HLA v2.0 Declarations Parser Design Documentation

This document describes the internal operation of HLA v2.0's declarations parser. This document assumes that the reader is familiar with compiler theory and terminology. This document also assumes that the reader is comfortable with the HLA language and 80x86 assembly language in general.

The parseDcls.hla source module in the HLA v2.0 source tree is responsible for processing declaration sections found in units, programs, procedures, iterators, namespaces, classes, records, unions, and methods. This document describes the code found in this module.

pgmDcls - Parsing Program Declarations

The main compiler unit calls the pgmDcls procedure to handle the processing of all the declarations between the `program` statement and the corresponding `begin` statement. Consider the following production that describes an HLA program:

HLAPgm →
  program identifier ';'
    pgmDcls
  begin identifier ';'
    statements
  end identifier ';'

The `pgmDcls` procedure handles the expansion of the *pgmDcls* non-terminal in this production. Here's the expansion of the pgmDcls production:

pgmDcls → $(\text{pDcls})^*$

pDcls → **var** Variables
  | uDcls

That is, a program declaration section may consist of zero or more `var, namespace, const, val, type, static, storage, readonly, segment, procedure, iterator,` or `method` declarations. The expansion of the non-terminals following each of the terminal symbols in this production is the subject of much of the rest of this document.

Implementation of the *pDcls* production is rather simple. Basically, the `pgmDcls` function (which implements both the *pgmDcls* and *pDcls* productions above) checks to see if there is a `var` keyword and calls `parseVar` to handle the `var` section declarations. If the current keyword is not `var`, then `pgmDcls` calls the `uDcls` (unit declarations) procedure to handle everything else. This is possible because units and programs share the same set of declarations except for the fact that units don't support a `var` declaration section.

`pgmDcls` begins by calling `resetTknQ` to clean out HLA's "attribute queue" that keeps track of attributes from past symbols it has scanned. Because `pgmDcls` is only called once when compiling an HLA program, this call to `resetTknQ` isn't strictly necessary (basically, only the tokens for `program`, the identifier, and a semicolon will be in the queue at this point), but it's always a good idea to call this whenever you begin parsing a new production and you don't need any inherited attributes from previous productions (as is certainly the case here).

The next thing that `pgmDcls` does is set the current lex level (curLexLevel) to one, as this is the lexical level for all declarations in a main HLA program. After setting the lex level to one, pgmDcls enters a loop that processes zero or more occurences of the various declaration sections. For each iteration of the loop, this procedure calls the lexical analyzer to get the next symbol from the source file. It checks the symbol to see if it's a `var` reserved word and calls `parseVar` if this is the case. Otherwise it calls `uDcls` to handle the other declaration sections. The `parseVar` procedure returns whenever it finishes processing the `var` declaration section and comes across a reserved word it doesn't know how to handle. `uDcls` returns when it finds something it cannot process -- either a `var` keyword, a `begin` keyword, or an error.

unitDcls - Parsing Unit Declarations

The `unitDcls` procedure is very similar to the `pgmDcls` procedure, except instead of processing `var` sections (which are not legal in a `unit`), this procedure reports an error if it comes across a `var` keyword.

uDcls - Parsing the declarations within a UNIT

The `uDcls` procedure is responsible for parsing the actual declaration sections that can be found in a `unit`. The production corresponding to these declarations is

```
uDcls      →   namespace parseNamespace optionalSemicolon
           |   label labels
           |   segment SegmentVars
           |   nsDcls
```

This takes the same “layered” approach that `pDcls` uses. That is, it processes all the productions that are unique to unit (and program) declarations, and then it call `nsDcls` to handle the parsing that is common to programs, units, and namespaces.

parseNamespace - Parsing a Namespace Declaration

The parseNamespace procedure handles the processing of a `namespace` declaration. Beyond processing the actual “namespace name;” and “end name;” clauses of this declaration, this parsing operation is very similar to processing a `program` except that you can’t have nested namespace declarations and namespaces don’t support a `var` declaration section.

parseNamespace → identifer ‘;’ (nsDcls)* **end** identifier ‘;’

The `parseNamespace` procedure actually does a lot of work in addition to simply calling `nsDcls`. Namespaces require their own local symbol table, so it is up to `parseNamespace` to create this new symbol table and activate it while processing all the namespace declarations (all the declarations between the `namespace` and `end` keywords wind up in the namespace’s local symbol table; see the documentation on HLA’s symbol table format for more details on local symbol tables). After creating the symbol table entry for the namespace identifier and attaching the local symbol table to it, the parseNamespace procedure processes all the declaration sections that are legal in a namespace. Afterwards, it checks the validity of the end clause (i.e., the identifier must match the one used in the namespace declaration, cleans up after itself, and then reactivates the original symbol table (the one in use prior to activating the namespace’s local symbol table).

nsDcls - Parsing the Declarations in a Namespace

The `nsDcls` procedure handles the processing of the declarations that actually appear between the `namespace` and the corresponding `end` statements. Here are the productions for the *nsDcls* non-terminal:

```
nsDcls     →   const Constants
           |   val Values
           |   type Types
           |   static StaticVars
           |   storage StorageVars
           |   readonly ReadonlyVars
           |   procedure  procStuff
           |   iterator procStuff
           |   method procStuff
```

Note that upon entry to the `nsDcls` procedure, the lexer must have already have been called and EAX/EBX/ECX should contain the values returned by `lex`. This function returns with the carry flag clear if it matches one of the productions above or it didn't match anything; it returns with the carry flag set if there was an error when matching one of the above productions or if it encounters a `segment` or `namespace` reserved word (which are illegal in a `namespace`). `var` declarations are also illegal in namespaces, but whomever calls `nsDcls` must explicitly check for this, if it is to be disallowed, because other (non-namespace related procedures also wind up calling `nsDcls`). Note that on return, whomever called `nsDcls` must call lex to read the next token from the source file.

parseType - Parsing a TYPE Declaration Section

The `parseType` function handles the parsing of the HLA type declaration section. Here's the grammar for the declarations following the `type` reserved word:

```
Types → ( typeDcls )*

typeDcls       →   ';'
               |   tID ':' tDcls

tDcls          →   typeID  optionalbounds   ';'
               |   enum '{' idList '}'    ';'
               |   pointer to TypeID   ';'
               |   forward '(' fID ')'   ';'
               |   record recStuff   ';'
               |   union recStuff   ';'
               |   class classStuff   ';'
               |   procedure optionalParms   ';'  protoOptions

dimList → constExpr ( ',' constExpr )*

optionalBounds   →   '[' dimList ']'
                 |   ε

protoOptions     →   ε
                 |   @returns '(' constExpr ')'  ';'  protoOptions
                 |   @pascal  ';'  protoOptions
                 |   @stdcall   ';'  protoOptions
                 |   @cdecl  ';'  protoOptions

typeID         →   identifier
               →   builtInTypes

builtInTypes   →   thunk
               |   constBITypes

constBITypes→   boolean
               |   uns8
               |   uns16
               |   uns32
               |   uns64
               |   uns128
               |   byte
               |   word
               |   dword
               |    qword
               |    tbyte
               |    lword
               |    int8
               |    int16
               |    int32
               |    int64
               |    int128
               |    char
               |    xchar
               |    unicode
               |    real32
               |    real64
               |    real80
               |    real128
               |    string
               |    ustring
               |    cset
               |    xcset
               |    text
```

`parseType` reads a lexeme from the source input stream and decides what to do based on the token. If it's a semicolon, `parseType` just eats the semicolon and loops back to read another token from the source file. If the input symbol is a locally defined symbol, then `parseType` reports a "duplicate defined symbol" error and then treats the symbol as though it were undefined (by calling the `makeUndefID` procedure, which adjusts the token's values and attributes as appropriate for an undefined symbol. If the input symbol is a global identifier, then `parseType` "converts" it to an undefined identifier token. If the symbol is an undefined identifier (or was converted to an undefined identifer), then `parseType` stores the pointer to the token in a local variable and it stores pointers to the identifier's true name and lower case name into the `trueName` and `lcName` local variables. The `tDcls` procedure (described in a moment) will refer to the values in these local variables when processing the actual type declaration.

After processing the undefined identifier, the `parseType` procedure checks to see if there is a colon following the identifier (type declarations consist of an identifier followed by a colon and the type info). After matching the token, the `parseType` procedure calls `tDcls` to extract the type information and process the specific type declaration. On return from `tDcls`, `parseType` loops back and repeats the process for each type declaration in the `type` section.

The `parseType` procedure declares the following local variables:

```
typeSym     :symNodePtr_t;
typeToken   :tokenPtr_t;
fwdName     :string;
trueName    :string;
lcName      :string;
dimensions  :dimPtr_t;
arrayType   :symNodePtr_t;
nameConst   :attr_t;
```

As it turns out, parseType only directly uses `typeSym, trueName,` and `typeToken`. The remaining variables are actually for use by the `tDcls` procedure. One technique you'll see used often throughout the HLA source code is that procedures that are nested inside other procedures (like `tDcls` is nested in `parseType`) are usually declared with the `@noframe` procedure option. Such procedures, like `tDcls`, cannot have their own local variables (as there is no stack frame in which to put them). This allows the local procedure (e.g., `tDcls`) to easily access "intermediate variables (variables global to that procedure, but local to some enclosing procedure) by simply using the "`ebp::`" prefix in front of the intermediate variable's name. In any case, the extra variables above (`typeSym,` fwdName, dimensions, arrayType, and nameConst) are actually local variables to the tDcls procedure. However, they wind up getting declared in parseType because tDcls doesn't have a stack frame.

tDcls - Parsing the Type Information in a TYPE Declaration

The tDcls procedure handles the job of determining the actual type for a type declaration and entering the type into the symbol table. Upon entry into this procedure, tDcls assumes that the parseType procedure has already processed the identifier at the beginning of the line and the colon that immediately follows. This procedure also assumes that the trueName and lcName variables point at the appropriate strings for the identifier and that typeToken contains a pointer to the token (returned by lex) for this identifier.

The tDcls procedure handles the following productions in the HLA grammar:

```
tDcls      →  typeID  optionalbounds   ';'
           |  enum '{' idList '}'    ';'
           |  pointer to TypeID   ';'
           |  forward '(' fID ')'   ';'
           |  record recStuff   ';'
           |  union recStuff   ';'
           |  class classStuff   ';'
           |  procedure optionalParms   ';'  protoOptions
```

The first production, "tDcls → typeID optionalbounds   ';'" handles type isomorphisms (renaming types) and array type declarations. Here are a couple of examples:

```
type
      integer :int32; //A simple type isomorphism
      intArray :int32[4];
      array2D :byte[4,4];
```

tDcls begins by calling getTypeID to see if the next token (after the ':') is a type identifier (the getTypeID procedure, by the way, is found in the hlautils.hla source file). getTypeID returns with the carry clear if the next token is a type identifier, it returns with the carry set if the next token is not a type identifier. Although getTypeID calls the lexer to fetch the next token, if it's not a type identifier, getTypeID pushes the token back onto the input stream, so whomever calls getTypeID must call lex again to read the next token if it was not a type identifier.

If getTypeID determines that the next item in the source file is one of the built-in types, then it will create a fake attribute for the type declaration. If the next item is a bonafide identifier, then getTypeID will determine its base type and return that. This function returns a pointer to the symbol table entry for that base type (or one of the symbol table entries created for the built-in types) in the EAX register.

Back in tDcls, if getTypeID returns with the carry flag clear (meaning it has found a type identifier), then it processes the declaration. First, tDcls calls createTypeSym to create a new symbol table entry for the symbol scanned at the beginning of the statement (i.e., the symbol that parseType processed). Next, tDcls checks to see if there are any array bounds following the type name (meaning that we're declaring a new array type) by calling optionalBounds (described later in this document). The optionalBounds procedure returns NULL in the EAX register if there were no array bounds, otherwise it returns a pointer to a dimensions_t object in EAX (a dimensions_t object is an integer containing the number of dimensions followed by that many integers specifying the bounds for each array dimension).

If array bounds are present and the array declaration specifies more than a single dimension, then the tDcls procedure needs to build a series of symbol table entries, one for each dimension of the array except the first. This is because the HLA symbol table format only allows single dimension array declarations; multi-dimensional arrays are handled by creating "arrays of arrays"[1]. The document on the symbol table describes the exact format for array dimension types in the symbol table; please see that document for more details on the exact nature of this structure. Note that for each of the intermediate types, the tDcls procedure creates a single-dimension type declaration with an identifier like “@array000” (substituting a unique integer value for “000”). For the first (or only) array dimension, tDcls places the array bounds information directly in the symbol table entry it creates for the type declaration (i.e., by storing the bounds for that dimension in the `numElements` field of the symbol table entry and setting the `objectSize` field accordingly).

If there are no optional array bounds following the type identifier, then we’ve got a type isomorphism and the `tDcls` procedure simply copies the base type information provided by the typeID into the symbol table entry of the new type that we’re creating.

If the first lexeme following the colon in the type declaration is not an identifier, then `tDcls` checks to see if it is one of the reserved words: `enum, forward, pointer, record, union, class,` or `procedure`. The following paragraphs describe how `tDcls` handles each of this possibile tokens that may legally appear in a type declaration.

tDcls $\rightarrow$ **enum** ’{’ idList ’} ’;’

If the reserved word enum appears immediately after the colon, then the user is creating an enumerated type. `tDcls` calls `createTypeSym` to create the symbol table entry for the new type and then calls `buildEnumType` (described later in this document) to process the enumeration list. On return from `buildEnumType`, the symbol table will not only contain a symbol table entry for the new type, but it will contain symbol table entries for each of the constants appearing in the enumeration list. See the discussion of `buildEnumType`, later, for more details.

tDcls $\rightarrow$ **pointer to** TypeID ’;’

If the two lexemes “pointer” and “to” follow the “id:” at the beginning of a declaration, the `tDcls` creates a pointer type. Following the “to” keyword must be a single identifier (or one of the predefined type reserved words). If the identifier is a type identifier (local, global, or built-in type) then `tDcls` enters the new symbol into the symbol table as a pointer object whose base type is set to the symbol table entry of `TypeID` parsed in the production above. If the symbol is undefined, then tDcls adds the symbol to a “forward declarations” list to check to see if the symbol is defined before the current program unit (program/unit/procedure/method/iterator) is finished. If the symbol is defined, but is not a type declaration, then HLA reports an error.

> **Point of contention**: what happens if the symbol is a global symbol but not a type ID. HLA reports an error. In fact, it could be a local type ID that hasn’t been defined yet. How to handle this? Perhaps the code should treat global symbols as undefined and report the class error (“not a type declaration”) at the end of the program unit?

tDcls $\rightarrow$ **forward** ’(’ fID ’)’ ’;’

A `forward` declaration isn’t an actual type declaration. In fact, it’s a `const` text declaration. Each declaration section, including type declarations, provides a forward declaration to allow the use of macros when processing declarations. A declaration like the following:

```
someID :forward( fID );
```

is really equivalent to the following:

```
?fID :text := “someID”;
```

1. Note that HLA v1.x’s symbol table format allowed multi-dimensional arrays to be declared in a single symbol table entry. Experience with HLA v1.x, however, shows that the “array of arrays” approach is more generic and easier to maintain.

The purpose of the `forward` declaration is to allow you to create a macro that defers the declaration for a given symbol while you insert some other declarations. What the forward declaration allows you to do is grab the identifier name appearing at the beginning of a type declaration and temporarily save it while you insert some other declarations within the `type` declaration section. Consider the following macro:

```
#macro myType( theType ):theID, IDtemp;
      forward( theID ); // Capture ID at beginning of declaration
      ?theID := @string( @text( (@string( IDtemp ))));
      @text( theID + "_t") : theType;
      ?@text( theID + "_size" ) := @size( theType );
      ?IDtemp :text := theID + "_t"
#endmacro
```

Consider the following type declaration:

```
type
      i :myType( int32 );
```

This results in the following declarations:

```
type
      i_t : int32;
      ?i_size := 4;
      ?i :text := "i_t";
```

Notice how each of these declarations use the name appearing in the original type declaration.

To process a forward declaration the `tDcls` procedure first matches the `forward` reserved word and an opening parenthesis. Then it calls the `lex` procedure to fetch (what better be) an identifier. The identifer must either be undefined, a global symbol, or a `val` object if it's a local symbol; otherwise `tDcls` reports a duplicate symbol error. If the symbol was previously undefined in the local scope, the `tDcls` code creates a `const` class text constant and initializes it with the string found in the `trueName` variable. If the symbol is defined in the local scope (meaning it's a `val` class object), then `tDcls` calls `setVal` to changes its value to the string specified by `trueName`.

An important thing to note about the code that processes forward declarations is that it must free the storage associated with `lcName`. When the lexical analyzer originally scanned this identifier, it allocated storage on the heap for both the `trueName` and `lcName` strings. In a normal declaration, the symbol table would retain pointers to both of those strings (for type neutrality checking). However, in the case of a forward declaration, the `lcName` string does not get used, so the `tDcls` procedure frees this storage to reclaim the memory the `lcName` string uses.

tDcls → **record** recStuff';'
tDcls → **union** recStuff ';'

From `tDcls`' point of view, records and unions are quite easy to handle. If `tDcls` sees the record or union reserved word, it creates a symbol table entry for the type name (at the beginning of the statement) and then calls `parseRecord` or `parseUnion` (respectively) to do the real work (which is not easy). A discussion of `parseRecord` and `parseUnion` appears later in this document.

tDcls → **class** classStuff ';'

At the time this document was being written, classes had yet to be implemented. Within `tDcls`, however, the code is pretty much the same as records and unions.

tDcls → **procedure** optionalParms ';' protoOptions

If tDcls sees the procedure reserved word, it calls the parseProcType to handle the actual processing of the procedure type. See the discussion of parseProcType elsewhere in this document.

parseConst - Parsing a CONST Declaration Section

The parseConst procedure handles the parsing of the HLA const declaration section. This section begins with the const keyword and supports the following grammar:

```
Constants → ( constDcls ) *
constDcls    →   ’;’
             |   identifier cDcls
```

Note that identifier must be an undefined identifier or a global ID.

Like the other parse* routines in the parse declarations module, the parseConst procedure contains a loop that processes all of the declarations in the current const section. Each iteration of this loop begins by resetting the token queue (which maintains attributes for each of the tokens parsed on the line), then it calls the lexer, expecting to find an identifier or a semicolon.

If parseConst encounters an identifier, it checks to see if it’s a globally-defined ID. If so, it converts it to an “undefined” identifier for local use. If the identifier is a local ID, then parseConst reports a duplicate symbol error (and converts it to an undefined ID just to ease further error recovery). If parseConst encounters an undefined ID (or converts a defined ID to an undefined ID), it then goes about the business of creating a new constant declaration in the local symbol table. It begins by copying the string pointers for the identifier into the constID and constlcID local variables (constID holds the actual identifer string, constlcID holds the lower-case version of the identifier’s name). It also sets up the constType and constToken local variables. The cDcls procedure (described in a moment) uses the values placed into these local variables. After setting up these locals, parseConst calls the cDcls procedure to handle the remainder of the constant declaration (see the productions above).

On return from cDcls, the parseConst procedure checks the constant it found to see if it matches the declared type (if there was one) and enters the constant into the symbol table if everything is correct. The cDcls routine will create a dummy constant (and type) to help prevent some indeterminate results when cDcls returns. After processing the symbol (or reporting a type mismatch error), the loop in the parseConst procedure repeats, calling the lexer to fetch another symbol from the input stream.

When the parseConst procedure scans a symbol that is not a semicolon or an identifier, it pushes the token back onto the input stream and returns. Note that this is not an error condition. When parseConst encounters something it doesn’t know how to deal with, it simply returns to whomever called it and lets them deal with that token.

cDcls - Parsing the Type and Value Portions of a Constant Declaration

The cDcls procedure handles everything to the right of the identifier in a constant declaration. The relative productions for the cDcls procedure are

```
cDcls        →   ’:’ ForC  ’;’
             |   ’:=’ constExpr ’;’
```

The ForC non-terminal (and the corresponding procedure) handle constant declarations that have an explicit type specification or use one of the special constant declarations (see the discussion of ForC in the next section). The other production that cDcls handles is a straight assignment without an explicit type.

Upon entry, the cDcls procedure calls lex to fetch the next token from the input stream. It checks this token against a colon or an assignment operator. If it is neither of these, then cDcls reports a syntax error and dummies up a return result. If it encounters a colon (‘:’) then it calls the ForC procedure to handle the remainder of the declaration up to the semicolon and checks for the presence of the semicolon upon return.

If the cDcls procedure encounters an assignment operation, it calls constExpr to process the expression which follows the assignment operator. As there has been no explicit type declaration, the cDcls procedure sets the return type to whatever type constExpr returns for the expression.

In either case (whether encountering a ‘:’ or the assignment operator, ‘:=’), the `cDcls` procedure scans for a semicolon at the end of the declaration to complete the parsing of a single constant declaration. If either `constExpr` or `ForC` fails, either call will stick a dummy value into the returned attribute (typically the boolean value ‘false’) to prevent cascading errors and indeterminate results later.

ForC - Parsing Special Constant Declarations and Those With Explicit Types

The `ForC` procedure handles the components of a constant declaration following the colon. This is either an explicitly typed constant declaration or one of three special constant declarations. Here are the productions for the `ForC` non-terminal:

ForC	→	<u>typeID</u> optionalbounds ’:=’ constExpr
	|	**enum** ’{’ idList ’}’ ’:=’ constExpr
	|	**pointer to** <u>ptrTypeID</u> ’:=’ constExpr
	|	**forward** ’(’ fID ’)’

<u>typeID</u> must be a predefined identifier that specifies some data type or it must be one of the predefined HLA data type reserved words. <u>ptrTypeID</u> must be a predefined type identifier or an undefined symbol; if it is undefined, you must define it as a type object before the end of the current program unit (`program, unit, procedure, iterator,` or `method`).

`ForC` begins by checking for a type identifier in the input stream. If it finds one, then it checks for an optional set of array bounds surrounded by square brackes (e.g., “[2,3,4]”). If the array bounds are present, and the bounds specify more than a single dimension, then `ForC` builds a set of anonymous array types for each of the dimensions but the first. This process is identical to that used by `tDcls`; please see the discussion appearing earlier for the `tDcls` procedure. After the array bounds (or if the bounds are not present), `ForC` scans for an assignment operator (‘:=’) and reports an error if it is not present. If the assignment operator is present, then `ForC` calls `constExpr` to evaluate the constant expression immediately following. Note that `ForC` does not check to see if the type of the expression matches the defined type. Instead, it simply stores the defined type into the `constType` variable and leaves it up to the `parseConst` procedure to check the types once control returns there.

If ForC does not encounter a type identifer in the input stream, it scans for a token and checks to see if it’s a `pointer, enum,` or `forward` token. Forward declarations are completely identical to those in the `type` section; please see the discussion in the section on `tDcls` for more information about the forward constant declaration.

If a bare enum declaration appears in a const section, `ForC` will create an anonymous type for the enumerated type, create a symbol table entry for each of the enumerated constants, and then initialize the constant declaration with the specified enumerated constant value. E.g.,

```
const
      eConst : enum{ a, b, c } := b;
```

Defines four constants and an anonymous type in the symbol table. The symbol `a` is a constant given the value zero, `b` is given the value one, and `c` is given the value two. This declaration (of course) also creates an entry for `eConst` having the value one. This declaration also enters a `type` symbol table entry of the form “@enum000” (where 000 represents some unique integer value). The other four symbols have this type.

At the time this was being written, pointer constants were not implemented yet. This document will be updated when pointer constant parsing is added to the HLA v2.0 source code.

parseVal - Parsing the VAL declaration Section

The `parseVal` procedure and operation is virtually identical to `parseConst`. It handles the `val` declaration section. The only difference is that `parseVal` doesn't require symbols to be unique - you may redefine a symbol declared in the `val` section. Here are the productions for the `val` section that `parseVal` handles:

```
Values → ( valDcls )*
valDcls    →   ';'
           |   identifier vDcls

vDcls      →   ':' ForV ';'
           |   ':=' constExpr ';'
           |   '+=' constExpr ';'
           |   '-=' constExpr ';'

ForV       →   typeID  optionalbounds  optAssign
           |   enum '{' idList '}'   optAssign
           |   pointer to ptrTypeID   optAssign
           |   forward '(' fID ')'
```

Another difference you will notice between the `val` productions and the `const` productions is the the `val` productions allow the C-like "+=" and "-=" operators (which wouldn't make sense in the `const` section, since the symbol must be undefined in a `const` declaration and these operators require a predefined value).

The other major difference between `parseConst` and `parseVal`, of course, is that `parseVal` calls `setVal` to change the value of a symbol table entry rather than `enterConst` to enter a new constant into the symbol table. See the discussion of `parseConst` for more details.

parseVar - Parsing the VAR Declaration Section

(still to be written)

parseStatic - Parsing the STATIC Declaration Section

(still to be written)

parseStorage - Parsing the STORAGE Declaration Section

(still to be written)

parseReadOnly - Parsing the READONLY Declaration Section

(still to be written)

parseSegment - Parsing the SEGMENT declaration Section

(still to be written)

parseProc - Parsing a Procedure declaration

(still to be written)

parseMethod - Parsing a Method declaration

(still to be written)

parseIterator - Parsing an Iterator Declaration

(still to be written)

parseRecord & parseUnion - Parsing record/union declarations

Record and union parsing are relatively complex because of the various options and facilities such as anonymous unions and records. Here are the productions for record and union declarations (note that the `record` or `union` keyword has already be parsed by a higher-level declaration section; the following productions start with the first token beyond the `record` or `union` keyword):

recordDcl → recOptions recunVars privateRecVars **endrecord**

unionDcl → recunVars **endunion**

recOptions → ε
| inherits ’(’ identifier ’)’
| setRecOffset
| recAlignment

recunVars → ruVars (ruVars)*

setRecOffset → ‘:=’ constExpr ‘;’

recAlignment→ ‘[‘ maxAlign constExpr ‘]’ ‘;’

maxAlign → ε
| constExpr ‘,’

privateRecVars → ε
| **private** ’:’ recunVars

ruVars → **align** ’(’ constExpr ’)’ ’;’
| **record** recunVars **endrecord** ’;’
| **union** recunVars **endunion** ’;’
| orID ’:’ ruDcls

ruDcls → **forward** ’(’ constExpr ’)’
| **record** recordDcl ’;’
| **union** unionDcl ’;’
| **procedure** optionalParms protoOptions ’;’
| **pointer to** typeID ’;’
| typeID optionalBounds’;’

orID → identifier
| **override** identifier

Record provide three options that the `recOptions` procedure handles. These options let you specify a set of fields to inherit from some other record, a starting offset for the record, or an alignment value for record fields (note that these options are mutually exclusive - you cannot specify both a starting offset and an inherited set of fields or an alignment value for the same record). Here are some examples of declarations that demonstrate these options:

```
type
      ancestor : record
            anscestor_field:dword;
      endrecord;
```

```
descendent : record inherits( ancestor );
      descendant_field:dword;
endrecord;

HasOffset : record := 4;
      offsetIs4:dword;
endrecord;

HasAlign : record [4];
      offset0:byte;
      offset4:word;
      offset8:dword;
endrecord;

HasAlign2 : record [4, 2];
      offset0:byte;
      offset2:word;
      offset4:dword;
endrecord;
```

The `recOptions` procedure checks for these three record options immediately after the record keyword in a record declaration. This procedure begins be resetting the token queue and then calling the lexical analyzer to fetch the next token from the input stream. If this token is for the `inherits` keyword, then `recOptions` processes the inherits option, if it's the assignment operation, `recOptions` processes the offset option, if it's an opening bracket, `recOptions` processes the alignment option. If it is none of these tokens, then `recOptions` pushes the token back onto the input stream and returns, letting whomever called `recOptions` handle that token.

If `recOptions` encounters the `inherits` keyword, then it calls the lexical analyer to fetch an opening right parenthesis, an identifier, and a closing right parenthesis. The identifier must be defined and it must be a type ID that defines a record; otherwise `recOptions` reports an error. If the identifier is a record type ID, then for each of the non-private classes in that record, `recOptions` copies the field definitions into the new record type being created. For type compatibility reasons, `recOptions` only copies the actual field defintions from the inherited type to the descendant type. Records may also contain certain other type and placeholder fields (for example, if you have a multidimensional array HLA will emit some anonymous type declarations); `recOptions` does not copy these fields because typechecking in HLA is done by comparing the pointers to the symbol table entries (if the pointers are the same, then the types are equal). By copying only the field definitions, `recOptions` ensures that the fields that reference these anonymous types have the same type as the ancestor fields.

While copying the (non-private) fields from the ancestor to the descendant record type, the `recOptions` procedure also computes the starting offset of the first field of the descendant record. The starting offset of the first new field is the offset of the last field in the ancestor record plus the size of that last field's type. Note that we cannot simply use the size of the inherited record as the starting offset of the first (non-inherited) field of the new record because the ancestor field could have had a non-zero starting offset (using the ":=" record option).

If the `recOptions` procedure encounters the assignment operator token (":=") immediately after the record keyword, it verifies that the following is a 32-bit (or smaller) numeric constant expression and sets the starting offset of the record to this value.

There are two forms of the alignment option that `recOptions` processes. One form expects a single constant expression. This value specifies a field alignment for the record's fields and the parser will add padding bytes to the record to ensure that each fields starts at an offset that is an even multiple of this value. The second form expects two expressions: the first is a maximum alignment, the second is a minimum alignment. When aligning fields, all data types whose size is less than the minimum alignment will be aligned to an offset in the record that is a multiple of the minimum alignment. All fields whose size is greater than the maximum alignment will be aligned to an offset that is an even multiple of the maximum alignment value. All fields whose size lies between the minimum and maximum sizes will be aligned to an offset that is a multiple of the object's size.

recunVars Procedure

The `recunVars` procedure handles the chore of processing all the field declarations that occur in a record or a union. It runs in a loop calling `ruVars` to process each field of the record or union until it encounters something that is not a record/union field. The `recunVars` procedure also has the responsibility of updating the size of the record or union after processing each field. To compute the size of a record, `recunVars` adds the size of each field to a running sum. To compute the size of a union, `recunVars` computes the maximum size of all the fields in the union.

ruVars Procedure

The `ruVars` procedure handles a single (legal) declaration inside a record or union. Here's the grammar productions for the text that `ruVars` recognizes:

ruVars	→	**align** '(' constExpr ')' ';'
	|	**record** recunVars **endrecord** ';'
	|	**union** recunVars **endunion** ';'
	|	orID ':' ruDcls

orID	→	identifier
	|	**override** identifier

If `ruVars` encounters an `align` keyword, it fetches an opening parenthesis, a constant expression, a closing parenthesis and a semicolon. If the constant expression is not a 32-bit (or smaller) integer value, `ruVars` reports an error. If the token sequence is syntactically and semantically correct, then ruVars takes the current offset into the record and adds a sufficient value to make the offset an even multiple of the value specified by the constant expression.

If `ruVars` encounters a record keyword (without the usual "id:" prefix indicating a standard nested record declaration), then the user is creating an anonymous record declaration within a record or union. In this case, `ruVars` emits a special `AnonRec_pt` type record to the current record to mark the start of the anonymous record and then it recursively calls `recunVars` to process the fields of that record. Upon return, it checks for the matching `endrecord` token and enters an `EndAnonRec_pt` type symbol ("@endAnonRecXXX" into the symbol table to mark the end of anonymous record. Note that for anonymous records, a local symbol table is not created. Instead, `ruVars` enters the fields of the anonymous record directly into the current record's or union's symbol table.

If `ruVars` encounters a union keyword, then it creates an anonymous union. The process is almost identical to that for creating an anonymous record except the "bracketing fields" are of type `AnonUnion_pt` and `EndAnonUnion_pt`.

The only remaining thing the `ruVars` handles is a bonafide record declaration. This takes one of two forms:

```
            identifier: type definition;
-or-
            override identifer : type definition;
```

The `override` keyword is used to reuse a fieldname present in an inherited record. If the `override` prefix is present, then `ruVars` will lookup the existing symbol in the record or union (copied from the ancestor object) and redefine its type fields according to whatever `ruDcls` finds after the colon. If the symbol does not already exist as an inherited field, then `ruVars` reports an error. Technically, the override prefix does not require the symbol to have been defined in the ancestor record - it can also override a symbol in the existing record or union. However, override is generally used to override inherited fields (there probably is no reason for applying it to fields existing in the current record or union, but this is allowed as a generalization).

If the current declaration is a straight record/union field declaration, then `ruVars` first checks to see if the symbol is a globally or locally defined identifier. Local symbols must have an override prefix or we've got a duplicate symbol error. If ruVars encounters an undefined identifier (or forces a global or local symbol to “undefined” after reporting an error or other correction), then it calls ruDcls to handle the remainder of the field’s declaration.

ruDcls Procedure

`ruDcls` has the task of actually processing a record or union field declaration. The `ruDcls` procedure gets called after `ruVars` processes an identifier and a colon. Here’s the grammar for the statements that this procedure handles:

```
ruDcls      →   forward ’(’ constExpr ’)’
            |   record recordDcl ’;’
            |   union unionDcl ’;’
            |   procedure optionalParms  protoOptions  ’;’
            |   pointer to typeID ’;’
            |   typeID optionalBounds’;’
```

Forward declarations in a `record` or `union` have the same purpose as in a `const` or `type` declaration. They are also handled exactly the same way by `ruDcls`. Please see the discussion of forward declarations in the earlier discussion of the `tDcls` procedure for more details.

The `ruDcls` encounters the `record` keyword in a declaration, then we’ve got a nested record declaration inside the current record or union. If this is the case, then `ruDcls` first creates an anonymous symbol table entry for a record type and then calls `parseRecord` (possibly recursively) in order to process the record declaration associated with the current field.

The `ruDcls` encounters the `union` keyword in a declaration, then we’ve got a nested union declaration inside the current record or union. If this is the case, then `ruDcls` first creates an anonymous symbol table entry for a union type and then calls `parseUnion` (possibly recursively) in order to process the union declaration associated with the current field.

If `ruDcls` encounters the procedure reserved word, then the field is a procedure pointer declaration. `ruDcls` calls `parseProcType` to handle the actual parsing of the procedure declaration and then sets the field’s size to four bytes (procedure pointers are always four bytes and adjusts

If `ruDcls` encounters the two token sequence “pointer to” after the “identifier:” sequence in the current declaration, then `ruDcls` creates a new symbol table entry holding an anonymous pointer type and then sets the current field’s type to this anonymous pointer type. As for type declarations, the identifier following the “pointer to” token sequence must either be a defined type identifier or an undefined symbol. If the symbol is currently undefined, `ruDcls` calls `addFwdPtr` to add it to the list of forward referenced pointer variables and the program must define that symbol prior to leaving the current lex level.

Last, but certainly not least, comes a standard field declaration consisting of a type identifier and an optional set of array dimension bounds. If `ruDcls` comes across an identifier, it first verifies that it is a type ID and then calls `getBaseIsoType` to produce the base type. After getting the base level type, ruDcls checks for the optional array bounds, generating any anonymous type symbol table entries, as necessary, to handle two or more dimensional arrays. Finally, `ruDcls` enters the original symbol into the symbol table, using the address of the typeID as the field’s type.

Utility Routines

makeLabel

optionalBounds

buildEnumType

parseProcType