ 10 {
      tok_mem = append(tok_mem, '0' + (c/10) % 10)
    }
    tok_mem = append(tok_mem, '0' + c % 10)
    tok_mem = append(tok_mem, constant)
  }
```

This code is used in section 149.

154. Scanning a section. The *scan_section* procedure starts when '`@␣`' or '`@*`' has been sensed in the input, and it proceeds until the end of that section. It uses *section_count* to keep track of the current section number; with luck, `GOWEAVE` and `GOTANGLE` will both assign the same numbers to sections.

155. The body of *scan_section* is a loop where we look for control codes that are significant to `GOTANGLE`: those that delimit a definition, the Go part of a module, or a new module.

```
func scan_section(){
  var p int32 = 0      /* section name for the current section */
  var q int32 = 0      /* text for the current section */
  var a int32 = 0      /* token for left-hand side of definition */
  section_count++
  no_where = true
  if loc⟨len(buffer) ∧ buffer[loc − 1] ≡ '*' ∧ show_progress() {      /* starred section */
    fmt.Printf("*%d", section_count)
    os.Stdout.Sync()
  }
  next_control = 0
  for true {
    ⟨Skip ahead until next_control corresponds to @d, @<, @␣ or the like 156⟩
    if next_control ≡ definition {      /* @d */
      ⟨Scan a definition 157⟩
      continue
    }
    if next_control ≡ begin_code {      /* @c or @p */
      p = −1
      break
    }
    if next_control ≡ section_name {      /* @< or @( */
      p = cur_section_name
      ⟨If section is not being defined, continue 158⟩
      break
    }
    return      /* @␣ or @* */
  }
  no_where = false
  print_where = false
  ⟨Scan the Go part of the current section 159⟩
}
```

156. At the top of this loop, if *next_control* ≡ *section_name*, the section name has already been scanned (see ⟨ Get control code and possible section name 137 ⟩). Thus, if we encounter *next_control* ≡ *section_name* in the skip-ahead process, we should likewise scan the section name, so later processing will be the same in both cases.

```
⟨ Skip ahead until next_control corresponds to @d, @<, @␣ or the like 156 ⟩ ≡
  for next_control⟨definition {
      /* definition is the lowest of the "significant" codes */
    if next_control = skip_ahead(); next_control ≡ section_name {
      loc −= 2
      next_control = get_next()
    }
  }
```

This code is used in section 155.

157.

```
⟨ Scan a definition 157 ⟩ ≡
  {
      /*allow newline before definition */
    for next_control = get_next(); next_control ≡ '\n'; next_control = get_next() {}
    if next_control ≠ identifier {
      err_print("!␣Definition␣flushed,␣must␣start␣with␣identifier")
      continue
    }
    a = id_lookup(id, 0)
    tok_mem = append(tok_mem, unicode.UpperLower + identifier)
    tok_mem = append(tok_mem, a)
        /* append the lhs */
    if loc⟨len(buffer) ∧ buffer[loc] ≠ '(' {      /* identifier must be separated from replacement text */
      tok_mem = append(tok_mem, strs)
      tok_mem = append(tok_mem, '␣')
      tok_mem = append(tok_mem, strs)
    }
    scan_repl(macro)
    text_info[cur_text].text_link = 0      /* text_link ≡ 0 characterizes a macro */
  }
```

This code is used in section 155.

158. If the section name is not followed by `=` or `+=`, no Go code is forthcoming: the section is being cited, not being defined. This use is illegal after the definition part of the current section has started, except inside a comment, but `GOTANGLE` does not enforce this rule; it simply ignores the offending section name and everything following it, up to the next significant control code.

```
⟨ If section is not being defined, continue 158 ⟩ ≡
      /* allow optional += */
  for next_control = get_next(); next_control ≡ '+'; next_control = get_next() {}
  if next_control ≠ '=' ∧ next_control ≠ eq_eq {
    continue
  }
```

This code is used in section 155.

159. ⟨ Scan the Go part of the current section 159 ⟩ ≡
```
⟨ Insert the section number into tok_mem 160 ⟩
scan_repl(section_name)      /* now cur_text points to the replacement text */
⟨ Update the data structure so that the replacement text is accessible 161 ⟩
```
This code is used in section 155.

160. ⟨ Insert the section number into tok_mem 160 ⟩ ≡
```
tok_mem = append(tok_mem, unicode.UpperLower + section_number)
tok_mem = append(tok_mem, section_count)
```
This code is used in section 159.

161. ⟨ Update the data structure so that the replacement text is accessible 161 ⟩ ≡
```
if p ≡ −1 {      /* unnamed section, or bad section name */
  text_info[last_unnamed].text_link = cur_text
  last_unnamed = cur_text
} else if name_dir[p].equiv ≡ −1 {
  name_dir[p].equiv = cur_text
      /* first section of this name */
} else {
  q = name_dir[p].equiv
  for text_info[q].text_link⟨max_texts {
    q = text_info[q].text_link      /* find end of list */
  }
  text_info[q].text_link = cur_text
}
text_info[cur_text].text_link = max_texts
      /* mark this replacement text as a nonmacro */
```
This code is used in section 159.

162.
```
func phase_one(){
  phase = 1
  section_count = 0
  reset_input()
  skip_limbo()
  for ¬input_has_ended {
    scan_section()
  }
  check_complete()
  phase = 2
}
```

163. Only a small subset of the control codes is legal in limbo, so limbo processing is straightforward.

```
func skip_limbo(){
  for true {
    if loc ≥ len(buffer) ∧ ¬get_line() {
      return
    }
    for loc⟨len(buffer) ∧ buffer[loc] ≠ '@' {
      loc++
    }
    if loc++; loc⟨len(buffer) {
      c := buffer[loc]
      loc++
      cc := ignore
      if c⟨int32(len(ccode)) {
        cc = ccode[c]
      }
      if cc ≡ new_section {
        break
      }
      switch cc {
        case format_code, '@':case control_text:
          if c ≡ 'q' ∨ c ≡ 'Q' {
            for c = skip_ahead(); c ≡ '@'; c = skip_ahead() {}
            if buffer[loc − 1] ≠ '>' {
              err_print("! Double @ should be used in control text")
            }
            break
          }
          fallthrough
        default:
          err_print("! Double @ should be used in limbo")
      }
    }
  }
}
```

164.

```
func print_stats(){
  fmt.Print("\nMemory usage statistics:\n")
  fmt.Printf("%v names\n", len(name_dir))
  fmt.Printf("%v replacement texts\n", len(text_info))
}
```

165. `GOTANGLE` specific creation of output file

⟨ Try to open output file 165 ⟩ ≡

```
var err error
if go_file, err = os.OpenFile(go_file_name, os.O_WRONLY | os.O_CREATE | os.O_TRUNC, °666); err ≠ nil {
  fatal("! Cannot open output file ", go_file_name)
}
```

This code is used in section 89.

166. ⟨ Print usage error message and quit 166 ⟩ ≡

```
{
  fatal("!␣Usage:␣gotangle␣[options]␣webfile[.w]␣[{changefile[.ch]|-}␣[outfile[.go]]]\n",
      "")
}
```

This code is used in section 83.

167. Index. Here is a cross-reference table for GOTANGLE. All sections in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in section names are not indexed. Underlined entries correspond to where the identifier was declared. Error messages and a few other things are indexed here too.

⟨ Case of a line number 123 ⟩ Used in section 119.
⟨ Case of a section number 122 ⟩ Used in section 119.
⟨ Case of an identifier 121 ⟩ Used in section 119.
⟨ Cases like `!=` 120 ⟩ Used in section 119.
⟨ Common constants 10, 31, 42, 55, 63, 65 ⟩ Used in section 6.
⟨ Compress two-symbol operator 91 ⟩ Used in section 133.
⟨ Compute the hash code h 46 ⟩ Used in section 45.
⟨ Compute the name location p 47 ⟩ Used in section 45.
⟨ Constants 1, 4, 100, 106, 111, 125, 129, 131, 144, 147 ⟩ Used in section 2.
⟨ Copy a comment 152 ⟩ Used in section 149.
⟨ Copy a string or verbatim construction or numerical constant 151 ⟩ Used in section 149.
⟨ Copy an ASCII constant 153 ⟩ Used in section 149.
⟨ Definitions that should agree with `GOTANGLE` and `GOWEAVE` 12, 17, 32, 40, 43, 68, 81, 88 ⟩ Used in section 6.
⟨ Enter a new name into the table at position p 49 ⟩ Used in section 45.
⟨ Expand section c, **goto** *restart* 109 ⟩ Used in section 108.
⟨ Get a constant 135 ⟩ Used in section 133.
⟨ Get a string 136 ⟩ Used in section 133.
⟨ Get an identifier 134 ⟩ Used in section 133.
⟨ Get control code and possible section name 137 ⟩ Cited in section 156. Used in section 133.
⟨ Global variables 93, 98, 102, 107, 112, 115, 126, 132, 145 ⟩ Used in section 2.
⟨ Handle flag argument 87 ⟩ Used in section 83.
⟨ If end of name or erroneous nesting, **break** 142 ⟩ Used in section 141.
⟨ If it's not there, add *cur_section_name* to the output file stack, or complain we're out of room 116 ⟩ Used in section 139.
⟨ If no match found, add new name to tree 61 ⟩ Used in section 59.
⟨ If one match found, check for compatibility and return match 62 ⟩ Used in section 59.
⟨ If section is not being defined, **continue** 158 ⟩ Used in section 155.
⟨ If the current line starts with `@y`, report any discrepancies and **return** 28 ⟩ Used in section 26.
⟨ Import packages 13, 16, 20, 27, 34 ⟩ Used in section 2.
⟨ In cases that a is a non-*char* token (*identifier*, *section_name*, etc.), either process it and change a to a byte that should be stored, or **continue** if a should be ignored, or **goto** *done* if a signals the end of this replacement text 149 ⟩ Used in section 146.
⟨ Initialization of a new identifier 97 ⟩ Used in section 49.
⟨ Initialize pointers 44, 51 ⟩ Used in section 8.
⟨ Initialize the output stacks 103 ⟩ Used in section 117.
⟨ Insert the line number into *tok_mem* 148 ⟩ Used in sections 133, 146, and 149.
⟨ Insert the section number into *tok_mem* 160 ⟩ Used in section 159.
⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 60 ⟩ Used in section 59.
⟨ Make *change_file_name* from *fname* 85 ⟩ Used in section 83.
⟨ Make *file_name*[0], *tex_file_name*, and *go_file_name* 84 ⟩ Used in section 83.
⟨ More elements of *name_info* structure 41, 50, 94 ⟩ Used in section 40.
⟨ Move *buffer* to *change_buffer* 23 ⟩ Used in sections 19 and 26.
⟨ Open input files 30 ⟩ Used in section 29.
⟨ Other definitions 7, 18 ⟩ Used in section 6.
⟨ Override *tex_file_name* and *go_file_name* 86 ⟩ Used in section 83.
⟨ Print error location based on input buffer 71 ⟩ Used in section 69.
⟨ Print the job *history* 74 ⟩ Used in section 73.
⟨ Print usage error message and quit 166 ⟩ Used in section 83.
⟨ Put section name into *section_text* 141 ⟩ Used in section 139.
⟨ Read from *change_file* and maybe turn off *changing* 37 ⟩ Used in section 33.
⟨ Read from *file*[*include_depth*] and maybe turn on *changing* 36 ⟩ Used in section 33.

⟨ Scan a definition 157 ⟩ Used in section 155.
⟨ Scan a verbatim string 143 ⟩ Used in section 137.
⟨ Scan an ASCII constant 138 ⟩ Used in section 137.
⟨ Scan arguments and open output files 89 ⟩ Used in section 8.
⟨ Scan the Go part of the current section 159 ⟩ Used in section 155.
⟨ Scan the section name and make *cur_section_name* point to it 139 ⟩ Used in section 137.
⟨ Set initial values 99, 127 ⟩ Used in section 3.
⟨ Set the default options common to `GOTANGLE` and `GOWEAVE` 82 ⟩ Used in section 8.
⟨ Skip ahead until *next_control* corresponds to `@d`, `@<`, `@␣` or the like 156 ⟩ Used in section 155.
⟨ Skip over comment lines in the change file; **return** if end of file 21 ⟩ Used in section 19.
⟨ Skip to the next nonblank line; **return** if end of file 22 ⟩ Used in section 19.
⟨ Try to open include file, abort push if unsuccessful, go to *restart* 35 ⟩ Used in section 33.
⟨ Try to open output file 165 ⟩ Used in section 89.
⟨ Typedef declarations 92, 101 ⟩ Used in section 2.
⟨ Update the data structure so that the replacement text is accessible 161 ⟩ Used in section 159.
⟨ Was an '@' missed here? 150 ⟩ Used in section 149.
⟨ Write all the named output files 118 ⟩ Used in section 117.

The GOTANGLE processor

(Version 0.82)