

The Arlington PDF Model

Overview

This is a DaeDaLus speicfication of the grammar for the Arlington PDF Model. The most recent developments for the model are available from Github:

<https://github.com/pdf-association/arlington-pdf-model>

The model describes a collection of datatypes, each in a separate file. The datatypes provide details and additional constraints about PDF dictionaries and arrays in the PDF specification.

```
def ArrayType      = Datatype ArrayKey
def DictionaryType = Datatype DictionaryKey
```

The name of each file determines if it contains an array or dictionary based type:

ArrayOf*, *Array, *ColorSpace	ArrayType
<i>other file</i>	DictionaryType

The files follow a standard tab-separated format, starting with a header row followed by a sequence of records. Each record contains information about a dictionary key or an array elemnt.

```
def Datatype KeyType =
  block
    Many $[! $recordTerminator]; $recordTerminator -- Skip header
    $$ = Many (Field KeyType)
  END
```

```
def $fieldSeparator = '\t'
def $recordTerminator = '\n'
```

```
def Field KeyType =
  block
    key          = KeyType;           $fieldSeparator
    type         = ReqAlts PrimitiveType; $fieldSeparator
    sinceVersion = Version;           $fieldSeparator
```

```

deprecatedIn      = Optional Version;           $fieldSeparator
required          = IsRequired;                 $fieldSeparator
indirectReference = Alts IsIndirect;            $fieldSeparator
inheritable       = BoolExpr;                  $fieldSeparator
defaultValue      = Alts DefaultValue;         $fieldSeparator
possibleValues    = MultiAlts PossibleValue;   $fieldSeparator
specialCase       = MultiAlts SpecialCase;     $fieldSeparator
link              = MultiAlts Link;            $fieldSeparator
note              = FreeText;                  $recordTerminator

```

Keys

```

def ArrayKey =
  block
    Array = FreeText

def DictionaryKey =
  block
    DictionaryKey = FreeText

```

Alternatives

```

def ReqAlts P = SepBy (KW ";") (Versioned P)

def Alts P =
  First
    SepBy (KW ";") (Bracketed (Optional (Versioned P))) -- must be first
    [ Optional (Versioned P) ]

def MultiAlts P =
  Optional (SepBy (KW ";")
    (Bracketed (Optional (SepBy (KW ",")
      (Versioned P))))))

```

Versions

PDF versions are specified as major and minor number separated by a .:

```

def Version =
  block
    major = Natural
    $['.']
    minor = Natural

```

Field components may be annotated with version dependent information:

```

def WithVersion P =

```

```

First
  Deprecated    = FnDeprecated (Versioned P)
  SinceVersion  = FnSinceVersion (Versioned P)
  BeforeVersion = FnBeforeVersion (Versioned P)
  IsVersion     = FnIsPDFVersion (Versioned P)

def Versioned P =
  First
    Versioned = WithVersion P
    Value     = P

```

Primitive Types

```

def PrimitiveType =
  First
    TArray          = @Match "array"
    TBitmask        = @Match "bitmask"
    TBoolean        = @Match "boolean"
    TDate           = @Match "date"
    TDictionary     = @Match "dictionary"
    TInteger        = @Match "integer"
    TMap            = @Match "matrix"
    TNameTree       = @Match "name-tree"
    TName           = @Match "name"
    TNull           = @Match "null"
    TNumberTree     = @Match "number-tree"
    TNumber         = @Match "number"
    TRectangle      = @Match "rectangle"
    TStream         = @Match "stream"
    TStringASCII    = @Match "string-ascii"
    TStringByte     = @Match "string-byte"
    TStringText     = @Match "string-text"
    TString         = @Match "string"

```

Required Fields

```

def IsRequired : BoolExpr = FnIsRequired <| BoolExpr

```

Direct Fields

This field component specifies if the field value may be a link, a non-link value, or if it could be either. Note that since there are three possible outcomes, this is not a boolean value.

```

def IsIndirect =
  First
    IndirectIf = BoolExpr

```

```

DirectIf    = {| TRUE = Fun0 "MustBeDirect" |}
DirectIf    = Fun1 "MustBeDirect" BoolExpr
IndirectIf  = Fun1 "MustBeIndirect" BoolExpr

```

Default Values

```

def DefaultValue =
  First
    ImplementationDependent = Fun0 "ImplementationDependent"
    Conditional              = ConditionalDefaultCases
    Value                   = Term

def ConditionalDefaultCases =
  First
    FnEval (SepBy (KW "||") ConditionalDefault)

  [ block
    let f = Fun2 "IsPresent" FieldName Term
    condition = {| IsPresent = f.arg1 |} : BoolExpr
    value      = f.arg2
  ]

def ConditionalDefault =
  block
    let f = Fun2 "DefaultValue" BoolExpr Term
    condition = f.arg1
    value      = f.arg2

```

Possible Values

```

def PossibleValue =
  First
    Conditional  = ConditionalValue
    Constraint   = BoolExpr
    Value        = Term
    Wild         = KW "*"

def ConditionalValue =
  block
    let f = Fun2 "RequiredValue" BoolExpr Term
    condition = f.arg1
    value      = f.arg2

```

Special Checks

```

def SpecialCase =

```

```

First
  Eval          = FnEval (SepBy (KW "&&") SpecialCase)
  Ignore0       = Fun0 "Ignore"
  Ignore        = Fun1 "Ignore" BoolExpr
  IgnoreF       = Fun1 "Ignore" FieldName
  Meaningful    = Fun1 "IsMeaningful" BoolExpr
  NoCycle       = Fun0 "NoCycle"
  NotPresentIf  = Fun1 "Not" (Fun1 "IsPresent" BoolExpr)
  NotRequiredIf = Fun1 "Not" FnIsRequired
  MustbeDirect  = Fun1 "MustBeDirect" FieldName
  Constraint    = BoolExpr

```

Links

```
def Link = Many (1..) $[ $alpha, $digit, '_' ]
```

Notes

```
def FreeText = Many $[! ($fieldSeparator | $recordTerminator)]
```

Boolean Expressions

```

def BoolExpr =
  block
    First
      {| OR = SepBy2 (KW "||") BoolAtomExpr |}
      {| AND = SepBy2 (KW "&&") BoolAtomExpr |}
      BoolAtomExpr

def BoolAtomExpr : BoolExpr =
  First

  block
    KW "("
    $$ = BoolExpr
    KW ")"

  FnEval BoolExpr

{| TRUE          = KW "TRUE" |}
{| FALSE        = KW "FALSE" |}
{| NOT          = Fun1 "Not" BoolExpr |}

{| FontHasLatinChars = Fun0 "FontHasLatinChars" |}
{| NotStandard14Font = Fun0 "NotStandard14Font" |}
{| KeyNameIsColorant = Fun0 "KeyNameIsColorant" |}

```

```

{| IsAssociatedFile          = Fun0 "IsAssociatedFile" |}
{| IsPDFTagged              = Fun0 "IsPDFTagged" |}
{| IsEncryptedWrapper       = Fun0 "IsEncryptedWrapper" |}
{| PageContainsStructContentItems = Fun0 "PageContainsStructContentItems" |}
{| ImageIsStructContentItem  = Fun0 "ImageIsStructContentItem" |}

{| IsPresent                = Fun1 "IsPresent" FieldName |}

{| InMap                    = Fun1 "InMap" FieldName |}
{| Contains                 = Fun2 "Contains" Term Term |}
{| ArraySortAscending       = Fun2 "ArraySortAscending" Term Term |}

{| Versioned                = WithVersion BoolExpr |}

{| IsAtLeastVersion         = Fun1 "SinceVersion" Version |}
{| IsBeforeVersion          = Fun1 "BeforeVersion" Version |}
{| IsPDFVersion             = Fun1 "IsPDFVersion" Version |}

{| BitClear                 = Fun1 "BitClear" Term |}
{| BitsClear                = Fun2 "BitsClear" Term Term |}
{| BitSet                   = Fun1 "BitSet" Term |}
{| BitsSet                  = Fun2 "BitsSet" Term Term |}

{| IsLastInNumberFormatArray = Fun1 "IsLastInNumberFormatArray" Term |}

{| EQ                       = BinOp "==" Term |}
{| NEQ                      = BinOp "!=" Term |}
{| LT                       = BinOp "<" Term |}
{| GT                       = BinOp ">" Term |}
{| LEQ                      = BinOp "<=" Term |}
{| GEQ                      = BinOp ">=" Term |}

```

Values

```

def Term =
  First
    {| add = BinOp "+" TermProduct |}
    {| sub = BinOp "-" TermProduct |}
    -- the space is to avoid conflict with names like a-b (hack)
  TermProduct

def TermProduct : Term =
  First
    {| mul = BinOp "*" TermAtom |}
    {| mod = BinOp "mod" TermAtom |}

```

```

TermAtom

def TermAtom : Term =
  First
  block
    KW "("
    $$ = Term
    KW ")"

    {| ValueOf          = ValueOf |}

    {| Float            = FloatValue |}
    {| Integer          = IntegerValue |}    -- Must be after Float
    {| String           = StringValue |}
    {| Bool             = BoolValue |}
    {| Null             = NullValue |}
    {| Array            = ArrayValue |}

    {| RectWidth        = Fun1 "RectWidth" Term |}
    {| RectHeight       = Fun1 "RectHeight" Term |}
    {| FileSize         = Fun0 "FileSize" |}
    {| ArrayLengthField = Fun1 "ArrayLength" FieldName |} -- ?
    {| ArrayLength      = Fun1 "ArrayLength" Term |}
    {| PageProperty     = Fun2 "PageProperty" Term FieldName |}
    {| StringLength     = Fun1 "StringLength" Term |}
    {| StreamLength     = Fun1 "StreamLength" Term |}
    {| NumberOfPages    = Fun0 "NumberOfPages" |}

    {| Name             = NameValue |}
    -- Needs to be after the functions, float, integer, bool, null

def Natural =
  block
    $$ = many (s = Digit) (10 * s + Digit)
    -- can't be followed by a letter
    case Optional $alpha of
      just    -> Fail "Expected number, found identifier"
      nothing -> Accept

def IntegerValue =
  Token
  First
  { $['-']; - Natural }
  Natural

```

```

{- Leave as text for now. We could parse this as double
but some of the literals (e.g. 1.2) are not exactly representable,
so they might print funny, although likely not due to rounding. -}
def FloatValue =
  Token
  block
    whole = Many (1..) Digit
    $['.']
    frac = Many (1..) Digit

def StringValue =
  block
    $['\\'']
    $$ = Many $['\\'']
    KW ""

def BoolValue =
  First
    { KW "true"; true }
    { KW "false"; false }

def NullValue = KW "null"

def ArrayValue =
  block
    KW "["
    $$ = Many Term
    KW "]"

def NameValue = Token (Many (1 .. ) $[ $alpha, $digit, '.', '_', '-'])

```

Field Names

```

def SimpleFieldName =
  First
    Text = NameValue
    Wild = $['*']

def FieldName = SepBy (Match "::") SimpleFieldName

def ValueOf =
  block
    qualifier = Optional { $$ = FieldName; Match "::" }
    $['@']
    field = SimpleFieldName

```


Functions

```
def Fun0 f =
  block
    Match "fn:"
    Match f
    KW "("
    KW ")"
    Accept

def Fun1 f Arg =
  block
    Match "fn:"
    Match f
    KW "("
    $$ = Arg
    KW ")"

def Fun2 f Arg1 Arg2 =
  block
    Match "fn:"
    Match f
    KW "("
    arg1 = Arg1
    KW ","
    arg2 = Arg2
    KW ")"

def versioned (f : Fun2) = { version = f.arg1 : Version, value = f.arg2 }

def FnSinceVersion Arg      = versioned (Fun2 "SinceVersion" Version Arg)
def FnBeforeVersion Arg     = versioned (Fun2 "BeforeVersion" Version Arg)
def FnDeprecated Arg       = versioned (Fun2 "Deprecated" Version Arg)
def FnIsPDFVersion Arg      = versioned (Fun2 "IsPDFVersion" Version Arg)

def FnEval Arg              = Fun1 "Eval" Arg

def FnIsRequired             = Fun1 "IsRequired" BoolExpr
```

Lexical and Utilities

Basic character classes:

```
def $alpha      = 'a' .. 'z' | 'A' .. 'Z'
def $digit      = '0' .. '9'
```

A parser for a single digit. The semantic value is the value corresponds to the

value of the digit, rather than the ASCII code of the character:

```
def Digit          = $digit - '0' as int
```

A *token* is a parser, followed by optional space:

```
def Token P =  
  block  
    $$ = P  
    Many $[' ']
```

A *keyword* is a token that matches a specific string:

```
def KW x = Token (@Match x)
```

Parse something enclosed in square brackets:

```
def Bracketed P =  
  block  
    KW "["  
    $$ = P  
    KW "]"
```

The following parsers are for sequences of **Thing** separated by **Sep**. **SepBy** parser a sequence of one or more elements, while **SeqBy2** parsers a sequence of at least two elements.

```
def SepBy Sep Thing =  
  build (many (s = emit builder Thing) { Sep; emit s Thing })
```

```
def SepBy2 Sep Thing =  
  block  
    let first  = emit builder Thing  
    let second = emit first { Sep; Thing }  
    let rest   = many (s = second) (emit s { Sep; Thing })  
    build rest
```

A helper for parsing infix operators.

```
def BinOp op P =  
  block  
    lhs = P  
    KW op  
    rhs = P
```