An Example LysKOM Client

Hugo Hörnquist (hugo)
hugo@lysator.liu.se

November 27, 2017

Contents

Chapter 1

Introduction

The goal of this document is to create a simple to understand (but not use) client for LysKOM protocol A. It starts with a BNF (Backus Naur Form) specification for the protocol, and ends with a complete terminal client.

The section Helpers define some stuff which is used in the whole document. You might want to keep a tab to there just in case.

It's probably safe to jump directly to chapter 4 if you only want information about handling incomming and outgoing data, and not the code for automatically generating a parser from a BNF specification.

1.1 Practical

This document is written in literate Haskell, appendix **??** details the main method. Note that the source in the document can't directly be compiled. Mostly due to imports being commented out from the TEX code.

The most up to date version of this document, along with its source code, can be found at `https://git.lysator.liu.se/hugo/hskom`.

Chapter 2

From BNF to AST

What we had from the outset was an info page detailing Protocol A, as well as a BNF (Backus Naur Form) specification on the protocol. Unfortunately it was written in the variant ASN.1 (Abstract Syntax Notation One) which doesn't seem to be to supported. It was also not properly implemented because he who wrote it lacked the ASN.1 specification when writing the Protocol A specification.

See appendix B for sample output for all parsers defined in this chapter.

2.1 AST creation

Our first goal is therefor to parse the BNF file into an AST (Abstract Syntax Tree) that Haskell understands.

In the following sections a bunch of sub parsers for different parts of the BNF. What binds them together and allows a complete syntax tree to be created is however this.

TODO

Fix E type and requestMetaParser

```
data AllTypes = A RPC
              | B TypeBinding
              | C String -- Comment
              | D (String, String) -- Alias
              | E () -- Request & Async meta-info
              deriving (Show)

documentParser :: GenParser Char () [AllTypes]
documentParser = do
    many $ withWS
         ( C <$> commentParser
       <|> E <$> try requestMetaParser
       <|> D <$> try versionParser
       <|> D <$> try typeAliasParser
       <|> A <$> try rpcParser
       <|> B <$> try bindingParser
```

```
<?> "Any valid BNF object")
```

2.2 Types & Bindings

> Data fields have been given names that start with a lower-case letter.
>
> Fundamental data type hve names in all-caps.
>
> Derived data types have names that start with an upper-case letter.
>
> — Info Page

TODO

- Refer to where value parsers for these types can be found. Ought to be chapter 4
- We still lack forms for ‘ENUMERATION-OF’ & ‘|’!
- We should also look over if we *really* needs both LysType and TypeField

---

A LysType our notation for our different “core” types, which all in some way hold a TypeField.

```
data LysType = SimpleType     TypeField
             | BitstringType [TypeField]
             | EnumType      [TypeField]
             | StructureType [TypeField]
             | SelectionType [TypeField]
             | ArrayType      TypeField
             deriving (Show)
```

A type field holds all the data we have about a type, including nested types. And is represented as:

```
data TypeField = OnlyType String
               | BitStringField String
               | StructureField TypeBinding
               | EnumField Int String
               | SelectionField Int String TypeBinding
               | ArrayField TypeField
               deriving (Show)
```

2.2.1 Parsing Types

Types come in a few forms. But they all share similarities.

First out are the simple primitive types. These are INT32, INT16, INT8, BOOL, FLOAT & HOLLERITH. They all represent a number of different (obvious) types, except for HOLLERITH which is a string (see section 4.1.5). They can all be parsed with:

```
typeWordParser :: GenParser Char () TypeField
typeWordParser = OnlyType <$> word
```

2.2.1.1 Arrays

Arrays are a simple type which represents a list of one type. In the specification they are written as

```
ARRAY <type>
```

Where <type> is which types they hold.

```
arrayParser :: GenParser Char () TypeField
arrayParser =  ArrayField
           <$> (string "ARRAY"
            *> whitespaces
            *> typeWordParser)
```

We then have the structure types BISTRINGS, ENUMERATIONS, SELECTIONS, & STRUCTURES.

They all have in common that the handle a list of declarations, surrounded by parenthesis. Therefore we start by creating a general listParser; Which takes a parser for each field and returns a list of fields.

```
listParser :: GenParser Char () TypeField → GenParser Char () [TypeField]
listParser fieldParser
    =  withDelim' "()" (many $ withWS fieldParser)
```

Many of them also have a name before the their list. This parsers takes a parser for the actual list, and checks if a set string appears before it.

```
specialTypeParser
    :: String
    → GenParser Char () [TypeField]
    → GenParser Char () [TypeField]
specialTypeParser str subParser =
    string str *> whitespaces *> subParser
```

2.2.1.2 Bitstring

Bitstrings are the simplest type. They represent a number of bits. A sample bistring structure in the BNF could look like:

```
BITSTRING ( name;
            other-name;
          )
```

Where both ‘name’ and ‘other-name’ declare one bit filed each, and each tell what that field should contain. It also implies that this specific bitstring holds exactly two bits, since it has two fields.

```
bitstringFieldParser :: GenParser Char () TypeField
bitstringFieldParser =  BitStringField <$> word
                                       <*  whitespaces
                                       <*  char ';'

bitstringParser = specialTypeParser "BITSTRING" (listParser bitstringFieldParser)
```

2.2.1.3 Selections

TODO
understand Selections

If I understand correctly selections are a form of type unions. [1] They declare a <name> <n> mapping which is used for specifing which type <type> will be used. I don't know why the <tail> field extists, but it creates a secound name.

```
selection (
    <n> = <name>   <tail> : <type>;
)
```

```
selectionFieldParser :: GenParser Char () TypeField
selectionFieldParser
     =  SelectionField
    <$> (intParser <* char '=')
    <*> (withWS word)
    <*> (bindingParser <* whitespaces <* char ';')

selectionParser = specialTypeParser "SELECTION" (listParser selectionFieldParser)
```

2.2.1.4 Enemurations

An enumeration works just as expected. It declares a number of symbols, as well as integer representations for them all.

An example BNF of it would be:

```
ENUMERATION ( name  = 1;
              other = 2;
            )
```

```
enumFieldParser :: GenParser Char () TypeField
enumFieldParser =  flip EnumField
              <$> word
              <*> withDelim' "=;" intParser
```

They can also appear on the form

```
ENUMERATION-OF (<selection-type>)
```

Which builds an enumeration from the <n> and <name> field in a selection. See above.

The parser for the BNF here would be [2]

```
-- enumSelectionParser :: GenParser Char () TypeField
-- enumSelectionParser =  SelectEnumField
--                  <$> word "ENUMERATION-OF"
--                   *> withDelim "()" (  typeWordParser
--                                    <|> selectionParser
--                                    <?> "Selection")
```

[1] Please correct me
[2] The parser might work. But I can't figure out the types for it.

```
enumParser =  specialTypeParser "ENUMERATION" (listParser enumFieldParser)
         -- <|> enumSelectionParser
         -- <?> "BNF Enum declaration"
```

TODO
move the following somewhere else.

On evaluating the inner selection is expanded, and bound translated to an enum with.

```
-- makeEnum :: LysType → LysType
-- makeEnum (SelectionType []) = []
-- makeEnum (SelectionType (s:xs))
--     = EnumField n name : makeEnum xs
--     where (SelectionField n name _ _) = s
```

2.2.1.5 Structures

A sturcture is just a simple compound data type, on the form

```
( field-name : TYPE;
  other-name : TYPE;
)
```

Note here that with my implementation the last semicolon is optional. This is to better work with how the BNF specifies RPC requests (see section 2.3)

```
structFieldParser :: GenParser Char () TypeField
structFieldParser =  StructureField
                 <$> bindingParser
                 <*  whitespaces
                 <*  (  char ';'
                    <|> lookAhead (char ')')
                    <?> "Struct Field End")
```

The reason for the `lookahead (char ')')` in the above code is since we want to check for the structure ending, but don't consume it. Since the actual parsing of the surrounding parenthisis is done in the "listParser".

```
structParser = listParser structFieldParser
```

We can now create a general type parser, which juts binds all our above parsers into one. It also shows that the reason for LysType existing besides TypeField was so that we didn't have to wory about a type being single or multiple.

```
typeParser :: GenParser Char () LysType
typeParser =  (BitstringType <$> try bitstringParser)
         <|> (     EnumType <$> try enumParser)
         <|> (SelectionType <$> try selectionParser)
         <|> (    ArrayType <$> try arrayParser)
         <|> (StructureType <$> try structParser)
         <|> (   SimpleType <$> typeWordParser)
         <?> "LysType"
```

2.2.2 Bindings

Now that we finanly can parse all our types it's time to bind them to new symbols. Some type bindings exists in the code above because multiple of the compound types names fields with specific types.

The BNF syntax for creating a top level binding is

```
<name> ::= <type>
```

which we can represent in Haskell as

```
data TypeBinding = TypeBinding String LysType
                 deriving (Show)
```

where the String is the <name> and LysType is the <type>.

Finally we need a parser for it:

```
bindingParser :: GenParser Char () TypeBinding
bindingParser
     =  TypeBinding
    <$> (word <* withWS (try (string "::=")
                         <|> string ":"
                         <?> "Name Type Separator"))
    <*> typeParser
```

2.2.3 Extra Helpers

TODO

These are here, they should maybe be moved.

```
maybeTypeParser    = option Nothing $ Just <$> typeParser

maybeBindingParser = option Nothing $ Just <$> bindingParser
```

2.3 RPC & Async

TODO

1. Async is for data from server only.
2. Detail how requests and Async differ
3. update the whole section with Async stuff
4. Figure out if <REQUEST> is any type or just a derived type.

RPC (Remote Procedure Call) notation is how calls to the server are specified.

The documenation is a bit confusing on this part, but I will assume that a each call is defined as

```
<CALL> [<N>] ( <REQUEST> ) -> ( <REPLY> ) ;
```

Where <CALL> is the name of the request, <N> is the actuall number sent to the server. The whole square bracket block might actually officialy not be part of the notation. But since the notation file I have includes it I will include it here.

<REQUEST> is a DerivedType, but without a name.

TODO
change Notation so that DerivedType is split into an LValue and an RValue.

Finally <REPLY> appears either a Structure of bindings, or a single binding.

Note that both requests and async (something) uses RPC. However with the difference that the async requests don't specify a reply. NOTE I currently haven't read the documentation for async responses.

2.3.1 Parser

We first declare a simple datatype for RPC rules:

```
data RPC = Request String Int (Maybe LysType) (Maybe LysType)
         | Async String Int (Maybe LysType)
         deriving (Show)
```

Where the String is the name of the call, the Int is the call number. And the two typefields are the requested and returned data, respectively.

An parser for RPC call specifications should therefor look something like:

```
rpcParser :: GenParser Char () RPC
rpcParser = do
    name ← word
    number ← withDelim' "[]" intParser
    request ← withDelim' "()"
               (Just ∘ StructureType ∘ (:[]) ∘ StructureField <$> bindingParser
           <|> Just <$> typeParser
           <|> lookAhead (whitespaces *> char ')') *> pure Nothing
           <?> "proper rpc request body")
    resp ← optionMaybe $
                string "→" *>
                withDelim' "()" maybeTypeParser
                <* char ';'
    return $ maybe (Async   name number request)
                   (Request name number request)
                   resp
```

In actual application an RPC object is always sent with a call- number before it. This number is an identificator of the request, and the response is returned with the same number before it. There's no need for a parser with the syntax "<N> <RPC>", since that never appears (and shouldn't apper) in the specification BNF. An incommind parser is however needed, and can be implemented as following:

```
responseParser :: GenParser Char () (Int, TypeField)
responseParser = liftA2 (,) intParser (whitespace *> valueParser)
```

Where ‘valueParser’ is a parser built from a ‘typeParser’, to parse incomming messages of that type. The code for creating the valueParser’s is however not written yet.

Also, the response parser might actually belong in ‘Datatype’, but that will be fixed once I actually get some structure.

2.3.2 Parser Meta

In the specification all BNC parsers are preceeded by some meta information on the form

```
%Request: <N>
    %name: <name>
    %Protocal version: <M>
    %Status: <status>
%End Request
```

which I think is either for human comsumption, or to be auto included in documentation files. Anyway, the values are <N> for request number. This is the same as what iss inside the square brackes in the BNC. <name> is the call name, which is also in the BNC. <M> and <status> is the update status within the protocol, where they show in which version it was introduced, along with if it still should be fine to use.

This part is currently not parsed, but a parser would be easy to write. [3]

2.4 Other

These are some extra parsers that are needed to parse the entire BNF document. Most of these will probably be moved some place else either, and possibly also get better versions.

2.4.1 Comments

A comment can begin anywhere on a line, and is defined as the text between an octothorpe and the end of the line.

```
commentParser :: GenParser Char () String
commentParser = (char '#')
             *> whitespaces
             *> manyTill anyChar eol
```

2.4.2 Meta (‘%’) commands

There are multiple types of parse commands which starts with an percent signs. They all follow the syntax

```
%key value ...
```

But in different ways.

[3] This is left as an exercise for the reader.

2.4.2.1 Version

Version numbers are presented with a number of specific strings as keys. Followed by a versino number as its value.

NOTE The word parser for the version number really should be replaced with a version number parser.

```
versionParser :: GenParser Char () (String, String)
versionParser =
    liftA2 (,)
        (char '%' *> (  try (string "PROTOEDITION")
                    <|>     string "PROTOVER"
                    <|>     string "LYSKOMDVERSION"
                    <?> "Protocol Version Name"))
        (whitespaces *> word <* eol)
```

2.4.2.2 Aliases

TODO

1. Explain aliases properly,
2. why they are there,
3. how they work,
4. and which different types there are

A type alias is simmilar, except it has two symbols after the string the key "type-alias". Where the first is the new name and the secound is the aliased name. Should work exactly like a c preprocessor define.

There also exists request-alias's and async-alias, this parser handles them all, but currently throws away which type of alias it is.

```
typeAliasParser :: GenParser Char () (String, String)
typeAliasParser =
    liftA2 (,)
        ((char '%') *> many letter
                    *> string "-alias"
                    *> whitespaces
                    *> word)
        (whitespaces *> word <* eol)
```

TODO

Rest of section should be moved to the RPC section

Requests and Async operations come with some meta information before them. The information is formatted with a start of either the key "Request:" or "Async:", follewed by the intenal numerical representation for it.

After that follows a number of (optionally?) indented percent rule lines. Finally the information ends with another command with "End" as the key and "Request" or "Async" as the value.

This parser reads the data and throws it out. It REALLY should be replaced. But is currently here since I want to be able to parse an entire BNF document.

```
requestMetaParser :: GenParser Char () ()
requestMetaParser = do
    char '%'
    type_ ← (string "Request" <|> string "Async")
    --manyTill (try anyChar) (string "%END" *> whitespaces *> string type_)
    manyTill anyChar (try $ string "%End" *> whitespaces *> string type_)
    return ()
```

Chapter 3

From AST to Haskell

This file is about how we best convert between our BNF and Haskell syntax.

TODO
propper scheme for (BNF-name -> Haskell-name)

3.1 Integers

```
type BOOL  = Bool
type INT8  = Int
type INT16 = Int
type INT32 = int
```

3.2 Floating point numbers

TODO
come on

3.3 Strings

TODO
this whole section is coppied verbatim from Datatype.lhs

Hollerith's are a type of strings defined as: <num>H<str> where <num> is the number of bytes in the string <str>.

The following parser works by reading any number of characters which isn't 'H', this should possibly instead be any number of digits. It then reads the literal letter 'H', followed by a string of length <num>, containing any characters.

NOTE that this uses haskell characters, while LysKOM expects a bytestring.

```
data Hollerith = Hollerith Int String

-- Replace this with own class
instance Show Hollerith where
    show (Hollerith size str) = show size ++ "H" ++ str

hollerithParser :: GenParser Char () Hollerith
hollerithParser = do
    num ← intParser <* char 'H'
    str ← count num anyChar
    return $ Hollerith num str
```

3.4 Bitstrings

```
BitName ::= BITSTRING ( a; b; )
```

```
type BitName = (BOOL,BOOL)
```

3.5 Enumerations

```
NewEnum ::= enumeration
           ( 0 = recept  ;
             1 = cc-rept ;
             2 = comm-to ;
             9 = sent-at ;
           )
```

```
data NewEnum = Recept
             | Cc_rept
             | Comm_to
             | Sent_at
             deriving (Show)

instance Enum NewEnum where
    fromEnum = fromJust ∘ flip lookup table
    toEnum = fromJust ∘ flip lookup (map swap table)
table = [(Recept, 0), (Cc_rept, 1), (Comm_to, 2), (Sent_at, 9)]

instance Enum NewEnum where
    fromEnum Recept   = 0
    fromEnum Cc_rept  = 1
    fromEnum Comm_to  = 2
    fromEnum Sent_at  = 9

    toEnum 0 = Recept
    toEnum 1 = Cc_rept
    toEnum 2 = Comm_to
    toEnum 9 = Sent_at
```

3.6 Arrays

```
data Array a = Array Int [a]
             deriving (Show)

-- TODO replace with own class
instance Show Array where
    show (Array n []) = show n ++ " *"
    show (Array n xs) = show n ++ " { " ++ show' xs ++ "}"
```

3.7 Selections

TODO

```
SELECTION (
    N=NAME TAIL : TYPE ;
    N=NAME TAIL : TYPE ;
)
```

3.8 RPC

```
RPC (
    CALL [N] ( REQUEST ) -> ( REPLY ) ;
)
```

```
Call :: Request → RPC Reply
```

```
create-anonymous-text-10 [87] (( text       : HOLLERITH;
                                 misc-info  : ARRAY Misc-Info-1;
                                 aux-items  : ARRAY Aux-Item-Input-10 ))
    -> ( Text-No-1 );
```

```
data RPC a = RPC Int a deriving (Show)

create_anonymous_text_10 :: HOLLERITH
                         → ARRAY Misc_Info_1
                         → ARRAY Aux_Item_Input_10
                         → RPC Text_No_1

create_anonymous_text_10 text misc_info aux_items =
    let str =
        show n ++ " 87 " ++ show' text
               ++ " "    ++ show' misc_info
               ++ " "    ++ show aux_items
     in RPC n str
     where n = next_num
-- RPC needs to be a monad
```

3.9 Structures

```
NewType ::= ( a : INT32; b : BOOL; )
```

```
type NewType = (INT32, BOOL)
```

I think this one is the best bet. It's an actual datatypes so I can do more with it. And it doesn't introduce getter functions. Which would have created namespaces conflicts.

```
data NewType = NewType INT32, BOOL deriving (Show)
```

One other possibility would be to create records. And then have them each in a sepparate namespace. Either with the help of modules. Or by prefixing each name with some kind of string.

```
data NewType = NewType
             { a :: INT32
             , b :: BOOL
             } deriving (Show)
```

Chapter 4

From Incomming Message to Haskell

Everything so far has simple been about parsing a BNF file, and generating Haskell code from it. Now we start actually looking towards actual data!

4.1 Primitive Datatypes

This section details the primitive datatypes of the protocol.

It also declares parsers to go from incomming data to haskell values.

TODO
Refer to "Parsing Types"

4.1.1 Integers

Integers are represented as base 10 ASCII strings, and have the variations INT32, INT16, INT8, & BOOL. Which contain 32, 16, 8, & 1 Bits of data respectivly.

For the 32, 16, & 8 variants a simple read can be used, note however that a check that it isn't larger than expected might be a good thing we have.

My current parser is simple to take read a series of digits. Note that this never checks if the number is the correct size

```
-- intParser was moved to Helpers.lhs
-- because int's needed to be parsed at quite a few places.
```

And while booleans could have the same parser, we might as well also run it through 'toEnum' to get a haskell boolean to work with.

```
boolParser :: GenParser Char () Bool
boolParser = (return $ toEnum ∘ digitToInt)
        <*> oneOf ['0', '1']
```

Floats works like integers, except they have a decimal place. For the time being, refer to C's 'printf("%g", val);' for formatting of floats.

I belive that floats follows the regex rule

```
[+-]?(\d+|\d*\.\d+)([eE](-|+?)\d+)
```

which isn't the easiest to express in parsec.

4.1.2 Bit Strings

Sequence of ASCII '0' & '1', behaves like many boolean values packed together without spaces between. The number of expected values are showed in the specification, as well as what each bit represents.

```
bitstringParser :: GenParser Char () [Bool]
bitstringParser = many boolParser
```

4.1.3 Enumerations

Enum values are represented as INT32, and the actual enum names are presented in the BNF.

The notation parser contains a parser for parsing enumeration deffinitions. There is however currently no way to actually go between the enumeration value and an integer. This will be solved once I can build parsers from the BNF.

4.1.4 Arrays

Arrays are generic types for storing items. And require a type when they are declared. The form for arrays is "<N> { <elem> <elem> ... }", where <N> is the length of the array, and each <elem> is an item of the type of the array.

The empty array is represented as either "0 *" or "0 { }". Note that the client must always send empty arrays using the secound variant. And that the server probably always use the first form. The secound form can also be transmitted by the server if the client requests the size of an array without its contents.

Note that if an array without a body, for example "19 *" is parsed then the value returned is "Right (19, [])". It's therefor up to the program to understand that the empty array here is not an error, but rather what's requested. This could be solved by wrapping the list in Maybe. But that would come at the expense of another nested monad.

Currently all sub arrays need to have the same type. I don't believe that protocol A ever uses nested arrays directly, but if it does this has to be updated.

```
emptyArrayParser = char '*' *> return []
asListBody = between (string "{ ") (char '}')

arrayParser :: GenParser Char () a → GenParser Char () (Int, [a])
arrayParser subParser = do
    len ← intParser <* space
    items  ← emptyArrayParser
          <|> asListBody (count len $ (subParser <* space))
          <?> "LysKOM Array"
    return (len, items)
```

4.1.5 Hollerith

Hollerith's are a type of strings defined as: <num>H<str> where <num> is the number of bytes in the string <str>.

The following parser works by reading any number of characters which isn't 'H', this should possibly instead be any number of digits. It then reads the literal letter 'H', followed by a string of length <num>, containing any characters.

NOTE that this uses haskell characters, while LysKOM expects a bytestring.

```
data Hollerith = Hollerith Int String
instance Show Hollerith where
    show (Hollerith size str) = show size ++ "H" ++ str

hollerithParser :: GenParser Char () Hollerith
hollerithParser = do
    num ← intParser <* char 'H'
    str ← count num anyChar
    return $ Hollerith num str
```

TODO

everything in this section from here on should *probably* be moved to the Temlate chapter

---

4.1.6 Structure

One of the most common compound data types is the structure.

A structure is on the form

```
( name : type ;
  name : type ; )
```

TODO

MORE, also, last ';' optional

4.1.7 Selections

If my understanding is correct selections is a form of union types.

A selection is on the form

```
<name> ::= SELECTION (
    <n>=<name>      <fieldName> : <type>;
)
```

Where <n> is a number, and messages over the protocol sends this number telling which instance of the selection is used.

<fieldName> is usually similar to <name>, according to the documentation. I however don't know why there are two different fields. Or why one should be used over the other.

The example of a SELECTION given in the documentation is

```
description ::= SELECTION (
    1=name           the_name : HOLLERITH;
    2=age            years    : INT32;
)
```

And then notes that "two legal messages of the type ‘description’ are ‘1 4HJohn’ and ‘2 18’."

Ideally, each SELECTION type should have its own parser defined, Which probably should return a ‘data’ type which instance is the <name>, and which value is of type <type>. Throwing away <fieldName> and having <n> implicit as a derivition from Enum.

The sample implementatino for the above mention ‘descrption’ type should therefor be

```
data Description
    = Name Hollerith
    | Years Int
    deriving (Show)
```

TODO
the above declaration should probably not actually be code which is run.

4.2 Notes

This file specifies some information about how to implement different datatypes. It shouldn’t really be run in its current form.

TODO
Most of this section should probably be spliced into other sections.

4.2.1 RPC request

```
requestName1 [] ( name : type ) -> ( )
requestName2 (( n1 : t1 ; n2 : t2 )) -> ( )

requestName3 ( name : type ) -> ( type )
requestName4 (( n1 : t1 ; n2 : t2 )) -> ( type2 )
```

```
requestName1 :: type → Request ()
requestName2 :: t1 → t2 → Request ()
requestName3 :: type → Request type
requestName4 :: t1 → t2 → Request type2
```

4.2.2 Types

Some information about how I plan to convert from LysKOM types to Haskell types.

```
Aux-Item-10 ::=
      ( aux-no                : Aux-No-10;
        tag                   : INT32;
        creator               : Pers-No-1;
        created-at            : Time-1;
        flags                 : Aux-Item-Flags-10;
        inherit-limit         : INT32;
        data                  : HOLLERITH;
      )
```

```
data Aux_Item_10
   = Aux_Item_10
         Aux_No_10
         INT32
         Pers_No_1
         Time_1
         Aux_Item_Flags_10
         INT32
         HOLLERITH
   deriving (Show)
```

Note that I preferably don't want to derive 'Show', but rather write my own which creates a string on the form to be sent over the line.

```
data BITSTRING = BITSTRING [BOOL]
instance Show BITSTRING where
    show (BITSTRING xs) = mconcat $ show <$> xs
```

HOLLERITH already in other file.

```
typename ::= ENUMERATION (
    a=1;
    b=2
)
```

```
data TypeName = A | B deriving (Enum)
```

```
ARRAY Aux-Item-10
```

```
type (ARRAY a) = [a]
```

4.2.3 Aliases

How does type aliases actually work?

```
%type-alias <new-name> <old-name>
```

```
type <new-name> = <old-name>
```

```
%request-alias
%async-alias
```

4.2.4 Types

```
BITSTRING ( a; b; c; )
```

```
data Bitstring = ()
```

Chapter 5

Binding it Together

Appendix A

Final Code

A.1 Helpers

We need some extra functions for making everything else go together. This includes extra parsers, a main method, and code for other things.

We start with some simple helping parsers.

```
whitespace :: GenParser Char () Char
whitespace = oneOf [' ', 'λt', 'λn', 'λr']

whitespaces = many whitespace

wordChars :: GenParser Char () Char
wordChars = alphaNum <|> oneOf ['-', '.']

word = many1 wordChars

eol =  try (string "λnλr")
   <|> try (string "λrλn")
   <|> string "λn"
   <|> string "λr"
   <?> "End of Line"
```

This is a function for allowing a parser to be wrapped in whitespace on both sides.

```
withWS = between whitespaces whitespaces
```

'withDelim' applies a parser with a set delimiter on each side. But with no padding whitespace anywhere. If there is a possibility for whitespace arround the delimiters then withDelim' should be used instead.

```
withDelim :: String → GenParser Char () a → GenParser Char () a
withDelim delims =  between (char $ delims !! 0)
                            (char $ delims !! 1)

withDelim' delims = between (withWS (char $ delims !! 0))
                            (withWS (char $ delims !! 1))
```

TODO

incorperate this into the actual document

```
intParser :: GenParser Char () Int
intParser = return read <*> many digit
```

Appendix B

AST examples

Here follows some examples on syntax trees constructed from the parser detailed in chapter 2. The section isn't really necessary, but should be good as a reference when figuring out how the parser contraction works.

B.1 Comments

```
# Comment
```

```
C "Comment"
```

B.2 Version Note

```
%PROTOEDITION 11.1
```

```
D ("PROTOEDITION","11.1")
```

B.3 Alias

```
%type-alias Any-Conf-Type-1 Any-Conf-Type
```

D ("Any-Conf-Type-1","Any-Conf-Type")

B.4 Types (& Structures)

```
Aux-Item-10 ::=
      ( aux-no              : Aux-No-10;
        tag                 : INT32;
        creator             : Pers-No-1;
        created-at          : Time-1;
        flags               : Aux-Item-Flags-10;
        inherit-limit       : INT32;
        data                : HOLLERITH;
      )
```

```
B (TypeBinding
      "Aux-Item-10"
      (StructureType
          [ StructureField
              (TypeBinding
                  "aux-no"
                  (SimpleType (OnlyType "Aux-No-10")))
          , StructureField
              (TypeBinding
                  "tag"
                  (SimpleType (OnlyType "INT32")))
          , StructureField
              (TypeBinding
                  "creator"
                  (SimpleType (OnlyType "Pers-No-1")))
          , StructureField
              (TypeBinding
                  "created-at"
                  (SimpleType (OnlyType "Time-1")))
          , StructureField
              (TypeBinding
              "flags"
              (SimpleType (OnlyType "Aux-Item-Flags-10")))
          , StructureField
              (TypeBinding
                  "inherit-limit"
                  (SimpleType (OnlyType "INT32")))
          , StructureField
              (TypeBinding
                  "data"
                  (SimpleType (OnlyType "HOLLERITH")))]))
```

B.5 RNC

```
create-anonymous-text-10 [87] (( text       : HOLLERITH;
                                 misc-info  : ARRAY Misc-Info-1;
                                 aux-items  : ARRAY Aux-Item-Input-10 ))
    -> ( Text-No-1 );
```

```
A (Request
      "create-anonymous-text-10"
      87
      (Just
          (StructureType
              [ StructureField
                  (TypeBinding
                      "text"
                      (SimpleType (OnlyType "HOLLERITH")))
              , StructureField
                  (TypeBinding
                      "misc-info"
                      (ArrayType
```

```
                    (ArrayField (OnlyType "Misc-Info-1"))))
            , StructureField
                (TypeBinding
                    "aux-items"
                    (ArrayType
                        (ArrayField
                            (OnlyType "Aux-Item-Input-10"))))]))
      (Just (SimpleType (OnlyType "Text-No-1"))))
```

B.6 Async

```
async-broadcast-1 [10] (( sender     : Pers-No-1;
                          message    : HOLLERITH ))
```

```
A (Async
      "async-broadcast-1"
      10
      (Just
          (StructureType
              [ StructureField
                  (TypeBinding
                      "sender"
                      (SimpleType (OnlyType "Pers-No-1")))
              , StructureField
                  (TypeBinding
                      "message"
                      (SimpleType (OnlyType "HOLLERITH")))])))
```