```
// /*
// Copyright 2018 The pdfcpu Authors.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
// */
//
// package pdfcpu
//
// import (
//     "bufio"
//     "bytes"
//     "io"
//     "os"
//     "sort"
//     "strconv"
//     "strings"
//
//     "github.com/pdfcpu/pdfcpu/pkg/filter"
//     "github.com/pdfcpu/pdfcpu/pkg/log"
//     "github.com/pkg/errors"
// )
//
// const (
//     defaultBufSize = 1024
//     //unknownDelimiter = byte(0)
// )
//
// // ReadFile reads in a PDF file and builds an internal structure holding its cross
// reference table aka the Context.
// func ReadFile(inFile string, conf *Configuration) (*Context, error) {
//
//     log.Info.Printf("reading %s..\n", inFile)
//
//     f, err := os.Open(inFile)
//     if err != nil {
//         return nil, errors.Wrapf(err, "can't open %q", inFile)
//     }
//
//     defer func() {
//         f.Close()
//     }()
//
//     return Read(f, conf)
// }
//
// // Read takes a readSeeker and generates a Context,
// // an in-memory representation containing a cross reference table.
// func Read(rs io.ReadSeeker, conf *Configuration) (*Context, error) {
//
```



```
//     fields := strings.Fields(line)
//     if len(fields) != 3 ||
//         len(fields[0]) != 10 || len(fields[1]) != 5 || len(fields[2]) != 1 {
//         return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
// header")
//     }
//
//     offset, err := strconv.ParseInt(fields[0], 10, 64)
//     if err != nil {
//         return err
//     }
//
//     generation, err := strconv.Atoi(fields[1])
//     if err != nil {
//         return err
//     }
//
//     entryType := fields[2]
//     if entryType != "f" && entryType != "n" {
//         return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
// entry")
//     }
//
//     var xRefTableEntry XRefTableEntry
//
//     if entryType == "n" {
//
//         // in use object
//
//         log.Read.Printf("parseXRefTableEntry: Object #%d is in use at offset=%d,
// generation=%d\n", objectNumber, offset, generation)
//
//         if offset == 0 {
//             log.Info.Printf("parseXRefTableEntry: Skip entry for in use object #%d
// with offset 0\n", objectNumber)
//             return nil
//         }
//
//         xRefTableEntry =
//             XRefTableEntry{
//                 Free:       false,
//                 Offset:     0offset,
//                 Generation: 0generation}
//
//     } else {
//
//         // free object
//
//         log.Read.Printf("parseXRefTableEntry: Object #%d is unused, next free is
// object#%d, generation=%d\n", objectNumber, offset, generation)
//
//         xRefTableEntry =
//             XRefTableEntry{
//                 Free:       true,
//                 Offset:     0offset,
//                 Generation: 0generation}
//
//     }
//
//     log.Read.Printf("parseXRefTableEntry: Insert new xrefTable entry for Object %d\n",
// objectNumber)
//     xRefTable.Table[objectNumber] = &xRefTableEntry
//
//     log.Read.Println("parseXRefTableEntry: end")
//
//     return nil
```



```
//     log.Read.Println("Read: begin")
//
//     ctx, err := NewContext(rs, conf)
//     if err != nil {
//         return nil, err
//     }
//
//     if ctx.Reader15 {
//         log.Info.Println("PDF Version 1.5 conforming reader")
//     } else {
//         log.Info.Println("PDF Version 1.4 conforming reader - no object streams or
// xrefstreams allowed")
//     }
//
//     // Populate xRefTable.
//     if err = readXRefTable(ctx); err != nil {
//         return nil, errors.Wrapf(err, "Read: xRefTable failed")
//     }
//
//     // Make all objects explicitly available (load into memory) in corresponding
// xRefTable entries.
//     // Also decode any involved object streams.
//     if err = dereferenceXRefTable(ctx, conf); err != nil {
//         return nil, err
//     }
//
//     // Some PDFWriters write an incorrent Size into trailer.
//     if *ctx.XRefTable.Size < len(ctx.XRefTable.Table) {
//         *ctx.XRefTable.Size = len(ctx.XRefTable.Table)
//     }
//
//     log.Read.Println("Read: end")
//
//     return ctx, nil
// }
//
// // ScanLines is a split function for a Scanner that returns each line of
// // text, stripped of any trailing end-of-line marker. The returned line may
// // be empty. The end-of-line marker is one carriage return followed
// // by one newline or one carriage return or one newline.
// // The last non-empty line of input will be returned even if it has no newline.
// func scanLines(data []byte, atEOF bool) (advance int, token []byte, err error) {
//
//     if atEOF && len(data) == 0 {
//         return 0, nil, nil
//     }
//
//     indCR := bytes.IndexByte(data, '\r')
//     indLF := bytes.IndexByte(data, '\n')
//
//     switch {
//
//     case indCR >= 0 && indLF >= 0:
//         if indCR < indLF {
//             if indLF == indCR+1 {
//                 // 0x0D0A
//                 return indLF + 1, data[0:indCR], nil
//             }
//             // 0x0D ... 0x0A
```



```
//         return indCR + 1, data[0:indCR], nil
//
//     case indCR >= 0:
//         // We have a full carriage return terminated line.
//         return indCR + 1, data[0:indCR], nil
//
//     case indLF >= 0:
//         // We have a full newline-terminated line.
//         return indLF + 1, data[0:indLF], nil
//
//     }
//
//     // If we're at EOF, we have a final, non-terminated line. Return it.
//     if atEOF {
//         return len(data), data, nil
//     }
//
//     // Request more data.
//     return 0, nil, nil
// }
//
// func newPositionedReader(rs io.ReadSeeker, offset *int64) (*bufio.Reader, error) {
//
//     _, err := rs.Seek(*offset, io.SeekStart); err != nil {
//         return nil, err
//     }
//
//     log.Read.Printf("newPositionedReader: positioned to offset: %d\n", *offset)
//
//     return bufio.NewReader(rs), nil
// }
//
// // Get the file offset of the last XRefSection.
// // Go to end of file and search backwards for the first occurrence of startxref
//     {offset} %%EOF.
// func offsetLastXRefSection(ctx *Context) (*int64, error) {
//
//     rs := ctx.Read.rs
//
//     var (
//         prevBuf, workBuf []byte
//         bufSize          int64 = 512
//         offset           int64
//     )
//
//     for i := 1; offset == 0; i++ {
//
//         off, err := rs.Seek(-int64(i)*bufSize, io.SeekEnd)
//         if err != nil {
//             return nil, errors.New("pdfcpu: can't find last xref section")
//         }
//
//         log.Read.Printf("scanning for offsetLastXRefSection starting at %d\n", off)
//
//         curBuf := make([]byte, bufSize)
//
```



```
//         if curBuf == nil {
//             curBuf = make([]byte, bufSize)
//         } else {
//             copy(curBuf[bufSize-len(prevBuf):], prevBuf)
//         }
//
//         if _, err = rs.Read(curBuf); err != nil {
//             return nil, errors.Wrapf(err, "can't read %q", inFile)
//         }
//
//         if workBuf == nil {
//             workBuf = append(curBuf, prevBuf...)
//         }
//
//         i = strings.LastIndex(string(workBuf), "startxref")
//         if i == -1 {
//             prevBuf = curBuf
//             continue
//         }
//
//         p := workBuf[i+len("startxref"):]
//         posEOF := strings.Index(string(p), "%%EOF")
//         if posEOF == -1 {
//             return nil, errors.New("pdfcpu: no matching %%EOF for startxref")
//         }
//
//         p = p[:posEOF]
//         offset, err = strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
//         if err != nil {
//             return nil, errors.New("pdfcpu: corrupted last xref section")
//         }
//
//         break
//
//     }
//
//     log.Read.Printf("Offset last xrefsection: %d\n", offset)
//
//     return &offset, nil
// }
//
// // read next subsection entry and generate correspondng xref table entry.
// func parseXRefTableEntry(s *bufio.Scanner, xRefTable *XRefTable, objectNumber int)
// error {
//
//     log.Read.Println("parseXRefTableEntry: begin")
//
//     line, err := scanLine(s)
//     if err != nil {
//         return err
//     }
//
//     if xRefTable.Exists(objectNumber) {
//         log.Read.Printf("parseXRefTableEntry: end - Skip entry %d - already
//     assigned\n", objectNumber)
//         return nil
//     }
//
//     fields := strings.Fields(line)
//     if len(fields) != 3 ||
//         len(fields[0]) != 10 || len(fields[1]) != 5 || len(fields[2]) != 1 {
//         return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
//     header")
//     }
//
```



```
//     offset, err := strconv.ParseInt(fields[0], 10, 64)
//     if err != nil {
//         return err
//     }
//
//     generation, err := strconv.Atoi(fields[1])
//     if err != nil {
//         return err
//     }
//
//     entryType := fields[2]
//     if entryType != "f" && entryType != "n" {
//         return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
// entry")
//     }
//
//     var xRefTableEntry XRefTableEntry
//
//     if entryType == "n" {
//
//         // in use object
//
//         log.Read.Printf("parseXRefTableEntry: Object #%d is in use at offset=%d,
// generation=%d\n", objectNumber, offset, generation)
//
//         if offset == 0 {
//             log.Info.Printf("parseXRefTableEntry: Skip entry for in use object #%d
// with offset 0\n", objectNumber)
//             return nil
//         }
//
//         xRefTableEntry =
//             XRefTableEntry{
//                 Free:       false,
//                 Offset:     0offset,
//                 Generation: 0generation}
//
//     } else {
//
//         // free object
//
//         log.Read.Printf("parseXRefTableEntry: Object #%d is unused, next free is
// object#%d, generation=%d\n", objectNumber, offset, generation)
//
//         xRefTableEntry =
//             XRefTableEntry{
//                 Free:       true,
//                 Offset:     0offset,
//                 Generation: 0generation}
//
//     }
//
//     log.Read.Printf("parseXRefTableEntry: Insert new xrefTable entry for Object %d\n",
// objectNumber)
//     xRefTable.Table[objectNumber] = &xRefTableEntry
//
//     log.Read.Println("parseXRefTableEntry: end")
//
//     return nil
```



```
// }
//
// // Process xRef table subsection and create corrresponding xRef table entries.
// func parseXRefTableSubSection(s *bufio.Scanner, xRefTable *XRefTable, fields []string)
// error {
//
//     log.Read.Println("parseXRefTableSubSection: begin")
//
//     startObjNumber, err := strconv.Atoi(fields[0])
//     if err != nil {
//         return err
//     }
//
//     objCount, err := strconv.Atoi(fields[1])
//     if err != nil {
//         return err
//     }
//
//     log.Read.Printf("detected xref subsection, startObj=%d length=%d\n",
// startObjNumber, objCount)
//
//     // Process all entries of this subsection into xRefTable entries.
//     for i := 0; i < objCount; i++ {
//         if err = parseXRefTableEntry(s, xRefTable, startObjNumber+i); err != nil {
//             return err
//         }
//     }
//
//     log.Read.Println("parseXRefTableSubSection: end")
//
//     return nil
// }
//
// // Parse compressed object.
// func compressedObject(s string) (Object, error) {
//
//     log.Read.Println("compressedObject: begin")
//
//     o, err := parseObject(&s)
//     if err != nil {
//         return nil, err
//     }
//
//     d, ok := o.(Dict)
//     if !ok {
//         // return trivial Object: Integer, Array, etc.
//         log.Read.Println("compressedObject: end, any other than dict")
//         return o, nil
//     }
//
//     streamLength, streamLengthRef := d.Length()
//     if streamLength == nil && streamLengthRef == nil {
//         // return Dict
//         log.Read.Println("compressedObject: end, dict")
//         return d, nil
//     }
//
//     return nil, errors.New("pdfcpu: compressedObject: stream objects are not to be
// stored in an object stream")
```



```
//     log.Read.Println("Read: begin")
//
//     ctx, err := NewContext(rs, conf)
//     if err != nil {
//         return nil, err
//     }
//
//     if ctx.Reader15 {
//         log.Info.Println("PDF Version 1.5 conforming reader")
//     } else {
//         log.Info.Println("PDF Version 1.4 conforming reader - no object streams or
// xrefstreams allowed")
//     }
//
//     // Populate xRefTable.
//     if err = readXRefTable(ctx); err != nil {
//         return nil, errors.Wrapf(err, "Read: xRefTable failed")
//     }
//
//     // Make all objects explicitly available (load into memory) in corresponding
// xRefTable entries.
//     // Also decode any involved object streams.
//     if err = dereferenceXRefTable(ctx, conf); err != nil {
//         return nil, err
//     }
//
//     // Some PDFWriters write an incorrent Size into trailer.
//     if *ctx.XRefTable.Size < len(ctx.XRefTable.Table) {
//         *ctx.XRefTable.Size = len(ctx.XRefTable.Table)
//     }
//
//     log.Read.Println("Read: end")
//
//     return ctx, nil
// }
//
// // ScanLines is a split function for a Scanner that returns each line of
// // text, stripped of any trailing end-of-line marker. The returned line may
// // be empty. The end-of-line marker is one carriage return followed
// // by one newline or one carriage return or one newline.
// // The last non-empty line of input will be returned even if it has no newline.
// func scanLines(data []byte, atEOF bool) (advance int, token []byte, err error) {
//
//     if atEOF && len(data) == 0 {
//         return 0, nil, nil
//     }
//
//     indCR := bytes.IndexByte(data, '\r')
//     indLF := bytes.IndexByte(data, '\n')
//
//     switch {
//
//     case indCR >= 0 && indLF >= 0:
//         if indCR < indLF {
//             if indLF == indCR+1 {
//                 // 0x0D0A
//                 return indLF + 1, data[0:indCR], nil
//             }
//             // 0x0D ... 0x0A
```



```
//         return indCR + 1, data[0:indCR], nil
//
//     case indCR >= 0:
//         // We have a full carriage return terminated line.
//         return indCR + 1, data[0:indCR], nil
//
//     case indLF >= 0:
//         // We have a full newline-terminated line.
//         return indLF + 1, data[0:indLF], nil
//
//     }
//
//     // If we're at EOF, we have a final, non-terminated line. Return it.
//     if atEOF {
//         return len(data), data, nil
//     }
//
//     // Request more data.
//     return 0, nil, nil
// }
//
// func newPositionedReader(rs io.ReadSeeker, offset *int64) (*bufio.Reader, error) {
//
//     _, err := rs.Seek(*offset, io.SeekStart); err != nil {
//         return nil, err
//     }
//
//     log.Read.Printf("newPositionedReader: positioned to offset: %d\n", *offset)
//
//     return bufio.NewReader(rs), nil
// }
//
// // Get the file offset of the last XRefSection.
// // Go to end of file and search backwards for the first occurrence of startxref
//     {offset} %%EOF.
// func offsetLastXRefSection(ctx *Context) (*int64, error) {
//
//     rs := ctx.Read.rs
//
//     var (
//         prevBuf, workBuf []byte
//         bufSize          int64 = 512
//         offset           int64
//     )
//
//     for i := 1; offset == 0; i++ {
//
//         off, err := rs.Seek(-int64(i)*bufSize, io.SeekEnd)
//         if err != nil {
//             return nil, errors.New("pdfcpu: can't find last xref section")
//         }
//
//         log.Read.Printf("scanning for offsetLastXRefSection starting at %d\n", off)
//
//         curBuf := make([]byte, bufSize)
//
```



```
//         if curBuf == nil {
//             curBuf = make([]byte, bufSize)
//         } else {
//             copy(curBuf[bufSize-len(prevBuf):], prevBuf)
//         }
//
//         if _, err = rs.Read(curBuf); err != nil {
//             return nil, errors.Wrapf(err, "can't read %q", inFile)
//         }
//
//         if workBuf == nil {
//             workBuf = append(curBuf, prevBuf...)
//         }
//
//         i = strings.LastIndex(string(workBuf), "startxref")
//         if i == -1 {
//             prevBuf = curBuf
//             continue
//         }
//
//         p := workBuf[i+len("startxref"):]
//         posEOF := strings.Index(string(p), "%%EOF")
//         if posEOF == -1 {
//             return nil, errors.New("pdfcpu: no matching %%EOF for startxref")
//         }
//
//         p = p[:posEOF]
//         offset, err = strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
//         if err != nil {
//             return nil, errors.New("pdfcpu: corrupted last xref section")
//         }
//
//         break
//
//     }
//
//     log.Read.Printf("Offset last xrefsection: %d\n", offset)
//
//     return &offset, nil
// }
//
// // read next subsection entry and generate correspondng xref table entry.
// func parseXRefTableEntry(s *bufio.Scanner, xRefTable *XRefTable, objectNumber int)
// error {
//
//     log.Read.Println("parseXRefTableEntry: begin")
//
//     line, err := scanLine(s)
//     if err != nil {
//         return err
//     }
//
//     if xRefTable.Exists(objectNumber) {
//         log.Read.Printf("parseXRefTableEntry: end - Skip entry %d - already
//     assigned\n", objectNumber)
//         return nil
//     }
//
//     fields := strings.Fields(line)
//     if len(fields) != 3 ||
//         len(fields[0]) != 10 || len(fields[1]) != 5 || len(fields[2]) != 1 {
//         return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
//     header")
//     }
//
```



```
//     offset, err := strconv.ParseInt(fields[0], 10, 64)
//     if err != nil {
//         return err
//     }
//
//     generation, err := strconv.Atoi(fields[1])
//     if err != nil {
//         return err
//     }
//
//     entryType := fields[2]
//     if entryType != "f" && entryType != "n" {
//         return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
// entry")
//     }
//
//     var xRefTableEntry XRefTableEntry
//
//     if entryType == "n" {
//
//         // in use object
//
//         log.Read.Printf("parseXRefTableEntry: Object #%d is in use at offset=%d,
// generation=%d\n", objectNumber, offset, generation)
//
//         if offset == 0 {
//             log.Info.Printf("parseXRefTableEntry: Skip entry for in use object #%d
// with offset 0\n", objectNumber)
//             return nil
//         }
//
//         xRefTableEntry =
//             XRefTableEntry{
//                 Free:       false,
//                 Offset:     0offset,
//                 Generation: 0generation}
//
//     } else {
//
//         // free object
//
//         log.Read.Printf("parseXRefTableEntry: Object #%d is unused, next free is
// object#%d, generation=%d\n", objectNumber, offset, generation)
//
//         xRefTableEntry =
//             XRefTableEntry{
//                 Free:       true,
//                 Offset:     0offset,
//                 Generation: 0generation}
//
//     }
//
//     log.Read.Printf("parseXRefTableEntry: Insert new xrefTable entry for Object %d\n",
// objectNumber)
//     xRefTable.Table[objectNumber] = &xRefTableEntry
//
//     log.Read.Println("parseXRefTableEntry: end")
//
//     return nil
```



```
// }
//
// // Process xRef table subsection and create corrresponding xRef table entries.
// func parseXRefTableSubSection(s *bufio.Scanner, xRefTable *XRefTable, fields []string)
// error {
//
//     log.Read.Println("parseXRefTableSubSection: begin")
//
//     startObjNumber, err := strconv.Atoi(fields[0])
//     if err != nil {
//         return err
//     }
//
//     objCount, err := strconv.Atoi(fields[1])
//     if err != nil {
//         return err
//     }
//
//     log.Read.Printf("detected xref subsection, startObj=%d length=%d\n",
// startObjNumber, objCount)
//
//     // Process all entries of this subsection into xRefTable entries.
//     for i := 0; i < objCount; i++ {
//         if err = parseXRefTableEntry(s, xRefTable, startObjNumber+i); err != nil {
//             return err
//         }
//     }
//
//     log.Read.Println("parseXRefTableSubSection: end")
//
//     return nil
// }
//
// // Parse compressed object.
// func compressedObject(s string) (Object, error) {
//
//     log.Read.Println("compressedObject: begin")
//
//     o, err := parseObject(&s)
//     if err != nil {
//         return nil, err
//     }
//
//     d, ok := o.(Dict)
//     if !ok {
//         // return trivial Object: Integer, Array, etc.
//         log.Read.Println("compressedObject: end, any other than dict")
//         return o, nil
//     }
//
//     streamLength, streamLengthRef := d.Length()
//     if streamLength == nil && streamLengthRef == nil {
//         // return Dict
//         log.Read.Println("compressedObject: end, dict")
//         return d, nil
//     }
//
//     return nil, errors.New("pdfcpu: compressedObject: stream objects are not to be
// stored in an object stream")
```



```
//     log.Read.Println("Read: begin")
//
//     ctx, err := NewContext(rs, conf)
//     if err != nil {
//         return nil, err
//     }
//
//     if ctx.Reader15 {
//         log.Info.Println("PDF Version 1.5 conforming reader")
//     } else {
//         log.Info.Println("PDF Version 1.4 conforming reader - no object streams or
// xrefstreams allowed")
//     }
//
//     // Populate xRefTable.
//     if err = readXRefTable(ctx); err != nil {
//         return nil, errors.Wrapf(err, "Read: xRefTable failed")
//     }
//
//     // Make all objects explicitly available (load into memory) in corresponding
// xRefTable entries.
//     // Also decode any involved object streams.
//     if err = dereferenceXRefTable(ctx, conf); err != nil {
//         return nil, err
//     }
//
//     // Some PDFWriters write an incorrent Size into trailer.
//     if *ctx.XRefTable.Size < len(ctx.XRefTable.Table) {
//         *ctx.XRefTable.Size = len(ctx.XRefTable.Table)
//     }
//
//     log.Read.Println("Read: end")
//
//     return ctx, nil
// }
//
// // ScanLines is a split function for a Scanner that returns each line of
// // text, stripped of any trailing end-of-line marker. The returned line may
// // be empty. The end-of-line marker is one carriage return followed
// // by one newline or one carriage return or one newline.
// // The last non-empty line of input will be returned even if it has no newline.
// func scanLines(data []byte, atEOF bool) (advance int, token []byte, err error) {
//
//     if atEOF && len(data) == 0 {
//         return 0, nil, nil
//     }
//
//     indCR := bytes.IndexByte(data, '\r')
//     indLF := bytes.IndexByte(data, '\n')
//
//     switch {
//
//     case indCR >= 0 && indLF >= 0:
//         if indCR < indLF {
//             if indLF == indCR+1 {
//                 // 0x0D0A
//                 return indLF + 1, data[0:indCR], nil
//             }
//             // 0x0D ... 0x0A
```



```
//         return indCR + 1, data[0:indCR], nil
//
//     case indCR >= 0:
//         // We have a full carriage return terminated line.
//         return indCR + 1, data[0:indCR], nil
//
//     case indLF >= 0:
//         // We have a full newline-terminated line.
//         return indLF + 1, data[0:indLF], nil
//
//     }
//
//     // If we're at EOF, we have a final, non-terminated line. Return it.
//     if atEOF {
//         return len(data), data, nil
//     }
//
//     // Request more data.
//     return 0, nil, nil
// }
//
// func newPositionedReader(rs io.ReadSeeker, offset *int64) (*bufio.Reader, error) {
//
//     _, err := rs.Seek(*offset, io.SeekStart); err != nil {
//         return nil, err
//     }
//
//     log.Read.Printf("newPositionedReader: positioned to offset: %d\n", *offset)
//
//     return bufio.NewReader(rs), nil
// }
//
// // Get the file offset of the last XRefSection.
// // Go to end of file and search backwards for the first occurrence of startxref
//     {offset} %%EOF.
// func offsetLastXRefSection(ctx *Context) (*int64, error) {
//
//     rs := ctx.Read.rs
//
//     var (
//         prevBuf, workBuf []byte
//         bufSize          int64 = 512
//         offset           int64
//     )
//
//     for i := 1; offset == 0; i++ {
//
//         off, err := rs.Seek(-int64(i)*bufSize, io.SeekEnd)
//         if err != nil {
//             return nil, errors.New("pdfcpu: can't find last xref section")
//         }
//
//         log.Read.Printf("scanning for offsetLastXRefSection starting at %d\n", off)
//
//         curBuf := make([]byte, bufSize)
//
```



```
//         if curBuf == nil {
//             curBuf = make([]byte, bufSize)
//         } else {
//             copy(curBuf[bufSize-len(prevBuf):], prevBuf)
//         }
//
//         if _, err = rs.Read(curBuf); err != nil {
//             return nil, errors.Wrapf(err, "can't read %q", inFile)
//         }
//
//         if workBuf == nil {
//             workBuf = append(curBuf, prevBuf...)
//         }
//
//         i = strings.LastIndex(string(workBuf), "startxref")
//         if i == -1 {
//             prevBuf = curBuf
//             continue
//         }
//
//         p := workBuf[i+len("startxref"):]
//         posEOF := strings.Index(string(p), "%%EOF")
//         if posEOF == -1 {
//             return nil, errors.New("pdfcpu: no matching %%EOF for startxref")
//         }
//
//         p = p[:posEOF]
//         offset, err = strconv.ParseInt(strings.TrimSpace(string(p)), 10, 64)
//         if err != nil {
//             return nil, errors.New("pdfcpu: corrupted last xref section")
//         }
//
//         break
//
//     }
//
//     log.Read.Printf("Offset last xrefsection: %d\n", offset)
//
//     return &offset, nil
// }
//
// // read next subsection entry and generate correspondng xref table entry.
// func parseXRefTableEntry(s *bufio.Scanner, xRefTable *XRefTable, objectNumber int)
// error {
//
//     log.Read.Println("parseXRefTableEntry: begin")
//
//     line, err := scanLine(s)
//     if err != nil {
//         return err
//     }
//
//     if xRefTable.Exists(objectNumber) {
//         log.Read.Printf("parseXRefTableEntry: end - Skip entry %d - already
//     assigned\n", objectNumber)
//         return nil
//     }
//
//     fields := strings.Fields(line)
//     if len(fields) != 3 ||
//         len(fields[0]) != 10 || len(fields[1]) != 5 || len(fields[2]) != 1 {
//         return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
//     header")
//     }
//
```



```
//     offset, err := strconv.ParseInt(fields[0], 10, 64)
//     if err != nil {
//         return err
//     }
//
//     generation, err := strconv.Atoi(fields[1])
//     if err != nil {
//         return err
//     }
//
//     entryType := fields[2]
//     if entryType != "f" && entryType != "n" {
//         return errors.New("pdfcpu: parseXRefTableEntry: corrupt xref subsection
// entry")
//     }
//
//     var xRefTableEntry XRefTableEntry
//
//     if entryType == "n" {
//
//         // in use object
//
//         log.Read.Printf("parseXRefTableEntry: Object #%d is in use at offset=%d,
// generation=%d\n", objectNumber, offset, generation)
//
//         if offset == 0 {
//             log.Info.Printf("parseXRefTableEntry: Skip entry for in use object #%d
// with offset 0\n", objectNumber)
//             return nil
//         }
//
//         xRefTableEntry =
//             XRefTableEntry{
//                 Free:       false,
//                 Offset:     0offset,
//                 Generation: 0generation}
//
//     } else {
//
//         // free object
//
//         log.Read.Printf("parseXRefTableEntry: Object #%d is unused, next free is
// object#%d, generation=%d\n", objectNumber, offset, generation)
//
//         xRefTableEntry =
//             XRefTableEntry{
//                 Free:       true,
//                 Offset:     0offset,
//                 Generation: 0generation}
//
//     }
//
//     log.Read.Printf("parseXRefTableEntry: Insert new xrefTable entry for Object %d\n",
// objectNumber)
//     xRefTable.Table[objectNumber] = &xRefTableEntry
//
//     log.Read.Println("parseXRefTableEntry: end")
//
//     return nil
```



```
// }
//
// // Process xRef table subsection and create corrresponding xRef table entries.
// func parseXRefTableSubSection(s *bufio.Scanner, xRefTable *XRefTable, fields []string)
// error {
//
//     log.Read.Println("parseXRefTableSubSection: begin")
//
//     startObjNumber, err := strconv.Atoi(fields[0])
//     if err != nil {
//         return err
//     }
//
//     objCount, err := strconv.Atoi(fields[1])
//     if err != nil {
//         return err
//     }
//
//     log.Read.Printf("detected xref subsection, startObj=%d length=%d\n",
// startObjNumber, objCount)
//
//     // Process all entries of this subsection into xRefTable entries.
//     for i := 0; i < objCount; i++ {
//         if err = parseXRefTableEntry(s, xRefTable, startObjNumber+i); err != nil {
//             return err
//         }
//     }
//
//     log.Read.Println("parseXRefTableSubSection: end")
//
//     return nil
// }
//
// // Parse compressed object.
// func compressedObject(s string) (Object, error) {
//
//     log.Read.Println("compressedObject: begin")
//
//     o, err := parseObject(&s)
//     if err != nil {
//         return nil, err
//     }
//
//     d, ok := o.(Dict)
//     if !ok {
//         // return trivial Object: Integer, Array, etc.
//         log.Read.Println("compressedObject: end, any other than dict")
//         return o, nil
//     }
//
//     streamLength, streamLengthRef := d.Length()
//     if streamLength == nil && streamLengthRef == nil {
//         // return Dict
//         log.Read.Println("compressedObject: end, dict")
//         return d, nil
//     }
//
//     return nil, errors.New("pdfcpu: compressedObject: stream objects are not to be
// stored in an object stream")
```



```
//     log.Read.Println("Read: begin")
//
//     ctx, err := NewContext(rs, conf)
//     if err != nil {
//         return nil, err
//     }
//
//     if ctx.Reader15 {
//         log.Info.Println("PDF Version 1.5 conforming reader")
//     } else {
//         log.Info.Println("PDF Version 1.4 conforming reader - no object streams or
// xrefstreams allowed")
//     }
//
//     // Populate xRefTable.
//     if err = readXRefTable(ctx); err != nil {
//         return nil, errors.Wrapf(err, "Read: xRefTable failed")
//     }
//
//     // Make all objects explicitly available (load into memory) in corresponding
// xRefTable entries.
//     // Also decode any involved object streams.
//     if err = dereferenceXRefTable(ctx, conf); err != nil {
//         return nil, err
//     }
//
//     // Some PDFWriters write an incorrent Size into trailer.
//     if *ctx.XRefTable.Size < len(ctx.XRefTable.Table) {
//         *ctx.XRefTable.Size = len(ctx.XRefTable.Table)
//     }
//
//     log.Read.Println("Read: end")
//
//     return ctx, nil
// }
//
// // ScanLines is a split function for a Scanner that returns each line of
// // text, stripped of any trailing end-of-line marker. The returned line may
// // be empty. The end-of-line marker is one carriage return followed
// // by one newline or one carriage return or one newline.
// // The last non-empty line of input will be returned even if it has no newline.
// func scanLines(data []byte, atEOF bool) (advance int, token []byte, err error) {
//
//     if atEOF && len(data) == 0 {
//         return 0, nil, nil
//     }
//
//     indCR := bytes.IndexByte(data, '\r')
//     indLF := bytes.IndexByte(data, '\n')
//
//     switch {
//
//     case indCR >= 0 && indLF >= 0:
//         if indCR < indLF {
//             if indLF == indCR+1 {
//                 // 0x0D0A
//                 return indLF + 1, data[0:indCR], nil
//             }
//             // 0x0D ... 0x0A
```



```
//         return indCR + 1, data[0:indCR], nil
//
//     case indCR >= 0:
//         // We have a full carriage return terminated line.
//         return indCR + 1, data[0:indCR], nil
//
//     case indLF >= 0:
//         // We have a full newline-terminated line.
//         return indLF + 1, data[0:indLF], nil
//
//     }
//
//     // If we're at EOF, we have a final, non-terminated line. Return it.
//     if atEOF {
//         return len(data), data, nil
//     }
//
//     // Request more data.
//     return 0, nil, nil
// }
//
// func newPositionedReader(rs io.ReadSeeker, offset *int64) (*bufio.Reader, error) {
//
//     _, err := rs.Seek(*offset, io.SeekStart); err != nil {
//         return nil, err
//     }
//
//     log.Read.Printf("newPositionedReader: positioned to offset: %d\n", *offset)
//
//     return bufio.NewReader(rs), nil
// }
//
// // Get the file offset of the last XRefSection.
// // Go to end of file and search backwards for the first occurrence of startxref
//     {offset} %%EOF.
// func offsetLastXRefSection(ctx *Context) (*int64, error) {
//
//     rs := ctx.Read.rs
//
//     var (
//         prevBuf, workBuf []byte
//         bufSize          int64 = 512
//         offset           int64
//     )
//
//     for i := 1; offset == 0; i++ {
//
//         off, err := rs.Seek(-int64(i)*bufSize, io.SeekEnd)
//         if err != nil {
```

```go
		return false
	}

	// We found the last >>. Return true if after end of dict only whitespace.
	ok := strings.TrimSpace(b[eod:]) == ">>"

	//log.Read.Printf("keywordStreamRightAfterEndOfDict: end, %v\n", ok)

	return ok
}

func buildFilterPipeline(ctx *Context, filterArray, decodeParmsArr Array, decodeParms
Object) ([]PDFFilter, error) {

	var filterPipeline []PDFFilter

	for i, f := range filterArray {

		filterName, ok := f.(Name)
		if !ok {
			return nil, errors.New("pdfcpu: buildFilterPipeline: filterArray elements
corrupt")
		}
		if decodeParms == nil || decodeParmsArr[i] == nil {
			filterPipeline = append(filterPipeline, PDFFilter{Name:
filterName.Value(), DecodeParms: nil})
			continue
		}

		dict, ok := decodeParmsArr[i].(Dict)
		if !ok {
			indRef, ok := decodeParmsArr[i].(IndirectRef)
			if !ok {
				return nil, errors.Errorf("buildFilterPipeline: corrupt Dict: %s\n",
dict)
			}
			d, err := dereferencedDict(ctx, indRef.ObjectNumber.Value())
			if err != nil {
				return nil, err
			}
			dict = d
		}

		filterPipeline = append(filterPipeline, PDFFilter{Name: filterName.String(),
DecodeParms: dict})
	}

	return filterPipeline, nil
}

// Return the filter pipeline associated with this stream dict.
func pdfFilterPipeline(ctx *Context, dict Dict) ([]PDFFilter, error) {

	log.Read.Println("pdfFilterPipeline: begin")

	var err error

	o, found := dict.Find("Filter")
	if !found {
		// stream is not compressed.
```



```go
		return nil, nil
	}

	// compressed stream.

	var filterPipeline []PDFFilter

	if indRef, ok := o.(IndirectRef); ok {
		o, err = dereferencedObject(ctx, indRef.ObjectNumber.Value())
		if err != nil {
			return nil, err
		}
	}

	//fmt.Printf("dereferenced filter obj: %s\n", obj)

	if name, ok := o.(Name); ok {

		// single filter.

		filterName := name.String()

		o, found := dict.Find("DecodeParms")
		if !found {
			// w/o decode parameters.
			log.Read.Println("pdfFilterPipeline: end w/o decode parms")
			return append(filterPipeline, PDFFilter{Name: filterName, DecodeParms:
nil}), nil
		}

		d, ok := o.(Dict)
		if !ok {
			ir, ok := o.(IndirectRef)
			if !ok {
				return nil, errors.Errorf("pdfFilterPipeline: corrupt Dict: %s\n", o)
			}
			d, err = dereferencedDict(ctx, ir.ObjectNumber.Value())
			if err != nil {
				return nil, err
			}
		}

		// with decode parameters.
		log.Read.Println("pdfFilterPipeline: end with decode parms")
		return append(filterPipeline, PDFFilter{Name: filterName, DecodeParms: d}),
nil
	}

	// filter pipeline.

	// Array of filternames
	filterArray, ok := o.(Array)
	if !ok {
		return nil, errors.Errorf("pdfFilterPipeline: Expected filterArray corrupt, %v
%T", o, o)
	}

	// Optional array of decode parameter dicts.
	var decodeParmsArr Array
```



```go
	decodeParms, found := dict.Find("DecodeParms")
	if found {
		decodeParmsArr, ok = decodeParms.(Array)
		if !ok {
			return nil, errors.New("pdfcpu: pdfFilterPipeline: expected decodeParms
array corrupt")
		}
	}

	//fmt.Printf("decodeParmsArr: %s\n", decodeParmsArr)

	filterPipeline, err = buildFilterPipeline(ctx, filterArray, decodeParmsArr,
decodeParms)

	log.Read.Println("pdfFilterPipeline: end")

	return filterPipeline, err
}

func streamDictForObject(ctx *Context, d Dict, objNr, streamInd int, streamOffset,
offset int64) (sd StreamDict, err error) {

	streamLength, streamLengthRef := d.Length()

	if streamInd <= 0 {
		return sd, errors.New("pdfcpu: streamDictForObject: stream object without
streamOffset")
	}

	filterPipeline, err := pdfFilterPipeline(ctx, d)
	if err != nil {
		return sd, err
	}

	streamOffset += offset

	// We have a stream object.
	sd = NewStreamDict(d, streamOffset, streamLength, streamLengthRef, filterPipeline)

	log.Read.Printf("streamDictForObject: end, Streamobject #%d\n", objNr)

	return sd, nil
}

func dict(ctx *Context, d1 Dict, objNr, genNr, endInd, streamInd int) (d2 Dict, err
error) {

	if ctx.EncKey != nil {
		_, err := decryptDeepObject(d1, objNr, genNr, ctx.EncKey, ctx.AES4Strings,
ctx.E.R)
		if err != nil {
			return nil, err
		}
	}

	if endInd >= 0 && (streamInd < 0 || streamInd > endInd) {
		log.Read.Printf("dict: end, #%d\n", objNr)
		d2 = d1
	}

```



```go
	return d2, nil
}

func object(ctx *Context, offset int64, objNr, genNr int) (o Object, endInd, streamInd
int, streamOffset int64, err error) {

	var rd io.Reader
	rd, err = newPositionedReader(ctx.Read.rs, &offset)
	if err != nil {
		return nil, 0, 0, 0, err
	}

	//log.Read.Printf("object: seeked to offset:%d\n", offset)

	// process: # gen obj ... obj dict ... {stream ... data ... endstream} endobj
	//                                      streamInd                        endInd
	//                                    -1 if absent                    -1 if absent
	var buf []byte
	buf, endInd, streamInd, streamOffset, err = buffer(rd)
	if err != nil {
		return nil, 0, 0, 0, err
	}

	//log.Read.Printf("streamInd:%d(#%x) streamOffset:%d(#%x) endInd:%d(#%x)\n",
streamInd, streamInd, streamOffset, streamOffset, endInd, endInd)
	//log.Read.Printf("buflen=%d\n%s", len(buf), hex.Dump(buf))

	line := string(buf)

	var l string

	if endInd < 0 { // && streamInd >= 0, streamdict
		// buf: # gen obj ... obj dict ... stream ... data
		// implies we detected no endobj and a stream starting at streamInd.
		// big stream, we parse object until "stream"
		log.Read.Println("object: big stream, we parse object until stream")
		l = line[:streamInd]
	} else if streamInd < 0 { // dict
		// buf: # gen obj ... obj dict ... endobj
		// implies we detected endobj and no stream.
		// small object w/o stream, parse until "endobj"
		log.Read.Println("object: small object w/o stream, parse until endobj")
		l = line[:endInd]
	} else if streamInd < endInd { // streamdict
		// buf: # gen obj ... obj dict ... stream ... data ... endstream endobj
		// implies we detected endobj and stream.
		// small stream within buffer, parse until "stream"
		log.Read.Println("object: small stream within buffer, parse until stream")
		l = line[:streamInd]
	} else { // dict
		// buf: # gen obj ... obj dict ... endobj # gen obj ... obj dict ... stream
		// small obj w/o stream, parse until "endobj"
		// stream in buf belongs to subsequent object.
		log.Read.Println("object: small obj w/o stream, parse until endobj")
		l = line[:endInd]
	}

	// Parse object number and object generation.
	var objectNr, generationNr *int
```



```go
	objectNr, generationNr, err = parseObjectAttributes(&l)
	if err != nil {
		return nil, 0, 0, 0, err
	}

	if objNr != *objectNr || genNr != *generationNr {
		return nil, 0, 0, 0, errors.Errorf("object: non matching objNr(%d) or
generationNumber(%d) tags found.", *objectNr, *generationNr)
	}

	o, err = parseObject(&l)

	return o, endInd, streamInd, streamOffset, err
}

// ParseObject parses an object from file at given offset.
func ParseObject(ctx *Context, offset int64, objNr, genNr int) (Object, error) {

	log.Read.Printf("ParseObject: begin, obj#%d, offset:%d\n", objNr, offset)

	obj, endInd, streamInd, streamOffset, err := object(ctx, offset, objNr, genNr)
	if err != nil {
		return nil, err
	}

	switch o := obj.(type) {

	case Dict:
		d, err := dict(ctx, o, objNr, genNr, endInd, streamInd)
		if err != nil || d != nil {
			// Dict
			return d, err
		}
		// StreamDict.
		return streamDictForObject(ctx, o, objNr, streamInd, streamOffset, offset)

	case Array:
		if ctx.EncKey != nil {
			if _, err = decryptDeepObject(o, objNr, genNr, ctx.EncKey,
ctx.AES4Strings, ctx.E.R); err != nil {
				return nil, err
			}
		}
		return o, nil

	case StringLiteral:
		if ctx.EncKey != nil {
			s1, err := decryptString(o.Value(), objNr, genNr, ctx.EncKey,
ctx.AES4Strings, ctx.E.R)
			if err != nil {
				return nil, err
			}
			return StringLiteral(*s1), nil
		}
		return o, nil

	case HexLiteral:
		if ctx.EncKey != nil {
			bb, err := decryptHexLiteral(o, objNr, genNr, ctx.EncKey, ctx.AES4Strings,
```



```go
ctx.E.R)
			if err != nil {
				return nil, err
			}
			return StringLiteral(string(bb)), nil
		}
		return o, nil

	default:
		return o, nil
	}
}

func dereferencedObject(ctx *Context, objectNumber int) (Object, error) {

	entry, ok := ctx.Find(objectNumber)
	if !ok {
		return nil, errors.New("pdfcpu: dereferencedObject: unregistered object")
	}

	if entry.Compressed {
		err := decompressXRefTableEntry(ctx.XRefTable, objectNumber, entry)
		if err != nil {
			return nil, err
		}
	}

	if entry.Object == nil {

		log.Read.Printf("dereferencedObject: dereferencing object %d\n", objectNumber)

		o, err := ParseObject(ctx, *entry.Offset, objectNumber, *entry.Generation)
		if err != nil {
			return nil, errors.Wrapf(err, "dereferencedObject: problem dereferencing
object %d", objectNumber)
		}

		if o == nil {
			return nil, errors.New("pdfcpu: dereferencedObject: object is nil")
		}

		entry.Object = o
	}

	return entry.Object, nil
}

func dereferencedInteger(ctx *Context, objectNumber int) (*Integer, error) {

	o, err := dereferencedObject(ctx, objectNumber)
	if err != nil {
		return nil, err
	}

	i, ok := o.(Integer)
	if !ok {
		return nil, errors.New("pdfcpu: dereferencedInteger: corrupt integer")
	}

```



```go
	return &i, nil
}

func dereferencedDict(ctx *Context, objectNumber int) (Dict, error) {

	o, err := dereferencedObject(ctx, objectNumber)
	if err != nil {
		return nil, err
	}

	d, ok := o.(Dict)
	if !ok {
		return nil, errors.New("pdfcpu: dereferencedDict: corrupt dict")
	}

	return d, nil
}

// dereference a Integer object representing an int64 value.
func int64Object(ctx *Context, objectNumber int) (*int64, error) {

	log.Read.Printf("int64Object begin: %d\n", objectNumber)

	i, err := dereferencedInteger(ctx, objectNumber)
	if err != nil {
		return nil, err
	}

	i64 := int64(i.Value())

	log.Read.Printf("int64Object end: %d\n", objectNumber)

	return &i64, nil

}

// Reads and returns a file buffer with length = stream length using provided reader
positioned at offset.
func readContentStream(rd io.Reader, streamLength int) ([]byte, error) {

	log.Read.Printf("readContentStream: begin streamLength:%d\n", streamLength)

	buf := make([]byte, streamLength)

	for totalCount := 0; totalCount < streamLength; {
		count, err := rd.Read(buf[totalCount:])
		if err != nil {
			return nil, err
		}
		log.Read.Printf("readContentStream: count=%d, buflen=%d(%X)\n", count,
len(buf), len(buf))
		totalCount += count
	}

	log.Read.Printf("readContentStream: end\n")

	return buf, nil
}

```



```go
// LoadEncodedStreamContent loads the encoded stream content from file into
StreamDict.
func loadEncodedStreamContent(ctx *Context, sd *StreamDict) ([]byte, error) {

	log.Read.Printf("LoadEncodedStreamContent: begin\n%v\n", sd)

	var err error

	// Return saved decoded content.
	if sd.Raw != nil {
		log.Read.Println("LoadEncodedStreamContent: end, already in memory.")
		return sd.Raw, nil
	}

	// Read stream content encoded at offset with stream length.

	// Dereference stream length if stream length is an indirect object.
	if sd.StreamLength == nil {
		if sd.StreamLengthObjNr == nil {
			return nil, errors.New("pdfcpu: loadEncodedStreamContent: missing
streamLength")
		}
		// Get stream length from indirect object
		sd.StreamLength, err = int64Object(ctx, *sd.StreamLengthObjNr)
		if err != nil {
			return nil, err
		}
		log.Read.Printf("LoadEncodedStreamContent: new indirect streamLength:%d\n",
*sd.StreamLength)
	}

	newOffset := sd.StreamOffset
	rd, err := newPositionedReader(ctx.Read.rs, &newOffset)
	if err != nil {
		return nil, err
	}

	log.Read.Printf("LoadEncodedStreamContent: seeked to offset:%d\n", newOffset)

	// Buffer stream contents.
	// Read content from disk.
	rawContent, err := readContentStream(rd, int(*sd.StreamLength))
	if err != nil {
		return nil, err
	}

	//log.Read.Printf("rawContent buflen=%d(#%x)\n%s", len(rawContent),
len(rawContent), hex.Dump(rawContent))

	// Save encoded content.
	sd.Raw = rawContent

	log.Read.Printf("LoadEncodedStreamContent: end: len(streamDictRaw)=%d\n",
len(sd.Raw))

	// Return encoded content.
	return rawContent, nil
}

// Decodes the raw encoded stream content and saves it to streamDict.Content.
```



```go
func saveDecodedStreamContent(ctx *Context, sd *StreamDict, objNr, genNr int, decode
bool) (err error) {

	log.Read.Printf("saveDecodedStreamContent: begin decode=%t\n", decode)

	// If the "Identity" crypt filter is used we do not need to decrypt.
	if ctx != nil && ctx.EncKey != nil {
		if len(sd.FilterPipeline) == 1 && sd.FilterPipeline[0].Name == "Crypt" {
			sd.Content = sd.Raw
			return nil
		}
	}

	// Special case: If the length of the encoded data is 0, we do not need to decode
anything.
	if len(sd.Raw) == 0 {
		sd.Content = sd.Raw
		return nil
	}

	// ctx gets created after XRefStream parsing.
	// XRefStreams are not encrypted.
	if ctx != nil && ctx.EncKey != nil {
		sd.Raw, err = decryptStream(sd.Raw, objNr, genNr, ctx.EncKey, ctx.AES4Streams,
ctx.E.R)
		if err != nil {
			return err
		}
		l := int64(len(sd.Raw))
		sd.StreamLength = &l
	}

	if !decode {
		return nil
	}

	// Actual decoding of content stream.
	err = decodeStream(sd)
	if err == filter.ErrUnsupportedFilter {
		err = nil
	}
	if err != nil {
		return err
	}

	log.Read.Println("saveDecodedStreamContent: end")

	return nil
}

// Resolve compressed xRefTableEntry
func decompressXRefTableEntry(xRefTable *XRefTable, objectNumber int, entry
*XRefTableEntry) error {

	log.Read.Printf("decompressXRefTableEntry: compressed object %d at %d[%d]\n",
objectNumber, *entry.ObjectStream, *entry.ObjectStreamInd)

	// Resolve xRefTable entry of referenced object stream.
	objectStreamXRefTableEntry, ok := xRefTable.Find(*entry.ObjectStream)
	if !ok {
```



```go
		return errors.Errorf("decompressXRefTableEntry: problem dereferencing object
stream %d, no xref table entry", *entry.ObjectStream)
	}

	// Object of this entry has to be a ObjectStreamDict.
	sd, ok := objectStreamXRefTableEntry.Object.(ObjectStreamDict)
	if !ok {
		return errors.Errorf("decompressXRefTableEntry: problem dereferencing object
stream %d, no object stream", *entry.ObjectStream)
	}

	// Get indexed object from ObjectStreamDict.
	o, err := sd.IndexedObject(*entry.ObjectStreamInd)
	if err != nil {
		return errors.Wrapf(err, "decompressXRefTableEntry: problem dereferencing
object stream %d", *entry.ObjectStream)
	}

	// Save object to XRefRableEntry.
	g := 0
	entry.Object = o
	entry.Generation = &g
	entry.Compressed = false

	log.Read.Printf("decompressXRefTableEntry: end, Obj %d[%d]:\n<%s>\n",
*entry.ObjectStream, *entry.ObjectStreamInd, o)

	return nil
}

// Log interesting stream content.
func logStream(o Object) {

	switch o := o.(type) {

	case StreamDict:

		if o.Content == nil {
			log.Read.Println("logStream: no stream content")
		}

		if o.IsPageContent {
			//log.Read.Printf("content <%s>\n", StreamDict.Content)
		}

	case ObjectStreamDict:

		if o.Content == nil {
			log.Read.Println("logStream: no object stream content")
		} else {
			log.Read.Printf("logStream: objectStream content = %s\n", o.Content)
		}

		if o.ObjArray == nil {
			log.Read.Println("logStream: no object stream obj arr")
		} else {
			log.Read.Printf("logStream: objectStream objArr = %s\n", o.ObjArray)
		}

	default:
```



```go
		log.Read.Println("logStream: no ObjectStreamDict")

	}

}

// Decode all object streams so contained objects are ready to be used.
func decodeObjectStreams(ctx *Context) error {

	// Note:
	// Entry "Extends" intentionally left out.
	// No object stream collection validation necessary.

	log.Read.Println("decodeObjectStreams: begin")

	// Get sorted slice of object numbers.
	var keys []int
	for k := range ctx.Read.ObjectStreams {
		keys = append(keys, k)
	}
	sort.Ints(keys)

	for _, objectNumber := range keys {

		// Get XRefTableEntry.
		entry := ctx.XRefTable.Table[objectNumber]
		if entry == nil {
			return errors.Errorf("decodeObjectStream: missing entry for obj#%d\n",
objectNumber)
		}

		log.Read.Printf("decodeObjectStreams: parsing object stream for obj#%d\n",
objectNumber)

		// Parse object stream from file.
		o, err := ParseObject(ctx, *entry.Offset, objectNumber, *entry.Generation)
		if err != nil || o == nil {
			return errors.New("pdfcpu: decodeObjectStreams: corrupt object stream")
		}

		// Ensure StreamDict
		sd, ok := o.(StreamDict)
		if !ok {
			return errors.New("pdfcpu: decodeObjectStreams: corrupt object stream")
		}

		// Load encoded stream content to xRefTable.
		if _, err = loadEncodedStreamContent(ctx, &sd); err != nil {
			return errors.Wrapf(err, "decodeObjectStreams: problem dereferencing
object stream %d", objectNumber)
		}

		// Save decoded stream content to xRefTable.
		if err = saveDecodedStreamContent(ctx, &sd, objectNumber, *entry.Generation,
true); err != nil {
			log.Read.Printf("obj %d: %s", objectNumber, err)
			return err
		}

```



```go
		// Ensure decoded objectArray for object stream dicts.
		if !sd.IsObjStm() {
			return errors.New("pdfcpu: decodeObjectStreams: corrupt object stream")
		}

		// We have an object stream.
		log.Read.Printf("decodeObjectStreams: object stream #%d\n", objectNumber)

		ctx.Read.UsingObjectStreams = true

		// Create new object stream dict.
		osd, err := objectStreamDict(&sd)
		if err != nil {
			return errors.Wrapf(err, "decodeObjectStreams: problem dereferencing
object stream %d", objectNumber)
		}

		log.Read.Printf("decodeObjectStreams: decoding object stream %d:\n",
objectNumber)

		// Parse all objects of this object stream and save them to
ObjectStreamDict.ObjArray.
		if err = parseObjectStream(osd); err != nil {
			return errors.Wrapf(err, "decodeObjectStreams: problem decoding object
stream %d\n", objectNumber)
		}

		if osd.ObjArray == nil {
			return errors.Wrap(err, "decodeObjectStreams: objArray should be set!")
		}

		log.Read.Printf("decodeObjectStreams: decoded object stream %d:\n",
objectNumber)

		// Save object stream dict to xRefTableEntry.
		entry.Object = *osd
	}

	log.Read.Println("decodeObjectStreams: end")

	return nil
}

func handleLinearizationParmDict(ctx *Context, obj Object, objNr int) error {

	if ctx.Read.Linearized {
		// Linearization dict already processed.
		return nil
	}

	// handle linearization parm dict.
	if d, ok := obj.(Dict); ok && d.IsLinearizationParmDict() {

		ctx.Read.Linearized = true
		ctx.LinearizationObjs[objNr] = true
		log.Read.Printf("handleLinearizationParmDict: identified linearizationObj
#%d\n", objNr)

		a := d.ArrayEntry("H")

```



```go
		if a == nil {
			return errors.Errorf("handleLinearizationParmDict: corrupt linearization
dict at obj:%d - missing array entry H", objNr)
		}

		if len(a) != 2 && len(a) != 4 {
			return errors.Errorf("handleLinearizationParmDict: corrupt linearization
dict at obj:%d - corrupt array entry H, needs length 2 or 4", objNr)
		}

		offset, ok := a[0].(Integer)
		if !ok {
			return errors.Errorf("handleLinearizationParmDict: corrupt linearization
dict at obj:%d - corrupt array entry H, needs Integer values", objNr)
		}

		offset64 := int64(offset.Value())
		ctx.OffsetPrimaryHintTable = &offset64

		if len(a) == 4 {

			offset, ok := a[2].(Integer)
			if !ok {
				return errors.Errorf("handleLinearizationParmDict: corrupt
linearization dict at obj:%d - corrupt array entry H, needs Integer values", objNr)
			}

			offset64 := int64(offset.Value())
			ctx.OffsetOverflowHintTable = &offset64
		}
	}

	return nil
}

func loadStreamDict(ctx *Context, sd *StreamDict, objNr, genNr int) error {

	var err error

	// Load encoded stream content for stream dicts into xRefTable entry.
	if _, err = loadEncodedStreamContent(ctx, sd); err != nil {
		return errors.Wrapf(err, "dereferenceObject: problem dereferencing stream %d",
objNr)
	}

	ctx.Read.BinaryTotalSize += *sd.StreamLength

	// Decode stream content.
	err = saveDecodedStreamContent(ctx, sd, objNr, genNr, ctx.DecodeAllStreams)

	return err
}

func updateBinaryTotalSize(ctx *Context, o Object) {

	switch o := o.(type) {

	case StreamDict:
		ctx.Read.BinaryTotalSize += *o.StreamLength

```



```go
	case ObjectStreamDict:
		ctx.Read.BinaryTotalSize += *o.StreamLength

	case XRefStreamDict:
		ctx.Read.BinaryTotalSize += *o.StreamLength

	}

}

func dereferenceObject(ctx *Context, objNr int) error {

	xRefTable := ctx.XRefTable
	xRefTableSize := len(xRefTable.Table)

	log.Read.Printf("dereferenceObject: begin, dereferencing object %d\n", objNr)

	entry := xRefTable.Table[objNr]

	if entry.Free {
		log.Read.Printf("free object %d\n", objNr)
		return nil
	}

	if entry.Compressed {
		err := decompressXRefTableEntry(xRefTable, objNr, entry)
		if err != nil {
			return err
		}
		//log.Read.Printf("dereferenceObject: decompressed entry,
Compressed=%v\n%s\n", entry.Compressed, entry.Object)
		return nil
	}

	// entry is in use.
	log.Read.Printf("in use object %d\n", objNr)

	if entry.Offset == nil || *entry.Offset == 0 {
		log.Read.Printf("dereferenceObject: already decompressed or used object w/o
offset -> ignored")
		return nil
	}

	o := entry.Object

	// Already dereferenced stream dict.
	if o != nil {
		logStream(entry.Object)
		updateBinaryTotalSize(ctx, o)
		log.Read.Printf("handleCachedStreamDict: using cached object %d of
%d\n<%s>\n", objNr, xRefTableSize, entry.Object)
		return nil
	}

	// Dereference (load from disk into memory).

	log.Read.Printf("dereferenceObject: dereferencing object %d\n", objNr)

	// Parse object from file: anything goes dict, array, integer, float,
```



```go
streamdicts...
	o, err := ParseObject(ctx, *entry.Offset, objNr, *entry.Generation)
	if err != nil {
		return errors.Wrapf(err, "dereferenceObject: problem dereferencing object %d",
objNr)
	}

	entry.Object = o

	// Linearization dicts are validated and recorded for stats only.
	err = handleLinearizationParmDict(ctx, o, objNr)
	if err != nil {
		return err
	}

	// Handle stream dicts.

	if _, ok := o.(ObjectStreamDict); ok {
		return errors.Errorf("dereferenceObject: object stream should already be
dereferenced at obj:%d", objNr)
	}

	if _, ok := o.(XRefStreamDict); ok {
		return errors.Errorf("dereferenceObject: xref stream should already be
dereferenced at obj:%d", objNr)
	}

	if sd, ok := o.(StreamDict); ok {

		err = loadStreamDict(ctx, &sd, objNr, *entry.Generation)
		if err != nil {
			return err
		}

		entry.Object = sd
	}

	log.Read.Printf("dereferenceObject: end obj %d of %d\n<%s>\n", objNr,
xRefTableSize, entry.Object)

	logStream(entry.Object)

	return nil
}

func processDictRefCounts(xRefTable *XRefTable, d Dict) {
	for _, e := range d {
		switch o1 := e.(type) {
		case IndirectRef:
			entry, ok := xRefTable.FindTableEntryForIndRef(&o1)
			if ok {
				entry.RefCount++
			}
		case Dict:
			processRefCounts(xRefTable, o1)
		case Array:
			processRefCounts(xRefTable, o1)
		}
	}
}
```



```go

func processArrayRefCounts(xRefTable *XRefTable, a Array) {
	for _, e := range a {
		switch o1 := e.(type) {
		case IndirectRef:
			entry, ok := xRefTable.FindTableEntryForIndRef(&o1)
			if ok {
				entry.RefCount++
			}
		case Dict:
			processRefCounts(xRefTable, o1)
		case Array:
			processRefCounts(xRefTable, o1)
		}
	}
}

func processRefCounts(xRefTable *XRefTable, o Object) {

	switch o := o.(type) {
	case Dict:
		processDictRefCounts(xRefTable, o)

	case StreamDict:
		processDictRefCounts(xRefTable, o.Dict)

	case Array:
		processArrayRefCounts(xRefTable, o)
	}
}

// Dereferences all objects including compressed objects from object streams.
func dereferenceObjects(ctx *Context) error {

	log.Read.Println("dereferenceObjects: begin")

	xRefTable := ctx.XRefTable

	// Get sorted slice of object numbers.
	// TODO Skip sorting for performance gain.
	var keys []int
	for k := range xRefTable.Table {
		keys = append(keys, k)
	}
	sort.Ints(keys)

	for _, objNr := range keys {
		err := dereferenceObject(ctx, objNr)
		if err != nil {
			return err
		}
	}

	for _, objNr := range keys {
		entry := xRefTable.Table[objNr]
		if entry.Free || entry.Compressed {
			continue
		}
		processRefCounts(xRefTable, entry.Object)
```



```go
	}

	log.Read.Println("dereferenceObjects: end")

	return nil
}

// Locate a possible Version entry (since V1.4) in the catalog
// and record this as rootVersion (as opposed to headerVersion).
func identifyRootVersion(xRefTable *XRefTable) error {

	log.Read.Println("identifyRootVersion: begin")

	// Try to get Version from Root.
	rootVersionStr, err := xRefTable.ParseRootVersion()
	if err != nil {
		return err
	}

	if rootVersionStr == nil {
		return nil
	}

	// Validate version and save corresponding constant to xRefTable.
	rootVersion, err := PDFVersion(*rootVersionStr)
	if err != nil {
		return errors.Wrapf(err, "identifyRootVersion: unknown PDF Root version:
%s\n", *rootVersionStr)
	}

	xRefTable.RootVersion = &rootVersion

	// since V1.4 the header version may be overridden by a Version entry in the
catalog.
	if *xRefTable.HeaderVersion < V14 {
		log.Info.Printf("identifyRootVersion: PDF version is %s - will ignore root
version: %s\n",
			xRefTable.HeaderVersion, *rootVersionStr)
	}

	log.Read.Println("identifyRootVersion: end")

	return nil
}

// Parse all Objects including stream content from file and save to the corresponding
xRefTableEntries.
// This includes processing of object streams and linearization dicts.
func dereferenceXRefTable(ctx *Context, conf *Configuration) error {

	log.Read.Println("dereferenceXRefTable: begin")

	xRefTable := ctx.XRefTable

	// Note for encrypted files:
	// Mandatory provide userpw to open & display file.
	// Access may be restricted (Decode access privileges).
	// Optionally provide ownerpw in order to gain unrestricted access.
	err := checkForEncryption(ctx)
	if err != nil {
```



```go
		return err
	}
	//fmt.Println("pw authenticated")

	// Prepare decompressed objects.
	err = decodeObjectStreams(ctx)
	if err != nil {
		return err
	}

	// For each xRefTableEntry assign a Object either by parsing from file or pointing
to a decompressed object.
	err = dereferenceObjects(ctx)
	if err != nil {
		return err
	}

	// Identify an optional Version entry in the root object/catalog.
	err = identifyRootVersion(xRefTable)
	if err != nil {
		return err
	}

	log.Read.Println("dereferenceXRefTable: end")

	return nil
}

func handleUnencryptedFile(ctx *Context) error {

	if ctx.Cmd == DECRYPT || ctx.Cmd == SETPERMISSIONS {
		return errors.New("pdfcpu: this file is not encrypted")
	}

	if ctx.Cmd != ENCRYPT {
		return nil
	}

	// Encrypt subcommand found.

	if ctx.OwnerPW == "" {
		return errors.New("pdfcpu: please provide owner password and optional user
password")
	}

	return nil
}

func idBytes(ctx *Context) (id []byte, err error) {

	if ctx.ID == nil {
		return nil, errors.New("pdfcpu: missing ID entry")
	}

	hl, ok := ctx.ID[0].(HexLiteral)
	if ok {
		id, err = hl.Bytes()
		if err != nil {
			return nil, err
```



```go
		}
	} else {
		sl, ok := ctx.ID[0].(StringLiteral)
		if !ok {
			return nil, errors.New("pdfcpu: ID must contain hex literals or string
literals")
		}
		id, err = Unescape(sl.Value())
		if err != nil {
			return nil, err
		}
	}

	return id, nil
}

func needsOwnerAndUserPassword(cmd CommandMode) bool {

	return cmd == CHANGEOPW || cmd == CHANGEUPW || cmd == SETPERMISSIONS
}

func handlePermissions(ctx *Context) error {

	// AES256 Validate permissions
	ok, err := validatePermissions(ctx)
	if err != nil {
		return err
	}

	if !ok {
		return errors.New("pdfcpu: corrupted permissions after upw ok")
	}

	// Double check minimum permissions for pdfcpu processing.
	if !hasNeededPermissions(ctx.Cmd, ctx.E) {
		return errors.New("pdfcpu: insufficient access permissions")
	}

	return nil
}

func setupEncryptionKey(ctx *Context, d Dict) (err error) {

	ctx.E, err = supportedEncryption(ctx, d)
	if err != nil {
		return err
	}

	ctx.E.ID, err = idBytes(ctx)
	if err != nil {
		return err
	}

	var ok bool

	//fmt.Printf("opw: <%s> upw: <%s> \n", ctx.OwnerPW, ctx.UserPW)

	// Validate the owner password aka. permissions/master password.
	ok, err = validateOwnerPassword(ctx)
```



```go
	if err != nil {
		return err
	}

	// If the owner password does not match we generally move on if the user password
is correct
	// unless we need to insist on a correct owner password due to the specific
command in progress.
	if !ok && needsOwnerAndUserPassword(ctx.Cmd) {
		return errors.New("pdfcpu: please provide the owner password with -opw")
	}

	// Generally the owner password, which is also regarded as the master password or
set permissions password
	// is sufficient for moving on. A password change is an exception since it
requires both current passwords.
	if ok && !needsOwnerAndUserPassword(ctx.Cmd) {
		// AES256 Validate permissions
		ok, err = validatePermissions(ctx)
		if err != nil {
			return err
		}
		if !ok {
			return errors.New("pdfcpu: corrupted permissions after opw ok")
		}
		return nil
	}

	// Validate the user password aka. document open password.
	ok, err = validateUserPassword(ctx)
	if err != nil {
		return err
	}
	if !ok {
		return errors.New("pdfcpu: please provide the correct password")
	}

	//fmt.Printf("upw ok: %t\n", ok)

	return handlePermissions(ctx)
}

func checkForEncryption(ctx *Context) error {

	ir := ctx.Encrypt

	if ir == nil {
		// This file is not encrypted.
		return handleUnencryptedFile(ctx)
	}

	// This file is encrypted.
	log.Read.Printf("Encryption: %v\n", ir)

	if ctx.Cmd == ENCRYPT {
		// We want to encrypt this file.
		return errors.New("pdfcpu: this file is already encrypted")
	}

	// Dereference encryptDict.
```



```go
	d, err := dereferencedDict(ctx, ir.ObjectNumber.Value())
	if err != nil {
		return err
	}
	log.Read.Printf("%s\n", d)

	// We need to decrypt this file in order to read it.
	return setupEncryptionKey(ctx, d)
}

```