DDL(v.2)

data definition language

Sergey Strukov 2014, version 1.00

1. Introduction.

DDL is the acronym of the “data definition language”. It is a computer language, designed to represent data of general kind in a human readable form. The **DDL** design is based on the following principles:

1) it must be simple,
2) it must look familiar,
3) it must be human friendly,
4) it must be capable to represent a complex data,
5) it must be a typed language,
6) it must be commutative wherever logically possible,
7) it must be modular,
8) it must be compatible with the modern programming languages.

Here is a simple **DDL** text example:

```
sint8 s8 = 1 ;
uint8 u8 = 2 ;

sint32 s32 = 3 ;
uint32 u32 = 4 ;

int i = 5 ;
sint s = 6 ;
uint u = 7 ;

text t = "text" ;
ip a = 192.168.1.10 ;

text *ptr_t = &t ;

text at_ptr_t = *ptr_t ;

int[10] array_len_i = {1,2,3,4,5,6,7,8,9} ;

int[] array_i = {1,2,3,4,5} ;
```

```
struct Struct
 {
  int fi = 1 ;
  uint fu = 2 ;
  text ft = "default" ;
 };

type List = Struct[] ;
```

Looks familiar, isn't it? **DDL** is a **C-**style language. **DDL** text defines **types** and **constants** of defined types. It does it pretty the same way, as **C** does. But there are some important differences. First, the language is commutative, i.e. the order of declarations is not significant. The second important feature is you can define default values for the structure fields. Unlike in **C**, in **DDL** to specify an array type square brackets are written after the element type, not after the variable name. The other important difference is the expression interpretation rules.

There are four main language entities: **types**, **constants**, **literals** and **expressions**. Types define the constant properties, constants comprise data, defined by the **DDL** source, literals defines values of base types, expressions combines one values to produce another using operations, defined for types. In **DDL** the result of expression evaluation is defined by the target type. For example,

```
uint8 a = 1000/500 ; // a ←0
uint32 b = 1000/500 ; // b ←2
```

The first expression is evaluated as `(uint8)1000/(uint8)500`, the second as `(uint32)1000/(uint32)500`.

There are following basic types in **DDL**:

```
sint8  uint8
sint16 uint16
sint32 uint32
sint64 uint64

sint   uint
       ulen

text
ip
```

Types `uintN` represent unsigned N-bit integers. Types `sintN` represent 2's complementary signed N-bit integers. The couple `(sint,uint)` is a platform dependent types, it must be identical either `(sint32,uint32)` or `(sint64,uint64)` (but different as types). `ulen` is a platform dependent type, it must be identical either `uint32` or `uint64` and has bit length at least the bit length of `uint`. All array dimensions are represented by this type. `int` is the same type as `sint`.

Types `text` and `ip` are special. First represents text strings, second – `IPv4` addresses.

Other types are derived types.

Pointer types:

```
type-definition *                          - simple pointer
{type-definition1,...,type-definitionn} * - polymorphic pointer
```

Array types:

```
type-definition[]    - variable length array
type-definition[Len] - fixed length array
```

Structure types:

```
struct Struct
 {
  type-definition1 field1 = expression1^opt ;
  ...
  type-definitionn fieldn = expressionn^opt ;
 }
```

Another language element is **type alias**:

```
type T = type-definition ;
```

Alias is not a new type, it is a synonym for a type.

Constants are defined as following:

```
type-definition C = expression ;
```

Expressions are used to set values to constants. Simple expression is a name of a constant or a literal:

```
int a = 12345 ; // literal expression
int b = a ;     // constant name as an expression
```

Operator expressions specify operation on the argument(s):

```
int a = 12345 ;
int b = 67890 ;
int c = a+b ;   // binary operator
int d = -c ;    // unary operator
```

There are following unary operators: `+`, `-`, `*`, `&`.

There are following binary operators: `+`, `-`, `*`, `/`, `%`.

Brackets may be used as usual:

```
int e = (a+b)*c ;
```

There is a special integer cast operator:

```
int f = sint8( a+d ) ;
```

The expression inside the brackets is evaluated to the specified target type.

Field selection expressions are used to access structure fields:

```
struct Point
 {
  int x;
  int y;
 } point = {100,200} ;

int X = point.x ;
int Y = (&point)->y ;
```

The index expression is used to refer to an array element:

```
int A[10] = {1,2,3,4,5} ;
int x = A[3] ;
```

There are following types of literals:

```
1234567890               - decimal literal
1234567890abcdefABCDEFh  - hexadecimal literal, suffix is h or H
10001b                   - binary literal, suffix is b or B
192.168.1.10             - ip literal
“abcdef\b\t\n\v\f\r“     - string literal with \-characters
'abcdef'                 - simple string literal
null                     - universal null literal
```

To organize a **DDL** content scopes can be used:

```
scope Name { scope-content }
```

Scopes can be nested. To refer an entity the following kinds of qualified names can be used:

```
name₁#name₂#...#nameₙ – relative name
#name₁#name₂#...#nameₙ – absolute name
.#name₁#name₂#...#nameₙ – based name, current scope is the base
..#name₁#name₂#...#nameₙ – based name, parent scope is the base
...#name₁#name₂#...#nameₙ – based name, grand-parent scope is the base
```

Finally, the file inclusion can be used:

```
include <file-name>
```

The content of the included file must be scope-content. Undefined names are permitted.

For example:

```
/* file1.ddl */

int X = 100 ;
int Y = 200+Custom#DY ;

/* file2.ddl */

scope Custom {

int DY = 10 ;

}

scope Default {

include <file1.ddl>

}

int x = #Default#X ;
int y = #Default#Y ;
```

The brief introduction to **DDL** is finished now. The following sections provides detailed description.

2. DDL source processing.

DDL source is a single file or a family of several files, included directly or indirectly in the main one. The term *file* means a unit, containing a text. I.e. it is some of generic kind. It may be a file from the host file system, or a virtual file from a virtual file system of any kind or simply a text, given as is. In the last case the file inclusion is not possible and the correspondent directive will produce error. The text encoding is not essential from the **DDL** definition perspective, but the current implementation assumes it is octet-based text with **ASCII** character encoding in the first 128 code positions. It is the best to follow this agreement. **UTF-8** text encoding is recommended.

A **DDL** source is processed in three phases. The first phase is the *atomization phase*. The text is parsed by tokens first and then tokens are converted into atoms. The second phase is the *parsing phase*. **DDL** is an **LR1** language, the sequence of atoms is folded into **AST** (abstract syntax tree) using an **LR1** parsing machine. The file inclusion is performed if necessary during this phase. If some file inclusion directive is found, the referred file is opened, it's text is processed as a **DDL** source to **AST** and the result is included at the point of inclusion. Finally, the **AST** is evaluated during the *evaluation phase* to produce values for all defined in the source constants.

3. Atomization phase.

During the atomization phase the source file is cut on tokens. This process is going in a loop. The file content comprises the initial value for the character sequence being processed. On each loop iteration the current sequence is scanned from the beginning to select the next token. Once an appropriate prefix is selected, it is cut from the sequence and out as the next token, the rest of the sequence goes to the next loop iteration.

Each token class is defined by some regular expression. Tokenizer looks up the next few characters and determines the class of token, then extracts the token either as the shortest sequence, which matches the expression, or the longest such sequence.

The file is divided on text lines to designate a character position. The end of line is determined by the one of the following character combinations (the longest is selected) : “`\r`”, “`\n`” or “`\r\n`”. Most tokens resides on a single line. Exceptions are: <u>`LongComment`</u> and <u>`Space`</u>.

<u>**symbol classes**</u>

L	*a..z A..Z _*
Q	*?*
D	*0..9*
B	*0 1*
H	*0..9 a..f A..F*
C	*[] {} () ; , # = & + - * / % .*
S	*space \t \v \f \r \n*
P	*printable symbols*
$P_{>}$	P \ { > }
$P_{'}$	P \ { ' }
$P_{"}$	P \ { " , \ }

<u>**token classes**</u>

Comments are cut up as the shortest sequence.

<u>ShortComment</u>	*/ / … end-of-line or end-of-file*
<u>LongComment</u>	*/ * … * /*

All other tokens are cut up as the longest sequence.

<u>Space</u>	$\mathbf{S}^{1..}$
<u>PunctSym</u>	**C** \ { . }
<u>PunctArrow</u>	- >

PunctDots	.$^{1..}$
Word	L (L|D)*
QWord	? L (L|D)*
Dec	D$^{1..}$
Bin	B$^{1..}$ (*B*|*b*)
Hex	H$^{1..}$ (*H*|*h*)
Number	Dec | Bin | Hex
BString	‹ P$_{›}$* ›
SString	' P$_{'}$* '
DString	" (P$_{"}$ | \ P)* "

Two consecutive tokens Number and Word are diagnosed as a error. Tokens from the set Number ∩ Word are also diagnosed as a error.

first letter → token class

S	Space
C	PunctSym PunctArrow PunctDots
L	Word
Q	QWord
D	Number
/	/ ShortComment LongComment
'	SString
"	DString
‹	BString

tokens to atoms

Tokens are converted to atoms according the following table.

Atom	Token	Token value
Number	Number	

String	SString DString	
FileName	BString	
Name	Word	
QName	QWord	
int	Word	“int”
sint	Word	“sint”
uint	Word	“uint”
ulen	Word	“ulen”
sint8	Word	“sint8”
sint16	Word	“sint16”
sint32	Word	“sint32”
sint64	Word	“sint64”
uint8	Word	“uint8”
uint16	Word	“uint16”
uint32	Word	“uint32”
uint64	Word	“uint64”
text	Word	“text”
ip	Word	“ip”
struct	Word	“struct”
type	Word	“type”
null	Word	“null”
scope	Word	“scope”
include	Word	“include”
const	Word	“const”
→	PunctArrow	“->”
.	PunctDots	“.”
...	PunctDots	“..” “...” ets.
*	PunctSym	“*”
,	PunctSym	“,”

;	PunctSym	“;”
=	PunctSym	“=”
+	PunctSym	“+”
-	PunctSym	“–”
&	PunctSym	“&”
#	PunctSym	“#”
/	PunctSym	“/”
%	PunctSym	“%”
(	PunctSym	“(”
)	PunctSym	“)”
[	PunctSym	“[”
]	PunctSym	“]”
{	PunctSym	“{”
}	PunctSym	“}”

4. Parsing phase.

During the parsing phase the sequence of atoms is folded into the **AST**. **DDL** is an **LR1** language, so it can be done using an **LR1** parser.

DDL grammar

Here is a *conditional recursive grammar* of **DDL**:

! BODY	*empty* BODY SCOPE BODY INCLUDE BODY TYPE BODY CONST BODY STRUCT ;
SCOPE	**scope Name** { BODY }
INCLUDE	**include FileName**
TYPE	**type Name =** TYPEDEF ;
CONST	TYPEDEF **Name =** EXPR ;
RNAME	**Name** RNAME **#** **Name**
NAME	RNAME **#** RNAME . **#** RNAME ... **#** RNAME
RQNAME	**Name** **QName** RQNAME **#** **Name** RQNAME **#** **QName**
QNAME	RQNAME **#** RQNAME . **#** RQNAME ... **#** RQNAME
INT_TYPE	**int** **sint** **uint** **ulen** **sint8** **uint8** **sint16** **uint16**

	sint32 **uint32** **sint64** **uint64**		
BASE_TYPE	INT_TYPE **text** **ip**		
TYPEDEF	NAME BASE_TYPE TYPEDEF * { TYPEDEF_LIST } * TYPEDEF [ ] TYPEDEF [EXPR] STRUCT		
TYPEDEF_LIST	TYPEDEF TYPEDEF_LIST , TYPEDEF		
STRUCT	**struct Name** { SBODY }		
SBODY	*empty* SBODY TYPE SBODY **const** CONST SBODY STRUCT ; SBODY TYPEDEF **Name** ; SBODY TYPEDEF **Name** = EXPR ;		

EXPR :
list ,
add ,
mul ,
un ,
num ,
post ,
prim
	(EXPR)
	QNAME
	LITERAL
	ITYPE (EXPR)
	
	EXPR.a . **Name**
	EXPR.a → **Name**
	EXPR.a [EXPR]
	
	Number
	
	* EXPR.a
	& EXPR.a
	+ EXPR.a
	– EXPR.a
		
		if(a>=post)
		if(a>=post)
		if(a>=num)
		
		if(a>=un)
		if(a>=un)
		if(a>=un)
		if(a>=un)
			= prim
			= prim
			= prim
			= prim
			
			= post
			= post
			= post
			
			= num
			
			= post
			= post
			= post
			= post

	EXPR.a * EXPR.b	if(a>=mul & b>=un)	= mul
	EXPR.a / EXPR.b	if(a>=mul & b>=un)	= mul
	EXPR.a % EXPR.b	if(a>=mul & b>=un)	= mul
	EXPR.a + EXPR.b	if(a>=add & b>=mul)	= add
	EXPR.a - EXPR.b	if(a>=add & b>=mul)	= add
	{ }		= list
	{ EXPR_LIST }		= list
	{ NAMED_EXPR_LIST }		= list
	EXPR { }		= list
	EXPR { NAMED_EXPR_LIST }		= list
EXPR_LIST	EXPR EXPR_LIST , EXPR		
NAMED_EXPR	. **Name** = EXPR		
NAMED_EXPR_LIST	NAMED_EXPR NAMED_EXPR_LIST , NAMED_EXPR		
ITYPE	INT_TYPE QNAME		
LITERAL	**null** **String** **Number . Number . Number . Number**		

BODY is the final language element. It represents a collection of different language entity, defined by the **DDL** source. BODY may contain SCOPEs, file inclusion directives, and definitions of type aliases, constants and structures.

SCOPE is the named scope of the entities, defined inside. It looks like

```
scope Name { BODY }
```

An inclusion directive INCLUDE looks like

```
include FileName
```

The directive is replaced by the BODY of the included file.

A type alias definition TYPE looks like

```
type Name = TYPEDEF ;
```

TYPEDEF is a “type expression”, i.e. a description of a type.

A constant definition looks like

```
TYPEDEF Name = EXPR ;
```

TYPEDEF here defines the constant type and EXPR must be evaluated to define the constant value.

To refer some language entity NAME is used. It may have four forms:

```
Name1 # Name2 # … # Namen
```

is a relative name. The correspondent entity is looked up first in the current scope. If there is no one the parent scope is searched, if it is failed again the grand-parent and so on. Here **$\texttt{Name}_1$** is a nested scope name in the base scope, **$\texttt{Name}_2$** is the name of double nested scope, … , finally, **$\texttt{Name}_n$** is the entity name.

```
# Name1 # Name2 # … # Namen
```

is an absolute name. The base scope is the global scope.

```
. # Name1 # Name2 # … # Namen
```

is an absolute name. The base scope is the current scope.

```
... # Name1 # Name2 # … # Namen
```

is an absolute name. The base scope is the one of parent scopes, determined by the number of dots.

In some situations QNAME may be used instead of NAME. QNAME is similar to NAME, but some names may be **QName**s. **QName** is formed from names, prefixed with the question mark.

The difference between NAME and QNAME is the entity lookup for QNAMESs is performed not from the point of definition, but from the point of usage. An example:

```
struct S
 {
  int x = ?Default#X ;
  int y = ?Default#Y ;
 };

scope A
 {
  scope Default
```

```
   {
    int X = 1 ;
    int Y = 2 ;
   }

  S s = {} ; // <- {1,2}
 }

scope B
 {
  scope Default
   {
    int X = 3 ;
    int Y = 4 ;
   }

  S s = {} ; // <- {3,4}
 }
```

Another example:

```
struct S
 {
  int x = 100*?M ;
 };

int M = 1 ;

S s = {} ; // <- 100

scope A
 {
  int M = 2 ;

```

```
  S x = {} ; // <- 200
 }
```

INT_TYPE is the one of base integer types.

BASE_TYPE is the one of base types.

TYPEDEF describes a type. The simplest way to do it is to provide a base type name (BASE_TYPE) or a type alias name (NAME). A structure name is also a type name.

A pointer description is:

```
TYPEDEF *
```

A polymorphic pointer description is:

```
{ TYPEDEF , TYPEDEF , … , TYPEDEF } *
```

An array description is:

```
TYPEDEF [ ]
```

A fixed length array description is:

```
TYPEDEF [ EXPR ]
```

Finally, a structure description STRUCT is also a type description. Using this kind of type description you simultaneously defines a structure and use it as a type description.

A structure description STRUCT looks like:

```
struct Name { SBODY }
```

Each structure definition defines a unique structure. You may refer to it using its name. Qualified names of structures must be distinct.

SBODY defines structure members. The main structure members are its fields:

```
TYPEDEF Name ;
TYPEDEF Name = EXPR ;
```

Each field has the name, type and optionally a default initialization expression. This expression is used for the field initialization in situation when the structure field initializer is missed. The order of fields is significant!

Each structure definition defines the same time a scope with the same name. You may extend this scope by additional scope definitions:

```
struct S
 {
```

```
  int a;
  int b;
 };

scope S
 {
  int X = 12345 ;
 }
```

But you may define type aliases, constants and structures in this scope directly in the structure definition:

```
struct S
 {
  int a; // field declaration
  int b; // field declaration

  type T = int ;        // scope type alias declaration

  struct A
   {
    int a;
   };                   // scope structure declaration

  const int X = 12345 ; // scope constant declaration
 };
```

To define a constant inside a structure you have to use the keyword **const**.

The last and most complicated fundamental language entity is the expression (EXPR). Expressions are used to give a value to a constant. This process is called the expression evaluation. An expression is built recursively from basic blocks using operations. Basic expressions are literals and names. Literals are:

```
null
String
Number
Number . Number . Number . Number
```

Names are either NAME or QNAME, where the last is permitted only in expressions for default field values.

Simple operations are:

```
( EXPR )
ITYPE ( EXPR )

EXPR . Name
EXPR → Name
EXPR [ EXPR ]

& EXPR
* EXPR
+ EXPR
- EXPR

EXPR + EXPR
EXPR - EXPR
EXPR * EXPR
EXPR / EXPR
EXPR % EXPR
```

ITYPE is either a base integer type keyword or NAME. This name must refer to a type alias, which eventually defines some integer type.

Compound operations are used to initialize arrays and structures:

```
{ }
{ EXPR , … , EXPR }
{ . Name = EXPR , … , . Name = EXPR }
EXPR { . Name = EXPR , … , . Name = EXPR }
```

Expression evaluation is described in the following section.

5. Evaluation phase.

During evaluation phase constants get its values. For each particular constant the initialization expression is evaluated with the constant type as the target type to produce the constant value. A fixed length array description has an implicitly declared associated constant – the length of the array. This constant has the type `ulen`. The key principle of expression evaluation is: an expression is evaluated with the target type. During evaluation this type determines target types for subexpressions evaluation. Exact rules are given below. For the evaluation purpose some special implicit types are used. They cannot be declared in the source and appears only during expression evaluation process. Constants evaluation introduces constant dependencies. The main principle here is: to use the constant or its part it must be evaluated before. But the constant address may be used before the constant is evaluated. If some dependencies make a circle the evaluation phase is failed. For example, the following text cannot be evaluated:

```
int a = b + 10 ;

int b = a + 20 ;
```

Another example:

```
struct S
 {
  int a = 10 ;
  int b = 20 ;
 };

S s = { .a = c } ;

int c = s.b +100 ;
```

But this one is OK:

```
struct S
 {
  int a = 10 ;
  int * b = &c ;
  S * ptr = &s ;
 };

S s = {} ;
```

```
int c = s.a+10+(s.b-s.b)+(s.ptr-s.ptr) ;
```

Two implicit types are: `ptr` and `slen`. `ptr` is a universal pointer. It may be the typeless null (`nullptr`), a typed null (`(T *)nullptr`) or point to some constant or subconstant. `slen` is a `ulen` with a sign. Operations with `slen` are performed with the overflow control. This type is used as an array index type. `I` below designates an integer type. Couple `(T,EXPR)` means an expression evaluated with the `T` as the target type. Signs such as $+_T$ denote operations on the correspondent types. Remember, that

`(ptr,EXPR)` cannot be `nullptr`,

`(ptr,EXPR)` and `(slen,EXPR)` cannot be evaluated both,

`(ptr,EXPR)` and `(I,EXPR)` cannot be evaluated both.

Type	**Expression**	**Evaluation**
I	+ EXPR	(I,EXPR)
I	- EXPR	$-_I$ (I,EXPR)
I	$EXPR_1$ + $EXPR_2$	(I,$EXPR_1$) $+_I$ (I,$EXPR_2$)
text	$EXPR_1$ + $EXPR_2$	(text,$EXPR_1$) $+_{text}$ (text,$EXPR_2$)
T *	$EXPR_1$ + $EXPR_2$	check_type<T *>((ptr,$EXPR_1$) $+_{ptr}$ (slen,$EXPR_2$)) check_type<T *>((ptr,$EXPR_2$) $+_{ptr}$ (slen,$EXPR_1$))
{T,...} *	$EXPR_1$ + $EXPR_2$	check_type<T *,...>((ptr,$EXPR_1$) $+_{ptr}$ (slen,$EXPR_2$)) check_type<T *,...>((ptr,$EXPR_2$) $+_{ptr}$ (slen,$EXPR_1$))
I	$EXPR_1$ - $EXPR_2$	(I,$EXPR_1$) $-_I$ (I,$EXPR_2$) cast_to<I>((ptr,$EXPR_1$) $-_{ptr}$ (ptr,$EXPR_2$))
T *	$EXPR_1$ - $EXPR_2$	check_type<T *>((ptr,$EXPR_1$) $+_{ptr}$ $-_{slen}$ (slen,$EXPR_2$))
{T,...} *	$EXPR_1$ - $EXPR_2$	check_type<T *,...>((ptr,$EXPR_1$) $+_{ptr}$ $-_{slen}$ (slen,$EXPR_2$))
I	$EXPR_1$ * $EXPR_2$	(I,$EXPR_1$) $*_I$ (I,$EXPR_2$)
I	$EXPR_1$ / $EXPR_2$	(I,$EXPR_1$) $/_I$ (I,$EXPR_2$)
I	$EXPR_1$ % $EXPR_2$	(I,$EXPR_1$) $\%_I$ (I,$EXPR_2$)

I	ITYPE (EXPR)	cast_to<I>((ITYPE,EXPR)) where ITYPE is determined from ITYPE
text	ITYPE (EXPR)	cast_to<text>((ITYPE,EXPR)) where ITYPE is determined from ITYPE
I	**Number**	determined from **Number** using “reduction by module”
text	**Number**	determined from **Number** as the literal string
text	**String**	determined from **String**
ip	**Number** . **Number** . **Number** . **Number**	determined from the literal, each **Number** is converted to uint8 using “reduction by module”
text	**Number** . **Number** . **Number** . **Number**	cast_to<text>((ip,EXPR)) where EXPR is the original expression
T *	& EXPR	check_type<T *>(address(EXPR))
{T,...} *	& EXPR	check_type<T *,...>(address(EXPR))
T	* EXPR	cast_obj<T>((ptr,EXPR))
T	EXPR$_1$ [EXPR$_2$]	cast_obj<T>(address(EXPR$_1$ [EXPR$_2$]))
T	EXPR . **Name**	cast_obj<T>(address(EXPR . **Name**))
T	EXPR → **Name**	cast_obj<T>(address(EXPR → **Name**))
T	QNAME	cast_obj<T>(address(QNAME))
I	null	0_I
text	null	“”
ip	null	0.0.0.0
T *	null	(T *)nullptr

{T,...} *	**null**	nullptr
T [ ]	**null**	{}
T [Len]	**null**	{ (T,**null**),... } (Len times)

struct S
{
 T_1 field$_1$;
 …
 T_n field$_n$;
}
	null	{
		 (T_1,**null**),
		 … ,
		 (T_n,**null**)
		}

I	**{ }**	0_I
text	**{ }**	“”
ip	**{ }**	0.0.0.0
T *	**{ }**	(T *)nullptr
{T,...} *	**{ }**	nullptr
T [ ]	**{ }**	{}
T [Len]	**{ }**	{ (T,**{ }**),... } (Len times)

struct S
{
 T_1 field$_1$;
 …
 T_n field$_n$;
}
	{ }	{
		 (T_1,E_1),
		 … ,
		 (T_n,E_n)
		}
		
		where E_k is the default initializer for the field$_k$ if any, or **{ }** if there is no one.
		Remember, that the name lookup for QNAMEs is performed at the point of usage for these expressions.

T [ ]	**{** EXPR$_1$ **,** … **,** EXPR$_n$ **}** (**n**>0)	{ (T,EXPR$_1$),...,(T,EXPR$_n$) }
T [Len]	**{** EXPR$_1$ **,** … **,** EXPR$_n$ **}** (**n**>0)	{ (T,EXPR$_1$),...,(T,EXPR$_n$), (T,**{ }**),... } (Len-n times) n must be <= Len
struct S { T_1 field$_1$; …	**{** EXPR$_1$ **,** … **,** EXPR$_n$ **}** (**n**>0)	{ (T_1,EXPR$_1$),...,(T_n,EXPR$_n$), (T_{n+1},E_{n+1}), … , (T_m,E_m)

T_m field$_m$; }		} n must be <= m where E_k is the default initializer for the field$_k$ if any, or { } if there is no one. Remember, that the name lookup for QNAMEs is performed at the point of usage for these expressions.

struct S
{
T_1 field$_1$;
…
T_m field$_m$;
}
	{
	. **Name**$_1$ = EXPR$_1$
	, … ,
	. **Name**$_n$ = EXPR$_n$
	} (n>0)
		This expression is evaluated by the same way, as in the case above, but the matching fields with expressions are base on the field names and atom names.
		
		{ … , (T_k,EXPR$_l$) , … } if field$_k$ == Name$_l$
		Name$_l$ is determined from **Name**$_l$
		
		Extra expressions are not evaluated.

struct S
{
T_1 field$_1$;
…
T_m field$_m$;
}
	EXPR { }	(struct S,EXPR)

struct S
{
T_1 field$_1$;
…
T_m field$_m$;
}
	EXPR
	{
	. **Name**$_1$ = EXPR$_1$
	, … ,
	. **Name**$_n$ = EXPR$_n$
	} (n>0)
		This case is similar to the case without EXPR. But EXPR is used to fill remaining fields, instead of default initializers. Extra expressions or subexpressions are not evaluated.

The following table describes the special type handling.

Type	**Expression**	**Evaluation**
ptr	& EXPR	address(EXPR)
ptr	EXPR$_1$ + EXPR$_2$	(ptr,EXPR$_1$) $+_{ptr}$ (slen,EXPR$_2$)

		(ptr,EXPR$_2$) $+_{ptr}$ (slen,EXPR$_1$)
ptr	EXPR$_1$ **-** EXPR$_2$	(ptr,EXPR$_1$) $+_{ptr}$ $-_{slen}$ (slen,EXPR$_2$)
ptr	***** EXPR	cast_obj<ptr>((ptr,EXPR))
ptr	EXPR$_1$ **[** EXPR$_2$ **]**	cast_obj<ptr>(address(EXPR$_1$ **[** EXPR$_2$ **]**))
ptr	EXPR **.** **Name**	cast_obj<ptr>(address(EXPR **.** **Name**))
ptr	EXPR **→** **Name**	cast_obj<ptr>(address(EXPR **→** **Name**))
ptr	QNAME	cast_obj<ptr>(address(QNAME))
slen	**+** EXPR	(slen,EXPR)
slen	**-** EXPR	$-_{slen}$ (slen,EXPR)
slen	EXPR$_1$ **+** EXPR$_2$	(slen,EXPR$_1$) $+_{slen}$ (slen,EXPR$_2$)
slen	EXPR$_1$ **-** EXPR$_2$	(slen,EXPR$_1$) $-_{slen}$ (slen,EXPR$_2$) (ptr,EXPR$_1$) $-_{ptr}$ (ptr,EXPR$_2$)
slen	EXPR$_1$ ***** EXPR$_2$	(slen,EXPR$_1$) $*_{slen}$ (slen,EXPR$_2$)
slen	EXPR$_1$ **/** EXPR$_2$	(slen,EXPR$_1$) $/_{slen}$ (slen,EXPR$_2$)
slen	EXPR$_1$ **%** EXPR$_2$	(slen,EXPR$_1$) $\%_{slen}$ (slen,EXPR$_2$)
slen	ITYPE **(** EXPR **)**	cast_to<slen>((ITYPE,EXPR)) where ITYPE is determined from ITYPE
slen	**Number**	determined from **Number** with overflow check
slen	***** EXPR	cast_obj<slen>((ptr,EXPR))
slen	EXPR$_1$ **[** EXPR$_2$ **]**	cast_obj<slen>(address(EXPR$_1$ **[** EXPR$_2$ **]**))
slen	EXPR **.** **Name**	cast_obj<slen>(address(EXPR **.** **Name**))
slen	EXPR **→** **Name**	cast_obj<slen>(address(EXPR **→** **Name**))
slen	QNAME	cast_obj<slen>(address(QNAME))
slen	**null**	0_{slen}
slen	**{ }**	0_{slen}

Address evaluation is described in the following table.

Expression	**Evaluation**
`* EXPR`	`(ptr,EXPR)`
`EXPR₁ [ EXPR₂ ]`	`(ptr, EXPR₁ + EXPR₂ )`
`EXPR . Name`	`address( EXPR ) → FieldName` where `FieldName` is determined from **`Name`**
`EXPR → Name`	`(ptr,EXPR) → FieldName` where `FieldName` is determined from **`Name`**
`QNAME`	`→ Const` where `Const` is determined from `QNAME` and must be a constant

The following table is the list of basic operations.

Operation	**Description**
$0_I \to I$	
$-_I\ I \to I$	residual ring operation
$I +_I I \to I$	residual ring operation
$I -_I I \to I$	residual ring operation
$I *_I I \to I$	residual ring operation
$I /_I I \to I$	lifted integer operation
$I \%_I I \to I$	lifted integer operation
$0_{slen} \to slen$	
$-_{slen} \to slen$	always successful
$slen +_{slen} slen \to slen$	integer operation with overflow check
$slen -_{slen} slen \to slen$	integer operation with overflow check
$slen *_{slen} slen \to slen$	integer operation with overflow check

slen $/_{slen}$ slen → slen	integer operation
slen $\%_{slen}$ slen → slen	integer operation
text $+_{text}$ text → text	text concatenation
ptr $+_{ptr}$ slen → ptr	
ptr $-_{ptr}$ ptr → slen	
check_type<T *,...>(ptr) → ptr	check the pointer type
cast_to<I>(I_1) → I	reduction by module
cast_to<slen>(I) → slen	with overflow check
cast_to<text>(I) → text	decimal representation
cast_to<text>(ip) → text	standard dot representation
cast_obj<T>(ptr) → T	
cast_obj<ptr>(ptr) → ptr	
cast_obj<slen>(ptr) → slen	
→ Const → ptr	core pointer to the constant
ptr → FieldName → ptr	structure field selection
ptr → [0] → ptr	array pointer to the array first element conversion

Integer type values have two interpretation: as the elements of the residual ring or as integer numbers from the representation set. For the type `uintN` the ring module is 2^N and the representation set is $\{0,...,2^N-1\}$. For the type `sintN` the ring module is 2^N and the representation set is $\{-2^{N-1},...,2^{N-1}-1\}$. Integer operations, except division operations, are performed in the correspondent residual ring of the integer type. But division operations are performed on the lifted integers. Divisor must not be zero. The following conditions are satisfied: `(a/b)*b+(a%b) == a`, `abs(a%b) < abs(b)`, `sign(a%b) == sign(a)`. These conditions define division operations. Division operations produce the output from the representation set except the one case: $-M \ / \ -1 = M = -M \pmod{2^N}$, where $-M == -2^{N-1}$ is the minimal negative value of the signed integer type. Integer cast operations are performed by reduction by module, i.e. the lifted integer residue by the module of the target type is produced.

slen is different than other integer types. It does not have the residual ring. It has the representation set $\{-2^N, \ldots, +2^N\}$. Here N is the number of bits of the type ulen. All slen operations are performed as an integer number operation and if the result is not representable then a error happens. Division operations do not cause overflow, but may fail if the divisor is null. Cast operations of integer values to slen preserve the integer value, but may cause a error if the value is not representable by slen.

In general, cast_obj operation reads the value of the object, specified by the given pointer and casts it to the target type. If the source type and the target type are the same, the value remains unchanged. Otherwise the one of the following cast operations is performed.

Source type	**Target type**	**Operation**
I_1	I	cast_to<I>(*ptr)
I	text	cast_to<text>(*ptr)
ip	text	cast_to<text>(*ptr)
I	slen	cast_to<slen>(*ptr)
T *	ptr	*ptr if *ptr is nullptr a error happens
{T,...} *	ptr	*ptr if *ptr is nullptr a error happens
struct S	struct T	recursive field-to-field cast_obj, based on field names, remaining fields are defaulted
T [ ] T [Len]	T *	ptr → [0]
T [ ] T [Len]	{...,T,...} *	ptr → [0]
T [ ] T [Len]	ptr	ptr → [0]

There is a special case, where so-called *decay* takes place. If the source type is an array type and the target type is the pointer type, the address of the array is converted to the address of the first array element.

Each non-null pointer is ether a core pointer, or derived from a core pointer by selection operations. There are two selection operations: one to select a structure field and another to select an array element.

Pointers	Description
→ Const	pointer to the constant
ptr → FieldName	pointer to the structure field
ptr → [index]	pointer to the array element

index has the type ulen.

During pointer operations pointer may point outside an array bound, but resulting constant value must point to an existing element. An example:

```
int[10] A = { } ;


int * ptr = A+100-99 ; // A+100 point outside array bounds
```

Overflows, however, produce errors.

6. File name processing.

If **DDL** processing is performed on the regular file system files, there is a recommended way to handle file names during file name inclusion operations. Each file has an associated file name. When a file inclusion operation is found in the file text, it is necessary to build the file name to be included. In this process two names are used: the name of the source file (`src_name`) and the name, given in the inclusion directive (`inc_name`).

The following table describes different file name classes.

File name class	**Definition**
general	$\texttt{(dev:)}^{opt}\texttt{(/)}^{opt}\texttt{(extname/)}^{*}\texttt{name}$
normalized	$\texttt{(dev:)}^{opt}\texttt{(/)}^{opt}\texttt{(name/)}^{*}\texttt{name}$ $\texttt{(dev:)}^{opt}\texttt{(../)}^{1..}\texttt{(name/)}^{*}\texttt{name}$
absolute	$\texttt{dev:(/)}^{opt}\texttt{(extname/)}^{*}\texttt{name}$ $\texttt{(dev:)}^{opt}\texttt{/(extname/)}^{*}\texttt{name}$
relative	$\texttt{(extname/)}^{*}\texttt{name}$

`dev` is a name of the device. There is no minimum limitation on this string.

`extname` is the file name. It is a non-empty string of allowed characters. The minimum requirement on the allowed character set is: it must not contain `:` `/` and `\`.

`name` is the regular file name. It is `extname` with excluded two special names: "`.`" and "`..`".

`/` means both `/` and `\`.

A general name can be normalized. It is a good practice to keep file names normalized to simplify a file name processing.

If `inc_name` is absolute, it is the resulting file name. If it is relative, it is combined with the `src_name` path component to produce the resulting file name.