SCL Binary Parser Generator and Runtime

v. 2021-1

Thomas R Dean

April 28, 2022

Abstract

The initial version of SCL was created by Sylvain Marquis as part of his M.Sc. thesis. It has evolved in parallel with the IBED language (ref) as multiple graduate students over the years have added to the system.

This manual provides a current description of the SCL, the generator and an outline of the code that it generates.

Contents

1 Overview of SCL and IBED

SCL and IBED are two languages part of a framework to generate custom grammar based fuzzing systems and intrusion detection systems. SCL is a domain specific language used to specify a packet parser for binary messaging protocols. It was originally used for grammar based fuzzing research and later for intrusion detection. IBED [**?**] is a domain specific language for specifying constraints on network protocols. It was developed as part of intrusion detection research.

1.1 SCL

SCL is a domain specific language is based on an extension to a subset of ASN.1. In the original research[6] the protocols we were working with were BER/DER[2] encoded protocols that were initially specified in ASN.1. One of the key features of these protocols is the use of Tag, Length, and Value(TLV) encoding of all elements of the grammar. We used custom code, and later a parameterised engine[5] to decode and re-encoded the mutated values. SCL was developed to specify this process more cleanly in a domain specific language (DSL), we adopted ASN.1 as a core of the language for this reason.

However, Several of the protocols, such as OSPF[3] and BGP[4] are not specified in ASN.1, and did not use the self decoding TLV provided by BER and DER. These protocols had mostly constant size primitive fields (e.g. the router id field in OSPF is 4 bytes in length), and encoded the length of other fields in a variety of ways. Thus we added a relatively rich set of size constraints to the language to direct the parser.

From the SCL description an XML configuration for a parser written in Java was created. The same XML configuration was used by the encoding engine to generate the mutated packets after fuzzing. Speed was not a concern, as the test data was done off line, and a significant number of test cases could be generated automatically for later injection into the test system. The system was used on several research projects[zhang,abo], and gradually grew to include constraints between packets and system to patch non-intentional errors introduced by mutations. For example, if a protocol has nested variable length fields, specified by specific fields, and a mutation is made to a value inside, all of the lengths must be changed to be consistent.

The goal of the next project was a real time Intrusion Detection System (IDS). This made the speed of the parser an issue. An encoder wasn't needed, and the protocols involved were not DER/BER encoded. Several weaknesses in the design of the original version of SCL had become apparent from the previous projects.

We incorporated some changes into the language itself, and the translation was changed to generate a C parser directly from the specification. We did not implement the entire capability of the SCL language in the new version, such as the use of DER/BER in the encoding engine. This is future work for the translation engine. Recently SCL will generate a serializer that will reencode the data structure into a binary buffer. It will also provide routines to print out a human friendly version of a message or part of a message.

2 SCL Language

We start with an explanation of the ASN.1 subset that is used in the SCL language. We then describe the SCL additions to this subset.

2.1 ASN.1 Subset

Like ASN.1, a SCL specification is a set of modules. Each module defines a set of *type names*. A type represents the type of an individual data item in a message, or it is a structured type that represents a group of data items. These types together give one or more grammars for the messages defined in a particular protocol. Listing 1 shows the skeleton the ASN.1 subset of a module. The module begins with the module name and ends with the keyword *END*. The type names that form the roots of the grammars are listed in the *EXPORTS* clause of the module. If the module needs types from another SCL module, it names the types and the modules they come from in the *IMPORTS* clause.

```
1 ModuleName DEFINITIONS ::= BEGIN
2     EXPORTS TypeName1;
3     IMPORTS TypeName2, TypeName3 FROM ExternalModule1
4             TypeName4 FROM ExternalModule2;
5     -- Rules
6 END
```

Listing 1: Skeleton Module

SCL currently supports two top level type specifications. These are *type decisions* and structured types. The first, type decisions, specify an alternate set of types to recognize. The general form is:

$$TypeName1 ::= (TypeName2 \mid TypeName3 \mid TypeName4 \mid ...)$$

The last form of type specification assembles other types into more complex types. Although ASN.1 allows both SETs and SEQUENCEs, SCL only supports the SEQUENCE type as shown in listing 2. The sequence has a name, and uses braces to delimit the list of fields in the record.

```
1     TypeName ::= SEQUENCE {
2        -- Field Definitions
3        Field1 FieldType1,
4        Field2 FieldType2
5     }
```

Listing 2: Skeleton Structure Type

Listing 3 shows the ARP protocol which is made up of a single packet type.

```
1 ARP DEFINITIONS ::= BEGIN
2     EXPORTS PDU;
```

```
3     PDU ::= SEQUENCE {
4         hwType          INTEGER,
5         protocolType    INTEGER,
6         hwSize          INTEGER,
7         protocolSize    INTEGER,
8         opcode          INTEGER,
9         senderMAC       OCTET STRING,
10        senderIP        INTEGER,
11        targetMAC       OCTET STRING,
12        targetIP        INTEGER
13    }
14 END
```

Listing 3: ASN.1 Subset for ARP Message

The ASN.1 built in types currently implement in SCL are:

- **BOOLEAN**: one or more bytes that represents a boolean value.
- **INTEGER**: one or more bytes that represents an integer number.
- **REAL**: one or more bytes that represents a floating point number.
- **OCTET STRING**: one or more bytes that are just bytes.

In addition to the built in types, there is also the SEQUENCE OF and SET OF modifiers for field types. These specify the field is an array of elements of a given type. In Listing 4, the topicData field contains multiple instances of TOPICPARMS.

```
1     TOPICS ::= SEQUENCE {
2         encapsKind  INTEGER,
3         encapsOpts  INTEGER,
4         topicData   SET OF TOPICPARMS
5     }
```

Listing 4: Example SET OF Modifier

2.2 SCL Additions

The subset is extended by SCL in to provide size information and other semantic information about the fields. SCL is also extended to provide parser optimizations and control how the parser calls the rest of the engine.

The first extension gives the size of each field in a structure. This takes the form of an annotation after the type of the field. The simplest form simply specifies the number of bytes (or bits) that the field occupies as shown in Listing 5 for the ARP packet. The annotation ENCODED BY CUSTOM, which is the default, specifies that the entire structure is not BER/DER encoded.

```
    NESTEDSTRING ::= SEQUENCE {
        nameLength      INTEGER (SIZE 4 BYTES),
        name            OCTET STRING (SIZE CONSTRAINED),
    }
    <transfer>
        Forward { LENGTH(name) == nameLength }
    </transfer>
```

Listing 7: Forward Length Constraint

```
ARP DEFINITIONS ::= BEGIN
    EXPORTS PDU;
    PDU ::= SEQUENCE {
        hwType          INTEGER (SIZE 2 BYTES),
        protocolType    INTEGER (SIZE 2 BYTES),
        hwSize          INTEGER (SIZE 1 BYTES),
        protocolSize    INTEGER (SIZE 1 BYTES),
        opcode          INTEGER (SIZE 2 BYTES),
        senderMAC       OCTET STRING (SIZE 6 BYTES),
        senderIP        INTEGER (SIZE 4 BYTES),
        targetMAC       OCTET STRING (SIZE 6 BYTES),
        targetIP        INTEGER (SIZE 4 BYTES)
    } (ENCODED BY CUSTOM)
END
```

Listing 5: Constant Field Sizes

The other options for the size annotation is *DER* which specifies that a field is DER encoded, *DEFINED* which is used on user defined types and states that the size of that field is derived from the type, and *CONSTRAINED* which states that the size of the field is given in a constraint. Listing 6 shows an example of the DEFINED size. in this case, HEADER is another structured type whose size is given in the type definition.

```
    PING ::= SEQUENCE {
        Header  HEADER (SIZE DEFINED),
        ping    OCTET STRING (SIZE 8 BYTES)
    } (ENCODED BY CUSTOM)
```

Listing 6: Fields with Defined Size

Listing 7 shows the use of a CONSTRAINED field. In this case, the value of the *nameLength* field specifies the number of characters in the OCTET STRING that contains the value of the field *name*. Constraints on parsing and well formedness of packets are given in the transfer block which takes the form of an xml markup after the type definition. The Forward keyword specifies that the constraint is for something that has not yet been parsed. In this case, the length of the name.

Listing 8 shows an example applied to the SET OF modifier for a field. The number of elements in the set of array can be specified in three ways. The first is the total number

of bytes used, which would be constrained using a LENGTH constraint as shown above. The second is a *CARDINALITY* constraint, where the parser uses one field to specify a count of the number of items. The third is the *TERMINATE* constraint which specifies the type of the last item in the list. The example is taken from the RTPS protocol where TOPICPARMS is a type decision that provides a variety of topic parameters types such as the topic name. One of the topic types is a type named PIDSENTINAL, which is used to specify the end of the data. The last is the *END* constraint that specifies the sequence runs to the end of the data in the message.

```
1     TOPICS ::= SEQUENCE {
2         encapsKind INTEGER (SIZE 2 BYTES),
3         encapsOpts INTEGER (SIZE 2 BYTES),
4         topicData SET OF TOPICPARMS (SIZE CONSTRAINED)
5     }
6     <transfer>
7         Forward { TERMINATE(topicData) == PIDSENTINAL}
8     </transfer>
```

Listing 8: Forward Terminate Constraint

Back constraints are constraints on the values of items that are checked after the items involved are parsed. For example, all RTPS packets have an initial four byte signature field that contains the value RTPS or RTPX. A topic parameter is a structure that contains 1) an integer identifier of the parameter, 2) the length of the value of the parameter and 3) the value of the parameter. The type of the value depends in part on the type of the parameter. Examples of both cases is shown in Listing 9. The first example shows the header for a full RTPS message. For the packet to be recognized as an RTPS packet, the protoName sub field of Header field must have the value RPTS or RTPX. The constraint on the set of sub messages is that the end of the message is reached when parsing subMsg.

The type PIDTYPECONSISTENCY is identified when the first two bytes of the structure have the value 116. In an unoptimized parser, the parser first parses the field (e.g. Header or parameterKind) and then checks that the value is correct before parsing. Our LL(K) optimization checks the value before parsing to minimize the amount of data manipulation.

Listing 10 shows the optional and endian extensions to the language. The field inlineQos in the RTPS protocol is optional, and is only present if the second bit in the flags field is set. While the standard network byte order is big endian, some protocols are more flexible. In particular RTPS allows different parts of messages to be encoded in different byte orders, and this is inherited by sub parts of the message. The listing shows the DATASUB type which is a sub message of the general RTPS message. The low order bit flags field specifies the byte order of the entire message, including the serialized data. However, the readerEnt and writerEnt fields are always transmitted in network byte order, so the BIGENDIAN option on that field overrides the endianness of the rest of the sub messages specified in the forward constraint.

```
1     DATASUB ::= SEQUENCE {
2         kind        INTEGER (SIZE 1 BYTES),
```

```
FULL ::= SEQUENCE {
    Header  HEADER (SIZE DEFINED),
    guidPrefix  GUIDPREFIX (SIZE DEFINED),
    subMsg  SET OF SUBMESSAGE (SIZE CONSTRAINED)
} (ENCODED BY CUSTOM)
<transfer>
    Back { Header.protoName == 'RTPS'
        || Header.protoName == 'RTPX' } -- 0x52545053
    Forward { END(subMsg) }
</transfer>

-- All RTPS have same header
HEADER ::= SEQUENCE {
    protoName   OCTET STRING  (SIZE 4 BYTES),
    version         INTEGER (SIZE 2 BYTES),
    vendorId    INTEGER (SIZE 2 BYTES)
} (ENCODED BY CUSTOM)

PIDTYPECONSISTENCY ::= SEQUENCE {
    parameterKind    INTEGER (SIZE 2 BYTES),
    parameterLength  INTEGER (SIZE 2 BYTES),
    typeConsistencyKind  TYPECONSISTENCYKIND
                                (SIZE DEFINED)
}
<transfer>
    Back { parameterKind == 116 } --  0x0074
    Forward { LENGTH(typeConsistencyKind)
                == parameterLength }
</transfer>
```

Listing 9: Eample Back Constraints

```
3         flags       INTEGER (SIZE 1 BYTES),
4         nextHeader  INTEGER (SIZE 2 BYTES),
5         extraFlags  INTEGER (SIZE 2 BYTES),
6         qosOffset   INTEGER (SIZE 2 BYTES),
7         readerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
8         writerEnt   ENTITYID (SIZE DEFINED) BIGENDIAN,
9         writerSEQ   INTEGER (SIZE 8 BYTES),
10        inlineQos   QOSPARM (SIZE DEFINED) OPTIONAL,
11        serializedData OCTET STRING (SIZE CONSTRAINED)
12    } (ENCODED BY CUSTOM)
13    <transfer>
14        Back {kind == 21} -- 0x15
15        Back {writerEnt.kind == 2 || writerEnt.kind == 3 }
16        Forward { ENDIANNESS == flags & 1 }
17        Forward { EXISTS(inlineQos) == flags & 2 }
18        -- other constraints
19    </transfer>
```

Listing 10: Example of Exists and Endian Constraints

There are several predefined names that can be used in the constraints. These are:

- **PDULENGTH**: the total length of the message passed to the parser.
- **PDUREMAINING**: the number of bytes left in the message.
- **DSTPORT**: The destination port (for IP messages). Some networks installations run protocols on non standard ports. This constraint allows the developer to specify which port is used.
- **SRCPORT**: The source port (for IP messages).
- **LENGTH(field)**: The length of a given field in a structure.
- **CARDINALITY(field)**: The number of elements in a SET or SEQUENCE.

In some cases, the grammar is ambiguous without some outside information. In particular, the application data parser that was built for RTPS. In this case the serializedData field in the data sub message, shown in figure 10, contains a structure from the application. The type of the data in that field is defined in the ENTITYID structure given by the writerEnt field. However that value is dynamically assigned, and given in a previous RTPS message. Our engine tracked the values of the entity id messages so at run time we know what type of data is present. While we could have specified a separate parser in a separate SCL file for each of the messages, it was a bit more convenient to have all of the application structures in a single file.

The GLOBAL annotation gives the name of an external integer variable that will be declared by the parser. As a constraint, it specifies that the variable must have that value in order to apply a given type rule to the parse. In the listing, the global variable is called *AppDat*. The first type definition is a type decision that gives all of the application data records carried by the protocol. If the writerEnt field for the DATASUB matches the one for a SSRModeCType data record, the parsing framework will set the variable AppDat

to the value 5 before calling the application parser on serializedData. This will direct the parser to parse the data as a Secondary Surveillance Radar Mode C data structure.

```
AppData DEFINITIONS ::= BEGIN
    EXPORTS PDU;
    PDU ::= (SSRModeSType | SSRModeCType | SSRModeAType | PSRType |
    FSDMessageType | FPType) < transfer >
        Back {GLOBAL (AppDat)}
        Callback
    </ transfer >

    SSRModeCType ::= SEQUENCE {
        timestamp INTEGER (SIZE 4 BYTES) ALIGN 4,
        equipment_id INTEGER (SIZE 4 BYTES) ALIGN 4,
        altitude REAL (SIZE 8 BYTES) ALIGN 8,
        target_id INTEGER (SIZE 4 BYTES) ALIGN 4
    } < transfer >
        Back {GLOBAL (AppDat) == 5}
    </ transfer >

    -- Other AppData Types
END
```

Listing 11: Example of Global Specification

There are some cases where one grammar might match the first part of another grammar. For example, some protocols do not contain magic numbers at the beginning of messages (such as the 'RTPS' at the beginning of RTPS messages). Since there are no direct clues, the beginning of an NFS message might be matched as an NTP message. To minimize this, the constraint *All Bytes Used* may be added to structured types to indicate that all of the bytes in the current buffer must be used by the end of the structure. Since a SCL module may be used to parse part of another message, this is left to developer control.

The last extension has to do with the interface from the parser to the rest of the software. The previous version of SCL generated a generic tree structure with the names and values stored dynamically in the tree. The new version generates a custom C data structure that is similar to the data structure specified by the ASN.1 subset of the language. Choice decisions are represented by a tag and union data type in C. If the parser returned a tagged union for the data type shown in Figure 12, then the rest of the sofware would have to test the tag to determine which stucture was returned. This is optimized by the use of the Callback annotation. When the parser recognizes a Type given in the type decision, it calls a function specific to that type. Listing the function called when a RTPS ping message is recognized. The first parameter is the data structure generated by the parser, while the second is a pointer to the raw data including the IP header for information about addresses and ports.

```
void PING_RTPS_callback (PING_RTPS * ping_rtps, PDU * thePDU);
```

Listing 12: RTPS Ping Callback

As part of the generation process, prototypes for those functions are declared. The rest of the software must implement the function that is called when the packet is recognized. A further optimization is implemented that recognizes protocols that contain a header followed by a sequence of sub messages. Examples are the Full type in RTPS (figure 9) where the header is followed by a set of SUBMESSAGE, or IGMP version 3 where a message contains a list of group addresses. In this case the parser generates a separate data structure for the entire packet, and a separate callback for each of the sub messages, such as the interface shown in Figure 13. In this case, the first parameter has the structure for the entire packet, but only the fields associated with the header are used, and the second parameter contains the data strucutre of the sub message, and the third is the same as above.

- Note that the pointer for the sub data structure is not created for submessage optimization
- the data structure is freed before returning, so cannot be used by the caller of the parser.

```
1 void DATASUB_RTPS_callback (FULL_RTPS * full_rtps, DATASUB_RTPS * datasub_rtps,
      PDU * thePDU);
```

Listing 13: Submessage Callback

3 Translator

The translator is written in the language TXL[1]. Txl is a strongly typed, functional, source transformation language. The approach taken is a sequence of transformations that add annotations to the SCL specification, and a final set of transformations that transform the specification to an XML file for a Python interpreter or to C code to provide a native parser.

Figure 1 shows the overall process of the translator. The first two stages, UID Decl and UID Ref give each entity in the specification a unique name. They also check that names are unique, and that all of the names used are declared.The Check Imports stage checks that the types imported in one module were a subset of the modules exported from the other module.

The next three stages, Callback Annotation, LL1 Optimization and LL(k) optimization analyze the specification and add annotations to assist the code generator. There are currently two code generators. One generates a C implementation of the parser, the other generates an XML file that can by used by an interpreter to parse the messages. We describe each of the stages of the translation in turn.

3.1 Rename Declarations

Rename Declarations is the first stage of the process. It generates a unique name for every entity declared in the system. Unlike languages like C or Java, there are only three scope

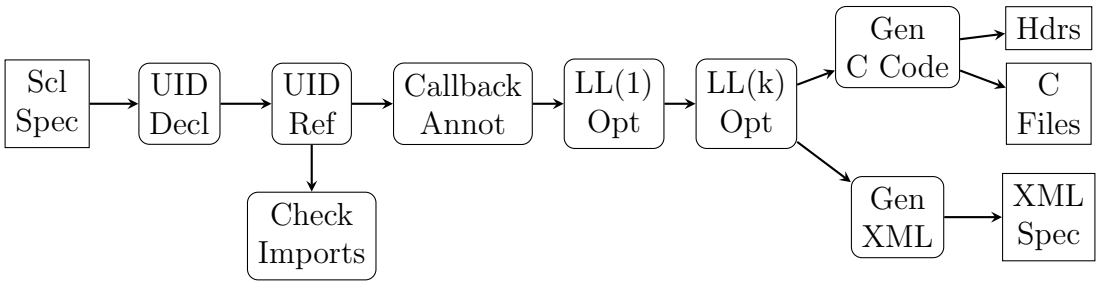

Figure 1: Generation Process

levels in the language, so unique naming is relatively simple. A file in SCL consists of one or more modules, each of which has a unique name at the global level. The approach taken is similar to the approach taken in LS/2000[ref] and Lei Wang's thesis [ref].

Each module consists of a sequence of type definitions, each of which has a name. Structured types (i.e. SEQUENCE { ... }) have fields. Two fields with the same name in different structured types are different fields. For example, in listing 14, both of the structured types B and C have a field named f. We use the same approach as LS2000 [ref]. Unique naming works from the outside end, appending the name of the outer scopes to names in the inner scopes. Thus B becomes B_A and the `f` in `B` becomes `f_B_A` while the `f` in `C` becomes `f_C_A`.

```
A DEFINITIONS ::= BEGIN
    EXPORTS PDU;
    IMPORTS M from N.
    PDU ::= (B | C )

    B ::= SEQUENCE {
        f INTEGER (SIZE 4 BYTES),
        g X (SIZE 4 BYTES),
    }
    <transfer>
       Back{ f == 2 }
    </transfer>

    C ::= SEQUENCE {
        h INTEGER (SIZE 4 BYTES),
        f INTEGER (SIZE 4 BYTES),
    }
    <transfer>
       Back{ h == 5 }
    </transfer>

    X ::= SEQUENCE {
        y INTEGER (SIZE 4 BYTES),
        z INTEGER (SIZE 4 BYTES),
    }
END
```

Listing 14: Lexical Levels

In some cases, the original shorter name is still needed. To preserve both of the

names, the transform includes both names by enclosing the names of declarations in square brackets(i.e. [] and separated by a caret (i.e. ^). An example is given in listing 15. Note that only the declarations of the names are changed, references to the name such as the reference to `B` and `C` in the type decision `PDU` and the reference to `X` in declaration of the field `g` are not changed at this point in time.

```
A DEFINITIONS ::= BEGIN
    EXPORTS PDU;
    IMPORTS [M_N ^ M] FROM N.
    [PDU_A ^ PDU] ::= (B | C )

    [B_A ^ B] ::= SEQUENCE {
        [f_B_A ^ f] INTEGER (SIZE 4 BYTES),
        [g_B_A ^ g] X (SIZE 4 BYTES),
    }
    <transfer>
       Back{ f == 2 }
    </transfer>

    [C_A ^ C] ::= SEQUENCE {
        [h_C_A ^ h] INTEGER (SIZE 4 BYTES),
        [f_C_A ^ f] INTEGER (SIZE 4 BYTES),
    }
    <transfer>
       Back{ h == 2 }
    </transfer>

    [X_A ^ X] ::= SEQUENCE {
        [y_X_A ^ y] INTEGER (SIZE 4 BYTES),
        [z_X_A ^ z] INTEGER (SIZE 4 BYTES),
    }
END
```

Listing 15: Rename Declarations

Since modules are processed independently, the processor does not know what types are exported from a given Module. SCL only allows top level types to be exported, so a name is generated for each import based on the type and module name in the imports statement. A later phase of the process will check that the imported names are exported from the module in question.

Since a name in SCL may include an underscore, the process first replaces all underscores in user defined names with dollar signs before the unique naming is started. Thus a type with the name `X_Y` in a module `Z` will become `X$Y_Z`.

3.2 Rename References

After all of the declarations have been given unique names, any references to the names are also uniquely named. There are four types of references:

Exports. A module may export one ore more grammar types. For example, the export of `PDU` in listing 15.

Type Decision. A type decision defines a type as a choice between several other types. For examp.e the defition of `PDU` in listing 15 defines `PDU` as either a `B` or a `C`.

Field Types. The type of a field in a structured type may be a primitive type such as INTEGER, or it may be a type defined elsewhere in the specification. For example, the field `g` of the type `B` is of type `X`.

Constraints. . Fields and Types may be referred to in transfer or constraints blocks that are appended to the definition of a type. For example, the reference to the fields `f` and `n` in listing 15. Some of the references in constraints are to elements other than types such as the reference to the source or destination port, to a global variable, or to the length of the PDU.

Unlike declarations, the references are renamed by replacing them the unique identifier, as shown in listing 16. In the listing the reference to `PDU` in the Exports list has been renamed and the references to `B` and `C` in the definition of `A`. The reference to `X` in the definition of `B` has been renamed as well as the refrences to `f` and `h` in the transfer blocks.

```
1 A DEFINITIONS ::= BEGIN
2     EXPORTS PDU_A;
3     IMPORTS [M_N ^ M] FROM N.
4     [PDU_A ^ PDU] ::= (B_A | C_A )
5
6     [B_A ^ B] ::= SEQUENCE {
7         [f_B_A ^ f] INTEGER (SIZE 4 BYTES),
8         [g_B_A ^ g] X_A (SIZE 4 BYTES),
9     }
10    <transfer>
11       Back{ f_B_A == 2 }
12    </transfer>
13
14    [C_A ^ C] ::= SEQUENCE {
15        [h_C_A ^ h] INTEGER (SIZE 4 BYTES),
16        [f_C_A ^ f] INTEGER (SIZE 4 BYTES),
17    }
18    <transfer>
19       Back{ h_C_A == 2 }
20    </transfer>
21
22    [X_A ^ X] ::= SEQUENCE {
23        [y_X_A ^ y] INTEGER (SIZE 4 BYTES),
24        [z_X_A ^ z] INTEGER (SIZE 4 BYTES),
25    }
26 END
```

Listing 16: Rename References

The rename references phase produces error messages for any references to types that have not been declared. It also generates a file that contains a list of the types that are exported. By default the name of this file is the module name with the extension `.exports`. For example, `A.exports`. The `-Intermediate` flag is used to specify the destination directory for the file (defualt is `INTERMEDIATE`).

```
1 FULL ::= SEQUENCE {
2     Header  HEADER (SIZE DEFINED),
3     guidPrefix  GUIDPREFIX (SIZE DEFINED),
4     subMsg  SET OF SUBMESSAGE (SIZE CONSTRAINED)
5 } (ENCODED BY CUSTOM)
6 <transfer>
7     -- Other transfer constraints here
8     Callback
9 </transfer>
10
11 SUBMESSAGE ::= (DATAPSUB  | ACKNACK |  ...)
12
13 DATAPSUB := SEQUENCE {
14     -- DATAPSUB fields here
15 }
16 <transfer>
17     -- DATAPSUB specific transfer constraints
18 </transfer>
```

Listing 17: Before Callback Annotation

3.3 Check Imports

This is a simple phase that checks that the types that were imported by one module were defined in the other module. Recall from rename declarations that the imports lists were assumed to be true for the purposes of generating unique names for declarations. For example the name `M_N` was generated for the import statement `IMPORT M FROM N`.

This phase runs after all of modules have been run through the rename references phase. At that point, an exports file for each module will have been created. This phase iterates over the module names in the imports list, and confirms that the type imported from the module is listed in the exports file for that module. It generates an error for any types that are imported that are not exported.

3.4 Callback Annotation

There are two types of callbacks. The first are the callbacks specifically identified by the user. In this case a callback is to be made each time the type is recognized. The second is the submessage callback optimization. If the user add the callback annotation to a structured type that ends with a field that is a set or sequence, then a separate callback can be made for each element of the set or sequence. In some cases, the set type is a single structured type as in the IGMP protocol. In other cases, the set type may be a type decision allowing the set to be a hetrogenous list. As shown previously in listings 12 and 13, each type of callback has a different signature.

Listing 17 shows a snippit of the RTPS protocol definition. In the figure , SCL generates a callback to the application logic whenever a FULL sequence is parsed. The annotation of the name of the SCL type (FULL) is added to the callback to aid in code generation. In the figure, the FULL sequence ends with a set of submessages, which

```
1 [FULL_RTPS ^ FULL] ::= SEQUENCE {
2     [Header_FULL_RTPS ^ Header] HEADER_RTPS (SIZE DEFINED),
3     [guidPrefix_FULL_RTPS ^ guidPrefix] GUIDPREFIX_RTPS (SIZE DEFINED),
4     [subMsg_FULL_RTPS ^ subMsg] SET OF SUBMESSAGE_RTPS (SIZE CONSTRAINED
      )
5 } (ENCODED BY CUSTOM)
6 < transfer >
7     -- Other constraints in internal form
8     Callback @ FULL_RTPS subMsg
9 </ transfer >
10
11 [SUBMESSAGE_RTPS ^ SUBMESSAGE] ::= (DATAPSUB_RTPS | ACKNACK_RTPS |  ...)
12 < transfer >
13     Callback ^ FULL_RTPS subMsg
14 </ transfer >
15
16 [DATAPSUB_RTPS ^ DATAPSUB] ::= SEQUENCE  {
17     -- DATAPSUB fields here
18 }
19 <transfer>
20     -- DATAPSUB specific transfer constraints
21     Callback FULL_RTPS subMsg
22 </transfer>
```

Listing 18: After Callback Annotation

in turn is a type decision allowing for a heterogenous list of submessages a part of the FULL message. To generate the callback from the DATAPSUB sequence, it must know the parent type that will be passed to the submessage callback, which must be passed through the SUBMESSAGE type decision. To support the code generation, this phase adds markup to both the SUBMESSAGE type decision and all of the types derived from the SUBMESSAGE type decision as shown in figure **??**. The fact that this is an original Callback annotation from the user is inidicated by the at (i.e. @) symbol.

In this case the annotation Callback Full

If this type is a type decision, then the callback statement is propagated without the $\wedge$ to each of the types in the type decision. For example, the callback statement `Callback FULL_RTPS subMsg` is added to each of the types referenced by the `SUBMESSAGE` field in RTPS. In both cases they markup is the same as in the original FULL sequence, except with an additional annotation that indicates the role. The caret (i.e.^) is used to indicate that it is a passthrough, and no annotation for the location of the submessage callback. Note that this phase only annotates the source with information, the callbacks are dealt with during the generation of the code and header files or by the interpreter. Flags to the generation phases or interpreter control how the annotations are used.

- free once called?

3.5 Optimization

The previous version of the generator used separate phases for the LL1 and the LLk optimizer. While this made the LLK optimizer optional, it also complicated the final code generator. The previous optimizaer also required the LL1/LLK fields to have the same name, not just the same type. The generator read the values and passed them on to the optimized field parser. the cost of copying the values outweighed the cost of reading the values twice. The generator also had to generate both optimized and non-optimized versions of the parsing methods as a record might be called from both an optimized choice and a non-optimized choice. The optimized versions assumed that the values were already read and that the current position had been updated.

The new approach identifies constant values with constant sizes at constant offsets in records independent of the name of the field. This includes and fields in any field records. This is used to look ahead into the input and check for the values at the offsets. However, when looking ahead, the current position is not changed, allowing the same parsing function to be used to read the fields into the internal data structures.

The LL optimization is done in four phases. In the first phase, fields of records are annotated as constant or variable. In the second, usable back constraints are converted to annotation on the records they govern. In the third phase, the back constraint annotations are copied to the type decisions that use of the types. The final the annotations of a type decision are analyzed and if LL optimization is possible, a decision annotation is added.

3.5.1 Back Constraints

While in general, back constraints can be any boolean expression, back constraints must have a specific format to be used in LL optimization.

The first condition on the format is that has the form `field_reference = constant_expression`. For example, in the IGMP protocol, the type field in a `QUERY` record has a value of 17 (in decimal), expressed as `Back{type == 17}`.

The second condition for use in LL optimization is that only one variable is referenced. If more that one value for the field is possible, then the constraints are expressed using a disjunction. For example, the RTPS protocol is identified by the first 4 bytes of a message. The value can be `'RTPS'` or `'RTPX'`, this is expressed as the constraint `Back { Header.protoName == 'RTPS' || Header.protoName == 'RTPX' }` . As also shown in this example, fields of fields (to an arbitrary depth) is also supported in back constraints and the LL optimization of back constraints.

If more than one field must be constrained, they are written as two back constraints as all backward constraints on a given type definition are implicitly conjoined.

```
TODO
- assumes the back constraint is at the point of decision. (should be covered in langua
```

3.5.2 Field and Record Annotation

The first phase of LL optimization locates and annotates fields that have a constant size and offset within the record. Since the types of some fields are records in turn, this phase

```
1
2 [FULL_RTPS ^ FULL] ::= SEQUENCE {
3     [Header_FULL_RTPS ^ Header] HEADER_RTPS (SIZE DEFINED),
4     [guidPrefix_FULL_RTPS ^ guidPrefix] GUIDPREFIX_RTPS (SIZE DEFINED),
5     [subMsg_FULL_RTPS ^ subMsg] SET OF SUBMESSAGE_RTPS (SIZE CONSTRAINED
      )
6 } (ENCODED BY CUSTOM)
7
8 [HEADER_RTPS ^ HEADER] ::= SEQUENCE {
9     [protoName_HEADER_RTPS ^ protoName] OCTET STRING (SIZE 4 BYTES),
10    [version_HEADER_RTPS ^ version] INTEGER (SIZE 2 BYTES),
11    [vendorId_HEADER_RTPS ^ vendorId] INTEGER (SIZE 2 BYTES)
12 } (ENCODED BY CUSTOM)
13
14 [GUIDPREFIX_RTPS ^ GUIDPREFIX] ::= SEQUENCE {
15    [hostID_GUIDPREFIX_RTPS ^ hostID] INTEGER (SIZE 4 BYTES),
16    [appID_GUIDPREFIX_RTPS ^ appID] INTEGER (SIZE 4 BYTES),
17    [counter_GUIDPREFIX_RTPS ^ counter] INTEGER (SIZE 4 BYTES)
18 } (ENCODED
```

Listing 19: Before Size Annotations

also identifies records that are a constant size. An example is given in figure 19. In the exmaple both the `HEADER` and `GUIDPREFIX` records have constant sizes). The field `Header` is at offset 0 and a size of 8, and the field `guildPrefix` is at an offset of 8 and has a size of 12. The last field, `subMsg` has a variable size, but a constant offset of 12.

The rules that implement the size annotation use a common TXL paradigm where they run until a fixed point is reached (no further changes are made to the code). In each iteration, the rules:

1. visit each record type declaration and visit each of the fields in turn.

   (a) If the field is of constant size, then the field is annotated with the size and current offset from the beginning of the enclosing struct. The annotation is of the form `@ CONST Offset Size`, where Offset and Size are the offset and size of the field in bytes. In the example in figure 19, the first iteration of the rules will annotate the six fields in the `HEADER` and `GUIDPREFIX` as constant. For exmaple, the field `counter` will be annotated with texttt@CONST 8 4 as it is a constant 4 bytes long, and is 8 bytes from the beginning of a `GUIDPREFIX`. At this point in time, even though the size of the fields textttHeader and `guildPrefix` in the `FULL` are constant sizes, they cannot yet be annotated.

   (b) If the field is not of constant size, then the field is marked as variable with the `@ VAR` annotation. All remaining fields are marked as variable, even if they are of constant size since the offset is not known. An example of the is the field `subMsg` in the user type `FULL`.

   (c) If the type of a field is a type decision, it is annotated as a variable size field.

```
1 FULL_RTPS ^ FULL] ::= SEQUENCE {
2   [Header_FULL_RTPS ^ Header] HEADER_RTPS (SIZE DEFINED),
3   [guidPrefix_FULL_RTPS ^ guidPrefix] GUIDPREFIX_RTPS (SIZE DEFINED),
4   [subMsg_FULL_RTPS ^ subMsg] SET OF SUBMESSAGE_RTPS (SIZE CONSTRAINED)
5 } (ENCODED BY CUSTOM)
6
7 [HEADER_RTPS ^ HEADER] @CONST 0 8 ::= SEQUENCE {
8   [protoName_HEADER_RTPS ^ protoName] @CONST 0 4 OCTET STRING (SIZE 4 BYTES),
9   [version_HEADER_RTPS ^ version] @CONST 0 2 INTEGER (SIZE 2 BYTES),
10  [vendorId_HEADER_RTPS ^ vendorId] @CONST 0 2INTEGER (SIZE 2 BYTES)
11 } (ENCODED BY CUSTOM)
12
13 [GUIDPREFIX_RTPS ^ GUIDPREFIX] @CONST 0 12 ::= SEQUENCE {
14   [hostID_GUIDPREFIX_RTPS ^ hostID] @CONST 0 4 INTEGER (SIZE 4 BYTES),
15   [appID_GUIDPREFIX_RTPS ^ appID]  @CONST 0 4  INTEGER (SIZE 4 BYTES),
16   [counter_GUIDPREFIX_RTPS ^ counter]  @CONST 0 4 INTEGER (SIZE 4 BYTES)
17 } (ENCODED
```

Listing 20: Before Size Annotations

(d) On each pass, the rules process as many fields whose size is known. If a field is a record, and the size is not yet computed, then that field and the remaining fields are left to the next pass. For example the field `Header` of the type `FULL` cannot be annotated until the size of the type `HEADER` is known.

(e) If some of the fields are annotated from a previous pass, they are skipped over and the current offset is recovered from the last annotated constant field.

2. Once all of the fields of a given type are annotated, the type itself is annotated either as a const (with and offset of 0) or as variable. After the first pass of the example, the type `HEADER` is annotated with a size of 8, and the type `GUIDPREFIX` is annotated with the size of 12.

So after the first pass of the rules, these types are annotated as:

On the second iteration, it can now annotate the size and offset of the fields in the type `FULL`.

Suggested improvements Identify constant sizes at variable offsets after the first var field. This will make code generation easier if it can identify constant size fields from the annotation.

3.5.3 Encoding Usable Back Constraints

A field that is used in a back constraint can be used for LL prediction must be at a constant offset within the field. The annotation `LL Offset Size list_of_values` is used to annotate a user type with LL optimizations. For example, the back constraint from lines 11 to 17 of listing **??** refers to the subfield `protoName` of the field `Header`, which is at offset 0 (i.e. offset 0 of a field at offset 0) of the record and has a size of 4 bytes. It has two possible values expressed as 4 byte character literals. The literals are converted to

```
1 FULL_RTPS ^ FULL] @VAR ::= SEQUENCE {
2   [Header_FULL_RTPS ^ Header] @CONST 0 8 HEADER_RTPS (SIZE DEFINED),
3   [guidPrefix_FULL_RTPS ^ guidPrefix] @CONST 8 12 GUIDPREFIX_RTPS (SIZE
      DEFINED),
4   [subMsg_FULL_RTPS ^ subMsg] @VAR SET OF SUBMESSAGE_RTPS (SIZE CONSTRAINED)
5 } (ENCODED BY CUSTOM)
6
7 [HEADER_RTPS ^ HEADER] @CONST 0 8 ::= SEQUENCE {
8   [protoName_HEADER_RTPS ^ protoName] @CONST 0 4 OCTET STRING (SIZE 4 BYTES),
9   [version_HEADER_RTPS ^ version] @CONST 4 2 INTEGER (SIZE 2 BYTES),
10  [vendorId_HEADER_RTPS ^ vendorId] @CONST 6 2 INTEGER (SIZE 2 BYTES)
11 } (ENCODED BY CUSTOM)
12
13 [GUIDPREFIX_RTPS ^ GUIDPREFIX] @CONST 0 12 ::= SEQUENCE {
14  [hostID_GUIDPREFIX_RTPS ^ hostID] @CONST 0 4 INTEGER (SIZE 4 BYTES),
15  [appID_GUIDPREFIX_RTPS ^ appID]  @CONST 4 4  INTEGER (SIZE 4 BYTES),
16  [counter_GUIDPREFIX_RTPS ^ counter]  @CONST 8 4 INTEGER (SIZE 4 BYTES)
17 } (ENCODED
```

Listing 21: Before Size Annotations

their numeric equivalents (1381257304 and 1381257299). The resulting annotation `@ LL 0 4 1381257304,1381257299` indicates that the first four bytes of this type must have one of these two values. More than one annotation may be added. For example, the type `DATASUB` in Listing 10 has two back constraints on the field `kind` (offset 0) and the subfield `kind` of the field `writerEnt` (offset 3 in 12 for a final offset of 15). This results in two annotations: `@ LL 0 1 21` and `@ LL 15 1 3,2`.

```
improvements:
 - must identify which back constraints are used in LL optimization,
    and which must be evaluated at parse time. (not currently used in
    any protocol yet)
 - also a field may be variable size as a record, but the field may be the
    first variable field (constant offset) and a subfield  may be at a constant
    offset from the beginning. This case is also optimizable, but not yet
    implemented.
```

3.5.4 Annotate Type Decisions

Once each user defined type has been annotated with LL optimization annotations, those annotations are copied to the use of those types in type decisions. Listing 22 shows the `SUBMESSAGE` type decision in RTPS. It shows a choice between 9 different types. Technically in the RTPS protocol, four of the types are the same submessage type: `DATA` (LL value 21 at offset 0), three of them are distinguished from the submessages used to carry user data by the indication that a predefined writer is used to create the data. User defined writers have a value of 2 or 3 at offset 15 in the submessage (`DATASUB_RTPS`), while predefined writers have the value 194 at offset 15 in the submessage. The value at offset

```
[SUBMESSAGE_RTPS ^ SUBMESSAGE] @ VAR ::= (
    DATAPSUB_RTPS @ LL 0 1 21 @ LL 12 3 256 @ LL 15 1 194
  | DATASUB_RTPS @ LL 0 1 21 @ LL 15 1 3, 2
  | ACKNACK_RTPS @  LL 0 1 6
  | HEARTBEAT_RTPS @ LL 0 1 7
  | INFO$DST_RTPS @ LL 0 1 14
  | INFO$TS_RTPS @ LL 0 1 9
  | DATAWSUB_RTPS @ LL 0 1 21 @ LL 12 3 3 @ LL 15 1 194
  | DATARSUB_RTPS @ LL 0 1 21 @ LL 12 3 4 @ LL 15 1 194
  | GAP_RTPS @ LL 0 1 8
 )
```

Listing 22: Type Decision Annotation

```
{0 1
    6 @ ACKNACK_RTPS
    7 @ HEARTBEAT_RTPS
    8 @ GAP_RTPS
    9 @ INFO$TS_RTPS
    14 @ INFO$DST_RTPS
    21 @
       DATAPSUB_RTPS @ LL 12 3 256 @ LL 15 1 194
       DATASUB_RTPS @ LL 15 1 3, 2
       DATAWSUB_RTPS @ LL 12 3 3 @ LL 15 1 194
       DATARSUB_RTPS @ LL 12 3 4 @ LL 15 1 194
}
```

Listing 23: Offset Zero Partition

12 identifies which predefined writer is used. As a last step, the annotations are sorted by offset.

3.5.5 Compute LL Decisions

The type decision is turned into a simple repeat in the TXL internal representation (no choice bars) that are in the same order as in the type decision. In the example in listing 22, all choices have a value at offset 0, size 1. Listing 23 shows the choices partitioned by the values at offset zero. Some values have more than choice so they end up in a repeat associated with the value. The order of the repeat is the same order as in the original type decision. The annotation for the clustered offset (offset 0) is also removed.

In the example, not all of the fields have an offset 12, so offset 12 cannot be used at this point in time as a lookahead decision. The internal annotation a second repeat (indicated by !), for unused lookahead annotations. Listing 24 shows the unused annotations moved to the second repeat.

In the first annotation repeat, all items have the same offset and size at 15 and 1, so we can partition again. When we build the new partition (as shown in Listing 25, we

```
{0 1
    6 @ ACKNACK_RTPS
    7 @ HEARTBEAT_RTPS
    8 @ GAP_RTPS
    9 @ INFO$TS_RTPS
    14 @ INFO$DST_RTPS
    21 @
       DATAPSUB_RTPS@ LL 15 1 194 ! @ LL 12 3 256
       DATASUB_RTPS @ LL 15 1 3, 2
       DATAWSUB_RTPS@ LL 15 1 194 ! @ LL 12 3 3
       DATARSUB_RTPS@ LL 15 1 194 ! @ LL 12 3 4
}
```

Listing 24: Offset 12 Moved

```
{0 1
    6 @ ACKNACK_RTPS
    7 @ HEARTBEAT_RTPS
    8 @ GAP_RTPS
    9 @ INFO$TS_RTPS
    14 @ INFO$DST_RTPS
    21 @{15 1
       194 @
  DATAPSUB_RTPS @ LL 12 3 256
  DATAWSUB_RTPS @ LL 12 3 3
  DATARSUB_RTPS @ LL 12 3 4
       3 @ DATASUB_RTPS
       2 @ DATASUB_RTPS
}
```

Listing 25: Offset 15 Partition

move the denied list back to the beginning of the main list because the partition may resolve the ambiguity. Again multiple options (value 194) are in the same order as the type decision). If an choice has more than one possible value for a given offset, it is repeated. For example, choice `DATASUB` has two values(2 and 3) at offset 15 size 1.

Separating the `DATASUB` choice from the rest based on the value 194 at offset 15 gave a list that can be partitioned at offset 12 with size 3. This can be used to make the final lookahead block shown in listing 26.

Any unresolved lookaheads are left in the block for possible use in backtracking. While it is not efficient to check a value close to the front of the current state of the parse as it the parse will backtrack as soon as the subfield is called, a value farther into the parse may provide a limit. Also the existence of forward constraints that must be managed may make checking a constraint before a given call useful.

If more than one option remains in a list, then the code generator must use backtracking to resolve it. One example may be if in one of the options we end up with a list that contains no annotations.

```
{0 1
  6 @ ACKNACK_RTPS
  7 @ HEARTBEAT_RTPS
  8 @ GAP_RTPS
  9 @ INFO$TS_RTPS
  14 @ INFO$DST_RTPS
  21 @  {15 1
    2 @ DATASUB_RTPS
    3 @ DATASUB_RTPS
    194: {12 3
      3 @  DATAWSUB_RTPS
      4 @  DATARSUB_RTPS
      256 @DATAPSUB_RTPS
    }
  }
}
```

Listing 26: Offset 12 Parittioned

3.5.6 Final Notes on LL

Currently the LL optimization is localized within a given module. For example, the UDP grammar contains a single type decision that calls each of the exported types from the imported UDP protocols (NTP, DNS, RTPS, etc). The back constraints on the exported types could be propagated to the using grammar to optimize that choice as well. But this is not implemented yet.

4 Python Interpreter

The original version of SCL provided a java interpreter that was presented by Marquis et al [ref]. In that version, SCL was translated into an XML specification that was used to drive the interpreter. It did not support modular descriptions, and a single XML file was used to read and write binary messages as part of an early grammar based fuzzer [refs].

```
TODO:
 - format of XML
 - generation of XML
 - structure of python interpreter.
```

5 C Code Generator

5.1 Data Structures

The parser generates C data types to hold the decoded packets. We start with the elementary types that are used to describe primitive items.

A previous version of SCL implemented bitfields, but they are not currently implemented. Integers are currently recognized with sizes up to 8 bytes, and the size is rounded up to the nearest integral C size, one of uint8_t, uint16_t, uint32_t, or uint64_t. For example, an INTEGER field that is declared to be 5 bytes long will be implemented as a uint64_t. The 5 bytes will be read from the message and stored in the lower 5 bytes of the integer.

Currently the only implemented sizes of REAL ASN types are - what do the generated data strucutes look like

- Integer types uint8_t, uint16_t, uint32_t, uint64_t, max 64 bits, sizes rounded up. (TODO How should the transformation be open ended?)

- Real types float, double

- Octet String 8 bytes or less get turned into integers, more than 8 bytes become strings - VAriable length become strings

- optionals are pointers. (maybe optional integers should just be integers with a flag?

- type decisions become union structures

- sets or and sequences of become arrays

- sequences become structures

5.2 Data Structure Future Work

6 Packet Parser Approach

- PDU structure and fields, what do they mean. - need to validate against generated code...

```
1 typedef struct _pdu {
2     unsigned char * data;
3     unsigned long len;
4     unsigned long watermark;
5     unsigned long curPos;
6     unsigned long curBitPos;
7     unsigned long remaining; // used when parsing flags??
8     struct HeaderInfo * header;
9 } PDU;
```

Listing 27: PDU Data Structure

```
struct HeaderInfo {
        uint32_t srcIP;
        uint32_t dstIP;
        uint16_t srcPort;
        uint16_t dstPort;
        long time;
        unsigned long pktCount;
        long refCount;
        void * mainpduPtr;
        int parserIndex;
};
```

Listing 28: HeaderInfo Data Structure

– general appraoch of parser, llk optimization.

After annotation, both the headers and the code are generated from the specification. Since some structures are used as fields in others, C imposes an order on the declaration of the structures. Since SCL does not impose an order on the declaration of the types, we first generate the structures and then sort them based on the partial order of their use.

6.0.1 Generate Headers

6.0.2 Sort Headers

- so compiler doesn't complain

6.0.3 Gen Source

```
- most complex transform 5-6K of code

- outline of code

- redefinitions

a rule definition is redefined to include a c declaration
 - this allows each rule to be replaced by a function

- deconstruct P into Module and body
    - assumes only one module in a file

- checks command line for three arguments:
    - nocallback
    - nosubmessage (new I just added, not implemented yet)
    - debug - creates an variable for indenting debug messages

- exports several global variable used to hold various
parts of the generated code
    - function prototypes
    - auxiliary functions
```

```
     -parsed paraemters (Llk)
     - parsed pair (llk)
     - pos list (llk)
     - lengthTocheck - involves generataion of length checks

constructs the name of the free function freePDU_PROTOCOL

Extracts a copy of the type rules (SEQ)
Extrats a copy of the type deicsions
and a copy of all the rules, which is exported

The main function calls 5 functions
    - checkGlobalConstraints
    - createFreeFunctionTYpeRule - creates the free funciton if the pUD is a SEQ
    - createFreeFuncitonTypeDec - creates free funciton if the PDU is a TD
    - createParseFunciton
    - assembleProgram (assembles all the parts into the final program

checkGlobalConstraints
    -creates a C definition for any global constraint variables and
      confirms that all global constraints use a created var

createFreeFunctionTypeRule generates a free rule assuming that the
main type is named PDU and is a SEQ

create FreeFynctionTypeDecision generates a free rule assuming that the
main type is named PDU and is a TD

- both should be changed to use the exports list.
- both generate into a global variable.
- both should be changed to generate in place and use a copy of the rules in a construc

createParseFunction runs in place and replaces tpe rules with parse funcitons for each

assembleProgram assembles all of code in the global variables with the parse functions
that are sill in the main scope.

createParseFunctions
    - scope is all of the rules
    - matches each rule in turn and replaces with code.
    - may add some code to auxiliary
    - parameters
        - ModName (the name of the moduel as an ID)
        - TP rules - all of the SEQ rules
        - TypeDec - all of the type decisions
```

```
    Rules are (all have the same three parameters passed through):
        - doSimpleTypeRule
        - doChoiceOPtimizedTypeRule
        - doSimpleTypeDecision
        - doLLKTypeDecision
        - doAnnotatedTypeDecision
        - doAnnoatedLLKTypeDecision
        - doOptimizedTypeDecision
        - doOptimizedTypeDecisionDotOp
        - doGloballyOptimizedTypeDecision

doSimpleTypeRule
    - only called by createParseFunctions
    - parameters
        - ModName (the name of the moduel as an ID)
        - TP rules - all of the SEQ rules
        - TypeDec - all of the type decisions
    - matches a TypeRule
    - constructs the function name which is parse and the short name of the
      type. (e.g.parseQuery)
    - if the type name is PDU, change to Module name (.e.g parseRTPS)
    - craete a function prototype
        - first paramter is pointer to long name, using long name in all lower case)
        - second paraemter is the PDU
        - third paramter is progrname
        - 4th parmater is endianness.
        - default ALLOCNAME is mainpud
        - variable lower is the long name of the type in lowercase
        - create an empty function body
        - set length to check to zero
            - as fields are converted to code, add all of the required constant length
              fields sizes together so can check the number of bytes once
        - Call function initializePointer to initialize all of the pointers
          - in the strcuture to NULL (must know whih fields are pointer/not pointer)

        - parse bySize

        - check for a global constraint and add code for a global constraint if there
            - calls addGlobalCheck


initializePointer
    - calle by doSimpleTypeRule
```

```
Section on neeeded improvements
- llk optimization should not be a list of all the fields whose names have to be maches
- more than one value for LL1/LLK optimization
```

7 Conclusion

foobar

References

[1] James R Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[2] ITU-T. *X.690 Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).* International Telecommunications Union, 2008.

[3] J. Moy. *OSPF Version 2.* Internet Engineering Task Force, April 1998.

[4] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4).* Internet Engineering Task Force, January 2006.

[5] Oded Tal, G. Scott Knight, and Thomas Roy Dean. Syntax-based vulnerability testing of frame-based network protocols. In *Proceedings of the 2nd Annual Conference on Privacy, Security and Trust*, pages 155–160, Fredericton, Canada, October 2004.

[6] Yves Turcott, Oded Tal, G. Scott Knight, and Thomas Roy Dean. Security vulnerabilities assessment of the x.509 protocol by syntax-based testing. In *Military Communications Conference 2004*, volume 3, pages 1572–1578, Monterey, CA, October 2004.