1. Introduction. This is the GOWEAVE program by Alexander Sychev based on CWEAVE by Silvio Levy and Donald E. Knuth.

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

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

See also sections 4, 97, 112, 121, 168, 172, 301, 324, 331, and 373.

This code is used in section 2.

2.

```
package main
import(
  ⟨ Import packages 13 ⟩
)
⟨ Typedef declarations 94 ⟩
⟨ Constants 1 ⟩
⟨ Global variables 93 ⟩
```

3. GOWEAVE has a fairly straightforward outline. It 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, finally it sorts and outputs the index.

```
func main(){
  flags['c'] = true
  flags['x'] = true
  flags['f'] = true
  flags['e'] = true      /* controlled by command-line options */
  common_init()
  ⟨ Set initial values 99 ⟩
  if show_banner() {
    fmt.Print(banner)      /* print a "banner line" */
  }
  ⟨ Store all the reserved words 109 ⟩
  phase_one()      /* read all the user's text and store the cross-references */
  phase_two()      /* read all the text again and translate it to TEX form */
  phase_three()      /* output the cross-reference index */
  os.Exit(wrap_up())      /* and exit gracefully */
}
```

4. The following parameters were sufficient in the original WEAVE to handle TEX, so they should be sufficient for most applications of GOWEAVE.

```
⟨ Constants 1 ⟩ +≡
  const(
    max_names = 4000      /* number of identifiers, strings, section names must be less than 10240 */
    line_length = 80
      /* lines of TEX output have at most this many characters should be less than 256 */
  )
```

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, 92, and 98.

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 108 ⟩
```
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 386⟩
}
}
}
if ¬found_web{⟨Print usage error message and quit 386⟩}
}
```

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 387 ⟩
```

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 123.

92. Data structures exclusive to GOWEAVE. As explained above, the field of a *name_info* structure that contains the *rlink* of a section name is used for a completely different purpose in the case of identifiers. It is then called the *ilk* of the identifier, and it is used to distinguish between various types of identifiers, as follows:

normal identifiers are part of the Go program that will appear in italic type (or in typewriter type if all uppercase).

custom identifiers are part of the Go program that will be typeset in special ways.

roman identifiers are index entries that appear after `@^` in the CWEB file.

wildcard identifiers are index entries that appear after `@:` in the CWEB file.

typewriter identifiers are index entries that appear after `@.` in the CWEB file.

zero, ... identifiers are Go reserved words and productions whose *ilk* explains how they are to be treated when Go code is being formatted.

```
⟨ More elements of name_info structure 41 ⟩ +≡
  ilk int32      /* used by identifiers in GOWEAVE only */
```

93. We keep track of the current section number in *section_count*, which is the total number of sections that have started. Sections which have been altered by a change file entry have their *changed_section* flag turned on during the first phase.

```
⟨ Global variables 93 ⟩ ≡
  var change_exists bool      /* has any section changed? */
```

See also sections 95, 115, 122, 135, 142, 147, 150, 169, 175, 179, 305, 327, 330, 347, 354, 365, 369, 371, and 381.

This code is used in section 2.

94. The other large memory area in GOWEAVE keeps the cross-reference data. All uses of the name p are recorded in a linked list beginning at $name\_dir[p].xref$, which is an index in the *xmem* array. The elements of *xmem* are structures consisting of an integer, *num*, and an index *xlink* to another element of *xmem*. If $x = name\_dir[p].xref$ is an index into *xmem*, the value of $xmem[x].num$ is either a section number where p is used, or *cite_flag* plus a section number where p is mentioned, or *def_flag* plus a section number where p is defined; and $xmem[x].xlink$ points to the next such cross-reference for p, if any. This list of cross-references is in decreasing order by section number. The linked list ends at -1.

The global variable *xref_switch* is set either to *def_flag* or to zero, depending on whether the next cross-reference to an identifier is to be underlined or not in the index. This switch is set to *def_flag* when `@!` is scanned, and it is cleared to zero when the next identifier or index entry cross-reference has been made. Similarly, the global variable *section_xref_switch* is either *def_flag* or *cite_flag* or zero, depending on whether a section name is being defined, cited or used in Go text.

```
⟨ Typedef declarations 94 ⟩ ≡
  type xref_info struct{
    num int32      /* section number plus zero or def_flag */
    xlink int32      /* index of the previous cross-reference */
  }
```

See also sections 174, 176, 178, 184, 323, and 325.

This code is used in section 2.

95.
```
⟨ Global variables 93 ⟩ +≡
  var xmem []xref_info      /* contains cross-reference information */
  var xref_switch int32
  var section_xref_switch int32      /* either zero or def_flag */
```

96. A section that is used for multi-file output (with the `@(` feature) has a special first cross-reference whose *num* field is *file_flag*.

97. ⟨ Constants 1 ⟩ +≡

```
const(
    cite_flag  =  10240
    file_flag  =  3 * cite_flag
    def_flag   =  2 * cite_flag
)
```

98. ⟨ More elements of *name_info* structure 41 ⟩ +≡

```
xref int32      /* info corresponding to names */
```

99. ⟨ Set initial values 99 ⟩ ≡

```
xmem = append(xmem, xref_info{})
xref_switch = 0
section_xref_switch = 0
```

See also sections 116, 153, 156, 170, and 370.

This code is used in section 3.

100. A new cross-reference for an identifier is formed by calling *new_xref*, which discards duplicate entries and ignores non-underlined references to one-letter identifiers or Go's reserved words.

If the user has sent the *flags*['x'] ≡ **false** flag (the `-x` option of the command line), it is unnecessary to keep track of cross-references for identifiers. If one were careful, one could probably make more changes around section 100 to avoid a lot of identifier looking up.

```
func append_xref(c int32){
    xmem = append(xmem, xref_info{})
    xmem[len(xmem) − 1].num = c
    xmem[len(xmem) − 1].xlink = 0
}
```

101.

```
func is_tiny(p int32) bool{
    return p⟨int32(len(name_dir)) ∧ len(name_dir[p].name) ≡ 1
}
```

102.

```
    /* tells if uses of a name are to be indexed */
func unindexed(p int32) bool{
    return p⟨res_wd_end ∧ name_dir[p].ilk ≥ custom
}
```

103.

```
func new_xref(p int32){
  if flags['x'] ≡ false {
    return
  }
  if (unindexed(p) ∨ is_tiny(p)) ∧ xref_switch ≡ 0 {
    return
  }
  m := section_count + xref_switch
  xref_switch = 0
  q := name_dir[p].xref       /* pointer to previous cross-reference */
  if q ≥ 0 {
    n := xmem[q].num          /* new and previous cross-reference value */
    if n ≡ m ∨ n ≡ m + def_flag {
      return
    } else if m ≡ n + def_flag {
      xmem[q].num = m
      return
    }
  }
  append_xref(m)
  xmem[len(xmem) − 1].xlink = int32(q)
  name_dir[p].xref = int32(len(xmem) − 1)
}
```

104. The cross-reference lists for section names are slightly different. Suppose that a section name is defined in sections $m_1, \ldots, m_k$, cited in sections $n_1, \ldots, n_l$, and used in sections $p_1, \ldots, p_j$. Then its list will contain $m_1 + def\_flag, \ldots, m_k + def\_flag, n_1 + cite\_flag, \ldots, n_l + cite\_flag, p_1, \ldots, p_j$, in this order.

Although this method of storage takes quadratic time with respect to the length of the list, under foreseeable uses of GOWEAVE this inefficiency is insignificant.

```
func new_section_xref(p int32){
  var r int32 = 0       /* pointers to previous cross-references */
  q := name_dir[p].xref
  if q ≥ 0 {
    for q ≥ 0 ∧ q⟨int32(len(xmem)) ∧ xmem[q].num⟩section_xref_switch {
      r = q
      q = xmem[q].xlink
    }
  }
  if r⟩0 ∧ r⟨int32(len(xmem)) ∧ xmem[r].num ≡ section_count + section_xref_switch {
    return      /* don't duplicate entries */
  }
  append_xref(section_count + section_xref_switch)
  xmem[len(xmem) − 1].xlink = q
  section_xref_switch = 0
  if r ≡ 0 {
    name_dir[p].xref = int32(len(xmem) − 1)
  } else {
    xmem[r].xlink = int32(len(xmem) − 1)
  }
}
```

105. The cross-reference list for a section name may also begin with *file_flag*. Here's how that flag gets put in.

```
func set_file_flag(p int32){
  q := name_dir[p].xref
  if xmem[q].num ≡ file_flag {
    return
  }
  append_xref(file_flag)
  xmem[len(xmem) − 1].xlink = q
  name_dir[p].xref = int32(len(xmem) − 1)
}
```

106. Here are the procedure needed to complete *id_lookup*:

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

107. *init_node* is used in *gocommon.w* to init a new node

```
func init_node(p int32){
  name_dir[p].xref = 0
}
```

108. With a next code GOWEAVE makes a specific initialization of a new identifier. .

```
⟨ Initialization of a new identifier 108 ⟩ ≡
  name_dir[p].ilk = t
  name_dir[p].xref = 0
```

This code is used in section 49.

109. We have to get Go's reserved words into the hash table, and the simplest way to do this is to insert them every time GOWEAVE is run. Fortunately there are relatively few reserved words.

⟨ Store all the reserved words 109 ⟩ ≡

id_lookup([]**rune**("break"), *break_token*)
id_lookup([]**rune**("case"), *case_token*)
id_lookup([]**rune**("chan"), *chan_token*)
id_lookup([]**rune**("const"), *const_token*)
id_lookup([]**rune**("continue"), *continue_token*)
id_lookup([]**rune**("default"), *default_token*)
id_lookup([]**rune**("defer"), *defer_token*)
id_lookup([]**rune**("else"), *else_token*)
id_lookup([]**rune**("fallthrough"), *fallthrough_token*)
id_lookup([]**rune**("for"), *for_token*)
id_lookup([]**rune**("func"), *func_token*)
id_lookup([]**rune**("go"), *go_token*)
id_lookup([]**rune**("goto"), *goto_token*)
id_lookup([]**rune**("if"), *if_token*)
id_lookup([]**rune**("import"), *import_token*)
id_lookup([]**rune**("interface"), *interface_token*)
id_lookup([]**rune**("map"), *map_token*)
id_lookup([]**rune**("package"), *package_token*)
id_lookup([]**rune**("range"), *range_token*)
id_lookup([]**rune**("return"), *return_token*)
id_lookup([]**rune**("select"), *select_token*)
id_lookup([]**rune**("struct"), *struct_token*)
id_lookup([]**rune**("switch"), *switch_token*)
id_lookup([]**rune**("type"), *type_token*)
id_lookup([]**rune**("var"), *var_token*)

id_lookup([]**rune**("bool"), *Type*)
id_lookup([]**rune**("byte"), *Type*)
id_lookup([]**rune**("complex64"), *Type*)
id_lookup([]**rune**("complex128"), *Type*)
id_lookup([]**rune**("error"), *Type*)
id_lookup([]**rune**("float32"), *Type*)
id_lookup([]**rune**("float64"), *Type*)
id_lookup([]**rune**("int"), *Type*)
id_lookup([]**rune**("int8"), *Type*)
id_lookup([]**rune**("int16"), *Type*)
id_lookup([]**rune**("int32"), *Type*)
id_lookup([]**rune**("int64"), *Type*)
id_lookup([]**rune**("rune"), *Type*)
id_lookup([]**rune**("string"), *Type*)
id_lookup([]**rune**("uint"), *Type*)
id_lookup([]**rune**("uint8"), *Type*)
id_lookup([]**rune**("uint16"), *Type*)
id_lookup([]**rune**("uint32"), *Type*)
id_lookup([]**rune**("uint64"), *Type*)
id_lookup([]**rune**("uintptr"), *Type*)

id_lookup([]**rune**("true"), *constant*)
id_lookup([]**rune**("false"), *constant*)
id_lookup([]**rune**("iota"), *constant*)

id_lookup([]**rune**("nil"), *constant*)

id_lookup([]**rune**("append"), *identifier*)
id_lookup([]**rune**("cap"), *identifier*)
id_lookup([]**rune**("close"), *identifier*)
id_lookup([]**rune**("complex"), *identifier*)
id_lookup([]**rune**("copy"), *identifier*)
id_lookup([]**rune**("delete"), *identifier*)
id_lookup([]**rune**("imag"), *identifier*)
id_lookup([]**rune**("len"), *identifier*)
id_lookup([]**rune**("make"), *identifier*)
id_lookup([]**rune**("new"), *identifier*)
id_lookup([]**rune**("panic"), *identifier*)
id_lookup([]**rune**("print"), *identifier*)
id_lookup([]**rune**("println"), *identifier*)
id_lookup([]**rune**("real"), *identifier*)
id_lookup([]**rune**("recover"), *identifier*)
res_wd_end = **int32**(**len**(*name_dir*))
id_lookup([]**rune**("TeX"), *custom*)

This code is used in section 3.

110. Lexical scanning. Let us now consider the subroutines that read the CWEB source file and break it into meaningful units. There are four such procedures: One simply skips to the next '@␣' or '@*' that begins a section; another passes over the TEX text at the beginning of a section; the third passes over the TEX text in a Go comment; and the last, which is the most interesting, gets the next token of a Go text.

111. Control codes in CWEB, which begin with '@', are converted into a numeric code designed to simplify GOWEAVE's logic; for example, larger numbers are given to the control codes that denote more significant milestones, and the code of *new_section* should be the largest of all. Some of these numeric control codes take the place of **rune** control codes that will not otherwise appear in the output of the scanning routines.

112. ⟨ Constants 1 ⟩ +≡

```
const(
    ignore rune = °0 /* control code of no interest to GOWEAVE */
    underline rune = '\n' /* this code will be intercepted without confusion */
    noop rune = °177 /* takes the place of ASCII delete */
    xref_roman rune = °213 /* control code for '@^' */
    xref_wildcard rune = °214 /* control code for '@:' */
    xref_typewriter rune = °215 /* control code for '@.' */
    TEX_string rune = °216 /* control code for '@t' */
    ord rune = °217 /* control code for '@'' */
    join rune = °220 /* control code for '@&' */
    thin_space rune = °221 /* control code for '@,' */
    math_break rune = °222 /* control code for '@|' */
    line_break rune = °223 /* control code for '@/' */
    big_line_break rune = °224 /* control code for '@#' */
    no_line_break rune = °225 /* control code for '@+' */
    pseudo_semi rune = °226 /* control code for '@;' */
    verbatim rune = °227 /* control code for '@=' */
    raw_TEX_string rune = °231 /* control code for '@r' */
    trace rune = °232 /* control code for '@0', '@1' and '@2' */
    format_code rune = °235 /* control code for '@f' and '@s' */
    begin_code rune = °237 /* control code for '@c' */
    section_name rune = °240 /* control code for '@<' */
    new_section rune = °241 /* control code for '@␣' and '@*' */
)
```

113. **format** *TeX_string* *TeX*

114. **format** *raw_TeX_string* *TeX*

115. Control codes are converted to GOWEAVE's internal representation by means of the table *ccode*.

⟨ Global variables 93 ⟩ +≡

```
var ccode [256]rune /* meaning of a char following @ */
```

116. ⟨ Set initial values 99 ⟩ +≡

```
{
  for c := 0; c⟨256; 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['@'] = '@'      /* 'quoted' at sign */
ccode['='] = verbatim
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['t'] = TEX_string
ccode['T'] = TEX_string
ccode['r'] = raw_TEX_string
ccode['R'] = raw_TEX_string
ccode['q'] = noop
ccode['Q'] = noop
ccode['&'] = join
ccode['<'] = section_name
ccode['('] = section_name
ccode['!'] = underline
ccode['^'] = xref_roman
ccode[':'] = xref_wildcard
ccode['.'] = xref_typewriter
ccode[','] = thin_space
ccode['|'] = math_break
ccode['/'] = line_break
ccode['#'] = big_line_break
ccode['+'] = no_line_break
ccode[';'] = pseudo_semi
ccode['\''] = ord
⟨ Special control codes for debugging 117 ⟩
```

117. Users can write from `@0` to `@9` to turn sets of different levels of tracing. The levels can be used like a bitmask combination.

```
⟨ Special control codes for debugging 117 ⟩ ≡
  ccode['0'] = trace      // turn the tracing off
  ccode['1'] = trace      // turn on a printing of irreducible scraps
  ccode['2'] = trace      // turn on a printing of a snapshot of the scrap_info
  ccode['4'] = trace      // turn on a printing of a category name is looking for
  ccode['8'] = trace      // turn on a printing of a resulting translation of a scrap
  ccode['3'] = trace
  ccode['5'] = trace
  ccode['6'] = trace
  ccode['7'] = trace
  ccode['9'] = trace
```

This code is used in section 116.

118. The *skip_limbo* routine is used on the first pass to skip through portions of the input that are not in any sections, i.e., that precede the first section. After this procedure has been called, the value of *input_has_ended* will tell whether or not a section has actually been found.

There's a complication that we will postpone until later: If the `@s` operation appears in limbo, we want to use it to adjust the default interpretation of identifiers.

```
func skip_limbo() { for { if loc ≥ len(buffer) ∧ ¬get_line() {
    return
  }
  for loc⟨len(buffer) ∧ buffer[loc] ≠ '@' {
    loc++      /* look for '@', then skip two symbols */
  }
  l := loc
  loc++
  if l⟨len(buffer) { c := new_section
  if loc⟨len(buffer) {
    c = ccode[buffer[loc]]
    loc++
  }
  if c ≡ new_section {
    return
  }
  if c ≡ noop { skip_restricted()
  } else if c ≡ format_code{⟨ Process simple format in limbo 145 ⟩}
  }
  }
  }
```

119. The *skip_TEX* routine is used on the first pass to skip through the TEX code at the beginning of a section. It returns the next control code or '|' found in the input. A *new_section* is assumed to exist at the very end of the file.

```
format skip_TeX TeX
    /* skip past pure TEX code */
func skip_TEX() rune{
  for{
    if loc ≥ len(buffer) ∧ ¬get_line() {
      return new_section
    }
    for loc⟨len(buffer) ∧ buffer[loc] ≠ '@' ∧ buffer[loc] ≠ '|' {
      loc++
    }
    l := loc
    loc++
    if l⟨len(buffer) ∧ buffer[l] ≡ '|' {
      return '|'
    }
    if loc⟨len(buffer) {
      l := loc
      loc++
      return ccode[buffer[l]]
    }
    if l⟨len(buffer) ∧ buffer[l] ≡ '@' {
      return new_section
    }
  }
  return 0
}
```

120. Inputting the next token. As stated above, GOWEAVE's most interesting lexical scanning routine is the *get_next* function that inputs the next token of Go input. However, *get_next* is not especially complicated.

The result of *get_next* is either a **rune** code for some special character, or it is a special code representing a pair of characters (e.g., '!='), or it is the numeric value computed by the *ccode* table, or it is one of the following special codes:

identifier: In this case the global variable *id* will contain an identifier, as required by the *id_lookup* routine.

str: The string will have been copied into the array *section_text*; *id* are set as above (now it is a slice of *section_text*).

constant: The constant is copied into *section_text*, with slight modifications; *id* is set.

Furthermore, some of the control codes cause *get_next* to take additional actions:

xref_roman, *xref_wildcard*, *xref_typewriter*, *TEX_string*, *raw_TEX_string*, *verbatim*: The values of *id* will have been set to the slice of the buffer.

section_name: In this case the global variable *cur_section* will point to the *byte_start* entry for the section name that has just been scanned. The value of *cur_section_char* will be '(' if the section name was preceded by @(instead of @<.

If *get_next* sees '@!' it sets *xref_switch* to *def_flag* and goes on to the next token.

In some cases a *pseudo_semi* will be added in end of line to help parse tokens more accurately.

121. ⟨ Constants 1 ⟩ +≡

```
const(
  constant rune = °210 /* Go constant */
  str rune = °211 /* Go string */
  identifier rune = °212 /* Go identifier or reserved word */
)
```

122. ⟨ Global variables 93 ⟩ +≡

```
var cur_section int32 /* name of section just scanned */
var cur_section_char rune /* the character just before that name */
```

123. As one might expect, *get_next* consists mostly of a big switch that branches to the various special cases that can arise. Go allows underscores to appear in identifiers, and some Go compilers even allow the dollar sign.

```
    /* produces the next input token */
func get_next() rune{
  for{
    if loc ≥ len(buffer) {
      // Looking for last non-insert scrap
      i := len(scrap_info) − 1
      for ; i ≥ 0 ∧ scrap_info[i].cat ≡ insert; i−− {}
      if i ≥ 0 ∧
        (scrap_info[i].cat ≡ identifier ∨
        scrap_info[i].cat ≡ constant ∨
        scrap_info[i].cat ≡ str ∨
        scrap_info[i].cat ≡ break_token ∨
        scrap_info[i].cat ≡ continue_token ∨
        scrap_info[i].cat ≡ fallthrough_token ∨
        scrap_info[i].cat ≡ return_token ∨
        scrap_info[i].cat ≡ plus_plus ∨
        scrap_info[i].cat ≡ minus_minus ∨
        scrap_info[i].cat ≡ rpar ∨
        scrap_info[i].cat ≡ rbracket ∨
        scrap_info[i].cat ≡ rbrace ∨
        scrap_info[i].cat ≡ Type) {
        return pseudo_semi
      }
      if ¬get_line() {
        return new_section
      }
    }
    c := buffer[loc]      /* the current character */
    loc++
    nc := ' '
    if loc⟨len(buffer) {
      nc = buffer[loc]
    }
    if unicode.IsDigit(c) ∨ (c ≡ '.' ∧ unicode.IsDigit(nc)) {
      ⟨ Get a constant 126 ⟩
    } else if c ≡ '\'' ∨ c ≡ '"' ∨ c ≡ '`' {
      ⟨ Get a string 127 ⟩
    } else if unicode.IsLetter(c) ∨ c ≡ '_' ∧ (unicode.IsLetter(c) ∨ unicode.IsDigit(c)) {
      ⟨ Get an identifier 125 ⟩
    } else if c ≡ '@' {
      ⟨ Get control code and possible section name 128 ⟩
    } else if unicode.IsSpace(c) {
      continue      /* ignore spaces and tabs */
    }
    mistake:
    ⟨ Compress two-symbol operator 91 ⟩
    return c
  }
  return 0
```

```
}
```

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

125. ⟨ Get an identifier 125 ⟩ ≡

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

This code is used in section 123.

126. Different conventions are followed by TEX and Go to express octal and hexadecimal numbers; it is reasonable to stick to each convention within its realm. Thus the Go part of a `CWEB` file has octals introduced by `0` and hexadecimals by `0x`, but `GOWEAVE` will print with TEX macros that the user can redefine to fit the context. In order to simplify such macros, we replace some of the characters.

Notice that in this section and the next, *id* is a slice of the array *section_text*, not of *buffer*.

```
⟨ Get a constant 126 ⟩ ≡
  {
    id = nil
    is_dec := false
    if loc⟨len(buffer) ∧ buffer[loc − 1] ≡ '0' {
      if buffer[loc] ≡ 'x' ∨ buffer[loc] ≡ 'X' {      /* hex constant */
        id = append(id, '^')
        loc++
        for loc⟨len(buffer) ∧ xisxdigit(buffer[loc]) {
          id = append(id, buffer[loc])
          loc++
        }
      } else if unicode.IsDigit(buffer[loc]) {      /* octal constant */
        id = append(id, '~')
        for loc⟨len(buffer) ∧ unicode.IsDigit(buffer[loc]) {
          id = append(id, buffer[loc])
          loc++
        }
      } else {
        is_dec = true      /* decimal constant */
      }
    } else {
      is_dec = true
    }
    if is_dec {      /* decimal constant */
      if loc⟨len(buffer) ∧ buffer[loc − 1] ≡ '.' ∧ ¬unicode.IsDigit(buffer[loc]) {
        goto mistake      /* not a constant */
      }
      id = append(id, buffer[loc − 1])
      for loc⟨len(buffer) ∧ (unicode.IsDigit(buffer[loc]) ∨ buffer[loc] ≡ '.') {
        id = append(id, buffer[loc])
        loc++
      }
      if loc⟨len(buffer) ∧ (buffer[loc] ≡ 'e' ∨ buffer[loc] ≡ 'E') {      /* float constant */
        id = append(id, '_')
        loc++
        if loc⟨len(buffer) ∧ (buffer[loc] ≡ '+' ∨ buffer[loc] ≡ '-') {
          id = append(id, buffer[loc])
          loc++
        }
        for loc⟨len(buffer) ∧ unicode.IsDigit(buffer[loc]) {
          id = append(id, buffer[loc])
          loc++
        }
      }
      if loc⟨len(buffer) ∧ buffer[loc] ≡ 'i' {
        id = append(id, '$')
```

```
        id = append(id, 'i')
        loc++
      }
    }
    return constant
  }
```

This code is used in section 123.

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

```
⟨ Get a string 127 ⟩ ≡
  {
    delim := c      /* what started the string */
    section_text = section_text[0:0]
    if delim ≡ '\'' ∧ loc − 2⟨len(buffer) ∧ loc − 2 ≥ 0 ∧ buffer[loc − 2] ≡ '@' {
      section_text = append(section_text, '@')
      section_text = append(section_text, '@')
    }
    section_text = append(section_text, delim)
    for{
      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 str
  }
```

This code is used in sections 123 and 128.

128. 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 128 ⟩ ≡
  {
    c = nc
    loc++
    switch ccode[c] {
      case underline:
        xref_switch = def_flag
        continue
      case trace:
        tracing = c - '0'
        continue
      case xref_roman, xref_wildcard, xref_typewriter, noop, TEX_string, raw_TEX_string:
        c = ccode[c]
        skip_restricted()
        return c
      case section_name:
        ⟨ Scan the section name and make cur_section point to it 129 ⟩
      case verbatim:
        ⟨ Scan a verbatim string 134 ⟩
      case ord:
        ⟨ Get a string 127 ⟩
      default:
        return ccode[c]
    }
  }
```

This code is used in section 123.

129. The occurrence of a section name sets *xref_switch* to zero, because the section name might (for example) follow **int**.

```
⟨ Scan the section name and make cur_section point to it 129 ⟩ ≡
  {
    section_text = section_text[0:0]
    cur_section_char = nc
    ⟨ Put section name into section_text 131 ⟩
    if len(section_text)⟩3 ∧ compare_runes(section_text[len(section_text) - 3:], []rune("...")) ≡ 0 {
      cur_section = section_lookup(section_text[0:len(section_text) - 3], true)      /* 1 means is a prefix */
    } else {
      cur_section = section_lookup(section_text, false)
    }
    xref_switch = 0
    return section_name
  }
```

This code is used in section 128.

130. 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. (We set *section_text*[0] = '␣' to facilitate this, since the *section_lookup* routine uses *section_text*[1] as the first character of the name.)

131. ⟨ Put section name into *section_text* 131 ⟩ ≡

```
for{
  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 132 ⟩
  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 129.

132. ⟨ If end of name or erroneous nesting, break 132 ⟩ ≡

```
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 131.

133. This function skips over a restricted context at relatively high speed.

```
func skip_restricted(){
  id_first := loc
  false_alarm:
  for loc⟨len(buffer) ∧ buffer[loc] ≠ '@' {
    loc++
  }
  id = buffer[id_first:loc]
  loc++
  if loc ≥ len(buffer) {
    err_print("! Control text didn't end")
    loc = len(buffer)
  } else {
    if buffer[loc] ≡ '@' ∧ loc ≤ len(buffer) {
      loc++
      goto false_alarm
    }
    l := loc
    loc++
    if buffer[l] ≠ '>' {
      err_print("! Control codes are forbidden in control text")
    }
  }
}
```

134. At the present point in the program we have $buffer[loc-1] \equiv verbatim$; we set *id* to the string itself. We also set *loc* to the position just after the ending delimiter.

```
⟨ Scan a verbatim string 134 ⟩ ≡
{
  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 verbatim
}
```

This code is used in section 128.

135. Phase one processing. We now have accumulated enough subroutines to make it possible to carry out GOWEAVE's first pass over the source file. If everything works right, both phase one and phase two of GOWEAVE will assign the same numbers to sections, and these numbers will agree with what GOTANGLE does.

The global variable *next_control* often contains the most recent output of *get_next*; in interesting cases, this will be the control code that ended a section or part of a section.

```
⟨ Global variables 93 ⟩ +≡
  var next_control rune      /* control code waiting to be acting upon */
```

136. The overall processing strategy in phase one has the following straightforward outline.

```
func phase_one() { phase = 1
reset_input()
section_count = 0
skip_limbo()
change_exists = false
for ¬input_has_ended{⟨ Store cross-reference data for the current section 137 ⟩}
changed_section[section_count] = change_exists
    /* the index changes if anything does */
phase = 2      /* prepare for second phase */
⟨ Print error messages about unused or undefined section names 149 ⟩
}
```

137. ⟨ Store cross-reference data for the current section 137 ⟩ ≡

```
{
  section_count++
  changed_section[section_count] = changing
      /* it will become 1 if any line changes */
  if loc − 1⟨len(buffer) ∧ buffer[loc − 1] ≡ '*' ∧ show_progress() {
    fmt.Printf("*%d", section_count)
    os.Stdout.Sync()      /* print a progress report */
  }
  ⟨ Store cross-references in the TEX part of a section 140 ⟩
  ⟨ Store cross-references in the format definition part of a section 143 ⟩
  ⟨ Store cross-references in the Go part of a section 146 ⟩
  if changed_section[section_count] {
    change_exists = true
  }
}
```

This code is used in section 136.

138. The *Go_xref* subroutine stores references to identifiers in Go text material beginning with the current value of *next_control* and continuing until *next_control* is '`{`' or '`|`', or until the next "milestone" is passed (i.e., *next_control* $\geq$ *format_code*). If *next_control* $\geq$ *format_code* when *Go_xref* is called, nothing will happen; but if *next_control* $\equiv$ '`|`' upon entry, the procedure assumes that this is the '`|`' preceding Go text that is to be processed.

The parameter *spec_ctrl* is used to change this behavior. In most cases *Go_xref* is called with *spec_ctrl* $\equiv$ *ignore*, which triggers the default processing described above. If *spec_ctrl* $\equiv$ *section_name*, section names will be gobbled. This is used when Go text in the TEX part or inside comments is parsed: It allows for section names to appear in `|` ... `|`, but these strings will not be entered into the cross reference lists since they are not definitions of section names.

The program uses the fact that our internal code numbers satisfy the relations *xref_roman* $\equiv$ *identifier* + *roman* and *xref_wildcard* $\equiv$ *identifier* + *wildcard* and *xref_typewriter* $\equiv$ *identifier* + *typewriter*, as well as *normal* $\equiv 0$.

```
    /* makes cross-references for Go identifiers */
func Go_xref(spec_ctrl rune) { for next_control⟨format_code ∨ next_control ≡ spec_ctrl { if
    next_control ≥ identifier ∧ next_control ≤ xref_typewriter { if next_control⟩identifier{⟨ Replace "@@"
    by "@" 141 ⟩}
p := id_lookup(id, next_control − identifier)
    /* a referenced name */
new_xref(p)
}
if next_control ≡ section_name {
  section_xref_switch = cite_flag
  new_section_xref(cur_section)
}
next_control = get_next()
if next_control ≡ '|' ∨ next_control ≡ begin_comment ∨ next_control ≡ begin_short_comment {
  return
}
}
}
```

139. The *outer_xref* subroutine is like *Go_xref* except that it begins with *next_control* ≠ `'|'` and ends with *next_control* ≥ *format_code*. Thus, it handles Go text with embedded comments.

```
  /* extension of Go_xref */
func outer_xref(){
  for next_control⟨format_code {
    if next_control ≠ begin_comment ∧ next_control ≠ begin_short_comment {
      Go_xref(ignore)
    } else {
      is_long_comment := (next_control ≡ begin_comment)
      bal, res := copy_comment(is_long_comment, 1, nil)      /* brace level in comment */
      next_control = '|'
      for bal⟩0 {
        Go_xref(section_name)      /* do not reference section names in comments */
        if next_control ≡ '|' {
          bal, res = copy_comment(is_long_comment, bal, res)
        } else {
          bal = 0      /* an error message will occur in phase two */
        }
      }
    }
  }
}
```

140. In the TEX part of a section, cross-reference entries are made only for the identifiers in Go texts enclosed in `|` ... `|`, or for control texts enclosed in `@^` ... `@>` or `@.` ... `@>` or `@:` ... `@>`.

```
⟨ Store cross-references in the TEX part of a section 140 ⟩ ≡
  for{
    next_control = skip_TEX()
    switch next_control {
      case underline:
        xref_switch = def_flag
        continue
      case trace:
        tracing = buffer[loc − 1] − '0'
        continue
      case '|':
        Go_xref(section_name)
      case xref_roman, xref_wildcard, xref_typewriter, noop, section_name:
        loc −= 2
        next_control = get_next()      /* scan to @> */
        if next_control ≥ xref_roman ∧ next_control ≤ xref_typewriter {
          ⟨ Replace "@@" by "@" 141 ⟩
          new_xref(id_lookup(id, next_control − identifier))
        }
    }
    if next_control ≥ format_code {
      break
    }
  }
```

This code is used in section 137.

141. ⟨ Replace `"@@"` by `"@"` 141 ⟩ ≡

```
{
  i := 0
  j := 0
  for i⟨len(id) {
    if id[i] ≡ '@' {
      i++
    }
    id[j] = id[i]
    j++
    i++
  }
  for j⟨i {
    id[j] = '␣'      /* clean up in case of error message display */
    j++
  }
}
```

This code is used in sections 138 and 140.

142. During the definition and Go parts of a section, cross-references are made for all identifiers except reserved words. However, the right identifier in a format definition is not referenced, and the left identifier is referenced only if it has been explicitly underlined (preceded by `@!`). The TEX code in comments is, of course, ignored, except for Go portions enclosed in `|` ... `|`; the text of a section name is skipped entirely, even if it contains `|` ... `|` constructions.

The variables *lhs* and *rhs* point to the respective identifiers involved in a format definition.

⟨ Global variables 93 ⟩ +≡

```
var lhs int32
var rhs int32      /* pointers to byte_start for format identifiers */
var res_wd_end int32
```

143. When we get to the following code we have *next_control* ≥ *format_code*.

⟨ Store cross-references in the format definition part of a section 143 ⟩ ≡

```
for next_control ≤ format_code {
  ⟨ Process a format definition 144 ⟩
  outer_xref()
}
```

This code is used in section 137.

144. Error messages for improper format definitions will be issued in phase two. Our job in phase one is to define the *ilk* of a properly formatted identifier, and to remove cross-references to identifiers that we now discover should be unindexed.

```
⟨ Process a format definition 144 ⟩ ≡
  {
    next_control = get_next()
    if next_control ≡ identifier {
      lhs = id_lookup(id, normal)
      name_dir[lhs].ilk = normal
      if xref_switch ≠ 0 {
        new_xref(lhs)
      }
      next_control = get_next()
      if next_control ≡ identifier {
        rhs = id_lookup(id, normal)
        name_dir[lhs].ilk = name_dir[rhs].ilk
        if unindexed(lhs) {
            /* retain only underlined entries */
          var r int32 = 0
          for q := name_dir[lhs].xref; q ≥ 0; q = xmem[q].xlink {
            if xmem[q].num⟨def_flag {
              if r ≠ 0 {
                xmem[r].xlink = xmem[q].xlink
              } else {
                name_dir[lhs].xref = xmem[q].xlink
              }
            } else {
              r = q
            }
          }
        }
        next_control = get_next()
      }
    }
  }
```

This code is used in section 143.

145. A much simpler processing of format definitions occurs when the definition is found in limbo.

```
⟨ Process simple format in limbo 145 ⟩ ≡
  {
    if get_next() ≠ identifier {
      err_print("! Missing left identifier of @s")
    } else {
      lhs = id_lookup(id, normal)
      if get_next() ≠ identifier {
        err_print("! Missing right identifier of @s")
      } else {
        rhs = id_lookup(id, normal)
        name_dir[lhs].ilk = name_dir[rhs].ilk
      }
    }
  }
```

This code is used in section 118.

146. Finally, when the TEX and definition parts have been treated, we have *next_control* ≥ *begin_code*.

```
⟨ Store cross-references in the Go part of a section 146 ⟩ ≡
  if next_control ≤ section_name {      /* begin_code or section_name */
    if next_control ≡ begin_code {
      section_xref_switch = 0
    } else {
      section_xref_switch = def_flag
      if cur_section_char ≡ '(' ∧ cur_section ≠ −1 {
        set_file_flag(cur_section)
      }
    }
    for{
      if next_control ≡ section_name ∧ cur_section ≠ −1 {
        new_section_xref(cur_section)
      }
      next_control = get_next()
      outer_xref()
      if next_control⟩section_name {
        break
      }
    }
  }
```

This code is used in section 137.

147. After phase one has looked at everything, we want to check that each section name was both defined and used. The variable *cur_xref* will point to cross-references for the current section name of interest.

```
⟨ Global variables 93 ⟩ +≡
  var cur_xref int32      /* temporary cross-reference pointer */
  var an_output bool      /* did file_flag precede cur_xref? */
```

148. The following recursive procedure walks through the tree of section names and prints out anomalies.

```
    /* print anomalies in subtree p */
  func section_check(p int32){
    if p ≠ −1 {
      section_check(name_dir[p].llink)
      cur_xref = name_dir[p].xref
      if xmem[cur_xref].num ≡ file_flag {
        an_output = true
        cur_xref = xmem[cur_xref].xlink
      } else {
        an_output = false
      }
      if xmem[cur_xref].num⟨def_flag {
        warn_print("! Never defined: <%s>", sprint_section_name(p))
      }
      for cur_xref ≠ 0 ∧ xmem[cur_xref].num ≥ cite_flag {
        cur_xref = xmem[cur_xref].xlink
      }
      if cur_xref ≡ 0 ∧ ¬an_output {
        warn_print("! Never used: <%s>", sprint_section_name(p))
      }
      section_check(name_dir[p].rlink)
    }
  }
```

149. ⟨ Print error messages about unused or undefined section names 149 ⟩ ≡

```
section_check(name_root)
```

This code is used in section 136.

150. Low-level output routines. The TEX output is supposed to appear in lines at most *line_length* characters long, so we place it into an output buffer. During the output process, *out_line* will hold the current line number of the line about to be output.

```
⟨ Global variables 93 ⟩ +≡
  var out_buf [line_length + 1]rune      /* assembled characters */
  var out_ptr int32      /* just after last character in out_buf */
  var out_buf_end int32 = line_length      /* end of out_buf */
  var out_line int      /* number of next line to be output */
```

151. The *flush_buffer* routine empties the buffer up to a given breakpoint, and moves any remaining characters to the beginning of the next line. If the *per_cent* parameter is 1 a `'%'` is appended to the line that is being output; in this case the breakpoint b should be strictly less than *out_buf_end*. If the *per_cent* parameter is 0, trailing blanks are suppressed. The characters emptied from the buffer form a new line of output; if the *carryover* parameter is true, a `"%"` in that line will be carried over to the next line (so that TEX will ignore the completion of commented-out text).

```
    /* outputs from out_buf + 1 to b,where b ≤ out_ptr */
  func flush_buffer(b int32, per_cent bool, carryover bool){
    j := b      /* pointer into out_buf */
    if ¬per_cent {      /* remove trailing blanks */
      for j⟩0 ∧ out_buf[j] ≡ '␣' {
        j−−
      }
    }
    fmt.Fprint(active_file, string(out_buf[1:j + 1]))
    if per_cent {
      fmt.Fprint(active_file, "%")
    }
    fmt.Fprint(active_file, "\n")
    out_line++
    if carryover {
      for j⟩0 {
        jj := j
        j−−
        if out_buf[jj] ≡ '%' ∧ (j ≡ 0 ∨ out_buf[j] ≠ '\\') {
          out_buf[b] = '%'
          b−−
          break
        }
      }
    }
    if b⟨out_ptr {
      copy(out_buf[1:], out_buf[b + 1:])
    }
    out_ptr −= b
  }
```

152. When we are copying TEX source material, we retain line breaks that occur in the input, except that an empty line is not output when the TEX source line was nonempty. For example, a line of the TEX file that contains only an index cross-reference entry will not be copied. The *finish_line* routine is called just before *get_line* inputs a new line, and just after a line break token has been emitted during the output of translated Go text.

```
    /* do this at the end of a line */
func finish_line(){
  if out_ptr>0 {
    flush_buffer(out_ptr, false, false)
  } else {
    for _, v := range buffer {
      if ¬unicode.IsSpace(v) {
        return
      }
    }
    flush_buffer(0, false, false)
  }
}
```

153. In particular, the *finish_line* procedure is called near the very beginning of phase two. We initialize the output variables in a slightly tricky way so that the first line of the output file will be '`\input gowebmac`'.

```
⟨ Set initial values 99 ⟩ +≡
  out_ptr = 1
  out_line = 1
  active_file = tex_file
  out_buf[out_ptr] = 'c'
  fmt.Fprint(active_file, "\\input␣gowebma")
```

154. When we wish to append one character c to the output buffer, we write '$out(c)$'; this will cause the buffer to be emptied if it was already full. If we want to append more than one character at once, we say $out\_str(s)$, where s is a string containing the characters.

A line break will occur at a space or after a single-nonletter TEX control sequence.

```
func out(c rune){
  if out_ptr ≥ out_buf_end {
    break_out()
  }
  out_ptr++
  out_buf[out_ptr] = c
}
```

155.

```
    /* output characters from s to end of string */
func out_str(s string){
  for _, v := range s {
    out(v)
  }
}
```

156. The *break_out* routine is called just before the output buffer is about to overflow. To make this routine a little faster, we initialize position 0 of the output buffer to '\'; this character isn't really output.

```
⟨ Set initial values 99 ⟩ +≡
  out_buf[0] = '\\'
```

157. A long line is broken at a blank space or just before a backslash that isn't preceded by another backslash. In the latter case, a `'%'` is output at the break.

```
    /* finds a way to break the output line */
  func break_out(){
    k := out_ptr      /* pointer into out_buf */
    for{
      if k ≡ 0 {
        ⟨ Print warning message, break the line, return 158 ⟩
      }
      if out_buf[k] ≡ '␣' {
        flush_buffer(k, false, true)
        return
      }
      kk := k
      k--
      if out_buf[kk] ≡ '\\' ∧ out_buf[k] ≠ '\\' {      /* we've decreased k */
        flush_buffer(k, true, true)
        return
      }
    }
  }
```

158. We get to this section only in the unusual case that the entire output line consists of a string of backslashes followed by a string of nonblank non-backslashes. In such cases it is almost always safe to break the line by putting a `'%'` just before the last character.

```
⟨ Print warning message, break the line, return 158 ⟩ ≡
  {
    warn_print("!␣Line␣had␣to␣be␣broken␣(output␣l.␣%d):\n%s\n", out_line,
        string(out_buf[1:out_ptr]))
    flush_buffer(out_ptr − 1, true, true)
    return
  }
```

This code is used in section 157.

159. Here is a function that make a section number in decimal notation. The number to be converted by *section_str* is known to be less than *def_flag*, so it cannot have more than five decimal digits. If the section is changed, we output '`\*`' just after the number.

```
  func section_str(n int32) string{
    s := fmt.Sprintf("%d", n)
    if changed_section[n] ∧ flags['c'] {
      s += "\\*"
    }
    return s
  }
```

160. The *out_name* procedure is used to output an identifier or index entry, enclosing it in braces.

```
func out_name(p int32, quote_xalpha bool){
  out('{')
  for _, v := range name_dir[p].name {
    if v ≡ '_' ∧ quote_xalpha {
      out('\\')
    }
    out(v)
  }
  out('}')
}
```

161. Routines that copy TEX material. During phase two, we use subroutines *copy_limbo*, *copy_TEX*, and *copy_comment* in place of the analogous *skip_limbo*, *skip_TEX*, and *skip_comment* that were used in phase one. (Well, *copy_comment* was actually written in such a way that it functions as *skip_comment* in phase one.)

The *copy_limbo* routine, for example, takes TEX material that is not part of any section and transcribes it almost verbatim to the output file. The use of '`@`' signs is severely restricted in such material: '`@@`' pairs are replaced by singletons; '`@l`' and '`@q`' and '`@s`' are interpreted.

```
func copy_limbo(){
  for{
    if loc ≥ len(buffer) {
      finish_line()
      if ¬get_line() {
        return
      }
    }
    for ; loc⟨len(buffer) ∧ buffer[loc] ≠ '@'; loc++ {
      out(buffer[loc])
    }
    l := loc
    loc++
    if l⟨len(buffer) {
      c := '␣'
      if loc⟨len(buffer) {
        c = buffer[loc]
        loc++
      }
      if ccode[c] ≡ new_section {
        break
      }
      switch ccode[c] {
        case '@':
          out('@')
        case noop:
          skip_restricted()
        case format_code:
          if get_next() ≡ identifier {
            get_next()
          }
          if loc ≥ len(buffer) {
            get_line()      /* avoid blank lines in output */
          }
              /* the operands of @s are ignored on this pass */
        default:
          err_print("!␣Double␣@␣should␣be␣used␣in␣limbo")
          out('@')
      }
    }
  }
}
```

162. The *copy_TEX* routine processes the TEX code at the beginning of a section; for example, the words you are now reading were copied in this way. It returns the next control code or '|' found in the input. We don't copy spaces or tab marks into the beginning of a line. This makes the test for empty lines in *finish_line* work.

```
format copy_TeX TeX
func copy_TEX() rune{
  for{
    if loc ≥ len(buffer) {
      finish_line()
      if ¬get_line() {
        return new_section
      }
    }
    c := buffer[loc]
    loc++
    for c ≠ '|' ∧ c ≠ '@' {
      out(c)
      if out_ptr ≡ 1 ∧ unicode.IsSpace(c) {
        out_ptr--
      }
      if loc ≡ len(buffer) {
        break
      }
      c = buffer[loc]
      loc++
    }
    if c ≡ '|' {
      return '|'
    }
    if c ≡ '@' ∧ len(buffer) ≡ 1 {
      return new_section
    }
    if loc⟨len(buffer) {
      l := loc
      loc++
      return ccode[buffer[l]]
    }
  }
  return 0
}
```

163. The *copy_comment* function issues a warning if more braces are opened than closed, and in the case of a more serious error it supplies enough braces to keep TEX from complaining about unbalanced braces. Instead of copying the TEX material into the output buffer, this function copies it into the token memory (in phase two only).

```
    /* copies TEX code in comments */
  func copy_comment(
    is_long_comment bool,
    bal int,      /* brace balance */
    tok_mem []interface{}
  ) (int, []interface{}){
    for{
      if loc ≥ len(buffer) {
        if is_long_comment {
          if ¬get_line() {
            err_print("! Input ended in mid-comment")
            loc = 1
            goto done
          }
        } else {
          if bal⟩1 {
            err_print("! Missing } in comment")
          }
          goto done
        }
      }
      c := buffer[loc]
      loc++
      if c ≡ '|' {
        return bal, tok_mem
      }
      if is_long_comment {
        ⟨ Check for end of comment 164 ⟩
      }
      if phase ≡ 2 {
        if c⟩°177 {
          tok_mem = append(tok_mem, quoted_char)
        }
        tok_mem = append(tok_mem, c)
      }
      ⟨ Copy special things when c ≡ '@', '\\' 165 ⟩
      if c ≡ '{' {
        bal++
      } else if c ≡ '}' {
        if bal⟩1 {
          bal--
        } else {
          err_print("! Extra } in comment")
          if phase ≡ 2 {
            tok_mem = tok_mem[:len(tok_mem) − 1]
          }
        }
      }
```

```
    }
    done:
    ⟨ Clear bal and return 166 ⟩
  }
```

164. ⟨ Check for end of comment 164 ⟩ ≡

```
  if c ≡ '*' ∧ loc⟨len(buffer) ∧ buffer[loc] ≡ '/' {
    loc++
    if bal⟩1 {
      err_print("! Missing } in comment")
    }
    goto done
  }
```

This code is used in section 163.

165. ⟨ Copy special things when $c \equiv$ '@', '\\' 165 ⟩ ≡

```
  if c ≡ '@' {
    l := loc
    loc++
    if l⟨len(buffer) ∧ buffer[l] ≠ '@' {
      err_print("! Illegal use of @ in comment")
      loc −= 2
      if phase ≡ 2 {
        tok_mem[len(tok_mem) − 1] = ' '
      }
      goto done
    }
  } else if c ≡ '\\' ∧ loc⟨len(buffer) ∧ buffer[loc] ≠ '@' {
    if phase ≡ 2 {
      tok_mem = append(tok_mem, buffer[loc])
    }
    loc++
  }
```

This code is used in section 163.

166. We output enough right braces to keep TEX happy.

⟨ Clear bal and return 166 ⟩ ≡

```
  if phase ≡ 2 {
    for bal−−; bal ≥ 0; bal−− {
      tok_mem = append(tok_mem, '}')
    }
  }
  return 0, tok_mem
```

This code is used in section 163.

167. Parsing. The most intricate part of `GOWEAVE` is its mechanism for converting Go-like code into TEX code, and we might as well plunge into this aspect of the program now. Parsing in `GOWEAVE` is different from parsing in `CWEAVE`. I decided to make a full parsing of Go-grammar, because the old variant seems to be quite difficult for me to reuse for parsing of Go grammar.

At the lowest level, the input is represented as a sequence of entities that we shall call *scraps*, where each scrap of information consists of two parts, its *category* and its *translation*. The category is essentially a syntactic class, and the translation is a token list that represents TEX code. Rules of syntax and semantics tell us how to combine adjacent scraps into larger sequence, and if we are lucky an entire Go text that starts out as hundreds of small scraps will join together into one gigantic scrap whose translation is the desired TEX code. If we are unlucky, we will be left with several scraps that don't combine; their translations will simply be output, one by one.

The combination rules are given as productions that are applied recursively from left to right. Suppose that we are currently working on the sequence of scraps $s_1\, s_2 \ldots s_n$. We try first to find the longest production that applies to an initial substring $s_1\, s_2 \ldots$; but if no such productions exist, we try to find the longest production applicable to the next substring $s_2\, s_3 \ldots$; and if that fails, we try to match $s_3\, s_4 \ldots$, etc.

A production applies if the category codes have a given pattern. For example, one of the productions is

$$\mathit{UnaryExpr}\ \{\ \mathit{binary\_op}\ \}\ \mathit{UnaryExpr}\ \rightarrow\ \mathit{Expression}$$

and it means that three consecutive scraps whose respective categories are *UnaryExpr*, *binary_op* and *UnaryExpr* are converted to one scrap whose category is *Expression*. The translations of the original scraps are simply concatenated. The case of

$$\mathit{Expression}\ \mathit{comma}\ \mathit{Expression} \rightarrow \mathit{ExpressionList} \qquad E_1 C\ \mathit{opt}9\, E_2$$

is only slightly more complicated: Here the resulting *exp* translation consists not only of the three original translations, but also of the tokens *opt* and 9 between the translations of the *comma* and the following *exp*. In the TEX file, this will specify an optional line break after the comma, with penalty 90.

Translation rules such as '$E_1 C\ \mathit{opt}9\, E_2$' above use subscripts to distinguish between translations of scraps whose categories have the same initial letter; these subscripts are assigned from left to right.

168. Here is a list of the category codes that scraps can have. (The *cat_name* array contains a complete list.)

```
⟨ Constants 1 ⟩ +≡
  const(
    normal rune = iota      /* ordinary identifiers have normal ilk */
    roman rune = iota       /* normal index entries have roman ilk */
    wildcard rune = iota    /* user-formatted index entries have wildcard ilk */
    typewriter rune = iota  /* 'typewriter type' entries have typewriter ilk */
    custom rune = iota      /* identifiers with user-given control sequence */
  )
  const(
    zero rune = iota
    ArrayType rune = iota
    StructType rune = iota
    PointerType rune = iota
    InterfaceType rune = iota
    SliceType rune = iota
    MapType rune = iota
    ChannelType rune = iota
    FieldDecl rune = iota
    AnonymousField rune = iota
    Signature rune = iota
    Parameters rune = iota
    ParameterList rune = iota
    ParameterDecl rune = iota
    MethodSpec rune = iota
    Block rune = iota
    Statement rune = iota
    ConstDecl rune = iota
    TypeDecl rune = iota
    VarDecl rune = iota
    FunctionDecl rune = iota
    MethodDecl rune = iota
    ConstSpec rune = iota
    IdentifierList rune = iota
    ExpressionList rune = iota
    TypeSpec rune = iota
    VarSpec rune = iota
    ShortVarDecl rune = iota
    Receiver rune = iota
    Operand rune = iota
    QualifiedIdent rune = iota
    MethodExpr rune = iota
    CompositeLit rune = iota
    FunctionLit rune = iota
    FunctionType rune = iota
    LiteralType rune = iota
    LiteralValue rune = iota
    ElementList rune = iota
    Element rune = iota
    PrimaryExpr rune = iota
    Conversion rune = iota
```

```
BuiltinCall rune = iota
Selector rune = iota
Index rune = iota
Slice rune = iota
TypeAssertion rune = iota
Call rune = iota
Expression rune = iota
UnaryExpr rune = iota
ReceiverType rune = iota
LabeledStmt rune = iota
SimpleStmt rune = iota
GoStmt rune = iota
ReturnStmt rune = iota
BreakStmt rune = iota
ContinueStmt rune = iota
GotoStmt rune = iota
fallthrough_token rune = iota
IfStmt rune = iota
SelectStmt rune = iota
ForStmt rune = iota
DeferStmt rune = iota
SendStmt rune = iota
IncDecStmt rune = iota
Assignment rune = iota
ExprSwitchStmt rune = iota
ExprCaseClause rune = iota
TypeSwitchStmt rune = iota
TypeSwitchGuard rune = iota
TypeCaseClause rune = iota
TypeSwitchCase rune = iota
ForClause rune = iota
RangeClause rune = iota
CommClause rune = iota
CommCase rune = iota
RecvStmt rune = iota
BuiltinArgs rune = iota
PackageClause rune = iota
ImportDecl rune = iota
ImportSpec rune = iota
Type rune = iota
package_token rune = iota     /* denotes package */
import_token rune = iota     /* denotes import */
type_token rune = iota     /* type */
interface_token rune = iota     /* interface */
const_token rune = iota     /* const */
go_token rune = iota     /* go */
return_token rune = iota     /* return */
break_token rune = iota     /* break */
continue_token rune = iota     /* continue */
goto_token rune = iota     /* goto */
if_token rune = iota     /* if */
switch_token rune = iota     /* switch */
```

```
    select_token rune = iota     /* select */
    case_token rune = iota     /* case */
    default_token rune = iota     /* default */
    for_token rune = iota     /* for */
    else_token rune = iota     /* else */
    defer_token rune = iota     /* denotes defer and go statements */
    func_token rune = iota     /* denotes a function declarator */
    struct_token rune = iota     /* struct */
    var_token rune = iota     /* var */
    range_token rune = iota     /* range */
    map_token rune = iota     /* map */
    chan_token rune = iota     /* chan */
    dot rune = iota     /* . */
    eq rune = iota     /* denotes an assign operator '=' */
    binary_op rune = iota
    rel_op rune = iota
    add_op rune = iota
    mul_op rune = iota
    unary_op rune = iota
    asterisk rune = iota
    assign_op rune = iota
    lbrace rune = iota     /* denotes a left brace */
    rbrace rune = iota     /* denotes a right brace */
    comma rune = iota     /* denotes a comma */
    lpar rune = iota     /* denotes a left parenthesis */
    rpar rune = iota     /* denotes a right parenthesis */
    lbracket rune = iota     /* denotes a left bracket */
    rbracket rune = iota     /* denotes a right bracket */
    semi rune = iota     /* denotes a semicolon */
    colon rune = iota     /* denotes a colon */
    insert rune = iota     /* a scrap that gets combined with its neighbor */
    section_scrap rune = iota     /* section name */
    dead rune = iota     /* scrap that won't combine */
)
```

169. ⟨ Global variables 93 ⟩ +≡

```
var cat_name [256]string
```

170. ⟨ Set initial values 99 ⟩ +≡

```
for cat_index := 0; cat_index⟨255; cat_index++ {
    cat_name[cat_index] = "UNKNOWN-" + fmt.Sprintf("%v", cat_index)
}
cat_name[Type] = "Type"
cat_name[ArrayType] = "ArrayType"
cat_name[StructType] = "StructType"
cat_name[PointerType] = "PointerType"
cat_name[InterfaceType] = "InterfaceType"
cat_name[SliceType] = "SliceType"
cat_name[MapType] = "MapType"
cat_name[ChannelType] = "ChannelType"
cat_name[FieldDecl] = "FieldDecl"
cat_name[AnonymousField] = "AnonymousField"
cat_name[Signature] = "Signature"
cat_name[Parameters] = "Parameters"
cat_name[ParameterList] = "ParameterList"
cat_name[ParameterDecl] = "ParameterDecl"
cat_name[MethodSpec] = "MethodSpec"
cat_name[Block] = "Block"
cat_name[Statement] = "Statement"
cat_name[ConstDecl] = "ConstDecl"
cat_name[TypeDecl] = "TypeDecl"
cat_name[VarDecl] = "VarDecl"
cat_name[FunctionDecl] = "FunctionDecl"
cat_name[MethodDecl] = "MethodDecl"
cat_name[ConstSpec] = "ConstSpec"
cat_name[IdentifierList] = "IdentifierList"
cat_name[ExpressionList] = "ExpressionList"
cat_name[TypeSpec] = "TypeSpec"
cat_name[VarSpec] = "VarSpec"
cat_name[ShortVarDecl] = "ShortVarDecl"
cat_name[Receiver] = "Receiver"
cat_name[Operand] = "Operand"
cat_name[QualifiedIdent] = "QualifiedIdent"
cat_name[MethodExpr] = "MethodExpr"
cat_name[CompositeLit] = "CompositeLit"
cat_name[FunctionLit] = "FunctionLit"
cat_name[FunctionType] = "FunctionType"
cat_name[LiteralType] = "LiteralType"
cat_name[LiteralValue] = "LiteralValue"
cat_name[ElementList] = "ElementList"
cat_name[Element] = "Element"
cat_name[PrimaryExpr] = "PrimaryExpr"
cat_name[Conversion] = "Conversion"
cat_name[BuiltinCall] = "BuiltinCall"
cat_name[Selector] = "Selector"
cat_name[Index] = "Index"
cat_name[Slice] = "Slice"
cat_name[TypeAssertion] = "TypeAssertion"
cat_name[Call] = "Call"
cat_name[Expression] = "Expression"
```

cat_name[*UnaryExpr*] = "UnaryExpr"
cat_name[*ReceiverType*] = "ReceiverType"
cat_name[*LabeledStmt*] = "LabeledStmt"
cat_name[*SimpleStmt*] = "SimpleStmt"
cat_name[*GoStmt*] = "GoStmt"
cat_name[*ReturnStmt*] = "ReturnStmt"
cat_name[*BreakStmt*] = "BreakStmt"
cat_name[*ContinueStmt*] = "ContinueStmt"
cat_name[*GotoStmt*] = "GotoStmt"
cat_name[*fallthrough_token*] = "fallthrough_token"
cat_name[*IfStmt*] = "IfStmt"
cat_name[*SelectStmt*] = "SelectStmt"
cat_name[*ForStmt*] = "ForStmt"
cat_name[*DeferStmt*] = "DeferStmt"
cat_name[*SendStmt*] = "SendStmt"
cat_name[*IncDecStmt*] = "IncDecStmt"
cat_name[*Assignment*] = "Assignment"
cat_name[*ExprSwitchStmt*] = "ExprSwitchStmt"
cat_name[*ExprCaseClause*] = "ExprCaseClause"
cat_name[*TypeSwitchStmt*] = "TypeSwitchStmt"
cat_name[*TypeSwitchGuard*] = "TypeSwitchGuard"
cat_name[*TypeCaseClause*] = "TypeCaseClause"
cat_name[*TypeSwitchCase*] = "TypeSwitchCase"
cat_name[*ForClause*] = "ForClause"
cat_name[*RangeClause*] = "RangeClause"
cat_name[*CommClause*] = "CommClause"
cat_name[*CommCase*] = "CommCase"
cat_name[*RecvStmt*] = "RecvStmt"
cat_name[*BuiltinArgs*] = "BuiltinArgs"
cat_name[*PackageClause*] = "PackageClause"
cat_name[*ImportDecl*] = "ImportDecl"
cat_name[*ImportSpec*] = "ImportSpec"
cat_name[*package_token*] = "package"
cat_name[*import_token*] = "import"
cat_name[*type_token*] = "type"
cat_name[*interface_token*] = "interface"
cat_name[*const_token*] = "const"
cat_name[*go_token*] = "go"
cat_name[*return_token*] = "return"
cat_name[*break_token*] = "break"
cat_name[*continue_token*] = "continue"
cat_name[*goto_token*] = "goto"
cat_name[*if_token*] = "if"
cat_name[*switch_token*] = "switch"
cat_name[*select_token*] = "select"
cat_name[*case_token*] = "case"
cat_name[*default_token*] = "default"
cat_name[*for_token*] = "for"
cat_name[*else_token*] = "else"
cat_name[*defer_token*] = "defer"
cat_name[*func_token*] = "func"
cat_name[*struct_token*] = "struct"

$cat\_name[var\_token] =$ "var"
$cat\_name[range\_token] =$ "range"
$cat\_name[map\_token] =$ "map"
$cat\_name[chan\_token] =$ "chan"
$cat\_name[dot] =$ "'.'"
$cat\_name[eq] =$ "'='"
$cat\_name[col\_eq] =$ "':='"
$cat\_name[binary\_op] =$ "binary_op"
$cat\_name[rel\_op] =$ "rel_op"
$cat\_name[add\_op] =$ "add_op"
$cat\_name[mul\_op] =$ "mul_op"
$cat\_name[unary\_op] =$ "unary_op"
$cat\_name[asterisk] =$ "'*'"
$cat\_name[assign\_op] =$ "assign_op"
$cat\_name[lbrace] =$ "'{'"
$cat\_name[rbrace] =$ "'}'"
$cat\_name[comma] =$ "','"
$cat\_name[lpar] =$ "'('"
$cat\_name[rpar] =$ "')'"
$cat\_name[lbracket] =$ "'['"
$cat\_name[rbracket] =$ "']'"
$cat\_name[semi] =$ "';'"
$cat\_name[colon] =$ "':'"
$cat\_name[insert] =$ "insert"
$cat\_name[section\_scrap] =$ "section_scrap"
$cat\_name[dead] =$ "@d"
$cat\_name[dot\_dot\_dot] =$ "'...'"
$cat\_name[constant] =$ "constant"
$cat\_name[str] =$ "str"
$cat\_name[identifier] =$ "identifier"
$cat\_name[0] =$ "zero"
$cat\_name[direct] =$ "'<-'"
$cat\_name[plus\_plus] =$ "'++'"
$cat\_name[minus\_minus] =$ "'--'"
$cat\_name[verbatim] =$ "verbatim"

171. This code allows GOWEAVE to display its parsing steps.

```
    /* symbolic printout of a category */
func print_cat(c int32){
  fmt.Printf("%s", cat_name[c])
}
```

172. The token lists for translated TEX output contain some special control symbols as well as ordinary characters. These control symbols are interpreted by `GOWEAVE` before they are written to the output file.

break_space denotes an optional line break or an en space;

force denotes a line break;

big_force denotes a line break with additional vertical space;

opt denotes an optional line break (with the continuation line indented two ems with respect to the normal starting position)—this code is followed by an integer n, and the break will occur with penalty $10n$;

backup denotes a backspace of one em;

cancel obliterates any *break_space*, *opt*, *force*, or *big_force* tokens that immediately precede or follow it and also cancels any *backup* tokens that follow it;

indent causes future lines to be indented one more em;

outdent causes future lines to be indented one less em.

All of these tokens are removed from the TEX output that comes from Go text between `|...|` signs; *break_space* and *force* and *big_force* become single spaces in this mode. The translation of other Go texts results in TEX control sequences `\1`, `\2`, `\3`, `\4`, `\5`, `\6`, `\7` corresponding respectively to *indent*, *outdent*, *opt*, *backup*, *break_space*, *force*, *big_force*. However, a sequence of consecutive '␣', *break_space*, *force*, and/or *big_force* tokens is first replaced by a single token (the maximum of the given sequence).

The token *math_rel* will be translated into `\MRL{`, and it will get a matching `}` later. Other control sequences in the TEX output will be '`\\{...}`' surrounding identifiers, '`\&{...}`' surrounding reserved words, '`\.{...}`' surrounding strings, '`\C{...}` *force*' surrounding comments, and '`\X`n`:...\X`' surrounding section names, where n is the section number.

```
⟨ Constants 1 ⟩ +≡
  const(
    math_rel rune = °244
    big_cancel rune = °245 /* like cancel, also overrides spaces */
    cancel rune = °246 /* overrides backup, break_space, force, big_force */
    indent rune = °247 /* one more tab (\1) */
    outdent rune = °250 /* one less tab (\2) */
    opt rune = °251 /* optional break in mid-statement (\3) */
    backup rune = °252 /* stick out one unit to the left (\4) */
    break_space rune = °253 /* optional break between statements (\5) */
    force rune = °254 /* forced break between statements (\6) */
    big_force rune = °255 /* forced break with additional space (\7) */
    quoted_char rune = °256 /* introduces a character token in the range °200–°377 */
    end_translation rune = °257 /* special sentinel token at end of list */
    inserted rune = °260 /* sentinel to mark translations of inserts */
  )
```

173. Implementing the productions. Parsing of Go code in `GOWEAVE` is different from one in CWEAVE. A scrap sequence to be reduced is been looking at the current position in the *scrap_info* recursively, but a reducing has to be proceeded if and only if a full sequence is found. Each search of the scrap sequence may initiate other search of a nested scrap sequence and so on. After the scrap sequence is found, a reducing closure is provided, that may calls other nested closures.

174. More specifically, a *scrap* is a structure consisting of a category *cat* and a *trans*, which contains the translation. When Go text is to be processed with the grammar, we form an array *scrap_info* containing the initial scraps.

```
⟨ Typedef declarations 94 ⟩ +≡
  type scrap struct{
    cat int32
    mathness int32
    trans []interface{}
    ⟨ Rest of scrap struct 368 ⟩
  }
```

175.
```
⟨ Global variables 93 ⟩ +≡
  var scrap_info []scrap      /* memory array for scraps */
```

176. Token lists in *tok_mem* are composed of the following kinds of items for TEX output.

- Character codes and special codes like *force* and *math_rel* represent themselves;
- a type *id_token* represents `\\{identifier}`;
- a type *res_token* represents `\&{identifier}`;
- a type *section_token* represents section name;
- a type *list_token* represents list of tokens;
- a type *inner_list_token* represents list of token, to be translated without line-break controls.

```
⟨ Typedef declarations 94 ⟩ +≡
  type id_token int
  type res_token int
  type section_token int32
  type list_token []interface{}
  type inner_list_token []interface{}
```

177. Several helper functions are defined here so that the program for each production is fairly short.

178.
```
⟨ Typedef declarations 94 ⟩ +≡
  type reducing func()
```

179.
```
⟨ Global variables 93 ⟩ +≡
  var shift = 0
  var empty reducing = func(){}
```

180. The function *call* is a helper to call all functions in a slice *fs* one by one.

```
  func call(fs []reducing){
    for i := len(fs) - 1; i ≥ 0; i-- {
      fs[i]()
    }
  }
```

181. The function *one* checks of slice of scraps *ss* has the specified category *c*. It returns a resting slice of scraps, a closure should be called to make a reducing of a found category *c* and a flag that *c* has been found. It returns a []*scrap* slice of a rest of scraps, a *reducing* closure and a **bool** flag points out the *c* is found. Actually, it is a heart of the parsing proccess.

```
func one(ss []scrap, c rune) ([]scrap, reducing, bool){
  m := "found"
  if (tracing & 4) ≡ 4 {
    fmt.Printf("%*cLooking␣for␣%q...\n", shift, '␣', cat_name[c])
    shift += 5
    defer func(){
      shift −= 5;
      fmt.Printf("%*c%q␣is␣%s\n", shift, '␣', cat_name[c], m)
    }()
    f := cat_name[c]
    fmt.Printf("%*c", shift, '␣')
    ⟨ Print a snapshot of the scrap list if debugging 306 ⟩
  }
  if len(ss) ≡ 0 {
    return ss, empty, false
  }
  if ss[0].cat ≡ c {
    return ss[1:], empty, true
  }
  switch c {
    case ConstDecl:
      ⟨ Cases for ConstDecl 195 ⟩
    case TypeDecl:
      ⟨ Cases for TypeDecl 197 ⟩
    case VarDecl:
      ⟨ Cases for VarDecl 199 ⟩
    case FunctionDecl:
      ⟨ Cases for FunctionDecl 203 ⟩
    case MethodDecl:
      ⟨ Cases for MethodDecl 205 ⟩
    case Receiver:
      ⟨ Cases for Receiver 207 ⟩
    case ConstSpec:
      ⟨ Cases for ConstSpec 208 ⟩
    case TypeSpec:
      ⟨ Cases for TypeSpec 209 ⟩
    case VarSpec:
      ⟨ Cases for VarSpec 210 ⟩
    case ImportSpec:
      ⟨ Cases for ImportSpec 212 ⟩
    case FieldDecl:
      ⟨ Cases for FieldDecl 213 ⟩
    case AnonymousField:
      ⟨ Cases for AnonymousField 214 ⟩
    case Type:
      ⟨ Cases for Type 215 ⟩
    case ArrayType:
      ⟨ Cases for ArrayType 216 ⟩
```

case *StructType*:
⟨ Cases for *StructType* 217 ⟩
case *PointerType*:
⟨ Cases for *PointerType* 219 ⟩
case *Signature*:
⟨ Cases for *Signature* 220 ⟩
case *Parameters*:
⟨ Cases for *Parameters* 221 ⟩
case *ParameterList*:
⟨ Cases for *ParameterList* 222 ⟩
case *ParameterDecl*:
⟨ Cases for *ParameterDecl* 223 ⟩
case *InterfaceType*:
⟨ Cases for *InterfaceType* 224 ⟩
case *MethodSpec*:
⟨ Cases for *MethodSpec* 225 ⟩
case *SliceType*:
⟨ Cases for *SliceType* 226 ⟩
case *MapType*:
⟨ Cases for *MapType* 227 ⟩
case *ChannelType*:
⟨ Cases for *ChannelType* 228 ⟩
case *IdentifierList*:
⟨ Cases for *IdentifierList* 229 ⟩
case *ExpressionList*:
⟨ Cases for *ExpressionList* 230 ⟩
case *Expression*:
⟨ Cases for *Expression* 231 ⟩
case *UnaryExpr*:
⟨ Cases for *UnaryExpr* 232 ⟩
case *binary_op*:
⟨ Cases for *binary_op* 233 ⟩
case *PrimaryExpr*:
⟨ Cases for *PrimaryExpr* 234 ⟩
case *Operand*:
⟨ Cases for *Operand* 235 ⟩
case *CompositeLit*:
⟨ Cases for *CompositeLit* 236 ⟩
case *LiteralType*:
⟨ Cases for *LiteralType* 237 ⟩
case *LiteralValue*:
⟨ Cases for *LiteralValue* 238 ⟩
case *ElementList*:
⟨ Cases for *ElementList* 239 ⟩
case *Element*:
⟨ Cases for *Element* 240 ⟩
case *FunctionLit*:
⟨ Cases for *FunctionLit* 241 ⟩
case *FunctionType*:
⟨ Cases for *FunctionType* 242 ⟩
case *Block*:
⟨ Cases for *Block* 243 ⟩

case *Statement*:
⟨ Cases for *Statement* 245 ⟩
case *LabeledStmt*:
⟨ Cases for *LabeledStmt* 246 ⟩
case *SimpleStmt*:
⟨ Cases for *SimpleStmt* 248 ⟩
case *GoStmt*:
⟨ Cases for *GoStmt* 249 ⟩
case *ReturnStmt*:
⟨ Cases for *ReturnStmt* 251 ⟩
case *BreakStmt*:
⟨ Cases for *BreakStmt* 253 ⟩
case *ContinueStmt*:
⟨ Cases for *ContinueStmt* 255 ⟩
case *GotoStmt*:
⟨ Cases for *GotoStmt* 257 ⟩
case *IfStmt*:
⟨ Cases for *IfStmt* 259 ⟩
case *ExprSwitchStmt*:
⟨ Cases for *ExprSwitchStmt* 261 ⟩
case *ExprCaseClause*:
⟨ Cases for *ExprCaseClause* 262 ⟩
case *TypeSwitchStmt*:
⟨ Cases for *TypeSwitchStmt* 263 ⟩
case *TypeSwitchGuard*:
⟨ Cases for *TypeSwitchGuard* 264 ⟩
case *TypeCaseClause*:
⟨ Cases for *TypeCaseClause* 265 ⟩
case *TypeSwitchCase*:
⟨ Cases for *TypeSwitchCase* 266 ⟩
case *SelectStmt*:
⟨ Cases for *SelectStmt* 268 ⟩
case *CommClause*:
⟨ Cases for *CommClause* 269 ⟩
case *CommCase*:
⟨ Cases for *CommCase* 270 ⟩
case *RecvStmt*:
⟨ Cases for *RecvStmt* 271 ⟩
case *SendStmt*:
⟨ Cases for *SendStmt* 272 ⟩
case *ForStmt*:
⟨ Cases for *ForStmt* 275 ⟩
case *ForClause*:
⟨ Cases for *ForClause* 276 ⟩
case *RangeClause*:
⟨ Cases for *RangeClause* 277 ⟩
case *DeferStmt*:
⟨ Cases for *DeferStmt* 279 ⟩
case *IncDecStmt*:
⟨ Cases for *IncDecStmt* 281 ⟩
case *Assignment*:
⟨ Cases for *Assignment* 283 ⟩

```
    case assign_op:
      ⟨ Cases for assign_op 285 ⟩
    case ShortVarDecl:
      ⟨ Cases for ShortVarDecl 286 ⟩
    case QualifiedIdent:
      ⟨ Cases for QualifiedIdent 288 ⟩
    case MethodExpr:
      ⟨ Cases for MethodExpr 289 ⟩
    case ReceiverType:
      ⟨ Cases for ReceiverType 290 ⟩
    case Conversion:
      ⟨ Cases for Conversion 291 ⟩
    case BuiltinCall:
      ⟨ Cases for BuiltinCall 292 ⟩
    case BuiltinArgs:
      ⟨ Cases for BuiltinArgs 293 ⟩
    case Selector:
      ⟨ Cases for Selector 294 ⟩
    case Index:
      ⟨ Cases for Index 295 ⟩
    case Slice:
      ⟨ Cases for Slice 296 ⟩
    case TypeAssertion:
      ⟨ Cases for TypeAssertion 297 ⟩
    case Call:
      ⟨ Cases for Call 298 ⟩
    case unary_op:
      ⟨ Cases for unary_op 299 ⟩
  }
  m = "not found"
  return ss, empty, false
}
```

182. The function *sequence* checks if corresponding scraps from start of *s* have the specified sequence of categories *cats*. All of the catigories *cats* is mandatory. A resulting []*scraps* contains a rest of scraps, a []*reducing* slice contains a chain of reducing closures should be called one by one to make a reducing full sequence. A **bool** points out the sequence of *cats* is found.

```
func sequence(ss []scrap, cats ...rune) ([]scrap, reducing, bool){
  var fs []reducing
  s := ss
  for _, v := range cats {
    f := empty
    ok := false
    if s, f, ok = one(s, v); ¬ok {
      return ss, empty, false
    }
    fs = append(fs, f)
  }
  return s, func(){call(fs)}, true
}
```

183. The function *any* checks if first of corresponding scraps from start of s have the specified category of categories *cats*. A resulting []*scraps* contains a rest of scraps, a *reducing* is a reducing closure should be called one by one to make a reducing full sequence. A **bool** points out one a category from *cats* is found.

```
func any(s []scrap, cats ...rune) ([]scrap, reducing, bool){
  for _, v := range cats {
    if s, f, ok := one(s, v); ok {
      return s, f, ok
    }
  }
  return s, empty, false
}
```

184. The *pair* struct helps to point out an optionality of a category *cat* by a flag *mand*

⟨ Typedef declarations 94 ⟩ +≡

```
type pair struct{
  cat int32
  mand bool
}
```

185. The function *optional* checks if corresponding scraps from start of *ss* have the specified sequence of categories *cats*. *g* is a start index of future scraps. Some of the catigories *cats* can be optional. A resulting []*scraps* contains a rest of scraps, a []*reducing* slice contains a chain of reducing closures should be called one by one to make a reducing full sequence. An **int** slice is contains indexes of scraps in a scrap sequence after processing of the reducing closure. A **bool** points out sequences of *cats* is found.

```
func optional(ss []scrap, g int, cats ...pair) ([]scrap, reducing, []int, bool){
  var trans []int
  var funcs []reducing
  ok := false
  for len(ss)⟩0 {
    var t []int
    var fs []reducing
    s := ss
    exit := false
    for _, v := range cats {
      f := empty
      if s, f, ok = one(s, v.cat); ok {
        t = append(t, g)
        fs = append(fs, f)
        g++
      } else if v.mand {
        exit = true
        break
      }
    }
    if exit ∨ len(fs) ≡ 0 {
      break
    }
    funcs = append(funcs, fs ...)
    trans = append(trans, t ...)
    ss = s
  }
  ok = true
  if len(funcs) ≡ 0 {
    ok = false
  }
  return ss, func(){call(funcs)}, trans, ok
}
```

186. Let us consider the big switch for productions now, before looking at its context. We want to design the program so that this switch works, so we might as well not keep ourselves in suspense about exactly what code needs to be provided with a proper environment.

```
⟨ Match a production at pp, or increase pp if there is no match 186 ⟩ ≡
  {
    _, f, ok := func(ss []scrap) ([]scrap, reducing, bool){
      switch ss[0].cat {
        case package_token:
          ⟨ Cases for PackageClause 193 ⟩
        case import_token:
          ⟨ Cases for ImportDecl 201 ⟩
        case struct_token:
          ⟨ Cases for StructType 217 ⟩
        case interface_token:
          ⟨ Cases for InterfaceType 224 ⟩
        case func_token:
          ⟨ Cases for FunctionDecl 203 ⟩
          ⟨ Cases for MethodDecl 205 ⟩
          ⟨ Cases for FunctionType 242 ⟩
        default:
          ⟨ Cases for ImportSpec 212 ⟩
          ⟨ Cases for Statement 245 ⟩
          ⟨ Cases for ConstSpec 208 ⟩
          ⟨ Cases for VarSpec 210 ⟩
          ⟨ Cases for TypeSpec 209 ⟩
          ⟨ Cases for FieldDecl 213 ⟩
          ⟨ Cases for ExprCaseClause 262 ⟩
          ⟨ Cases for TypeCaseClause 265 ⟩
          ⟨ Cases for CommClause 269 ⟩
          ⟨ Cases for ElementList 239 ⟩
      }
      return ss, empty, false
    }(scrap_info[pp:])
    if ok {
      f()
    }
    pp++      /* if no match was found, we move to the right */
  }
```

This code is used in section 304.

187. In Go, new specifier names can be defined via **type** , and we want to make the parser recognize future occurrences of the identifier thus defined as specifiers. This is done by the procedure *make_reserved*, which changes the *ilk* of the relevant identifier.

We first need a procedure to recursively seek the first identifier in a token list, because the identifier might be enclosed in parentheses, as when one defines a function returning a pointer.

```
func find_first_ident(p []interface{}) []interface{}{
  for i, j := range p {
    switch r := j.(type) {
      case res_token:
        if name_dir[r].ilk ≠ Type {
          break
        }
        return p[i:i + 1]
      case id_token:
        return p[i:i + 1]
      case list_token:
        if q := find_first_ident(r); q ≠ nil {
          return q
        }
      case inner_list_token:
        if q := find_first_ident(r); q ≠ nil {
          return q
        }
      case rune:      /* char, section_token, fallthru: move on to next token */
        if r ≡ inserted {
          return nil      /* ignore inserts */
        }
    }
  }
  return nil
}
```

188. The scraps currently being parsed must be inspected for any occurrence of the identifier that we're making reserved; hence the **for** loop below.

```
    /* make the first identifier in scrap_info[p].trans like c */
func make_reserved(p []interface{}){
  tok_ptr := find_first_ident(p)
  if tok_ptr ≡ nil {
    return      /* this should not happen */
  }
  tok_ptr[0] = res_token(tok_ptr[0].(id_token))
}
```

189. In the following situations we want to mark the occurrence of an identifier as a definition: when *make_reserved* is just about to be used; after a specifier, as in *argv* []**string**; before a colon, as in *found*:; and in the declaration of a function, as in *main*(){...;}. This is accomplished by the invocation of *make_underlined* at appropriate times. Notice that, in the declaration of a function, we find out that the identifier is being defined only after it has been swallowed up by an *Expression*.

```
    /* underline the entry for the first identifier in scrap_info[p].trans */
func make_underlined(p []interface{}){
  tok_ptr := find_first_ident(p)
  if tok_ptr ≡ nil {
    return      /* this happens, for example, in case found: */
  }
  xref_switch = def_flag
  underline_xref(tok_ptr[0].(id_token))
}
```

190. We cannot use *new_xref* to underline a cross-reference at this point because this would just make a new cross-reference at the end of the list. We actually have to search through the list for the existing cross-reference.

```
func underline_xref(p id_token){
  q := name_dir[p].xref      /* pointer to cross-reference being examined */
  if flags['x'] ≡ false {
    return
  }
  m := section_count + xref_switch      /* cross-reference value to be installed */
  for q ≠ 0 {
    n := xmem[q].num      /* cross-reference value being examined */
    if n ≡ m {
      return
    } else if m ≡ n + def_flag {
      xmem[q].num = m
      return
    } else if n ≥ def_flag ∧ n⟨m {
      break
    }
    q = xmem[q].xlink
  }
  ⟨Insert new cross-reference at q, not at beginning of list 191⟩
}
```

191. We get to this section only when the identifier is one letter long, so it didn't get a non-underlined entry during phase one. But it may have got some explicitly underlined entries in later sections, so in order to preserve the numerical order of the entries in the index, we have to insert the new cross-reference not at the beginning of the list (namely, at *name_dir*[*p*].*xref*), but rather right before *q*.

```
⟨ Insert new cross-reference at q, not at beginning of list 191 ⟩ ≡
  append_xref(0)      /* this number doesn't matter */
  xmem[len(xmem) − 1].xlink = name_dir[p].xref
  r := int32(len(xmem) − 1)      /* temporary pointer for permuting cross-references */
  name_dir[p].xref = r
  for xmem[r].xlink ≠ q {
    xmem[r].num = xmem[xmem[r].xlink].num
    r = xmem[r].xlink
  }
  xmem[r].num = m      /* everything from q on is left undisturbed */
```

This code is used in section 190.

192. Now comes the code that tries to match each production starting with a particular type of scrap. Whenever a match is discovered, a closure is formed to reduce nested scrap sequence and matched scrap sequence. This closure is returned with rest of scraps and a flag of success.

193.

```
⟨ Cases for PackageClause 193 ⟩ ≡
  if s, f, ok := sequence(ss, package_token, identifier); ok {
    return s, func(){
      f()
      reduce(ss, 2, PackageClause, 0, break_space, 1, big_force)
    }, true
  }
```

This code is used in section 186.

194. Test for **package**

```
⟨ goweave/package.w 194 ⟩ ≡
  @
  @ c
  package main
```

195. ⟨ Cases for *ConstDecl* 195 ⟩ ≡

```
if s, f1, ok := one(ss, const_token); ok {
  if s, f2, ok := one(s, ConstSpec); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, ConstDecl, 0, break_space, 1, force)
    }, true
  } else if s, f2, ok := one(s, lpar); ok {
    tok_mem := append([]interface{}{}, 0, 1)
    s, f3, t, ok := optional(s, 2, pair{ConstSpec, true})
    if ok {
      tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    if s, f4, ok := one(s, rpar); ok {
      tok_mem = append(tok_mem, 2 + len(t), force)
      return s, func(){
        f4()
        f3()
        f2()
        f1()
        reduce(ss, 3 + len(t), ConstDecl, tok_mem ...)
      }, true
    }
  }
}
```

This code is used in section 181.

196. Tests for **const**

```
⟨ goweave/const.w 196 ⟩ ≡
  @
  @ c
  const Pi float64 = 3.14159265358979323846
  @
  @ c
  const zero = 0.0
  @
  @ c
  const(
    size int64 = 1024
    eof = -1
  )
  @
  @ c
  const a, b, c = 3, 4, "foo"
  @
  @ c
  const u, v float32 = 0, 3
  @
  @ c
  const(
    a t = 1 << iota
    b
    c
  )
```

197. ⟨ Cases for *TypeDecl* 197 ⟩ ≡

```
if s, f1, ok := one(ss, type_token); ok {
  if s, f2, ok := one(s, TypeSpec); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, TypeDecl, 0, break_space, 1, force)
    }, true
  } else if s, f2, ok := one(s, lpar); ok {
    tok_mem := append([]interface{}{}, 0, 1)
    s, f3, t, ok := optional(s, 2, pair{cat: TypeSpec, mand: true})
    if ok {
      tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    if s, f4, ok := one(s, rpar); ok {
      tok_mem = append(tok_mem, 2 + len(t), force)
      return s, func(){
        f4()
        f3()
        f2()
        f1()
        reduce(ss, 3 + len(t), TypeDecl, tok_mem ...)
      }, true
    }
  }
}
```

This code is used in section 181.

198. Tests for **type**

```
⟨ goweave/type.w 198 ⟩ ≡
  @
  @ c
  type IntArray [16]int
  @
  @ c
  type(
    Point struct{
      x, y float64
    }
    Polar Point
  )
  @
  @ c
  type TreeNode struct{
    left, right *TreeNode
    value *Comparable
  }
  @
  @ c
  type Block interface{
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
  }
```

199. ⟨ Cases for *VarDecl* 199 ⟩ ≡

```
if s, f1, ok := one(ss, var_token); ok {
  if s, f2, ok := one(s, VarSpec); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, VarDecl, 0, break_space, 1)
    }, true
  } else if s, f2, ok := one(s, lpar); ok {
    tok_mem := append([]interface{}{}, 0, 1)
    s, f3, t, ok := optional(s, 2, pair{cat: VarSpec, mand: true})
    if ok {
      tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    if s, f4, ok := one(s, rpar); ok {
      tok_mem = append(tok_mem, 2 + len(t), force)
      return s, func(){
        f4()
        f3()
        f2()
        f1()
        reduce(ss, 3 + len(t), VarDecl, tok_mem ...)
      }, true
    }
  }
}
```

This code is used in section 181.

200. Tests for **var**

```
⟨ goweave/var.w 200 ⟩ ≡
  @
  @ c
  var i int
  @
  @ c
  var U, V, W float64
  @
  @ c
  var k = 0
  @
  @ c
  var x, y float32 = −1, −2
  @
  @ c
  var(
    i int
    u, v, s = 2.0, 3.0, "bar"
  )
  @
  @ c
  var re, im = complexSqrt(−1)
  @
  @ c
  var _, found = entries[name]
```

201. ⟨ Cases for *ImportDecl* 201 ⟩ ≡

```
if s, f1, ok := one(ss, import_token); ok {
  if s, f2, ok := one(s, ImportSpec); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, ImportDecl, 0, break_space, 1, force)
    }, true
  } else if s, f2, ok := one(s, lpar); ok {
    tok_mem := append([]interface{}{}, 0, 1)
    s, f3, t, ok := optional(s, 2, pair{cat: ImportSpec, mand: true})
    if ok {
      tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    if s, f4, ok := one(s, rpar); ok {
      tok_mem = append(tok_mem, 2 + len(t), force)
      return s, func(){
        f4()
        f3()
        f2()
        f1()
        reduce(ss, 3 + len(t), ImportDecl, tok_mem ...)
      }, true
    }
  }
}
```

This code is used in section 186.

202. Tests for **import**

⟨ `goweave/import.w` 202 ⟩ ≡

```
@
@ c
import "im1"
@
@ c
import _ "im2";      /*im2*/
@
@ c
import . "im3"       //im3
@
@ c
import IM "im4"
@
@ c
import(
  "nim1/subnim1"
  . "nim2";      // nim2
  _ "nim3"       /*nim3*/
  NIM "nim4"
)
```

203. ⟨ Cases for *FunctionDecl* 203 ⟩ ≡

```
if s, f1, ok := sequence(ss, func_token, identifier, Signature); ok {
  if s, f2, ok := sequence(s, Block, semi); ok {
    return s, func(){
      f2()
      f1()
      make_underlined(ss[1].trans)
      reduce(ss, 5, FunctionDecl, 0, break_space, 1, 2, 3, 4, big_force)
    }, true
  } else if s, f2, ok := one(s, semi); ok {
    return s, func(){
      f2()
      f1()
      make_underlined(ss[1].trans)
      reduce(ss, 4, FunctionDecl, 0, break_space, 1, 2, 3, big_force)
    }, true
  }
}
```

This code is used in sections 181 and 186.

204. Tests for **func**

⟨ goweave/func.w 204 ⟩ ≡

```
@
@ c
func min(x int, y int) int{
  if x⟨y {
    return x
  }
  return y
}
@
@ c
func flushICache(begin, end uintptr)
```

205. ⟨ Cases for *MethodDecl* 205 ⟩ ≡

```
if s, f1, ok := sequence(ss, func_token, Receiver, identifier, Signature); ok {
  if s, f2, ok := one(s, Block); ok {
    return s, func(){
      f2()
      f1()
      make_underlined(ss[2].trans)
      reduce(ss, 5, MethodDecl, 0, break_space, 1, break_space, 2, 3, 4)
    }, true
  } else {
    return s, func(){
      f1()
      make_underlined(ss[2].trans)
      reduce(ss, 4, MethodDecl, 0, break_space, 1, break_space, 2, 3)
    }, true
  }
}
```

This code is used in sections 181 and 186.

206. Tests for *method*

```
⟨ goweave/method.w 206 ⟩ ≡
  @
  @ c
  func (p * Point) Length() float64{
    return math.Sqrt(p.x * p.x + p.y * p.y)
  }
  @
  @ c
  func (p * Point) Scale(factor float64){
    p.x *= factor
    p.y *= factor
  }
```

207. ⟨ Cases for *Receiver* 207 ⟩ ≡

```
  if s, f1, ok := one(ss, lpar); ok {
    if s, f2, ok := one(s, identifier); ok {
      if s, f3, ok := sequence(s, asterisk, identifier, rpar); ok {
        return s, func(){
          f3()
          f2()
          f1()
          reduce(ss, 5, Receiver, 0, 1, 2, 3, 4)
        }, true
      } else if s, f, ok := sequence(s, identifier, rpar); ok {
        return s, func(){
          f()
          reduce(ss, 4, Receiver, 0, 1, 2, 3)
        }, true
      } else if s, f, ok := one(s, rpar); ok {
        return s, func(){
          f()
          reduce(ss, 3, Receiver, 0, 1, 2)
        }, true
      }
    } else if s, f, ok := sequence(s, asterisk, identifier, rpar); ok {
      return s, func(){
        f()
        reduce(ss, 4, Receiver, 0, 1, 2, 3)
      }, true
    }
  }
```

This code is used in section 181.

208. ⟨ Cases for *ConstSpec* 208 ⟩ ≡

```
if s, f1, ok := one(ss, IdentifierList); ok {
  if s, f2, ok := sequence(s, Type, eq, ExpressionList); ok {
    if s, f3, ok := one(s, semi); ok {
      return s, func(){
        f3()
        f2()
        f1()
        reduce(ss, 5, ConstSpec, 0, break_space, 1, break_space, 2, break_space, 3, 4, force)
      }, true
    } else if _, _, ok := any(s, rpar, rbrace); ok {
      return s, func(){
        f2()
        f1()
        reduce(ss, 4, ConstSpec, 0, break_space, 1, break_space, 2, break_space, 3, force)
      }, true
    }
  } else if s, f2, ok := sequence(s, eq, ExpressionList); ok {
    if s, f3, ok := one(s, semi); ok {
      return s, func(){
        f3()
        f2()
        f1()
        reduce(ss, 4, ConstSpec, 0, break_space, 1, break_space, 2, 3, force)
      }, true
    } else if _, _, ok := any(s, rpar, rbrace); ok {
      return s, func(){
        f2()
        f1()
        reduce(ss, 3, ConstSpec, 0, break_space, 1, break_space, 2, force)
      }, true
    }
  } else if s, f2, ok := one(s, semi); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, ConstSpec, 0, 1, force)
    }, true
  }
} else if s, f, ok := one(ss, section_scrap); ok {
  return s, func(){
    f()
    reduce(ss, 1, ConstSpec, 0, force)
  }, true
}
```

This code is used in sections 181 and 186.

209. ⟨ Cases for *TypeSpec* 209 ⟩ ≡

```
if s, f1, ok := sequence(ss, identifier, Type); ok {
  if s, f2, ok := one(s, semi); ok {
    return s, func(){
      f2()
      f1()
      make_underlined(ss[0].trans)
      make_reserved(ss[0].trans)
      reduce(ss, 3, TypeSpec, 0, break_space, 1, 2, force)
    }, true
  } else if _, _, ok := any(s, rpar, rbrace); ok {
    return s, func(){
      f1()
      make_underlined(ss[0].trans)
      make_reserved(ss[0].trans)
      reduce(ss, 2, TypeSpec, 0, break_space, 1, force)
    }, true
  }
} else if s, f, ok := one(ss, section_scrap); ok {
  return s, func(){
    f()
    reduce(ss, 1, TypeSpec, 0, force)
  }, true
}
```

This code is used in sections 181 and 186.

210. ⟨ Cases for *VarSpec* 210 ⟩ ≡

```
if s, f1, ok := one(ss, IdentifierList); ok {
  if s, f2, ok := one(s, Type); ok {
    if s, f3, ok := sequence(s, eq, ExpressionList); ok {
      if s, f4, ok := one(s, semi); ok {
        return s, func(){
          f4()
          f3()
          f2()
          f1()
          reduce(ss, 5, VarSpec, 0, break_space, 1, 2, 3, 4, force)
        }, true
      } else if _, _, ok := any(s, rpar, rbrace); ok {
        return s, func(){
          f3()
          f2()
          f1()
          reduce(ss, 4, VarSpec, 0, break_space, 1, 2, 3, force)
        }, true
      }
    } else if s, f3, ok := one(s, semi); ok {
      return s, func(){
        f3()
        f2()
        f1()
        reduce(ss, 3, VarSpec, 0, break_space, 1, 2, force)
      }, true
    } else if _, _, ok := any(s, rpar, rbrace); ok {
      return s, func(){
        f2()
        f1()
        reduce(ss, 2, VarSpec, 0, break_space, 1, force)
      }, true
    }
  } else if s, f2, ok := sequence(s, eq, ExpressionList); ok {
    if s, f3, ok := one(s, semi); ok {
      return s, func(){
        f3()
        f2()
        f1()
        reduce(ss, 4, VarSpec, 0, 1, 2, 3, force)
      }, true
    } else if _, _, ok := any(s, rpar, rbrace); ok {
      return s, func(){
        f2()
        f1()
        reduce(ss, 3, VarSpec, 0, 1, 2, force)
      }, true
    }
  }
} else if s, f, ok := one(ss, section_scrap); ok {
  return s, func(){
```

```
    f()
    reduce(ss, 1, VarSpec, 0, force)
  }, true
}
```

This code is used in sections 181 and 186.

211. The function *underline_import* helps to extract a package name from **import** string and to make it underlined. It searches a last filename in quoted string, adds this filename like an identifier and makes an underlined xref.

```
func underline_import(s []interface{}){
  var id []rune
  var i int
  for i = 0; i⟨len(s); i++ {
    if c, ok := s[i].(rune); ok ∧ c ≡ '"' {
      i++
      break
    }
  }
  for ; i⟨len(s); i++ {
    if c, ok := s[i].(rune); ¬ok ∨ c ≡ '/' {
      id = nil
    } else if c ≡ '"' {
      break
    } else {
      id = append(id, c)
    }
  }
  if len(id) ≡ 0 {
    return
  }
  xref_switch = def_flag
  underline_xref(id_token(id_lookup(id, normal)))
}
```

212. ⟨ Cases for *ImportSpec* 212 ⟩ ≡

```
if s, f1, ok := sequence(ss, identifier, str); ok {
  if s, f2, ok := one(s, semi); ok {
    return s, func(){
      f2()
      f1()
      make_reserved(ss[0].trans)
      reduce(ss, 3, ImportSpec, 0, break_space, 1, 2, force)
    }, true
  } else if _, _, ok := any(s, rpar, rbrace); ok {
    return s, func(){
      f1()
      make_reserved(ss[0].trans)
      reduce(ss, 2, ImportSpec, 0, break_space, 1, force)
    }, true
  }
} else if s, f1, ok := sequence(ss, dot, str); ok {
  if s, f2, ok := one(s, semi); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 3, ImportSpec, 0, break_space, 1, 2, force)
    }, true
  } else if _, _, ok := any(s, rpar, rbrace); ok {
    return s, func(){
      f1()
      reduce(ss, 2, ImportSpec, 0, break_space, 1, force)
    }, true
  }
} else if s, f1, ok := one(ss, str); ok {
  if s, f2, ok := one(s, semi); ok {
    return s, func(){
      f2()
      f1()
      underline_import(ss[0].trans)
      reduce(ss, 2, ImportSpec, 0, 1, force)
    }, true
  } else if _, _, ok := any(s, rpar, rbrace); ok {
    return s, func(){
      f1()
      underline_import(ss[0].trans)
      reduce(ss, 1, ImportSpec, 0, force)
    }, true
  }
} else if s, f, ok := one(ss, section_scrap); ok {
  return s, func(){
    f()
    reduce(ss, 1, ImportSpec, 0, force)
  }, true
}
```

This code is used in sections 181 and 186.

213. ⟨ Cases for *FieldDecl* 213 ⟩ ≡

```
if s, f1, ok := sequence(ss, IdentifierList, Type); ok {
    tok_mem := append([]interface{}{}, 0, break_space, 1)
    c := 2
    s, f2, ok := one(s, str)
    if ok {
        tok_mem = append(tok_mem, break_space, 2)
        c++
    }
    if s, f3, ok := one(s, semi); ok {
        tok_mem = append(tok_mem, c, force)
        c++
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, c, FieldDecl, tok_mem ...)
        }, true
    } else if _, _, ok := any(s, rpar, rbrace); ok {
        tok_mem = append(tok_mem, force)
        return s, func(){
            f2()
            f1()
            reduce(ss, c, FieldDecl, tok_mem ...)
        }, true
    }
} else if s, f1, ok := one(ss, AnonymousField); ok {
    tok_mem := append([]interface{}{}, 0)
    c := 1
    s, f2, ok := one(s, str)
    if ok {
        tok_mem = append(tok_mem, break_space, 1)
        c++
    }
    tok_mem = append(tok_mem, force)
    if s, f3, ok := one(s, semi); ok {
        c++
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, c, FieldDecl, tok_mem ...)
        }, true
    } else if _, _, ok := any(s, rpar, rbrace); ok {
        return s, func(){
            f2()
            f1()
            reduce(ss, c, FieldDecl, tok_mem ...)
        }, true
    }
} else if s, f, ok := one(ss, section_scrap); ok {
    return s, func(){
```

```
      f()
      reduce(ss, 1, FieldDecl, 0, force)
    }, true
  }
```

This code is used in sections 181 and 186.

214. ⟨ Cases for *AnonymousField* 214 ⟩ ≡

```
if s, f, ok := sequence(ss, asterisk, Type); ok {
  return s, func(){
    f()
    reduce(ss, 2, AnonymousField, 0, 1)
  }, true
} else if s, f, ok := one(ss, Type); ok {
  return s, func(){
    f()
    reduce(ss, 1, AnonymousField, 0)
  }, true
}
```

This code is used in section 181.

215. ⟨ Cases for *Type* 215 ⟩ ≡

```
if s, f, ok := any(ss,
  ArrayType,
  StructType,
  PointerType, FunctionType,
  InterfaceType, SliceType,
  MapType,
  ChannelType,
  QualifiedIdent); ok {
  return s, func(){
    f()
    reduce(ss, 1, Type, 0)
  }, true
}
```

This code is used in section 181.

216. ⟨ Cases for *ArrayType* 216 ⟩ ≡

```
if s, f, ok := sequence(ss, lbracket, Expression, rbracket, Type); ok {
  return s, func(){
    f()
    reduce(ss, 4, ArrayType, 0, 1, 2, 3)
  }, true
}
```

This code is used in section 181.

217. ⟨ Cases for *StructType* 217 ⟩ ≡

```
if s, f1, ok := sequence(ss, struct_token, lbrace); ok {
    tok_mem := append([]interface{}{}, 0, 1)
    s, f2, t, ok := optional(s, 2, pair{cat: FieldDecl, mand: true})
    if ok {
        tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    if s, f3, ok := one(s, rbrace); ok {
        tok_mem = append(tok_mem, 2 + len(t))
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, 3 + len(t), StructType, tok_mem ...)
        }, true
    }
}
```

This code is used in sections 181 and 186.

218. Tests for **struct**

⟨ goweave/struct.w 218 ⟩ ≡

```
@
@ c
struct{}
@
@ c
struct{
    x, y int
    u float32
    _ float32
    A *[]int
    F func()
}
@
@ c
struct{
    T1
    *T2
    P.T3
    *P.T4
    x, y int
}
@
@ c
struct{
    microsec uint64 "field 1"
    serverIP6 uint64 "field 2"
    process string "field 3"
}
```

219. ⟨ Cases for *PointerType* 219 ⟩ ≡

```
if s, f, ok := sequence(ss, asterisk, Type); ok {
  return s, func(){
    f()
    reduce(ss, 2, PointerType, 0, 1)
  }, true
}
```

This code is used in section 181.

220. ⟨ Cases for *Signature* 220 ⟩ ≡

```
if s, f1, ok := one(ss, Parameters); ok {
  if s, f2, ok := any(s, Type, Parameters); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, Signature, 0, break_space, 1)
    }, true
  } else {
    return s, func(){
      f1()
      reduce(ss, 1, Signature, 0)
    }, true
  }
} else if s, f, ok := one(ss, section_scrap); ok {
  return s, func(){
    f()
    reduce(ss, 1, Signature, 0, force)
  }, true
}
```

This code is used in section 181.

221. ⟨ Cases for *Parameters* 221 ⟩ ≡

```
if s, f1, ok := one(ss, lpar); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1, pair{cat: ParameterList, mand: true}, pair{cat: comma, mand: false})
    if ok {
        tok_mem = append(tok_mem, t)
    }
    if s, f3, ok := one(s, rpar); ok {
        tok_mem = append(tok_mem, 1 + len(t))
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, 2 + len(t), Parameters, tok_mem ...)
        }, true
    }
} else if s, f, ok := one(ss, section_scrap); ok {
    return s, func(){
        f()
        reduce(ss, 1, Signature, 0, force)
    }, true
}
```

This code is used in section 181.

222. ⟨ Cases for *ParameterList* 222 ⟩ ≡

```
if s, f1, ok := one(ss, ParameterDecl); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1, pair{cat: comma, mand: true}, pair{cat: ParameterDecl, mand: true})
    if ok {
        tok_mem = append(tok_mem, t)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 1 + len(t), ParameterList, tok_mem ...)
    }, true
}
```

This code is used in section 181.

223. ⟨ Cases for *ParameterDecl* 223 ⟩ ≡

```
if s, f, ok := sequence(ss, IdentifierList, dot_dot_dot, Type); ok {
    return s, func(){
        f()
        reduce(ss, 3, ParameterDecl, 0, "\\,", 1, 2)
    }, true
} else if s, f, ok := sequence(ss, IdentifierList, Type); ok {
    return s, func(){
        f()
        reduce(ss, 2, ParameterDecl, 0, break_space, 1)
    }, true
} else if s, f, ok := sequence(ss, dot_dot_dot, Type); ok {
    return s, func(){
        f()
        reduce(ss, 2, ParameterDecl, 0, "\\,", 1)
    }, true
} else if s, f, ok := one(ss, Type); ok {
    return s, func(){
        f()
        reduce(ss, 1, ParameterDecl, 0)
    }, true
}
```

This code is used in section 181.

224. ⟨ Cases for *InterfaceType* 224 ⟩ ≡

```
if s, f1, ok := sequence(ss, interface_token, lbrace); ok {
    tok_mem := append([]interface{}{}, 0, 1)
    s, f2, t, ok := optional(s, 2, pair{cat: MethodSpec, mand: true})
    if ok {
        tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    if s, f3, ok := one(s, rbrace); ok {
        tok_mem = append(tok_mem, 2 + len(t))
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, 3 + len(t), InterfaceType, tok_mem...)
        }, true
    }
}
```

This code is used in sections 181 and 186.

225. ⟨ Cases for *MethodSpec* 225 ⟩ ≡

```
if s, f1, ok := sequence(ss, identifier, Signature); ok {
  if s, f2, ok := one(s, semi); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 3, MethodSpec, 0, 1, 2, force)
    }, true
  } else if _, _, ok := any(s, rpar, rbrace); ok {
    return s, func(){
      f1()
      reduce(ss, 2, MethodSpec, 0, 1, force)
    }, true
  }
} else if s, f1, ok := sequence(ss, Type); ok {
  if s, f2, ok := one(s, semi); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, MethodSpec, 0, 1, force)
    }, true
  } else if _, _, ok := any(s, rpar, rbrace); ok {
    return s, func(){
      f1()
      reduce(ss, 1, MethodSpec, 0, force)
    }, true
  }
} else if s, f, ok := one(ss, section_scrap); ok {
  return s, func(){
    f()
    reduce(ss, 1, MethodSpec, 0, force)
  }, true
}
```

This code is used in section 181.

226. ⟨ Cases for *SliceType* 226 ⟩ ≡

```
if s, f, ok := sequence(ss, lbracket, rbracket, Type); ok {
  return s, func(){
    f()
    reduce(ss, 3, SliceType, 0, 1, 2)
  }, true
}
```

This code is used in section 181.

227. ⟨ Cases for *MapType* 227 ⟩ ≡

```
if s, f, ok := sequence(ss, map_token, lbracket, Type, rbracket, Type); ok {
  return s, func(){
    f()
    reduce(ss, 5, MapType, 0, 1, 2, 3, 4)
  }, true
}
```

This code is used in section 181.

228. ⟨ Cases for *ChannelType* 228 ⟩ ≡

```
if s, f, ok := sequence(ss, direct, chan_token, Type); ok {
    return s, func(){
        f()
        reduce(ss, 3, ChannelType, 0, 1, break_space, 2)
    }, true
} else if s, f1, ok := one(ss, chan_token); ok {
    if s, f2, ok := sequence(s, direct, Type); ok {
        return s, func(){
            f2()
            f1()
            reduce(ss, 3, ChannelType, 0, 1, 2)
        }, true
    } else if s, f2, ok := one(s, Type); ok {
        return s, func(){
            f2()
            f1()
            reduce(ss, 2, ChannelType, 0, break_space, 1)
        }, true
    }
}
```

This code is used in section 181.

229. ⟨ Cases for *IdentifierList* 229 ⟩ ≡

```
if s, f1, ok := one(ss, identifier); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1, pair{cat: comma, mand: true}, pair{cat: identifier, mand: true})
    if ok {
        tok_mem = append(tok_mem, t)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 1 + len(t), IdentifierList, tok_mem ...)
    }, true
}
```

This code is used in section 181.

230. ⟨ Cases for *ExpressionList* 230 ⟩ ≡

```
if s, f1, ok := one(ss, Expression); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1, pair{cat: comma, mand: true}, pair{cat: Expression, mand: true})
    if ok {
        tok_mem = append(tok_mem, t)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 1 + len(t), ExpressionList, tok_mem ...)
    }, true
}
```

This code is used in section 181.

231. ⟨ Cases for *Expression* 231 ⟩ ≡

```
if s, f1, ok := one(ss, UnaryExpr); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1, pair{binary_op, true}, pair{UnaryExpr, true});
    if ok {
        tok_mem = append(tok_mem, t)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 1 + len(t), Expression, tok_mem...)
    }, true
}
```

This code is used in section 181.

232. ⟨ Cases for *UnaryExpr* 232 ⟩ ≡

```
if s, f, ok := one(ss, PrimaryExpr); ok {
    return s, func(){
        f()
        reduce(ss, 1, UnaryExpr, 0)
    }, true
} else if s, f, ok := sequence(ss, unary_op, UnaryExpr); ok {
    return s, func(){
        f()
        reduce(ss, 2, UnaryExpr, 0, 1)
    }, true
}
```

This code is used in section 181.

233. ⟨ Cases for *binary_op* 233 ⟩ ≡

```
if s, f, ok := any(ss, rel_op, add_op, mul_op, asterisk); ok {
    return s, func(){
        f()
        reduce(ss, 1, binary_op, 0)
    }, true
}
```

This code is used in section 181.

234. ⟨ Cases for *PrimaryExpr* 234 ⟩ ≡

```
if s, f1, ok := any(ss, BuiltinCall, Conversion, Operand); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1,
        pair{cat: Selector, mand: false},
        pair{cat: Index, mand: false},
        pair{cat: Slice, mand: false},
        pair{cat: TypeAssertion, mand: false},
        pair{cat: Call, mand: false});
    if ok {
        tok_mem = append(tok_mem, t)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 1 + len(t), PrimaryExpr, tok_mem ...)
    }, true
}
```

This code is used in section 181.

235. ⟨ Cases for *Operand* 235 ⟩ ≡

```
if s, f, ok := any(ss, CompositeLit, FunctionLit, MethodExpr, str, constant, QualifiedIdent); ok {
    return s, func(){
        f()
        reduce(ss, 1, Operand, 0)
    }, true
} else if s, f, ok := sequence(ss, lpar, Expression, rpar); ok {
    return s, func(){
        f()
        reduce(ss, 3, Operand, 0, 1, 2)
    }, true
}
```

This code is used in section 181.

236. ⟨ Cases for *CompositeLit* 236 ⟩ ≡

```
if s, f, ok := sequence(ss, LiteralType, LiteralValue); ok {
    return s, func(){
        f()
        reduce(ss, 2, CompositeLit, 0, 1)
    }, true
}
```

This code is used in section 181.

237. ⟨ Cases for *LiteralType* 237 ⟩ ≡

```
if s, f, ok := one(ss, Type); ok {
    return s, func(){
        f()
        reduce(ss, 1, LiteralType, 0)
    }, true
} else if s, f, ok := sequence(ss, lbracket, dot_dot_dot, rbracket, Type); ok {
    return s, func(){
        f()
        reduce(ss, 4, LiteralType, 0, 1, 2, 3)
    }, true
}
```

This code is used in section 181.

238. ⟨ Cases for *LiteralValue* 238 ⟩ ≡

```
if s, f1, ok := one(ss, lbrace); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1, pair{cat: ElementList, mand: true}, pair{cat: comma, mand: false})
    if ok {
        tok_mem = append(tok_mem, t)
    }
    if s, f3, ok := one(s, rbrace); ok {
        tok_mem = append(tok_mem, 1 + len(t))
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, 2 + len(t), LiteralValue, tok_mem...)
        }, true
    }
}
```

This code is used in section 181.

239. ⟨ Cases for *ElementList* 239 ⟩ ≡

```
if s, f1, ok := one(ss, Element); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1, pair{cat: comma, mand: true}, pair{cat: Element, mand: true})
    if ok {
        tok_mem = append(tok_mem, t)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 1 + len(t), ElementList, tok_mem...)
    }, true
}
```

This code is used in sections 181 and 186.

240. ⟨ Cases for *Element* 240 ⟩ ≡

```
if s, f1, ok := any(ss, identifier, Expression); ok {
  if s, f2, ok := one(s, colon); ok {
    if s, f3, ok := any(s, Expression, LiteralValue); ok {
      return s, func(){
        f3()
        f2()
        f1()
        reduce(ss, 3, Element, 0, 1, break_space, 2)
      }, true
    }
  }
}
if s, f, ok := any(ss, Expression, LiteralValue); ok {
  return s, func(){
    f()
    reduce(ss, 1, Element, 0)
  }, true
}
if s, f, ok := one(ss, section_scrap); ok {
  return s, func(){
    f()
    reduce(ss, 1, Element, 0)
  }, true
}
```

This code is used in section 181.

241. ⟨ Cases for *FunctionLit* 241 ⟩ ≡

```
if s, f, ok := sequence(ss, FunctionType, Block); ok {
  return s, func(){
    f()
    reduce(ss, 2, FunctionLit, 0, 1)
  }, true
}
```

This code is used in section 181.

242. ⟨ Cases for *FunctionType* 242 ⟩ ≡

```
if s, f, ok := sequence(ss, func_token, Signature); ok {
  return s, func(){
    f()
    reduce(ss, 2, FunctionType, 0, 1)
  }, true
}
```

This code is used in sections 181 and 186.

243. ⟨ Cases for *Block* 243 ⟩ ≡

```
if s, f1, ok := one(ss, lbrace); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1, pair{cat: Statement, mand: true})
    if ok {
        tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    if s, f3, ok := one(s, rbrace); ok {
        tok_mem = append(tok_mem, 1 + len(t))
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, 2 + len(t), Block, tok_mem ...)
        }, true
    }
}
```

This code is used in section 181.

244. Tests for *block*

⟨ goweave/block.w 244 ⟩ ≡

```
@
@ c
{
    a := b
}
```

245. ⟨ Cases for *Statement* 245 ⟩ ≡

```
if s, f, ok := any(ss,
    ImportDecl,
    ConstDecl,
    VarDecl,
    TypeDecl,
    LabeledStmt); ok {
    return s, func(){
        f()
        reduce(ss, 1, Statement, 0)
    }, true
} else if s, f1, ok := any(ss,
    GoStmt,
    ReturnStmt,
    BreakStmt,
    ContinueStmt,
    GotoStmt,
    fallthrough_token,
    Block,
    IfStmt,
    ExprSwitchStmt,
    TypeSwitchStmt,
    SelectStmt,
    ForStmt,
    DeferStmt,
    SimpleStmt); ok {
    if s, f2, ok := one(s, semi); ok {
        return s, func(){
            f2()
            f1()
            reduce(ss, 2, Statement, 0, 1, force)
        }, true
    } else if _, _, ok := any(s, rpar, rbrace); ok {
        return s, func(){
            f1()
            reduce(ss, 1, Statement, 0, force)
        }, true
    }
} else if s, f, ok := one(ss, semi); ok {
    return s, func(){
        f()
        reduce(ss, 1, Statement, 0, force)
    }, true
} else if s, f, ok := one(ss, section_scrap); ok {
    return s, func(){
        f()
        reduce(ss, 1, Statement, 0, force)
    }, true
}
```

This code is used in sections 181 and 186.

246. ⟨ Cases for *LabeledStmt* 246 ⟩ ≡

```
if s, f, ok := sequence(ss, identifier, colon, Statement); ok {
  return s, func(){
    f()
    reduce(ss, 3, LabeledStmt, 0, 1, force, 2)
  }, true
}
```

This code is used in section 181.

247. Tests for *label*

⟨ goweave/label.w 247 ⟩ ≡

```
@
@ c
Error:
log.Panic("error encountered")
```

248. ⟨ Cases for *SimpleStmt* 248 ⟩ ≡

```
if s, f, ok := any(ss, SendStmt, IncDecStmt, Assignment, ShortVarDecl, Expression); ok {
  return s, func(){
    f()
    reduce(ss, 1, SimpleStmt, 0)
  }, true
}
```

This code is used in section 181.

249. ⟨ Cases for *GoStmt* 249 ⟩ ≡

```
if s, f, ok := sequence(ss, go_token, Expression); ok {
  return s, func(){
    f()
    reduce(ss, 2, GoStmt, 0, break_space, 1)
  }, true
}
```

This code is used in section 181.

250. Tests for **go**

⟨ goweave/go.w 250 ⟩ ≡

```
@
@ c
go Server()
@
@ c
go func(ch chan ← bool){
  for{
    sleep(10);
    ch ← true;
  }
}(c)
```

251. ⟨ Cases for *ReturnStmt* 251 ⟩ ≡

```
if s, f, ok := sequence(ss, return_token, ExpressionList); ok {
    return s, func(){
        f()
        reduce(ss, 2, ReturnStmt, 0, break_space, 1)
    }, true
} else if s, f, ok := one(ss, return_token); ok {
    return s, func(){
        f()
        reduce(ss, 1, ReturnStmt, 0)
    }, true
}
```

This code is used in section 181.

252. Tests for **return**

⟨ goweave/return.w 252 ⟩ ≡

```
@
@ c
return
@
@ c
return -7.0, -4.0
@
@ c
return complexF1()
```

253. ⟨ Cases for *BreakStmt* 253 ⟩ ≡

```
if s, f1, ok := one(ss, break_token); ok {
    if s, f2, ok := one(s, identifier); ok {
        return s, func(){
            f2()
            f1()
            reduce(ss, 2, BreakStmt, 0, break_space, 1)
        }, true
    } else {
        return s, func(){
            f1()
            reduce(ss, 1, BreakStmt, 0)
        }, true
    }
}
```

This code is used in section 181.

254. Tests for **break**

```
⟨ goweave/break.w 254 ⟩ ≡
  @
  @ c
  for i⟨n {
    switch i {
      case 5:
        break
    }
  }
  @
  @ c
  L:
  for i⟨n {
    switch i {
      case 5:
        break L
    }
  }
```

255. ⟨ Cases for *ContinueStmt* 255 ⟩ ≡

```
  if s, f, ok := sequence(ss, continue_token, identifier); ok {
    return s, func(){
      f()
      reduce(ss, 2, ContinueStmt, 0, break_space, 1)
    }, true
  } else if s, f, ok := one(ss, continue_token); ok {
    return s, func(){
      f()
      reduce(ss, 1, ContinueStmt, 0)
    }, true
  }
```

This code is used in section 181.

256. Tests for **continue**

```
⟨ goweave/continue.w 256 ⟩ ≡
  @
  @ c
  for i⟨n {
    switch i {
      case 5:
        continue
    }
  }
  @
  @ c
  L:
  for i⟨n {
    switch i {
      case 5:
        continue L
    }
  }
```

257. ⟨ Cases for *GotoStmt* 257 ⟩ ≡

```
  if s, f, ok := sequence(ss, goto_token, identifier); ok {
    return s, func(){
      f()
      reduce(ss, 2, GotoStmt, 0, break_space, 1)
    }, true
  }
```

This code is used in section 181.

258. Tests for **goto**

```
⟨ goweave/goto.w 258 ⟩ ≡
  @
  @ c
  goto Label
```

259. ⟨ Cases for *IfStmt* 259 ⟩ ≡

```
if s, f1, ok := one(ss, if_token); ok {
    tok_mem := append([]interface{}{}, 0)
    c := 1
    f2, f3, f4 := empty, empty, empty
    if s, f2, ok = sequence(s, SimpleStmt, semi, Expression, Block); ok {
        tok_mem = append(tok_mem, break_space, c)
        if len(scrap_info[c + 1].trans) ≠ 0 {
            tok_mem = append(tok_mem, c + 1)
        } else {
            tok_mem = append(tok_mem, ';')
        }
        tok_mem = append(tok_mem, break_space, c + 2, break_space, c + 3)
        c += 4
    } else if s, f2, ok = sequence(s, SimpleStmt, semi, QualifiedIdent, Block); ok {
        tok_mem = append(tok_mem, break_space, c)
        if len(scrap_info[c + 1].trans) ≠ 0 {
            tok_mem = append(tok_mem, c + 1)
        } else {
            tok_mem = append(tok_mem, ';')
        }
        tok_mem = append(tok_mem, break_space, c + 2, break_space, c + 3)
        c += 4
    } else if s, f2, ok = sequence(s, Expression, Block); ok {
        tok_mem = append(tok_mem, break_space, c, break_space, c + 1)
        c += 2
    } else if s, f2, ok = sequence(s, QualifiedIdent, Block); ok {
        tok_mem = append(tok_mem, break_space, c, break_space, c + 1)
        c += 2
    } else {
        break
    }
    if s, f3, ok = one(s, else_token); ok {
        if s, f4, ok = any(s, IfStmt, Block); ok {
            tok_mem = append(tok_mem, break_space, c, break_space, c + 1)
            c += 2
        } else {
            break
        }
    }
    return s, func(){
        f4()
        f3()
        f2()
        f1()
        reduce(ss, c, IfStmt, tok_mem ...)
    }, true
}
```

This code is used in section 181.

260. Tests for **if**

```
⟨ goweave/if.w 260 ⟩ ≡
  @
  @ c
  if x⟩max {
    x = max
  }
  @
  @ c
  if x := f(); x⟨y {
    return x
  } else if x⟩z {
    return z
  } else {
    return y
  }
  @
  @ c
  if err := input_ln(change_file); err ≠ nil {
    return
  }
  @
  @ c
  if test ≠ 1 {@ ⟨ Section@ ⟩ }
```

261. ⟨ Cases for *ExprSwitchStmt* 261 ⟩ ≡

```
if s, f1, ok := one(ss, switch_token); ok {
    tok_mem := append([]interface{}{}, 0)
    c := 1
    f2, f3, f4 := empty, empty, empty
    if s, f2, ok = sequence(s, SimpleStmt, semi); ok {
        tok_mem = append(tok_mem, break_space, c, c + 1)
        if len(scrap_info[c + 1].trans) ≠ 0 {
            tok_mem = append(tok_mem, break_space, c + 1)
        } else {
            tok_mem = append(tok_mem, ';')
        }
        c += 2
    }
    if s, f3, ok = one(s, Expression); ok {
        tok_mem = append(tok_mem, break_space, c, break_space)
        c++
    }
    if s, f4, ok = one(s, lbrace); ok {
        tok_mem = append(tok_mem, c)
        c++
        s, f5, t, ok := optional(s, c, pair{cat: ExprCaseClause, mand: false})
        if ok {
            tok_mem = append(tok_mem, force, indent, t, outdent)
            c += len(t)
        }
        if s, f6, ok := one(s, rbrace); ok {
            tok_mem = append(tok_mem, c)
            c++
            return s, func(){
                f6()
                f5()
                f4()
                f3()
                f2()
                f1()
                reduce(ss, c, ExprSwitchStmt, tok_mem ...)
            }, true
        }
    }
}
```

This code is used in section 181.

262. ⟨ Cases for *ExprCaseClause* 262 ⟩ ≡

```
if s, f1, ok := sequence(ss, case_token, ExpressionList, colon); ok {
    tok_mem := append([]interface{}{}, 0, break_space, 1, 2)
    s, f2, t, ok := optional(s, 3, pair{cat: Statement, mand: true})
    if ok {
        tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 3 + len(t), ExprCaseClause, tok_mem ...)
    }, true
} else if s, f1, ok := sequence(ss, default_token, colon); ok {
    tok_mem := append([]interface{}{}, 0, 1, force)
    s, f2, t, ok := optional(s, 2, pair{cat: Statement, mand: true})
    if ok {
        tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 2 + len(t), ExprCaseClause, tok_mem ...)
    }, true
} else if s, f, ok := one(ss, section_scrap); ok {
    return s, func(){
        f()
        reduce(ss, 1, ExprCaseClause, 0, force)
    }, true
}
```

This code is used in sections 181 and 186.

263. ⟨ Cases for *TypeSwitchStmt* 263 ⟩ ≡

```
if s, f1, ok := one(ss, switch_token); ok {
    tok_mem := append([]interface{}{}, 0)
    c := 1
    f2 := empty
    if s, f2, ok = sequence(s, SimpleStmt, semi); ok {
        tok_mem = append(tok_mem, break_space, c, c + 1)
        if len(scrap_info[c + 1].trans) ≠ 0 {
            tok_mem = append(tok_mem, break_space, c + 1)
        } else {
            tok_mem = append(tok_mem, ';')
        }
        c += 2
    }
    if s, f3, ok := sequence(s, TypeSwitchGuard, lbrace); ok {
        tok_mem = append(tok_mem, break_space, c, break_space, c + 1)
        c += 2
        s, f4, t, ok := optional(s, c, pair{cat: TypeCaseClause, mand: true})
        if ok {
            tok_mem = append(tok_mem, force, indent, t, outdent)
            c += len(t)
        }
        if s, f5, ok := one(s, rbrace); ok {
            tok_mem = append(tok_mem, c)
            c++
            return s, func(){
                f5()
                f4()
                f3()
                f2()
                f1()
                reduce(ss, c, TypeSwitchStmt, tok_mem ...)
            }, true
        }
    }
}
```

This code is used in section 181.

264. ⟨ Cases for *TypeSwitchGuard* 264 ⟩ ≡

```
if s, f, ok := sequence(ss, identifier, col_eq, PrimaryExpr, dot, lpar, type_token, rpar); ok {
    return s, func(){
        f()
        reduce(ss, 7, TypeSwitchGuard, 0, 1, 2, 3, 4, 5, 6)
    }, true
} else if s, f, ok := sequence(ss, PrimaryExpr, dot, lpar, type_token, rpar); ok {
    return s, func(){
        f()
        reduce(ss, 5, TypeSwitchGuard, 0, 1, 2, 3, 4)
    }, true
}
```

This code is used in section 181.

265. ⟨ Cases for *TypeCaseClause* 265 ⟩ ≡

```
if s, f1, ok := sequence(ss, TypeSwitchCase, colon); ok {
    tok_mem := append([]interface{}{}, 0, 1, force)
    s, f2, t, ok := optional(s, 2, pair{cat: Statement, mand: true})
    if ok {
        tok_mem = append(tok_mem, indent, t, outdent)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 2 + len(t), TypeCaseClause, tok_mem ...)
    }, true
} else if s, f, ok := one(ss, section_scrap); ok {
    return s, func(){
        f()
        reduce(ss, 1, TypeCaseClause, 0, force)
    }, true
}
```

This code is used in sections 181 and 186.

266. ⟨ Cases for *TypeSwitchCase* 266 ⟩ ≡

```
if s, f1, ok := sequence(ss, case_token); ok {
    tok_mem := append([]interface{}{}, 0)
    if s, f2, ok := any(s, Type, constant); ok {
        tok_mem = append(tok_mem, break_space, 1)
        s, f3, t, ok := optional(s, 2, pair{cat: comma, mand: true}, pair{cat: Type, mand: true})
        if ok {
            tok_mem = append(tok_mem, t)
        }
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, 2 + len(t), TypeSwitchCase, tok_mem ...)
        }, true
    }
} else if s, f, ok := one(ss, default_token); ok {
    return s, func(){
        f()
        reduce(ss, 1, TypeSwitchCase, 0)
    }, true
} else if s, f, ok := one(ss, section_scrap); ok {
    return s, func(){
        f()
        reduce(ss, 1, TypeSwitchCase, 0, force)
    }, true
}
```

This code is used in section 181.

267. Tests for **switch**

```
⟨ goweave/switch.w 267 ⟩ ≡
  @
  @ c
  switch tag {
    default:
      s3()
    case 0, 1, 2, 3:
      s1()
    case 4, 5, 6, 7:
      s2()
  }
  @
  @ c
  switch x := f(); ;{
    case x⟨0:
      return -x
    default:
      return x
  }
  @
  @ c
  switch{
    case x⟨y:
      f1()
    case x⟨z:
      f2()
    case x ≡ 4:
      f3()
  }
  @
  @ c
  switch i := x.(type) {
    case nil:
      printString("x␣is␣nil")
    case int:
      printInt(i)
    case float64:
      printFloat64(i)
    case func(int) float64:
      printFunction(i)
    case bool, string:
      printString("type␣is␣bool␣or␣string")
    default:
      printString("don't␣know␣the␣type")
  }
```

268. ⟨ Cases for *SelectStmt* 268 ⟩ ≡

```
if s, f1, ok := sequence(ss, select_token, lbrace); ok {
    tok_mem := append([]interface{}{}, 0, 1)
    s, f2, t, ok := optional(s, 2, pair{cat: CommClause, mand: false})
    if ok {
        tok_mem = append(tok_mem, force, indent, t, outdent)
    }
    if s, f3, ok := one(s, rbrace); ok {
        tok_mem = append(tok_mem, 2 + len(t))
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, 3 + len(t), SelectStmt, tok_mem ...)
        }, true
    }
}
```

This code is used in section 181.

269. ⟨ Cases for *CommClause* 269 ⟩ ≡

```
if s, f1, ok := sequence(ss, CommCase, colon); ok {
    tok_mem := append([]interface{}{}, 0, 1, force)
    s, f2, t, ok := optional(s, 2, pair{cat: Statement, mand: true})
    if ok {
        tok_mem = append(tok_mem, indent, t, outdent)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 2 + len(t), CommClause, tok_mem ...)
    }, true
} else if s, f, ok := one(ss, section_scrap); ok {
    return s, func(){
        f()
        reduce(ss, 1, CommClause, 0, force)
    }, true
}
```

This code is used in sections 181 and 186.

270. ⟨ Cases for *CommCase* 270 ⟩ ≡

```
if s, f1, ok := one(ss, case_token); ok {
  if s, f2, ok := any(s, SendStmt, RecvStmt); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, CommCase, 0, break_space, 1)
    }, true
  }
} else if s, f, ok := one(ss, default_token); ok {
  return s, func(){
    f()
    reduce(ss, 1, CommCase, 0)
  }, true
} else if s, f, ok := one(ss, section_scrap); ok {
  return s, func(){
    f()
    reduce(ss, 1, CommCase, 0, force)
  }, true
}
```

This code is used in section 181.

271. ⟨ Cases for *RecvStmt* 271 ⟩ ≡

```
if s, f1, ok := one(ss, ExpressionList); ok {
  if s, f2, ok := any(s, eq, col_eq); ok {
    if s, f3, ok := one(s, Expression); ok {
      return s, func(){
        f3()
        f2()
        f1()
        reduce(ss, 3, RecvStmt, 0, 1, 2)
      }, true
    }
  }
} else if s, f, ok := one(s, Expression); ok {
  return s, func(){
    f()
    reduce(ss, 1, RecvStmt, 0)
  }, true
}
```

This code is used in section 181.

272. ⟨ Cases for *SendStmt* 272 ⟩ ≡

```
if s, f, ok := sequence(ss, Expression, direct, Expression); ok {
  return s, func(){
    f()
    reduce(ss, 3, SendStmt, 0, 1, 2)
  }, true
}
```

This code is used in section 181.

273. Tests for *send*

```
⟨ goweave/send.w 273 ⟩ ≡
  @
  @ c
  ch ← 3
```

274. Tests for **select**

```
⟨ goweave/select.w 274 ⟩ ≡
  @
  @ c
  select{
    case i1 =← c1:
      print("received␣", i1, "␣from␣c1\n")
    case c2 ← i2:
      print("sent␣", i2, "␣to␣c2\n")
    case i3, ok := (← c3):      // same as: i3, ok := ¡-c3
      if ok {
        print("received␣", i3, "␣from␣c3\n")
      } else {
        print("c3␣is␣closed\n")
      }
    default:
      print("no␣communication\n")
  }
  @
  @ c
  select {@ ⟨ Case@ ⟩ case c ← 0:      // note: no statement, no fallthrough, no folding of cases
  case c ← 1:
  }
  @
  @ c
  select{}
```

275. ⟨ Cases for *ForStmt* 275 ⟩ ≡

```
if s, f1, ok := one(ss, for_token); ok {
  if s, f2, ok := sequence(s, Expression, Block); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 3, ForStmt, 0, break_space, 1, break_space, 2)
    }, true
  } else if s, f2, ok := sequence(s, ForClause, Block); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 3, ForStmt, 0, break_space, 1, break_space, 2)
    }, true
  } else if s, f2, ok := sequence(s, RangeClause, Block); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 3, ForStmt, 0, break_space, 1, break_space, 2)
    }, true
  } else if s, f2, ok := one(s, Block); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, ForStmt, 0, 1)
    }, true
  }
}
```

This code is used in section 181.

276. ⟨ Cases for *ForClause* 276 ⟩ ≡

```
var tok_mem []interface{}
c := 0
s, f1, ok := one(ss, SimpleStmt)
if ok {
  tok_mem = append(tok_mem, c)
  c++
}
f2 := empty
s, f2, ok = one(s, semi)
if ok {
  if len(scrap_info[c].trans) ≡ 0 {
    tok_mem = append(tok_mem, c)
  } else {
    tok_mem = append(tok_mem, ';')
  }
  c++
  f3 := empty
  if s, f3, ok = one(s, Expression); ok {
    tok_mem = append(tok_mem, break_space, c)
    c++
  }
  if s, f4, ok := one(s, semi); ok {
    if len(scrap_info[c].trans) ≡ 0 {
      tok_mem = append(tok_mem, c)
    } else {
      tok_mem = append(tok_mem, ';')
    }
    c++
    f5 := empty
    if s, f5, ok = one(s, SimpleStmt); ok {
      tok_mem = append(tok_mem, break_space, c)
      c++
    }
    return s, func(){
      f5()
      f4()
      f3()
      f2()
      f1()
      reduce(ss, c, ForClause, tok_mem ...)
    }, true
  }
}
```

This code is used in section 181.

277. ⟨ Cases for *RangeClause* 277 ⟩ ≡

```
if s, f1, ok := one(ss, ExpressionList); ok {
  if s, f2, ok := any(s, eq, col_eq); ok {
    if s, f3, ok := sequence(s, range_token, Expression); ok {
      return s, func(){
        f3()
        f2()
        f1()
        reduce(ss, 4, RangeClause, 0, 1, 2, break_space, 3)
      }, true
    }
  }
}
```

This code is used in section 181.

278. Tests for **for**

⟨ goweave/for.w 278 ⟩ ≡

```
@
@ c
for a⟨b {
  a *= 2
}
@
@ c
for i := 0; i⟨10; i++ {
  f(i)
}
@
@ c
for i, _ := range testdata.a {
  f(i)
}
@
@ c
for i, s := range a {
  g(i, s)
}
@
@ c
for{
  sleep(10);
  ch ← true;
}
@
@ c
for _, err := f.Read(b[:]); err ≡ nil; _, err = f.Read(b[:]) {}
```

279. ⟨ Cases for *DeferStmt* 279 ⟩ ≡

```
if s, f, ok := sequence(ss, defer_token, Expression); ok {
  return s, func(){
    f()
    reduce(ss, 2, DeferStmt, 0, break_space, 1)
  }, true
}
```

This code is used in section 181.

280. Tests for **defer**

⟨ goweave/defer.w 280 ⟩ ≡

```
@
@ c
defer unlock(l)
@
@ c
defer func(){
  result++
}()
```

281. ⟨ Cases for *IncDecStmt* 281 ⟩ ≡

```
if s, f1, ok := one(ss, Expression); ok {
  if s, f2, ok := any(s, plus_plus, minus_minus); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 2, IncDecStmt, 0, 1)
    }, true
  }
}
```

This code is used in section 181.

282. Tests for *incdec*

⟨ goweave/incdec.w 282 ⟩ ≡

```
@
@ c
i++
@
@ c
j--
```

283. ⟨ Cases for *Assignment* 283 ⟩ ≡

```
if s, f, ok := sequence(ss, ExpressionList, assign_op, ExpressionList); ok {
  return s, func(){
    f()
    reduce(ss, 3, Assignment, 0, 1, 2)
  }, true
}
```

This code is used in section 181.

284. Tests for assignments

⟨ goweave/assign.w 284 ⟩ ≡

```
@
@ c
x = 1
@
@ c
*p = f()
@
@ c
a[i] = 23
@
@ c
(k) =<- ch
@
@ c
a[i] <<= 2
@
@ c
i &^= 1 << n
@
@ c
x, y = f()
@
@ c
x, _ = f()
@
@ c
a, b = b, a
@
@ c
i, x[i] = 1, 2
@
@ c
i = 0
@
@ c
x[i], i = 2, 1
@
@ c
x[0], x[0] = 1, 2
@
@ c
x[1], x[3] = 4, 5
@
@ c
x[2], p.x = 6, 7
@
@ c
i = 2
@
@ c
```

```
x = []int{3, 5, 7}
```

285. ⟨ Cases for *assign_op* 285 ⟩ ≡

```
if s, f, ok := sequence(ss, binary_op, eq); ok {
    return s, func(){
        f()
        reduce(ss, 2, assign_op, math_rel, '{', 0, '}', '{', 1, '}', '}')
    }, true
} else if s, f, ok := one(ss, eq); ok {
    return s, func(){
        f()
        reduce(ss, 1, assign_op, 0)
    }, true
}
```

This code is used in section 181.

286. ⟨ Cases for *ShortVarDecl* 286 ⟩ ≡

```
if s, f, ok := sequence(ss, IdentifierList, col_eq, ExpressionList); ok {
    return s, func(){
        f()
        reduce(ss, 3, ShortVarDecl, 0, 1, 2)
    }, true
}
```

This code is used in section 181.

287. Tests for short var declarations

⟨ goweave/shortvar.w 287 ⟩ ≡

```
@
@ c
i, j := 0, 10
@
@ c
f := func() int{
    return 7
}
@
@ c
ch := make(chan int)
@
@ c
r, w := os.Pipe(fd)
@
@ c
_, y, _ := coord(p)
@
@ c
ints = make(map[string]int)
```

288. ⟨ Cases for *QualifiedIdent* 288 ⟩ ≡

```
if s, f1, ok := one(ss, identifier); ok {
  if s, f2, ok := sequence(s, dot, identifier); ok {
    return s, func(){
      f2()
      f1()
      reduce(ss, 3, QualifiedIdent, 0, 1, 2)
          // make_reserved(ss[0],ss[0].cat)
    }, true
  } else {
    return s, func(){
      f1()
      reduce(ss, 1, QualifiedIdent, 0)
    }, true
  }
}
```

This code is used in section 181.

289. ⟨ Cases for *MethodExpr* 289 ⟩ ≡

```
if s, f, ok := sequence(ss, ReceiverType, dot, identifier); ok {
  return s, func(){
    f()
    reduce(ss, 3, MethodExpr, 0, 1, 2)
  }, true
}
```

This code is used in section 181.

290. ⟨ Cases for *ReceiverType* 290 ⟩ ≡

```
if s, f, ok := one(ss, Type); ok {
  return s, func(){
    f()
    reduce(ss, 1, ReceiverType, 0)
  }, true
} else if s, f, ok := sequence(ss, lpar, asterisk, Type, rpar); ok {
  return s, func(){
    f()
    reduce(ss, 4, ReceiverType, 0, 1, 2, 3)
  }, true
}
```

This code is used in section 181.

291. ⟨ Cases for *Conversion* 291 ⟩ ≡

```
if s, f, ok := sequence(ss, Type, lpar, Expression, rpar); ok {
  return s, func(){
    f()
    reduce(ss, 4, Conversion, 0, 1, 2, 3)
  }, true
}
```

This code is used in section 181.

292. ⟨ Cases for *BuiltinCall* 292 ⟩ ≡

```
if s, f1, ok := sequence(ss, identifier, lpar); ok {
    tok_mem := append([]interface{}{}, 0, 1)
    s, f2, t, ok := optional(s, 2, pair{cat: BuiltinArgs, mand: true}, pair{cat: comma, mand: false})
    if ok {
        tok_mem = append(tok_mem, t)
    }
    if s, f3, ok := one(s, rpar); ok {
        tok_mem = append(tok_mem, 2 + len(t))
        return s, func(){
            f3()
            f2()
            f1()
            reduce(ss, 3 + len(t), BuiltinCall, tok_mem ...)
        }, true
    }
}
```

This code is used in section 181.

293. ⟨ Cases for *BuiltinArgs* 293 ⟩ ≡

```
if s, f1, ok := one(ss, Type); ok {
    tok_mem := append([]interface{}{}, 0)
    s, f2, t, ok := optional(s, 1, pair{cat: comma, mand: true}, pair{cat: ExpressionList, mand: true})
    if ok {
        tok_mem = append(tok_mem, t)
    }
    return s, func(){
        f2()
        f1()
        reduce(ss, 1 + len(t), BuiltinArgs, tok_mem ...)
    }, true
} else if s, f, ok := one(s, ExpressionList); ok {
    return s, func(){
        f()
        reduce(ss, 1, BuiltinArgs, 0)
    }, true
}
```

This code is used in section 181.

294. ⟨ Cases for *Selector* 294 ⟩ ≡

```
if s, f, ok := sequence(ss, dot, identifier); ok {
    return s, func(){
        f()
        reduce(ss, 2, Selector, 0, 1)
    }, true
}
```

This code is used in section 181.

295. ⟨ Cases for *Index* 295 ⟩ ≡

```
if s, f, ok := sequence(ss, lbracket, Expression, rbracket); ok {
  return s, func(){
    f()
    reduce(ss, 3, Index, 0, 1, 2)
  }, true
}
```

This code is used in section 181.

296. ⟨ Cases for *Slice* 296 ⟩ ≡

```
if s, f1, ok := one(ss, lbracket); ok {
  tok_mem := append([]interface{}{}, 0)
  s, f2, t1, ok := optional(s, 1, pair{cat: Expression, mand: false})
  if ok {
    tok_mem = append(tok_mem, t1)
  }
  if s, f3, ok := one(s, colon); ok {
    tok_mem = append(tok_mem, 1 + len(t1))
    s, f4, t2, ok := optional(s, 1 + len(t1) + 1, pair{cat: Expression, mand: false})
    if ok {
      tok_mem = append(tok_mem, t2)
    }
    if s, f5, ok := one(s, rbracket); ok {
      tok_mem = append(tok_mem, 2 + len(t1) + len(t2))
      return s, func(){
        f5()
        f4()
        f3()
        f2()
        f1()
        reduce(ss, 3 + len(t1) + len(t2), Slice, tok_mem...)
      }, true
    }
  }
}
```

This code is used in section 181.

297. ⟨ Cases for *TypeAssertion* 297 ⟩ ≡

```
if s, f, ok := sequence(ss, dot, lpar, Type, rpar); ok {
  return s, func(){
    f()
    reduce(ss, 4, TypeAssertion, 0, 1, 2, 3)
  }, true
}
```

This code is used in section 181.

298. ⟨ Cases for *Call* 298 ⟩ ≡

```
if s, f1, ok := one(ss, lpar); ok {
    tok_mem := append([]interface{}{}, 0)
    c := 1
    s, f2, ok := one(s, ExpressionList)
    f3 := empty
    if ok {
        tok_mem = append(tok_mem, c)
        c++
        s, f3, ok = one(s, dot_dot_dot)
        if ok {
            tok_mem = append(tok_mem, c)
            c++
        }
    }
    if s, f4, ok := one(s, rpar); ok {
        tok_mem = append(tok_mem, c)
        c++
        return s, func(){
            f4()
            f3()
            f2()
            f1()
            reduce(ss, c, Call, tok_mem ...)
        }, true
    }
}
```

This code is used in section 181.

299. ⟨ Cases for *unary_op* 299 ⟩ ≡

```
if s, f, ok := any(ss, asterisk, direct, add_op, mul_op); ok {
    return s, func(){
        f()
        reduce(ss, 1, unary_op, 0)
    }, true
}
```

This code is used in section 181.

300. Now here's the *reduce* procedure used in our code for productions.

```
⟨ Making translation for an element v of scrap sequence 300 ⟩ ≡
  switch s.mathness % 4 {      /* left boundary */
    case no_math:
      if cur_mathness ≡ maybe_math {
        init_mathness = no_math
      } else if cur_mathness ≡ yes_math {
        trans = append(trans, "{}$")
      }
      cur_mathness = s.mathness/4      /* right boundary */
    case yes_math:
      if cur_mathness ≡ maybe_math {
        init_mathness = yes_math
      } else if cur_mathness ≡ no_math {
        trans = append(trans, "${}")
      }
      cur_mathness = s.mathness/4      /* right boundary */
    case maybe_math:      /* no changes */
  }
  trans = append(trans, s.trans ...)
```

This code is used in section 302.

301. The *reduce* function makes a reducing of scraps and a correcting of a *mathness* of an expression.

The *mathness* is an attribute of scraps that says whether they are to be printed in a math mode context or not. It is separate from the "part of speech" (the *cat*) because to make each *cat* have a fixed *mathness*.

The low two bits (i.e. *mathness* % 4) control the left boundary. (We need two bits because we allow cases *yes_math*, *no_math* and *maybe_math*, which can go either way.) The next two bits (i.e. *mathness*/4) control the right boundary. If we combine two scraps and the right boundary of the first has a different mathness from the left boundary of the second, we insert a `$` in between. Similarly, if at printing time some irreducible scrap has a *yes_math* boundary the scrap gets preceded or followed by a `$`. The left boundary is *maybe_math* if and only if the right boundary is.

A reducing is made by moving a tail of the slices *ss* and *scrap_info* at position 1.

```
⟨ Constants 1 ⟩ +≡
  const(
    maybe_math rune = iota      /* works in either horizontal or math mode */
    yes_math rune = iota      /* should be in math mode */
    no_math rune = iota      /* should be in horizontal mode */
  )
```

302.

```
func reduce(ss []scrap, k int, c rune, s ...interface{}){
  var trans []interface{}
  cur_mathness := maybe_math
  init_mathness := maybe_math
  for _, t := range s {
    switch v := t.(type) {
      case rune:
        if v ≡ '␣' ∨ (v ≥ big_cancel ∧ v ≤ big_force)      /* non-math token */ {
          if cur_mathness ≡ maybe_math {
            init_mathness = no_math
          } else if cur_mathness ≡ yes_math {
            trans = append(trans, "{}$")
          }
          cur_mathness = no_math
        } else {
          if cur_mathness ≡ maybe_math {
            init_mathness = yes_math
          } else if cur_mathness ≡ no_math {
            trans = append(trans, "${}")
          }
          cur_mathness = yes_math
        }
        trans = append(trans, v)
      case int:
        s := ss[v]
        ⟨ Making translation for an element v of scrap sequence 300 ⟩
      case []int:
        for _, v := range v {
          if v ≡ −1 {
            continue
          }
          s := ss[v]
          ⟨ Making translation for an element v of scrap sequence 300 ⟩
        }
      case string:
        trans = append(trans, v)
      default:
        panic(fmt.Sprintf("Invalid␣type␣of␣translation:␣%T(%v)", v, v))
    }
  }
  if init_mathness ≡ maybe_math ∧ cur_mathness ≠ maybe_math {
    init_mathness = cur_mathness
  }
  ss[0] = scrap{cat: c, trans: trans, mathness: 4 * cur_mathness + init_mathness, }
  if k⟩1 {
    copy(ss[1:], ss[k:])
    ss = ss[:len(ss) − k + 1]
    scrap_info = scrap_info[:len(scrap_info) − k + 1]
  }
  f := fmt.Sprintf("reduce␣%q␣%v", cat_name[c], k)
  ⟨ Print a snapshot of the scrap list if debugging 306 ⟩
```

```
    if (tracing & 8) ≡ 8 {
      fmt.Printf("translation␣of␣%s:␣%v\n", cat_name[c], trans)
    }
  }
```

303. And here now is the code that applies productions as long as possible.

304. ⟨ Reduce the scraps using the productions until no more rules apply 304 ⟩ ≡

```
for{
  if pp ≥ len(scrap_info) {
    break
  }
  ⟨ Match a production at pp, or increase pp if there is no match 186 ⟩
}
```

This code is used in section 307.

305. If GOWEAVE is being run in debugging mode, the productions and current stack categories will be printed out when *tracing* is set to 2; a sequence of two or more irreducible scraps will be printed out when *tracing* is set to 1.

⟨ Global variables 93 ⟩ +≡

```
var tracing int32      /* can be used to show parsing details */
```

306. ⟨ Print a snapshot of the scrap list if debugging 306 ⟩ ≡

```
{
  if (tracing & 2) ≡ 2 {
    fmt.Printf("%s:", f)
    for i, v := range scrap_info {
      if i ≡ len(scrap_info) − len(ss) {
        fmt.Print("␣*")
      } else {
        fmt.Print("␣")
      }
      if v.mathness % 4 ≡ yes_math {
        fmt.Print("+")
      } else if v.mathness % 4 ≡ no_math {
        fmt.Print("-")
      }
      print_cat(v.cat)
      if v.mathness/4 ≡ yes_math {
        fmt.Print("+")
      } else if v.mathness/4 ≡ no_math {
        fmt.Print("-")
      }
    }
    fmt.Println()
  }
}
```

This code is used in sections 181 and 302.

307. The *translate* function assumes that scraps have been stored in *scrap_info* of *cat* and *trans*. It applies productions as much as possible. The result is a token list containing the translation of the given sequence of scraps.

```
    /* converts a sequence of scraps */
func translate() []interface{}{
  pp := 0
  ⟨ If tracing, print an indication of where we are 310 ⟩
  ⟨ Reduce insert productions 311 ⟩
  ⟨ Reduce the scraps using the productions until no more rules apply 304 ⟩
  ⟨ Combine the irreducible scraps that remain 308 ⟩
}
```

308. If the initial sequence of scraps does not reduce to a single scrap, we concatenate the translations of all remaining scraps, separated by blank spaces, with dollar signs surrounding the translations of scraps where appropriate.

```
⟨ Combine the irreducible scraps that remain 308 ⟩ ≡
  {
    ⟨ If semi-tracing, show the irreducible scraps 309 ⟩
    var tok_mem []interface{}
    for i, v := range scrap_info {
      if i ≠ 0 {
        tok_mem = append(tok_mem, '␣')
      }
      if v.mathness % 4 ≡ yes_math {
        tok_mem = append(tok_mem, '$')
      }
      tok_mem = append(tok_mem, v.trans ...)
      if v.mathness/4 ≡ yes_math {
        tok_mem = append(tok_mem, '$')
      }
    }
    return tok_mem
  }
```

This code is used in section 307.

309.

```
⟨ If semi-tracing, show the irreducible scraps 309 ⟩ ≡
  if len(scrap_info)⟩0 ∧ (tracing & 1) ≡ 1 {
    s := ""
    for i, _ := range scrap_info {
      s += fmt.Sprintf("␣%s", cat_name[scrap_info[i].cat])
    }
    warn_print("Irreducible␣scrap␣sequence␣in␣section␣%d:%s", section_count, s)
  }
```

This code is used in section 308.

310.

```
⟨ If tracing, print an indication of where we are 310 ⟩ ≡
  if (tracing & 2) ≡ 2 {
    warn_print("Tracing␣after␣%s:%d:\n", file_name[include_depth], line[include_depth])
  }
```

This code is used in section 307.

311. ⟨ Reduce *insert* productions 311 ⟩ ≡

```
for i := 1; i⟨len(scrap_info); {
  if scrap_info[i].cat ≡ insert {
    reduce(scrap_info[i − 1:], 2, scrap_info[i − 1].cat, 0, 1)
    continue
  }
  i++
}
if len(scrap_info)⟩1 ∧ scrap_info[0].cat ≡ insert ∧ scrap_info[1].cat ≠ zero {
  reduce(scrap_info, 2, scrap_info[1].cat, 0, 1)
}
```

This code is used in section 307.

312. Initializing the scraps. If we are going to use the powerful production mechanism just developed, we must get the scraps set up in the first place, given a Go text. A table of the initial scraps corresponding to Go tokens appeared above in the section on parsing; our goal now is to implement that table. We shall do this by implementing a subroutine called *Go_parse* that is analogous to the *Go_xref* routine used during phase one.

Like *Go_xref*, the *Go_parse* procedure starts with the current value of *next_control* and it uses the operation *next_control* = *get_next*() repeatedly to read Go text until encountering the next '`|`' or '`/*`', or until *next_control* ≥ *format_code*. The scraps corresponding to what it reads are appended into the *cat* and *trans* arrays, and *scrap_ptr* is advanced.

```
  /* creates scraps from Go tokens */
func Go_parse(spec_ctrl rune){
  for next_control⟨format_code ∨ next_control ≡ spec_ctrl {
    ⟨ Append the scrap appropriate to next_control 314 ⟩
    next_control = get_next()
    if next_control ≡ '|' ∨ next_control ≡ begin_comment ∨ next_control ≡ begin_short_comment {
      return
    }
  }
}
```

313. The following function is used to append a scrap whose tokens have just been appended:

```
func app_scrap(c int32, b int32, t ...interface{}){
  scrap_info = append(scrap_info, scrap{cat: c, trans: t, mathness: 5 * b, })
}
```

314. ⟨ Append the scrap appropriate to *next_control* 314 ⟩ ≡

```
switch (next_control) {
  case section_name:
    app_scrap(section_scrap, maybe_math, section_token(cur_section))
  case str, constant, verbatim:
    ⟨ Append a string or constant 316 ⟩
  case identifier:
    app_id(id)
  case TEX_string:
    ⟨ Append a TEX string 317 ⟩
  case raw_TEX_string:
    ⟨ Append a raw TEX string 318 ⟩
  case '/':
    app_scrap(mul_op, yes_math, next_control)
    next_control = mul_op
  case '.':
    app_scrap(dot, yes_math, next_control)
    next_control = dot
  case '_':
    app_scrap(identifier, maybe_math, "\\_")
    next_control = identifier
  case '<':
    app_scrap(rel_op, yes_math, "\\langle")
    next_control = rel_op
  case '>':
    app_scrap(rel_op, yes_math, "\\rangle")
    next_control = rel_op
  case '=':
    app_scrap(eq, yes_math, "\\K")
    next_control = eq
  case '|':
    app_scrap(add_op, yes_math, "\\OR")
    next_control = add_op
  case '^':
    app_scrap(add_op, yes_math, "\\CF")
    next_control = add_op
  case '%':
    app_scrap(mul_op, yes_math, "\\MOD")
    next_control = mul_op
  case '!':
    app_scrap(unary_op, yes_math, "\\R")
    next_control = unary_op
  case '+', '-':
    app_scrap(add_op, yes_math, next_control)
    next_control = add_op
  case '*':
    app_scrap(asterisk, yes_math, next_control)
    next_control = asterisk
  case '&':
    app_scrap(mul_op, yes_math, "\\AND")
    next_control = mul_op
  case ignore, xref_roman, xref_wildcard, xref_typewriter, noop:
```

```
    break
  case '(':
    app_scrap(lpar, maybe_math, next_control)
    next_control = lpar
  case ')':
    app_scrap(rpar, maybe_math, next_control)
    next_control = rpar
  case '[':
    app_scrap(lbracket, maybe_math, next_control)
    next_control = lbracket
  case ']':
    app_scrap(rbracket, maybe_math, next_control)
    next_control = rbracket
  case '{':
    app_scrap(lbrace, yes_math, "\\{")
    next_control = lbrace
  case '}':
    app_scrap(rbrace, yes_math, "\\}")
    next_control = rbrace
  case ',':
    app_scrap(comma, yes_math, next_control, opt, '9',)
    next_control = comma
  case ';':
    app_scrap(semi, maybe_math, next_control)
    next_control = semi
  case ':':
    app_scrap(colon, no_math, next_control)
    next_control = colon
    ⟨ Cases involving nonstandard characters 315 ⟩
  case thin_space:
    app_scrap(insert, maybe_math, "\\,")
    next_control = thin_space
  case math_break:
    app_scrap(insert, maybe_math, opt, "0")
    next_control = insert
  case line_break:
    app_scrap(insert, no_math, force)
    next_control = insert
  case big_line_break:
    app_scrap(insert, no_math, big_force)
    next_control = insert
  case no_line_break:
    app_scrap(insert, no_math, big_cancel, noop, break_space, noop, big_cancel)
    next_control = insert
  case pseudo_semi:
    next_control = semi
    app_scrap(semi, maybe_math)
  case join:
    app_scrap(insert, no_math, "\\J")
    next_control = insert
  default:
    app_scrap(insert, maybe_math, inserted, next_control)
```

next_control = *insert*
}

This code is used in section 312.

315. Some nonstandard characters may have entered GOWEAVE by means of standard sequence. They are converted to TEX control sequences so that it is possible to keep GOWEAVE from outputting unusual **rune** codes.

⟨ Cases involving nonstandard characters 315 ⟩ ≡

```
case not_eq:
  app_scrap(rel_op, yes_math, "\\I")
case lt_eq:
  app_scrap(rel_op, yes_math, "\\Z")
case gt_eq:
  app_scrap(rel_op, yes_math, "\\G")
case eq_eq:
  app_scrap(rel_op, yes_math, "\\E")
case and_and:
  app_scrap(binary_op, yes_math, "\\W")
case or_or:
  app_scrap(binary_op, yes_math, "\\V")
case plus_plus:
  app_scrap(plus_plus, yes_math, "\\PP")
case minus_minus:
  app_scrap(minus_minus, yes_math, "\\MM")
case gt_gt:
  app_scrap(mul_op, yes_math, "\\GG")
case lt_lt:
  app_scrap(mul_op, yes_math, "\\LL")
case dot_dot_dot:
  app_scrap(dot_dot_dot, yes_math, "\\ldots")
case col_eq:
  app_scrap(col_eq, yes_math, ":\\K")
case direct:
  app_scrap(direct, yes_math, "\\leftarrow")
case and_not:
  app_scrap(mul_op, yes_math, "\\AND\\CF")
```

This code is used in section 314.

316. Many of the special characters in a string must be prefixed by '\' so that TEX will print them properly.

```
⟨ Append a string or constant 316 ⟩ ≡
  count := −1
  var tok_mem []interface{}
  if next_control ≡ constant {
    tok_mem = append(tok_mem, "\\T{")
  } else if next_control ≡ str {
    count = 20
    tok_mem = append(tok_mem, "\\.{")
  } else {
    tok_mem = append(tok_mem, "\\vb{")
  }
  for i := 0; i⟨len(id); {
    if count ≡ 0 {      /* insert a discretionary break in a long string */
      tok_mem = append(tok_mem, "}\\)\\.{")
      count = 20
    }
    switch id[i] {
      case '␣', '\\', '#', '%', '$', '^', '{', '}', '~', '&', '_':
        tok_mem = append(tok_mem, '\\')
      case '@':
        if i + 1⟨len(id) ∧ id[i + 1] ≡ '@' {
          i++
        } else {
          err_print("!␣Double␣@␣should␣be␣used␣in␣strings")
        }
    }
    tok_mem = append(tok_mem, id[i])
    i++
    count−−
  }
  tok_mem = append(tok_mem, '}')
  app_scrap(next_control, maybe_math, tok_mem ...)
```

This code is used in section 314.

317.

```
⟨ Append a TEX string 317 ⟩ ≡
  tok_mem := append([]interface{}{}, "\\hbox{")
  for i := 0; i⟨len(id); {
    if id[i] ≡ '@' {
      i++
    }
    tok_mem = append(tok_mem, id[i])
    i++
  }
  tok_mem = append(tok_mem, '}')
  app_scrap(insert, no_math, tok_mem ...)
```

This code is used in section 314.

318.

```
⟨ Append a raw TEX string 318 ⟩ ≡
  tok_mem := make([]interface{}, 0, len(id))
  for i := 0; i⟨len(id); {
    if id[i] ≡ '@' {
      i++
    }
    tok_mem = append(tok_mem, id[i])
    i++
  }
  app_scrap(insert, no_math, tok_mem ...)
```

This code is used in section 314.

319. The function *app_id* appends an identifier *id* to the token list.

```
func app_id(id []rune) id_token{
  p := id_lookup(id, normal)
  if name_dir[p].ilk ≤ custom {      /* not a reserved word */
    a1 := identifier
    a2 := maybe_math
    if name_dir[p].ilk ≡ custom {
      a2 = yes_math
    }
    app_scrap(a1, a2, id_token(p))
  } else {
    if
      name_dir[p].ilk ≡ binary_op ∨ name_dir[p].ilk ≡ rel_op ∨ name_dir[p].ilk ≡ add_op ∨ name_dir[p].ilk ≡
          mul_op {
      app_scrap(name_dir[p].ilk, yes_math, res_token(p))
    } else {
      app_scrap(name_dir[p].ilk, maybe_math, res_token(p))
    }
  }
  return id_token(p)
}
```

320. When the '|' that introduces Go text is sensed, a call on *Go_translate* will return a pointer to the TEX translation of that text. If scraps exist in *scrap_info*, they are unaffected by this translation process.

```
func Go_translate() []interface{}{
  save_scraps := scrap_info      /* holds original value of scrap_info */
  scrap_info = nil
  Go_parse(section_name)      /* get the scraps together */
  if next_control ≠ '|' {
    err_print("! Missing '|' after Go text")
  }
  app_scrap(semi, no_math)
  app_scrap(insert, maybe_math, cancel)
      /* place a cancel token as a final "comment" */
  p := translate()      /* make the translation */
  scrap_info = save_scraps      /* scrap the scraps */
  return p
}
```

321. The *outer_parse* routine is to *Go_parse* as *outer_xref* is to *Go_xref*: It constructs a sequence of scraps for Go text until *next_control* $\geq$ *format_code*. Thus, it takes care of embedded comments.

The token list created from within '`|...|`' brackets is output as an argument to `\PB`, if the user has invoked `GOWEAVE` with the `+e` flag. Although `gowebmac` ignores `\PB`, other macro packages might use it to localize the special meaning of the macros that mark up program text.

```
    /* makes scraps from Go tokens and comments */
func outer_parse(){
  for next_control⟨format_code {
    var tok_mem []interface{}
    if next_control ≠ begin_comment ∧ next_control ≠ begin_short_comment {
      Go_parse(ignore)
    } else {
      is_long_comment := (next_control ≡ begin_comment)
      tok_mem = append(tok_mem, inserted)
          // checking if a comment is placed at start of the line
      s := true
      for i := 0; i⟨loc − 2; i++ {
        if ¬unicode.IsSpace(buffer[i]) {
          s = false
          break
        }
      }
      if s {
        tok_mem = append(tok_mem, force)
      }
      if is_long_comment {
        tok_mem = append(tok_mem, "\\C{")
      } else {
        tok_mem = append(tok_mem, "\\SHC{")
      }
      var bal int
      bal, tok_mem = copy_comment(is_long_comment, 1, tok_mem)      /* brace level in comment */
      next_control = ignore
      for bal⟩0 {
        p := tok_mem
        tok_mem = nil
        q := Go_translate()      /* partial comments */
        tok_mem = append(tok_mem, list_token(p))
        if flags['e'] {
          tok_mem = append(tok_mem, "\\PB{")
        }
        tok_mem = append(tok_mem, inner_list_token(q))
        if flags['e'] {
          tok_mem = append(tok_mem, '}')
        }
        if next_control ≡ '|' {
          bal, tok_mem = copy_comment(is_long_comment, bal, tok_mem)
          next_control = ignore
        } else {
          bal = 0      /* an error has been reported */
        }
      }
```

```
            // checking if the comment is a last entity in the line
          for loc⟨len(buffer) ∧ unicode.IsSpace(buffer[loc]) {
            loc++
          }
          if loc ≥ len(buffer) {
            tok_mem = append(tok_mem, force)
          }
          app_scrap(insert, no_math, tok_mem ...)
            /* the full comment becomes a scrap */
        }
      }
    }
```

322. Output of tokens. So far our programs have only built up multi-layered token lists in GOWEAVE's internal memory; we have to figure out how to get them into the desired final form. The job of converting token lists to characters in the TEX output file is not difficult, although it is an implicitly recursive process. Four main considerations had to be kept in mind when this part of GOWEAVE was designed. (a) There are two modes of output: *outer* mode, which translates tokens like *force* into line-breaking control sequences, and *inner* mode, which ignores them except that blank spaces take the place of line breaks. (b) The *cancel* instruction applies to adjacent token or tokens that are output, and this cuts across levels of recursion since '*cancel*' occurs at the beginning or end of a token list on one level. (c) The TEX output file will be semi-readable if line breaks are inserted after the result of tokens like *break_space* and *force*. (d) The final line break should be suppressed, and there should be no *force* token output immediately after '`\Y\B`'.

323. The output process uses a stack to keep track of what is going on at different "levels" as the token lists are being written out. Entries on this stack have three parts:

tok_field is a slice of tokens from begin of which the next token on a particular level will be read;

mode_field is the current mode, either *inner* or *outer*.

The current values of these 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.end_field*, *cur_state.tok_field*, and *cur_state.mode_field*.

The end of output occurs when an *end_translation* token is found, so the stack is never empty except when we first begin the output process.

```
⟨ Typedef declarations 94 ⟩ +≡
  type mode int
```

324.
```
⟨ Constants 1 ⟩ +≡
  const(
    inner mode = 0 /* value of mode for Go texts within TEX texts */
    outer mode = 1 /* value of mode for Go texts in sections */
  )
```

325.
```
⟨ Typedef declarations 94 ⟩ +≡
  type output_state struct{
    tok_field []interface{} /* present location of token list */
    mode_field mode /* interpretation of control tokens */
  }
```

326.
```
  func init_stack(){
    stack = make([]output_state, 0, 100)
    cur_state.mode_field = outer
  }
```

327.
```
⟨ Global variables 93 ⟩ +≡
  var cur_state output_state /* cur_state.tok_field, cur_state.mode_field */
  var stack []output_state /* info for non-current levels */
```

328. To insert token-list p into the output, the *push_level* subroutine is called; it saves the old level of output and gets a new one going. The value of *cur_state.mode_field* is not changed.

```
    /* suspends the current level */
func push_level(tokens []interface{}){
  stack = append(stack, output_state{tok_field: cur_state.tok_field, mode_field: cur_state.mode_field, })
  cur_state.tok_field = tokens
}
```

329. Conversely, the *pop_level* routine restores the conditions that were in force when the current level was begun.

```
func pop_level() bool{
  if len(stack) ≡ 0 {
    return false
  }
  p := len(stack) − 1
  cur_state = stack[p]
  stack = stack[:p]
  return true
}
```

330. The *get_output* function returns the next byte of output that is not a reference to a token list. It returns the values *identifier* or *res_word* or *section_code* if the next token is to be an identifier (typeset in italics), a reserved word (typeset in boldface), or a section name (typeset by a complex routine that might generate additional levels of output). In these cases *cur_name* points to the identifier or section name in question.

```
⟨ Global variables 93 ⟩ +≡
  var cur_name int32 = −1
```

331.

```
⟨ Constants 1 ⟩ +≡
  const(
    res_word rune = °242    /* returned by get_output for reserved words */
    section_code rune = °243    /* returned by get_output for section names */
  )
```

332.

```
    /* returns the next token of output */
  func get_output() rune{
    restart:
    for len(cur_state.tok_field) ≡ 0 {
      if ¬pop_level() {
        return −1
      }
    }
    val := cur_state.tok_field[0]
    cur_state.tok_field = cur_state.tok_field[1:]
    switch tok := val.(type) {
      case id_token:
        cur_name = int32(tok)
        return identifier
      case res_token:
        cur_name = int32(tok)
        return res_word
      case section_token:
        cur_name = int32(tok)
        return section_code
      case inner_list_token:
        push_level(tok)
        cur_state.mode_field = inner
        goto restart
      case list_token:
        push_level(tok)
        goto restart
      case rune:
        return tok
      case []interface{}:
        push_level(tok)
        goto restart
      case string:
        var tok_mem []interface{}
        for _, v := range tok {
          tok_mem = append(tok_mem, v)
        }
        push_level(tok_mem)
        goto restart
    }
    panic(fmt.Sprintf("Invalid␣type␣of␣scrap:␣%T(%v)", val, val))
  }
```

333. The real work associated with token output is done by *make_output*. This procedure appends an *end_translation* token to the current token list, and then it repeatedly calls *get_output* and feeds characters to the output buffer until reaching the *end_translation* sentinel. It is possible for *make_output* to be called recursively, since a section name may include embedded Go text; however, the depth of recursion never exceeds one level, since section names cannot be inside of section names.

A procedure called *output_Go* does the scanning, translation, and output of Go text within '| ... |' brackets, and this procedure uses *make_output* to output the current token list. Thus, the recursive call of *make_output* actually occurs when *make_output* calls *output_Go* while outputting the name of a section.

```
  /* outputs the current token list */
func output_Go(){
  save_next_control := next_control  /* values to be restored */
  next_control = ignore
  p := Go_translate()  /* translation of the Go text */
  if flags['e'] {
    out_str("\\PB{")
    make_output(inner_list_token(p))
    out('}')
  } else {
    make_output(inner_list_token(p))  /* output the list */
  }
  next_control = save_next_control  /* restore next_control to original state */
}
```

334. Here is GOWEAVE's major output handler.

```
    /* outputs the equivalents of tokens */
  func make_output(p interface{}){
    var c int      /* count of indent and outdent tokens */
    tok_mem := append([]interface{}{}, p, end_translation)      /* append a sentinel */
    push_level(tok_mem)
    tok_mem = nil
    var b rune
    for{
      a := get_output()      /* current output byte */
      reswitch:
      switch a {
        case end_translation:
          return
        case identifier, res_word:
          ⟨ Output an identifier 335 ⟩
        case section_code:
          ⟨ Output a section name 340 ⟩
        case math_rel:
          out_str("\\MRL{")
        case noop, inserted:
          break
        case cancel, big_cancel:
          c = 0
          b = a
          for{
            a = get_output()
            if a ≡ inserted {
              continue
            }
            if a⟨indent ∧ ¬(b ≡ big_cancel ∧ a ≡ '␣') ∨ a⟩big_force {
              break
            }
            if a ≡ indent {
              c++
            } else if a ≡ outdent {
              c--
            } else if a ≡ opt {
              a = get_output()
            }
          }
          ⟨ Output saved indent or outdent tokens 339 ⟩
          goto reswitch
        case indent, outdent, opt, backup, break_space, force, big_force:
          ⟨ Output a control, look ahead in case of line breaks, possibly goto reswitch 337 ⟩
        case quoted_char:
          out(cur_state.tok_field[0].(rune))
          cur_state.tok_field = cur_state.tok_field[1:]
        default:
          out(a)      /* otherwise a is an ordinary character */
      }
    }
```

```
}
```

335. An identifier of length one does not have to be enclosed in braces, and it looks slightly better if set in a math-italic font instead of a (slightly narrower) text-italic font. Thus we output '`\|a`' but '`\\{aa}`'.

```
⟨ Output an identifier 335 ⟩ ≡
  out('\\')
  if a ≡ identifier { if name_dir[cur_name].ilk ≡ custom ∧ ¬doing_format{⟨ Custom out 336 ⟩} else if
    is_tiny(cur_name) {
    out('|')
  } else {
    delim := '.'
    for _, v := range name_dir[cur_name].name {
      if unicode.IsLower(v) {      /* not entirely uppercase */
        delim = '\\'
        break
      }
    }
    out(delim)
  }
  } else {
    out('&')      /* a ≡ res_word */
  }
  if is_tiny(cur_name) {
    if name_dir[cur_name].name[0] ≡ '_' {
      out('\\')
    }
    out(name_dir[cur_name].name[0])
  } else {
    out_name(cur_name, true)
  }
```

This code is used in section 334.

336.

```
⟨ Custom out 336 ⟩ ≡
  for _, v := range name_dir[cur_name].name {
    if v ≡ '_' {
      out('x')
    } else {
      out(v)
    }
  }
  break
```

This code is used in section 335.

337. The current mode does not affect the behavior of GOWEAVE's output routine except when we are outputting control tokens.

```
⟨ Output a control, look ahead in case of line breaks, possibly goto reswitch 337 ⟩ ≡
  if a⟨break_space {
    if cur_state.mode_field ≡ outer {
      out('\\')
      out(a − cancel + '0')
      if a ≡ opt {
        b = get_output()      /* opt is followed by a digit */
        if b ≠ '0' ∨ flags['f'] ≡ false {
          out(b)
        } else {
          out_str("{-1}")      /* flags['f'] encourages more @| breaks */
        }
      }
    } else if a ≡ opt {
      b = get_output()      /* ignore digit following opt */
    }
  } else {
    ⟨ Look ahead for strongest line break, goto reswitch 338 ⟩
  }
```

This code is used in section 334.

338. If several of the tokens *break_space*, *force*, *big_force* occur in a row, possibly mixed with blank spaces (which are ignored), the largest one is used. A line break also occurs in the output file, except at the very end of the translation. The very first line break is suppressed (i.e., a line break that follows '`\Y\B`').

```
⟨ Look ahead for strongest line break, goto reswitch 338 ⟩ ≡
  {
    b = a
    save_mode := cur_state.mode_field      /* value of cur_state.mode_field before a sequence of breaks */
    c = 0
    for{
      a = get_output()
      if a ≡ inserted {
        continue
      }
      if a ≡ cancel ∨ a ≡ big_cancel {
        ⟨ Output saved indent or outdent tokens 339 ⟩
        goto reswitch      /* cancel overrides everything */
      }
      if a ≠ '␣' ∧ a⟨indent ∨ a ≡ backup ∨ a⟩big_force {
        if save_mode ≡ outer {
          if out_ptr⟩3 ∧ compare_runes(out_buf[out_ptr − 3:out_ptr + 1], []rune("\\Y\\B")) ≡ 0 {
            goto reswitch
          }
          ⟨ Output saved indent or outdent tokens 339 ⟩
          out('\\')
          out(b − cancel + '0')
          if a ≠ end_translation {
            finish_line()
          }
        } else if a ≠ end_translation ∧ cur_state.mode_field ≡ inner {
          out('␣')
        }
        goto reswitch
      }
      if a ≡ indent {
        c++
      } else if a ≡ outdent {
        c−−
      } else if a ≡ opt {
        a = get_output()
      } else if a⟩b {
        b = a      /* if a ≡ '␣' we have a⟨b */
      }
    }
  }
```

This code is used in section 337.

339. ⟨ Output saved *indent* or *outdent* tokens 339 ⟩ ≡

```
for ; c⟨0; c−− {
  out_str("\\1")
}
for ; c⟨0; c++ {
  out_str("\\2")
}
```

This code is used in sections 334 and 338.

340. The remaining part of *make_output* is somewhat more complicated. When we output a section name, we may need to enter the parsing and translation routines, since the name may contain Go code embedded in | ... | constructions. This Go code is placed at the end of the active input buffer and the translation process uses the end of the active *tok_mem* area.

⟨ Output a section name 340 ⟩ ≡

```
{
  out_str("\\X")
  cur_xref = name_dir[cur_name].xref
  if xmem[cur_xref].num ≡ file_flag {
    an_output = true
    cur_xref = xmem[cur_xref].xlink
  } else {
    an_output = false
  }
  if xmem[cur_xref].num ≥ def_flag {
    out_str(section_str(xmem[cur_xref].num − def_flag))
    if phase ≡ 3 {
      cur_xref = xmem[cur_xref].xlink
      for xmem[cur_xref].num ≥ def_flag {
        out_str(",␣")
        out_str(section_str(xmem[cur_xref].num − def_flag))
        cur_xref = xmem[cur_xref].xlink
      }
    }
  } else {
    out('0')      /* output the section number, or zero if it was undefined */
  }
  out(':')
  if an_output {
    out_str("\\.{")
  }
  ⟨ Output the text of the section name 341 ⟩
  if an_output {
    out_str("␣}")
  }
  out_str("\\X")
}
```

This code is used in section 334.

341. ⟨ Output the text of the section name 341 ⟩ ≡

```
scratch, _ := get_section_name(cur_name)
cur_section_name := cur_name
for i := 0; i⟨len(scratch); {
  b = scratch[i]
  i++
  if b ≡ '@' {
    ⟨ Skip next character, give error if not '@' 342 ⟩
  }
  if an_output {
    switch b {
      case '␣', '\\', '#', '%', '$', '^', '{', '}', '~', '&', '_':
        out('\\')
        fallthrough
      default:
        out(b)
    }
  } else if b ≠ '|' {
    out(b)
  } else {
    var buf []rune
    ⟨ Copy the Go text into the buffer array 343 ⟩
    save_buf := buffer
    save_loc := loc
    buf = append(buf, '|')
    buffer = buf
    loc = 0
    output_Go()
    loc = save_loc
    buffer = save_buf
  }
}
```

This code is used in section 340.

342. ⟨ Skip next character, give error if not '@' 342 ⟩ ≡

```
ii := i
i++
if ii⟨len(scratch) ∧ scratch[ii] ≠ '@' {
  err_print("!␣Illegal␣control␣code␣in␣section␣name:␣<%s>",
      sprint_section_name(cur_section_name))
}
```

This code is used in section 341.

343. The Go text enclosed in `|...|` should not contain '`|`' characters, except within strings. We put a '`|`' at the front of the buffer, so that an error message that displays the whole buffer will look a little bit sensible. The variable *delim* is zero outside of strings, otherwise it equals the delimiter that began the string being copied.

```
⟨ Copy the Go text into the buffer array 343 ⟩ ≡
  var delim rune
  for{
    if i ≥ len(scratch) {
      err_print("!␣Go␣text␣in␣section␣name␣didn't␣end:␣<%s>",
          sprint_section_name(cur_section_name))
      break
    }
    b = scratch[i]
    i++
    if b ≡ '@' ∨ b ≡ '\\' ∧ delim ≠ 0 {
      ⟨ Copy a quoted character into the buf 344 ⟩
    } else {
      if b ≡ '\'' ∨ b ≡ '"' {
        if delim ≡ 0 {
          delim = b
        } else if delim ≡ b {
          delim = 0
        }
      }
      if b ≠ '|' ∨ delim ≠ 0 {
        buf = append(buf, b)
      } else {
        break
      }
    }
  }
```

This code is used in section 341.

344. ⟨ Copy a quoted character into the *buf* 344 ⟩ ≡

```
  {
    buf = append(buf, b)
    buf = append(buf, scratch[i])
    i++
  }
```

This code is used in section 343.

345. Phase two processing. We have assembled enough pieces of the puzzle in order to be ready to specify the processing in GOWEAVE's main pass over the source file. Phase two is analogous to phase one, except that more work is involved because we must actually output the TEX material instead of merely looking at the CWEB specifications.

```
func phase_two() { reset_input()
if show_progress() {
  fmt.Print("\nWriting␣the␣output␣file...")
}
section_count = 0
format_visible = true
copy_limbo()
finish_line()
flush_buffer(0, false, false)      /* insert a blank line, it looks nice */
for ¬input_has_ended{⟨ Translate the current section 348 ⟩}
}
```

346. The output file will contain the control sequence \Y between non-null sections of a section, e.g., between the TEX and definition parts if both are nonempty. This puts a little white space between the parts when they are printed. However, we don't want \Y to occur between two definitions within a single section. The variables *out_line* or *out_ptr* will change if a section is non-null, so the following functions '*save_position*' and '*emit_space_if_needed*' are able to handle the situation:

```
func save_position(){
  save_line = out_line
  save_place = out_ptr
}
func emit_space_if_needed(){
  if save_line ≠ out_line ∨ save_place ≠ out_ptr {
    out_str("\\Y")
  }
  space_checked = true
}
```

347. ⟨ Global variables 93 ⟩ +≡

```
var save_line int      /* former value of out_line */
var save_place int32   /* former value of out_ptr */
var sec_depth int32    /* the integer, if any, following @* */
var space_checked bool     /* have we done emit_space_if_needed? */
var format_visible bool    /* should the next format declaration be output? */
var doing_format bool = false      /* are we outputting a format declaration? */
var group_found bool = false       /* has a starred section occurred? */
```

348. ⟨ Translate the current section 348 ⟩ ≡

```
{
  section_count++
  ⟨ Output the code for the beginning of a new section 349 ⟩
  save_position()
  ⟨ Translate the TEX part of the current section 350 ⟩
  ⟨ Translate the definition part of the current section 351 ⟩
  ⟨ Translate the Go part of the current section 355 ⟩
  ⟨ Show cross-references to this section 358 ⟩
  ⟨ Output the code for the end of a section 361 ⟩
}
```

This code is used in section 345.

349. Sections beginning with the CWEB control sequence '@␣' start in the output with the TEX control sequence '\M', followed by the section number. Similarly, '@*' sections lead to the control sequence '\N'. In this case there's an additional parameter, representing one plus the specified depth, immediately after the \N. If the section has changed, we put * just after the section number.

⟨ Output the code for the beginning of a new section 349 ⟩ ≡

```
if loc − 1 ≥ len(buffer) ∨ buffer[loc − 1] ≠ '*' {
  out_str("\\M")
} else {
  for loc⟨len(buffer) ∧ buffer[loc] ≡ '␣' {
    loc++
  }
  if loc⟨len(buffer) ∧ buffer[loc] ≡ '*' {      /* "top" level */
    sec_depth = −1
    loc++
  } else {
    for sec_depth = 0; loc⟨len(buffer) ∧ unicode.IsDigit(buffer[loc]); loc++ {
      sec_depth = sec_depth * 10 + buffer[loc] − '0'
    }
  }
  for loc⟨len(buffer) ∧ buffer[loc] ≡ '␣' {
    loc++      /* remove spaces before group title */
  }
  group_found = true
  out_str("\\N")
  {
    s := fmt.Sprintf("{%d}", sec_depth + 1)
    out_str(s)
  }
  if show_progress() {
    fmt.Printf("*%d", section_count)
  }
  os.Stdout.Sync()      /* print a progress report */
}
out_str("{")
out_str(section_str(section_count))
out_str("}")
```

This code is used in section 348.

350. In the TEX part of a section, we simply copy the source text, except that index entries are not copied and Go text within | ... | is translated.

```
⟨ Translate the TEX part of the current section 350 ⟩ ≡
  for{
    next_control = copy_TEX()
    switch next_control {
      case '|':
        init_stack()
        output_Go()
      case '@':
        out('@')
      case TEX_string, raw_TEX_string, noop, xref_roman, xref_wildcard, xref_typewriter, section_name:
        loc −= 2
        next_control = get_next()      /* skip to @> */
        if next_control ≡ TEX_string ∨ next_control ≡ raw_TEX_string {
          err_print("! TeX string should be in Go text only")
        }
      case thin_space, math_break, ord, line_break, big_line_break, no_line_break, join, pseudo_semi:
        err_print("! You can't do that in TeX text")
    }
    if next_control ≥ format_code {
      break
    }
  }
```

This code is used in section 348.

351. When we get to the following code we have *next_control* ≥ *format_code*, and the token memory is in its initial empty state.

```
⟨ Translate the definition part of the current section 351 ⟩ ≡
  space_checked = false
  for next_control ≤ format_code {      /* format_code or definition */
    init_stack()
    ⟨ Start a format definition 353 ⟩
    outer_parse()
    finish_Go(format_visible)
    format_visible = true
    doing_format = false
  }
```

This code is used in section 348.

352. The *finish_Go* procedure outputs the translation of the current scraps, preceded by the control sequence '`\B`' and followed by the control sequence '`\par`'. It also restores the token and scrap memories to their initial empty state.

A *force* token is appended to the current scraps before translation takes place, so that the translation will normally end with `\6` or `\7` (the TEX macros for *force* and *big_force*). This `\6` or `\7` is replaced by the concluding `\par` or by `\Y\par`.

```
    /* finishes a definition or a Go part */
func finish_Go(
  visible bool      /* visible is nonzero if we should produce TEX output */){
  if visible {
    out_str("\\B")
    app_scrap(insert, no_math, force)
    p := translate()      /* translation of the scraps */
    scrap_info = nil
    make_output(list_token(p))      /* output the list */
    if out_ptr⟩1 {
      if out_buf[out_ptr − 1] ≡ '\\' {
        if out_buf[out_ptr] ≡ '6' {
          out_ptr −= 2
        } else if out_buf[out_ptr] ≡ '7' {
          out_buf[out_ptr] = 'Y'
        }
      }
    }
    out_str("\\par")
    finish_line()
  }
}
```

353. ⟨ Start a format definition 353 ⟩ ≡

```
{
  doing_format = true
  if buffer[loc − 1] ≡ 's' ∨ buffer[loc − 1] ≡ 'S' {
    format_visible = false
  }
  if ¬space_checked {
    emit_space_if_needed()
    save_position()
  }
  tok_mem := append([]interface{}{}, "\\F")      /* this will produce 'format ' */
  next_control = get_next()
  if next_control ≡ identifier {
    tok_mem = append(tok_mem, id_token(id_lookup(id, normal)), '␣', break_space)
      /* this is syntactically separate from what follows */
    next_control = get_next()
    if next_control ≡ identifier {
      tok_mem = append(tok_mem, id_token(id_lookup(id, normal)))
      app_scrap(Expression, maybe_math, tok_mem ...)
      app_scrap(semi, maybe_math)
      next_control = get_next()
    }
  }
  if len(scrap_info) ≠ 2 {
    err_print("!␣Improper␣format␣definition")
  }
}
```

This code is used in section 351.

354. Finally, when the TEX and definition parts have been treated, we have *next_control* ≥ *begin_code*. We will make the global variable *this_section* point to the current section name, if it has a name.

⟨ Global variables 93 ⟩ +≡

```
var this_section int32      /* the current section name, or zero */
```

355. ⟨ Translate the Go part of the current section 355 ⟩ ≡

```
this_section = −1
if next_control ≤ section_name {
  emit_space_if_needed()
  init_stack()
  if next_control ≡ begin_code {
    next_control = get_next()
  } else {
    this_section = cur_section
    ⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section
        definition 356 ⟩
  }
  for next_control ≤ section_name {
    outer_parse()
    ⟨ Emit the scrap for a section name if present 357 ⟩
  }
  finish_Go(true)
}
```

This code is used in section 348.

356. The title of the section and an ≡ or +≡ are made into a scrap that should not take part in the parsing.

```
⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section definition 356 ⟩ ≡
  for{
    next_control = get_next()
    if next_control ≠ '+' {
      break
    }
  }     /* allow optional '+=' */
  var tok_mem []interface{}
  if next_control ≠ '=' ∧ next_control ≠ eq_eq {
    err_print("! You need an = sign after the section name")
  } else {
    next_control = get_next()
  }
  if out_ptr⟩1 ∧ out_buf[out_ptr] ≡ 'Y' ∧ out_buf[out_ptr − 1] ≡ '\\' {
    tok_mem = append(tok_mem, backup)
  }
      /* the section name will be flush left */
  tok_mem = append(tok_mem, section_token(this_section))
  cur_xref = name_dir[this_section].xref
  if xmem[cur_xref].num ≡ file_flag {
    cur_xref = xmem[cur_xref].xlink
  }
  tok_mem = append(tok_mem, "${}")
  if xmem[cur_xref].num ≠ section_count + def_flag {
    tok_mem = append(tok_mem, "\\mathrel+")      /*section name is multiply defined*/
    this_section = −1      /*so we won't give cross-reference info here*/
  }
  tok_mem = append(tok_mem, "\\E", "{}$", force)      /* output an equivalence sign */
  app_scrap(dead, no_math, tok_mem ...)
      /* this forces a line break unless '@+' follows */
```

This code is used in section 355.

357. ⟨ Emit the scrap for a section name if present 357 ⟩ ≡

```
  if next_control⟨section_name {
    err_print("! You can't do that in Go text")
    next_control = get_next()
  } else if next_control ≡ section_name {
    app_scrap(section_scrap, maybe_math, section_token(cur_section))
    next_control = get_next()
  }
```

This code is used in section 355.

358. Cross references relating to a named section are given after the section ends.

```
⟨ Show cross-references to this section 358 ⟩ ≡
  if this_section ≠ −1 {
    cur_xref = name_dir[this_section].xref
    if xmem[cur_xref].num ≡ file_flag {
      an_output = true
      cur_xref = xmem[cur_xref].xlink
    } else {
      an_output = false
    }
    if xmem[cur_xref].num⟩def_flag {
      cur_xref = xmem[cur_xref].xlink      /* bypass current section number */
    }
    footnote(def_flag)
    footnote(cite_flag)
    footnote(0)
  }
```

This code is used in section 348.

359. The *footnote* procedure gives cross-reference information about multiply defined section names (if the *flag* parameter is *def_flag*), or about references to a section name (if *flag* ≡ *cite_flag*), or to its uses (if *flag* ≡ 0). It assumes that *cur_xref* points to the first cross-reference entry of interest, and it leaves *cur_xref* pointing to the first element not printed. Typical outputs: '`\A101.`'; '`\Us 370\ET1009.`'; '`\As 8, 27\*\ETs64.`'.

Note that the output of `GOWEAVE` is not English-specific; users may supply new definitions for the macros `\A`, `\As`, etc.

```
    /* outputs section cross-references */
  func footnote(flag int32){
    if xmem[cur_xref].num ≤ flag {
      return
    }
    finish_line()
    out('\\')
    switch flag {
      case 0:
        out('U')
      case cite_flag:
        out('Q')
      default:
        out('A')
    }
    ⟨ Output all the section numbers on the reference list cur_xref 360 ⟩
    out('.')
  }
```

360. The following code distinguishes three cases, according as the number of cross-references is one, two, or more than two. Variable q points to the first cross-reference, and the last link is a zero.

```
⟨ Output all the section numbers on the reference list cur_xref 360 ⟩ ≡
  q := cur_xref      /* cross-reference pointer variable */
  if xmem[xmem[q].xlink].num⟩flag {
    out('s')      /* plural */
  }
  for{
    out_str(section_str(xmem[cur_xref].num − flag))
    cur_xref = xmem[cur_xref].xlink      /* point to the next cross-reference to output */
    if xmem[cur_xref].num ≤ flag {
      break
    }
    if xmem[xmem[cur_xref].xlink].num⟩flag {
      out_str(",␣")      /* not the last */
    } else {
      out_str("\\ET")      /* the last */
      if cur_xref ≠ xmem[q].xlink {
        out('s')      /* the last of more than two */
      }
    }
  }
```

This code is used in section 359.

361. ⟨ Output the code for the end of a section 361 ⟩ ≡

```
  out_str("\\fi")
  finish_line()
  flush_buffer(0, false, false)      /* insert a blank line, it looks nice */
```

This code is used in section 348.

362. Phase three processing. We are nearly finished! GOWEAVE's only remaining task is to write out the index, after sorting the identifiers and index entries.

If the user has set the *flags*['x'] ≡ 0 flag (the -x option on the command line), just finish off the page, omitting the index, section name list, and table of contents.

```
func phase_three(){
  if ¬flags['x'] {
    finish_line()
    out_str("\\end")
    finish_line()
  } else {
    phase = 3
    if show_progress() {
      fmt.Print("\nWriting␣the␣index...")
    }
    finish_line()
    if f, err := os.OpenFile(idx_file_name, os.O_WRONLY | os.O_CREATE | os.O_TRUNC, °666); err ≠ nil {
      fatal("!␣Cannot␣open␣index␣file␣", idx_file_name)
    } else {
      idx_file = f
    }
    if change_exists ∧ flags['c'] {
      ⟨ Tell about changed sections 364 ⟩
      finish_line()
      finish_line()
    }
    out_str("\\inx")
    finish_line()
    active_file = idx_file      /* change active file to the index file */
    ⟨ Do the first pass of sorting 366 ⟩
    ⟨ Sort and output the index 375 ⟩
    finish_line()
    active_file.Close()      /* finished with idx_file */
    active_file = tex_file      /* switch back to tex_file for a tic */
    out_str("\\fin")
    finish_line()
    if f, err := os.OpenFile(scn_file_name, os.O_WRONLY | os.O_CREATE | os.O_TRUNC, °666); err ≠ nil {
      fatal("!␣Cannot␣open␣section␣file␣", scn_file_name)
    } else {
      scn_file = f
    }
    active_file = scn_file      /* change active file to section listing file */
    ⟨ Output all the section names 384 ⟩
    finish_line()
    active_file.Close()      /* finished with scn_file */
    active_file = tex_file
    if group_found {
      out_str("\\con")
    } else {
      out_str("\\end")
    }
```

```
    finish_line()
    active_file.Close()
  }
  if show_happiness() {
    fmt.Print("\nDone.")
  }
  check_complete()      /* was all of the change file used? */
}
```

363. Just before the index comes a list of all the changed sections, including the index section itself.

364. ⟨ Tell about changed sections 364 ⟩ ≡

```
{
    /* remember that the index is already marked as changed */
  var k_section int32 = 0      /* runs through the sections */
  for k_section++; ¬changed_section[k_section]; k_section++ {}
  out_str("\\ch␣")
  out_str(section_str(k_section))
  for k_section⟨section_count {
    for k_section++; ¬changed_section[k_section]; k_section++ {}
    out_str(",␣")
    out_str(section_str(k_section))
  }
  out('.')
}
```

This code is used in section 362.

365. A left-to-right radix sorting method is used, since this makes it easy to adjust the collating sequence and since the running time will be at worst proportional to the total length of all entries in the index. We put the identifiers into 102 different lists based on their first characters. (Uppercase letters are put into the same list as the corresponding lowercase letters, since we want to have 't < *TeX* < **to**'.) The list for character c begins at location $bucket[c]$ and continues through the *blink* array.

⟨ Global variables 93 ⟩ +≡

```
var bucket [256]int32
var blink [max_names]int32      /* links in the buckets */
```

366. To begin the sorting, we go through all the hash lists and put each entry having a nonempty cross-reference list into the proper bucket.

```
⟨ Do the first pass of sorting 366 ⟩ ≡
  {
    for c := 0; c ≤ 255; c++ {
      bucket[c] = −1
    }
    for _, next_name := range hash {
      for next_name ≠ −1 {
        cur_name = next_name
        next_name = name_dir[cur_name].llink
        if name_dir[cur_name].xref ≠ 0 {
          c := name_dir[cur_name].name[0]
          if unicode.IsUpper(c) {
            c = unicode.ToLower(c)
          }
          blink[cur_name] = bucket[c]
          bucket[c] = cur_name
        }
      }
    }
  }
```

This code is used in section 362.

367. During the sorting phase we shall use the *cat* and *trans* arrays from `GOWEAVE`'s parsing algorithm and rename them *depth* and *head*. They now represent a stack of identifier lists for all the index entries that have not yet been output. The variable *sort_ptr* tells how many such lists are present; the lists are output in reverse order (first *sort_ptr*, then *sort_ptr* − 1, etc.). The jth list starts at *head*$[j]$, and if the first k characters of all entries on this list are known to be equal we have *depth*$[j] \equiv k$.

368.
```
⟨ Rest of scrap struct 368 ⟩ ≡
  head int32
```

This code is used in section 174.

369.
```
⟨ Global variables 93 ⟩ +≡
  var cur_depth int32      /* depth of current buckets */
  var cur_byte int32       /* index into byte_mem */
  var cur_val int32        /* current cross-reference number */
  var max_sort_ptr int32   /* largest value of sort_ptr */
  var sort_ptr int32       /* ditto */
```

370.
```
⟨ Set initial values 99 ⟩ +≡
  max_sort_ptr = 0
```

371. The desired alphabetic order is specified by the *collate* array; namely, $collate[0] < collate[1] < \cdots < collate[100]$.

```
⟨ Global variables 93 ⟩ +≡
    /* collation order */
  var collate = [102 + 128]rune{0, '␣', °01, °02, °03, °04, °05, °06, °07, °10, °11, °12, °13, °14, °15, °16,
    °17, °20, °21, °22, °23, °24, °25, °26, °27, °30, °31, °32, °33, °34, °35, °36, °37, '!', °42, '#', '$',
    '%', '&', '\'', '(', ')', '*', '+', ',', '-', '.', '/', ':', ';', '<', '=', '>', '?', '@', '[', '\\',
    ']', '^', '`', '{', '|', '}', '~', '_', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
    'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', °200, °201, °202, °203, °204, °205, °206, °207, °210, °211, °212, °213, °214, °215, °216,
    °217, °220, °221, °222, °223, °224, °225, °226, °227, °230, °231, °232, °233, °234, °235, °236, °237,
    °240, °241, °242, °243, °244, °245, °246, °247, °250, °251, °252, °253, °254, °255, °256, °257, °260,
    °261, °262, °263, °264, °265, °266, °267, °270, °271, °272, °273, °274, °275, °276, °277, °300, °301,
    °302, °303, °304, °305, °306, °307, °310, °311, °312, °313, °314, °315, °316, °317, °320, °321, °322,
    °323, °324, °325, °326, °327, °330, °331, °332, °333, °334, °335, °336, °337, °340, °341, °342, °343,
    °344, °345, °346, °347, °350, °351, °352, °353, °354, °355, °356, °357, °360, °361, °362, °363, °364,
    °365, °366, °367, °370, °371, °372, °373, °374, °375, °376, °377}
```

372. We use the order null $<$ ␣ $<$ other characters $<$ `_` $<$ `A` $=$ `a` $< \cdots <$ `Z` $=$ `z` $<$ `0` $< \cdots <$ `9`. Warning: The collation mapping needs to be changed if ASCII code is not being used.

We initialize *collate* by copying a few characters at a time, because some Go compilers choke on long strings.

373. Procedure *unbucket* goes through the buckets and adds nonempty lists to the stack, using the collating sequence specified in the *collate* array. The parameter to *unbucket* tells the current depth in the buckets. Any two sequences that agree in their first 255 character positions are regarded as identical.

```
⟨ Constants 1 ⟩ +≡
  const infinity = −1    /* ∞ (approximately) */
```

374.

```
    /* empties buckets having depth d */
  func unbucket(d int32){
    /* index into bucket; cannot be a simple char because of sign comparison below */
    for c := 100 + 128; c ≥ 0; c−− {
      if bucket[collate[c]] ≠ −1 {
        sort_ptr++
        scrap_info = append(scrap_info, scrap{})
        if sort_ptr⟩max_sort_ptr {
          max_sort_ptr = sort_ptr
        }
        if c ≡ 0 {
          scrap_info[sort_ptr].cat = infinity
        } else {
          scrap_info[sort_ptr].cat = d
        }
        scrap_info[sort_ptr].head = bucket[collate[c]]
        bucket[collate[c]] = −1
      }
    }
  }
```

375. ⟨ Sort and output the index 375 ⟩ ≡

```
sort_ptr = 0
scrap_info = append(scrap_info, scrap{})
unbucket(1)
for sort_ptr⟩0 { cur_depth = scrap_info[sort_ptr].cat
if blink[scrap_info[sort_ptr].head] ≡ −1 ∨ cur_depth ≡ infinity{⟨ Output index entries for the list at
        sort_ptr 377 ⟩} else {
  ⟨ Split the list at sort_ptr into further lists 376 ⟩
}
}
```

This code is used in section 362.

376. ⟨ Split the list at *sort_ptr* into further lists 376 ⟩ ≡

```
{
  next_name := scrap_info[sort_ptr].head
  for{
    var c rune
    cur_name = next_name
    next_name = blink[cur_name]
    cur_byte = cur_depth
    if cur_byte ≥ int32(len(name_dir[cur_name].name)) {
      c = 0      /* hit end of the name */
    } else {
      c = name_dir[cur_name].name[cur_byte]
      if unicode.IsUpper(c) {
        c = unicode.ToLower(c)
      }
    }
    blink[cur_name] = bucket[c]
    bucket[c] = cur_name
    if next_name ≡ −1 {
      break
    }
  }
  sort_ptr−−
  unbucket(cur_depth + 1)
}
```

This code is used in section 375.

377. ⟨ Output index entries for the list at *sort_ptr* 377 ⟩ ≡

```
{
  cur_name = scrap_info[sort_ptr].head
  for{
    out_str("\\I")
    ⟨ Output the name at cur_name 378 ⟩
    ⟨ Output the cross-references at cur_name 380 ⟩
    cur_name = blink[cur_name]
    if cur_name ≡ −1 {
      break
    }
  }
  sort_ptr −−
}
```

This code is used in section 375.

378. ⟨ Output the name at *cur_name* 378 ⟩ ≡

```
switch name_dir[cur_name].ilk {
  case normal:
    if is_tiny(cur_name) {
      out_str("\\|")
    } else {
      lowcase := false
      for _, v := range name_dir[cur_name].name {
        if unicode.IsLower(v) {
          lowcase = true
          break
        }
      }
      if ¬lowcase {
        out_str("\\.")
      } else {
        out_str("\\\\")
      }
    }
  case wildcard:
    out_str("\\9")
    out_name(cur_name, false)
    goto name_done
  case typewriter:
    out_str("\\.")
    fallthrough
  case roman:
    out_name(cur_name, false)
    goto name_done
  case custom:
    {
      out_str("$\\")
      for _, v := range name_dir[cur_name].name {
        if v ≡ '_' {
          out('x')
        } else {
          out(v)
        }
      }
      out('$')
      goto name_done
    }
  default:
    out_str("\\&")
}
out_name(cur_name, true)
name_done :
```

This code is used in section 377.

379.

380. Section numbers that are to be underlined are enclosed in '`\[...]`'.

```
⟨ Output the cross-references at cur_name 380 ⟩ ≡
  ⟨ Invert the cross-reference list at cur_name, making cur_xref the head 382 ⟩
  for{
    out_str(",␣")
    cur_val = xmem[cur_xref].num
    if cur_val⟨def_flag {
      out_str(section_str(cur_val))
    } else {
      out_str("\\[")
      out_str(section_str(cur_val − def_flag))
      out(']')
    }
    cur_xref = xmem[cur_xref].xlink
    if cur_xref ≡ 0 {
      break
    }
  }
  out('.')
  finish_line()
```

This code is used in section 377.

381. List inversion is best thought of as popping elements off one stack and pushing them onto another. In this case *cur_xref* will be the head of the stack that we push things onto.

```
⟨ Global variables 93 ⟩ +≡
  var next_xref int32
  var this_xref int32
      /* pointer variables for rearranging a list */
```

382.

```
⟨ Invert the cross-reference list at cur_name, making cur_xref the head 382 ⟩ ≡
  this_xref = name_dir[cur_name].xref
  cur_xref = 0
  for{
    next_xref = xmem[this_xref].xlink
    xmem[this_xref].xlink = cur_xref
    cur_xref = this_xref
    this_xref = next_xref
    if this_xref ≡ 0 {
      break
    }
  }
```

This code is used in section 380.

383. The following recursive procedure walks through the tree of section names and prints them.

```
    /* print all section names in subtree p */
  func section_print(p int32){
    if p ≠ −1 {
      section_print(name_dir[p].llink)
      out_str("\\I")
      init_stack()
      make_output(section_token(p))
      footnote(cite_flag)
      footnote(0)      /* cur_xref was set by make_output */
      finish_line()
      section_print(name_dir[p].rlink)
    }
  }
```

384. ⟨ Output all the section names 384 ⟩ ≡

```
section_print(name_root)
```

This code is used in section 362.

385. Because on some systems the difference between two pointers is a *long* rather than an **int**, we use `%ld` to print these quantities.

```
  func print_stats(){
    fmt.Print("\nMemory␣usage␣statistics:\n")
    fmt.Printf("%v␣names\n", len(name_dir))
    fmt.Println("Sorting:")
    fmt.Printf("%v␣levels\n", max_sort_ptr)
  }
```

386. ⟨ Print usage error message and quit 386 ⟩ ≡

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

This code is used in section 83.

387. `GOWEAVE` specific creation of output file

⟨ Try to open output file 387 ⟩ ≡

```
if f, err := os.OpenFile(tex_file_name, os.O_WRONLY | os.O_CREATE | os.O_TRUNC, °666); err ≠ nil {
  fatal("!␣Cannot␣open␣output␣file␣", tex_file_name)
} else {
  tex_file = f
}
```

This code is used in section 89.

388. Index. If you have read and understood the code for Phase III above, you know what is in this index and how it got here. 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, control sequences put into the output, and a few other things like "recursion" are indexed here too.

⟨ Append a TEX string 317 ⟩ Used in section 314.
⟨ Append a raw TEX string 318 ⟩ Used in section 314.
⟨ Append a string or constant 316 ⟩ Used in section 314.
⟨ Append the scrap appropriate to *next_control* 314 ⟩ Used in section 312.
⟨ Cases for *AnonymousField* 214 ⟩ Used in section 181.
⟨ Cases for *ArrayType* 216 ⟩ Used in section 181.
⟨ Cases for *Assignment* 283 ⟩ Used in section 181.
⟨ Cases for *Block* 243 ⟩ Used in section 181.
⟨ Cases for *BreakStmt* 253 ⟩ Used in section 181.
⟨ Cases for *BuiltinArgs* 293 ⟩ Used in section 181.
⟨ Cases for *BuiltinCall* 292 ⟩ Used in section 181.
⟨ Cases for *Call* 298 ⟩ Used in section 181.
⟨ Cases for *ChannelType* 228 ⟩ Used in section 181.
⟨ Cases for *CommCase* 270 ⟩ Used in section 181.
⟨ Cases for *CommClause* 269 ⟩ Used in sections 181 and 186.
⟨ Cases for *CompositeLit* 236 ⟩ Used in section 181.
⟨ Cases for *ConstDecl* 195 ⟩ Used in section 181.
⟨ Cases for *ConstSpec* 208 ⟩ Used in sections 181 and 186.
⟨ Cases for *ContinueStmt* 255 ⟩ Used in section 181.
⟨ Cases for *Conversion* 291 ⟩ Used in section 181.
⟨ Cases for *DeferStmt* 279 ⟩ Used in section 181.
⟨ Cases for *ElementList* 239 ⟩ Used in sections 181 and 186.
⟨ Cases for *Element* 240 ⟩ Used in section 181.
⟨ Cases for *ExprCaseClause* 262 ⟩ Used in sections 181 and 186.
⟨ Cases for *ExprSwitchStmt* 261 ⟩ Used in section 181.
⟨ Cases for *ExpressionList* 230 ⟩ Used in section 181.
⟨ Cases for *Expression* 231 ⟩ Used in section 181.
⟨ Cases for *FieldDecl* 213 ⟩ Used in sections 181 and 186.
⟨ Cases for *ForClause* 276 ⟩ Used in section 181.
⟨ Cases for *ForStmt* 275 ⟩ Used in section 181.
⟨ Cases for *FunctionDecl* 203 ⟩ Used in sections 181 and 186.
⟨ Cases for *FunctionLit* 241 ⟩ Used in section 181.
⟨ Cases for *FunctionType* 242 ⟩ Used in sections 181 and 186.
⟨ Cases for *GoStmt* 249 ⟩ Used in section 181.
⟨ Cases for *GotoStmt* 257 ⟩ Used in section 181.
⟨ Cases for *IdentifierList* 229 ⟩ Used in section 181.
⟨ Cases for *IfStmt* 259 ⟩ Used in section 181.
⟨ Cases for *ImportDecl* 201 ⟩ Used in section 186.
⟨ Cases for *ImportSpec* 212 ⟩ Used in sections 181 and 186.
⟨ Cases for *IncDecStmt* 281 ⟩ Used in section 181.
⟨ Cases for *Index* 295 ⟩ Used in section 181.
⟨ Cases for *InterfaceType* 224 ⟩ Used in sections 181 and 186.
⟨ Cases for *LabeledStmt* 246 ⟩ Used in section 181.
⟨ Cases for *LiteralType* 237 ⟩ Used in section 181.
⟨ Cases for *LiteralValue* 238 ⟩ Used in section 181.
⟨ Cases for *MapType* 227 ⟩ Used in section 181.
⟨ Cases for *MethodDecl* 205 ⟩ Used in sections 181 and 186.
⟨ Cases for *MethodExpr* 289 ⟩ Used in section 181.
⟨ Cases for *MethodSpec* 225 ⟩ Used in section 181.
⟨ Cases for *Operand* 235 ⟩ Used in section 181.
⟨ Cases for *PackageClause* 193 ⟩ Used in section 186.
⟨ Cases for *ParameterDecl* 223 ⟩ Used in section 181.

⟨ Cases for *ParameterList* 222 ⟩ Used in section 181.
⟨ Cases for *Parameters* 221 ⟩ Used in section 181.
⟨ Cases for *PointerType* 219 ⟩ Used in section 181.
⟨ Cases for *PrimaryExpr* 234 ⟩ Used in section 181.
⟨ Cases for *QualifiedIdent* 288 ⟩ Used in section 181.
⟨ Cases for *RangeClause* 277 ⟩ Used in section 181.
⟨ Cases for *ReceiverType* 290 ⟩ Used in section 181.
⟨ Cases for *Receiver* 207 ⟩ Used in section 181.
⟨ Cases for *RecvStmt* 271 ⟩ Used in section 181.
⟨ Cases for *ReturnStmt* 251 ⟩ Used in section 181.
⟨ Cases for *SelectStmt* 268 ⟩ Used in section 181.
⟨ Cases for *Selector* 294 ⟩ Used in section 181.
⟨ Cases for *SendStmt* 272 ⟩ Used in section 181.
⟨ Cases for *ShortVarDecl* 286 ⟩ Used in section 181.
⟨ Cases for *Signature* 220 ⟩ Used in section 181.
⟨ Cases for *SimpleStmt* 248 ⟩ Used in section 181.
⟨ Cases for *SliceType* 226 ⟩ Used in section 181.
⟨ Cases for *Slice* 296 ⟩ Used in section 181.
⟨ Cases for *Statement* 245 ⟩ Used in sections 181 and 186.
⟨ Cases for *StructType* 217 ⟩ Used in sections 181 and 186.
⟨ Cases for *TypeAssertion* 297 ⟩ Used in section 181.
⟨ Cases for *TypeCaseClause* 265 ⟩ Used in sections 181 and 186.
⟨ Cases for *TypeDecl* 197 ⟩ Used in section 181.
⟨ Cases for *TypeSpec* 209 ⟩ Used in sections 181 and 186.
⟨ Cases for *TypeSwitchCase* 266 ⟩ Used in section 181.
⟨ Cases for *TypeSwitchGuard* 264 ⟩ Used in section 181.
⟨ Cases for *TypeSwitchStmt* 263 ⟩ Used in section 181.
⟨ Cases for *Type* 215 ⟩ Used in section 181.
⟨ Cases for *UnaryExpr* 232 ⟩ Used in section 181.
⟨ Cases for *VarDecl* 199 ⟩ Used in section 181.
⟨ Cases for *VarSpec* 210 ⟩ Used in sections 181 and 186.
⟨ Cases for *assign_op* 285 ⟩ Used in section 181.
⟨ Cases for *binary_op* 233 ⟩ Used in section 181.
⟨ Cases for *unary_op* 299 ⟩ Used in section 181.
⟨ Cases involving nonstandard characters 315 ⟩ Used in section 314.
⟨ Check for end of comment 164 ⟩ Used in section 163.
⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section definition 356 ⟩ Used in section 355.
⟨ Clear *bal* and **return** 166 ⟩ Used in section 163.
⟨ Combine the irreducible scraps that remain 308 ⟩ Used in section 307.
⟨ Common constants 10, 31, 42, 55, 63, 65 ⟩ Used in section 6.
⟨ Compress two-symbol operator 91 ⟩ Used in section 123.
⟨ Compute the hash code h 46 ⟩ Used in section 45.
⟨ Compute the name location p 47 ⟩ Used in section 45.
⟨ Constants 1, 4, 97, 112, 121, 168, 172, 301, 324, 331, 373 ⟩ Used in section 2.
⟨ Copy a quoted character into the *buf* 344 ⟩ Used in section 343.
⟨ Copy special things when $c \equiv$ '@', '\\' 165 ⟩ Used in section 163.
⟨ Copy the Go text into the *buffer* array 343 ⟩ Used in section 341.
⟨ Custom out 336 ⟩ Used in section 335.
⟨ Definitions that should agree with GOTANGLE and GOWEAVE 12, 17, 32, 40, 43, 68, 81, 88 ⟩ Used in section 6.
⟨ Do the first pass of sorting 366 ⟩ Used in section 362.
⟨ Emit the scrap for a section name if present 357 ⟩ Used in section 355.

⟨ Enter a new name into the table at position p 49 ⟩ Used in section 45.
⟨ Get a constant 126 ⟩ Used in section 123.
⟨ Get a string 127 ⟩ Used in sections 123 and 128.
⟨ Get an identifier 125 ⟩ Used in section 123.
⟨ Get control code and possible section name 128 ⟩ Used in section 123.
⟨ Global variables 93, 95, 115, 122, 135, 142, 147, 150, 169, 175, 179, 305, 327, 330, 347, 354, 365, 369, 371, 381 ⟩ Used in section 2.
⟨ Handle flag argument 87 ⟩ Used in section 83.
⟨ If end of name or erroneous nesting, **break** 132 ⟩ Used in section 131.
⟨ 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 semi-tracing, show the irreducible scraps 309 ⟩ Used in section 308.
⟨ If the current line starts with `@y`, report any discrepancies and **return** 28 ⟩ Used in section 26.
⟨ If tracing, print an indication of where we are 310 ⟩ Used in section 307.
⟨ Import packages 13, 16, 20, 27, 34 ⟩ Used in section 2.
⟨ Initialization of a new identifier 108 ⟩ Used in section 49.
⟨ Initialize pointers 44, 51 ⟩ Used in section 8.
⟨ Insert new cross-reference at q, not at beginning of list 191 ⟩ Used in section 190.
⟨ Invert the cross-reference list at *cur_name*, making *cur_xref* the head 382 ⟩ Used in section 380.
⟨ Look ahead for strongest line break, **goto** *reswitch* 338 ⟩ Used in section 337.
⟨ 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.
⟨ Making translation for an element v of scrap sequence 300 ⟩ Used in section 302.
⟨ Match a production at pp, or increase pp if there is no match 186 ⟩ Used in section 304.
⟨ More elements of *name_info* structure 41, 50, 92, 98 ⟩ 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.
⟨ Output a control, look ahead in case of line breaks, possibly **goto** *reswitch* 337 ⟩ Used in section 334.
⟨ Output a section name 340 ⟩ Used in section 334.
⟨ Output all the section names 384 ⟩ Used in section 362.
⟨ Output all the section numbers on the reference list *cur_xref* 360 ⟩ Used in section 359.
⟨ Output an identifier 335 ⟩ Used in section 334.
⟨ Output index entries for the list at *sort_ptr* 377 ⟩ Used in section 375.
⟨ Output saved *indent* or *outdent* tokens 339 ⟩ Used in sections 334 and 338.
⟨ Output the code for the beginning of a new section 349 ⟩ Used in section 348.
⟨ Output the code for the end of a section 361 ⟩ Used in section 348.
⟨ Output the cross-references at *cur_name* 380 ⟩ Used in section 377.
⟨ Output the name at *cur_name* 378 ⟩ Used in section 377.
⟨ Output the text of the section name 341 ⟩ Used in section 340.
⟨ Override *tex_file_name* and *go_file_name* 86 ⟩ Used in section 83.
⟨ Print a snapshot of the scrap list if debugging 306 ⟩ Used in sections 181 and 302.
⟨ Print error location based on input buffer 71 ⟩ Used in section 69.
⟨ Print error messages about unused or undefined section names 149 ⟩ Used in section 136.
⟨ Print the job *history* 74 ⟩ Used in section 73.
⟨ Print usage error message and quit 386 ⟩ Used in section 83.
⟨ Print warning message, break the line, **return** 158 ⟩ Used in section 157.
⟨ Process a format definition 144 ⟩ Used in section 143.
⟨ Process simple format in limbo 145 ⟩ Used in section 118.
⟨ Put section name into *section_text* 131 ⟩ Used in section 129.

The GOWEAVE processor

(Version 0.82)