```
// Compile Ninth FORTH into 6809 OS-9 Assembly Module.
package main

import "io"
import "io/ioutil"
import "fmt"
import "os"
import "strconv"
import "strings"

var F = fmt.Sprintf

func P(format string, args ...interface{}) {
	fmt.Printf(format+"\n", args...)
}

type Ninth struct {
	Lines []string
	L     int

	Words []string
	W     int

	Latest string
	Here   int

	Allots map[string]int
	IfStack  []string
	LoopStack  []string
	Serial int
}

func (o *Ninth) NextWord() string {
	for o.W >= len(o.Words) {
		t := o.NextLine()
		if t == ">EOF<" {
			return t
		}
		o.Words = []string{}
		for _, w := range strings.Split(t, " ") {
			if w != "" {
				o.Words = append(o.Words, w)
			}
		}
		o.W = 0
	}
	z := o.Words[o.W]
	o.W++
	return z
}

func (o *Ninth) NextLine() string {
	o.Words = nil
	o.W = 0
	if o.L >= len(o.Lines) {
		return ">EOF<"
	}
	z := o.Lines[o.L]
	o.L++
	return strings.Replace(z, "\t", "        ", -1)
}
```

```go
func NewNinth(r io.Reader) *Ninth {
	all, err := ioutil.ReadAll(r)
	if err != nil {
		panic("can't ioutil.ReadAll")
	}
	lines := strings.Split(string(all), "\n")
	return &Ninth{
		Lines:  lines,
		Allots: make(map[string]int),
	}
}

func EncodeFunnyChars(s string) string {
	var bb []byte
	for _, ch := range s {
		if '0' <= ch && ch <= '9' ||
			'A' <= ch && ch <= 'Z' ||
			'a' <= ch && ch <= 'z' ||
			ch == '_' {
			bb = append(bb, byte(ch))
		} else {
			bb = append(bb, []byte(F("_%02x", ch))...)
		}
	}
	return string(bb)
}

func (o *Ninth) DoPrelude(name string, code string) {
	ename := EncodeFunnyChars(name)
	ecode := EncodeFunnyChars(code)
	elatest := EncodeFunnyChars(o.Latest)
	P("\n\n***  %s  ***\n", name)
	P("l_%s", ename)
	if o.Latest == "" {
		P("  fcb 0,0 ;link")
	} else {
		P("  fcb ($10000+l_%s-*)/256 ;link", elatest)
		P("  fcb ($10000+l_%s-*)+1", elatest)
	}
	P("  fcb %d ;len", len(name)) // For going forwards >CFA
	P("  fcc ~%s~", name)
	P("  fcb 0")                  // NUL terminate C-style.
	P("  fcb %d ;len", len(name)) // For going backwards

	P("c_%s", ename)
	P("  fcb ($10000+%s-*)/256 ;codeword", ecode)
	P("  fcb ($10000+%s-*)+1", ecode)
	P("d_%s", ename)

	o.Latest = name
}

func (o *Ninth) InsertAllot(offset int) {
	P("  tfr dp,a")
	P("  clrb")
	P("  addd #%d", offset)
	P("  pshU d")
	P("  jmp Next,pcr")
}

func (o *Ninth) InsertCode() {
	for {
```

```
            s := o.NextLine()
            if strings.Trim(s, " \t") == ";" {
                  break
            }
            P("%s", s)
      }
      P("  jmp Next,pcr")
}

func (o *Ninth) InsertBegin() {
      o.Serial++
      label := F("begin%d", o.Serial)
      o.LoopStack = append(o.LoopStack, label)
      P("%s", label)
}

func (o *Ninth) InsertWhile() {
      o.Serial++
      new_label := F("while%d", o.Serial)
      o.LoopStack = append(o.LoopStack, new_label)
      o.Comma("c_0branch", "0branch")
      o.Comma(F("%s-2", new_label), new_label)
}

func (o *Ninth) InsertRepeat() {
      new_label := o.LoopStack[len(o.LoopStack)-1]
      o.LoopStack = o.LoopStack[:len(o.LoopStack)-1]

      old_label := o.LoopStack[len(o.LoopStack)-1]
      o.LoopStack = o.LoopStack[:len(o.LoopStack)-1]
      o.Comma("c_branch", "branch")
      o.Comma(F("%s-2", old_label), old_label)

      P("%s", new_label)
}
func (o *Ninth) InsertDo() {
      o.CommaEncode("c_>r", "save starting current")
      o.CommaEncode("c_>r", "save limit")
      o.InsertBegin()
      o.CommaEncode("c_r0", "limit")
      // o.CommaEncode("c_dup", "---"); o.CommaEncode("c_.", "---");
      o.CommaEncode("c_r1", "current")
      // o.CommaEncode("c_dup", "---"); o.CommaEncode("c_.", "---");
      o.CommaEncode("c_>", "not finished?")
      // o.CommaEncode("c_dup", "---"); o.CommaEncode("c_.", "---");
      o.InsertWhile()
}
func (o *Ninth) InsertLoop() {
      o.CommaEncode("c_r1", "current")
      o.CommaEncode("c_1+", "incr current")
      o.CommaEncode("c_r1!", "save current")
      o.InsertRepeat()
      o.CommaEncode("c_rdrop", "drop limit")
      o.CommaEncode("c_rdrop", "drop current")
}

func (o *Ninth) InsertColon() {
      for {
            s := o.NextWord()
            P("  ******  %s", s)

            // Stop at the ";"
```

```
if s == ";" {
      break
}

// Special handling for decimal integers.
n, err := strconv.ParseInt(s, 10, 64)
if err == nil {
      // Compile: lit
      P("  fcb ($10000+c_lit-*)/256 ;; %s ;;", s)
      P("  fcb ($10000+c_lit-*)+1")
      // Compile: the integer.
      P("  fcb ($10000+(%d))/256", n)
      P("  fcb (%d)", n)
      continue
}

// Special handling for "$" and hex integers.
if s[0] == '$' {
      // Compile: lit
      P("  fcb ($10000+c_lit-*)/256 ;; %s ;;", s)
      P("  fcb ($10000+c_lit-*)+1")
      x, err := strconv.ParseInt(s[1:], 16, 64)
      if err != nil {
            panic(s)
      }
      // Compile: the integer.
      P("  fcb ($10000+(%d))/256", x)
      P("  fcb (%d)", x)
      continue
}

// if, else, then.
if s == "if" {
      o.Serial++
      label := F("if%d", o.Serial)
      o.IfStack = append(o.IfStack, label)
      o.Comma("c_0branch", "0branch")
      o.Comma(F("%s-2", label), label)
      continue
}

if s == "else" {
      o.Serial++
      new_label := F("if%d", o.Serial)
      o.Comma("c_branch", "branch")
      o.Comma(F("%s-2", new_label), new_label)

      old_label := o.IfStack[len(o.IfStack)-1]
      o.IfStack = append(o.IfStack, old_label)
      P("%s", old_label)

      o.IfStack = append(o.IfStack, new_label)
      continue
}

if s == "then" {
      label := o.IfStack[len(o.IfStack)-1]
      o.IfStack = append(o.IfStack, label)
      P("%s", label)
      continue
}
```

```
            // begin ... while ... repeat

            switch s {
            case "begin":
                  o.InsertBegin()
                  continue
            case "while":
                  o.InsertWhile()
                  continue
            case "repeat":
                  o.InsertRepeat()
                  continue
            case "do":
                  o.InsertDo()
                  continue
            case "loop":
                  o.InsertLoop()
                  continue
            case "\\":
                  o.Words = nil
      continue
            }

            // Normal non-immediate words.
            es := EncodeFunnyChars(s)
            P("  fcb ($10000+c_%s-*)/256 ;; %s ;;", es, s)
            P("  fcb ($10000+c_%s-*)+1", es)
      }
      P("  fcb ($10000+c_exit-*)/256 ;; exit ;;")
      P("  fcb ($10000+c_exit-*)+1")
}

func (o *Ninth) Comma(s string, rem string) {
      P("  fcb ($10000+%s-*)/256 ;; %s ;;", s, rem)
      P("  fcb ($10000+%s-*)+1", s)
}
func (o *Ninth) CommaEncode(s string, rem string) {
      P("  fcb ($10000+%s-*)/256 ;; %s ;;", EncodeFunnyChars(s), rem)
      P("  fcb ($10000+%s-*)+1", EncodeFunnyChars(s))
}

func (o *Ninth) DoCode() {
      name := o.NextWord()
      o.DoPrelude(name, "d_"+name)
      o.InsertCode()
}
func (o *Ninth) DoColon() {
      name := o.NextWord()
      o.DoPrelude(name, "Enter")
      o.InsertColon()
}
func (o *Ninth) DoAllot(n int) {
      name := o.NextWord()
      offset := o.Here
      o.Here += n
      o.DoPrelude(name, "d_"+name)
      o.InsertAllot(offset)
      o.Allots[name] = offset
}
func (o *Ninth) DoInit() {
      // Save our dynamic o.Here into the "here" variable in RAM.
      P("Init")
      // The location of the "here" variable into X.
      P("  tfr dp,a")
      P("  clrb")
      P("  addd #%d", o.Allots["here"])
      P("  tfr d,x")
      // The current runtime o.Here in D.
      P("  tfr dp,a")
      P("  clrb")
      P("  addd #%d", o.Here)
      // Save D at X.
      P("  std ,x")

      // Save our dynamic o.Latest into the "latest" variable in RAM.
      // The current runtime o.Latest's link address onto stack.
      P("  leax l_%s,pcr", o.Latest)
      P("  pshu x")

      // The location of the "latest" variable into X.
      P("  tfr dp,a")
      P("  clrb")
      P("  addd #%d", o.Allots["latest"])
      P("  tfr d,x")
      // pop d & Save D at X.
      P("  pulu d")
      P("  std ,x")

      // Return
      P("  rts")
}

func CompileFile(w io.Writer, r io.Reader) {
      var hold int
      o := NewNinth(r)
      for {
            w := o.NextWord()
            if w == ">EOF<" {
                  break
            }
            n, err := strconv.ParseInt(w, 10, 64)
            if err == nil {
                  hold = int(n)
                  continue
            }
            switch w {
            case "\\":
                  o.Words = nil
            case ":":
                  o.DoColon()
            case "code":
                  o.DoCode()
            case "allot":
                  o.DoAllot(hold)
            default:
                  panic(F("Unknown Command: %q", w))
            }
      }
      o.DoInit()
}

func main() {
      CompileFile(os.Stdout, os.Stdin)
}
```