DDL-1

DDL (v.1) – Data Definition Language, язык определения данных. Предназначен для описания связного набора данных произвольной природы. Имеет C-образный синтаксис.

Конспект языка.

Далее представлен «конспект языка».

основные понятия

тип	`T`	
константа	`C`	
литерал	`L`	
выражение	`E`	выражения вычисляются в контексте результирующего типа

базовые типы

целые

```
sint8     uint8
sint16    uint16
sint32    uint32
sint64    uint64
sint      uint
          ulen
```

`( sint, uint ) = ( sint32, uint32 ) или ( sint64, uint64 )`

`int = sint`

`ulen = uint32 или uint64 ≥ uint`

специальные

`text`	строки
`ip`	IP адреса

производные типы

`T *`	указатель
`T[E]`	массив (`ulen E`)
`T[]`	массив с выводимой размерностью

```
struct T
 {
    T₁    name₁[=E₁];
    T₂    name₂[=E₂];
    ...
    Tₙ    nameₙ[=Eₙ];
 };
```

структура

переименование типов

```
type T = T' ;
```

константы

```
T C = E ;
```

выражения

составные

$\{ E_1 , E_2 , \ldots , E_n \}$

$[E] \{ .name_1 = E_1 , .name_2 = E_2 , \ldots , .name_n = E_n \}$

простые

L

C

операционные

(E)

T(E) T целочисленный тип, операции производятся в кольце T

$E_1\ op_2\ E_2$ op_2 = + − ∗ / %

$op_1\ E$ op_1 = + − ∗ &

E.name

E->name

E[E'] slen E' – специальный внутренний тип, ±ulen с контролем переполнения

литералы

`1234567890`	десятичный
`1234567890ABCDEFabcdefH`	шестнадцатеричный (`H|h`)
`1000001B`	двоичный (`B|b`)
`192.168.1.10`	IP адрес
`null`	универсальный нуль
`"abcdef\b\t\n\v\f\r"`	строки с \-символами
`'abcdef'`	простые строки

области

```
scope name { <content> }
```

$name_1$#$name_2$#...#$name_n$

#$name_1$#$name_2$#...#$name_n$

.#$name_1$#$name_2$#...#$name_n$

..#$name_1$#$name_2$#...#$name_n$

...#$name_1$#$name_2$#...#$name_n$

включение файлов

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

содержимое файла – scope-level, допускаются неопределённые имена

Токены.

Исходный файл последовательно разрезается на токены. На каждом шаге из остатка выделяется префикс, являющийся токеном. Префиксы выделяются следуя либо правилу наименьшего префикса, либо правилу наибольшего префикса. Для обозначения позиции файл делится на строки. Конец строки определяется по одной из последовательностей (выбирается наиболее длинная) : “\r” “\n” “\r\n” .

классы символов

L	*a..z A..Z _*
D	*0..9*
B	*0 1*
H	*0..9 a..f A..F*
C	*[] {} () ; , # = & + – * / % .*
S	*пробел \t \v \f \r \n*
P	*печатные символы*
P›	**P** \ { › }
P'	**P** \ { ' }
P"	**P** \ { " , \ }

классы токенов

Комментарии выбираются по правилу наименьшего префикса.

ShortComment	*/ / … конец-строки или конец-файла*
LongComment	*/ * … * /*

Все остальные токены выбираются по правилу наибольшего префикса.

Space	**S**$^{1..}$
PunctSym	**C** \ { . }
PunctArrow	– ›
PunctDots	.$^{1..}$

Word	**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**)* "

Два подряд токена Number и Word диагностируются как ошибка. Токены из Number ∩ Word тоже диагностируются как ошибка.

первая буква → класс токена

S	Space
C	Punct PunctArrow PunctDots
L	Word
D	Number
/	/ ShortComment LongComment
'	SString
"	DString
<	BString

Атомы.

Атомы	Токены	Значения токенов
Number	Dec Bin Hex	
String	SString DString	
FileName	BString	
Name	Word	
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	“}”

Формальное определение языка.

BODY	пусто 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
INAME	**int** **sint** **uint** **ulen** **sint8** **uint8** **sint16** **uint16** **sint32** **uint32** **sint64** **uint64**
TNAME	INAME **text** **ip**

TYPEDEF	NAME TNAME TYPEDEF ***** TYPEDEF **[ ]** TYPEDEF **[** EXPR **]** STRUCT
STRUCT	**struct Name {** SBODY **}**
SBODY	пусто SBODY TYPE SBODY **const** CONST SBODY STRUCT **;** SBODY TYPEDEF **Name ;** SBODY TYPEDEF **Name =** EXPR **;**
EXPR	**{ }** **{** ELIST **}** **{** NELIST **}** EXPR **{ }** EXPR **{** NELIST **}** EXPR_ADD
EXPR_ADD	EXPR_MUL EXPR_ADD **+** EXPR_MUL EXPR_ADD **–** EXPR_MUL
EXPR_MUL	EXPR_UN EXPR_MUL ***** EXPR_UN EXPR_MUL **/** EXPR_UN EXPR_MUL **%** EXPR_UN
EXPR_UN	EXPR_POST ***** EXPR_UN **&** EXPR_UN **+** EXPR_UN **–** EXPR_UN
EXPR_POST	EXPR_NNPOST **Number**
EXPR_NNPOST	EXPR_NNPRIM EXPR_POST **[** EXPR **]** EXPR_NNPOST **.** **Name** EXPR_NNPOST → **Name**
EXPR_NNPRIM	**(** EXPR **)**

	ITYPE **(** EXPR **)** NAME NNLIT
ELIST	EXPR ELIST **,** EXPR
NEXPR	**. Name =** EXPR
NELIST	NEXPR NELIST **,** NEXPR
ITYPE	NAME INAME
NNLIT	**null** **String** **Number . Number . Number . Number**

Модель данных.

Body

list<Alias>

list<Const>

list<Struct>

list<Len>

list<Scope> // level 1

Alias

Name

●→ Scope

depth

●→ Type

Const

Name

●→ Scope

depth

●→ Type

●→ Expr

Field

Name

●→ Type

●→ Expr

Struct

Name

●→ Scope

depth

Scope

olist‹Field›

Len

●→ Expr

Scope

Name

●→ Scope

●→ Body

NameRef

- rel
- abs
- this
- dots

olist‹Name›

Len

●→ Expr

Type

●→ Struct

- Type_suint‹SUInt›
- Type_text
- Type_ip
- Type_ptr

●→ Type

Type_array

●→ Type

Type_array_len

●→ Type

Len

Type_struct

Type_ref (name link)

●→ NameRef

●→ Alias[opt]

●→ Struct[opt]

DomainType (name link[opt])

●→ Type[opt]

●→ NameRef[opt]

●→ Alias[opt]

Expr

PosName

Expr_add (_sub _mul _div _rem _ind)

●→ Expr

●→ Expr

Expr_deref (_address _plus _minus)

●→ Expr

Expr_number

Expr_ptr_select

●→ Expr

Name

Expr_select

●→ Expr

Name

Expr_domain

●→ Expr

●→ DomainType

Expr_var (name link)

●→ NameRef

●→ Const

Expr_null

Expr_string

Expr_ip

Number

Number

Number

Number

Expr_noname_list

olist‹●→ Expr›

Expr_named_list

list‹●→ Name,●→ Expr›

Expr_apply_named_list

●→ Expr

list‹●→ Name,●→ Expr›

Имена файлов.

extname – непустое имя файла, не содержит '/' '\' ':' , может быть специальным (“.” “..”).

name – регулярное имя файла.

Общее имя файла:

(dev:)^{opt}(/)^{opt}(extname/)*name .

Нормализованное имя файла:

(dev:)^{opt}(/)^{opt}(name/)*name

(dev:)^{opt}(../)$^{1..}$(name/)*name .

Имена, обрабатываемые как самостоятельные пути:

(dev:)^{opt}/(name/)*name

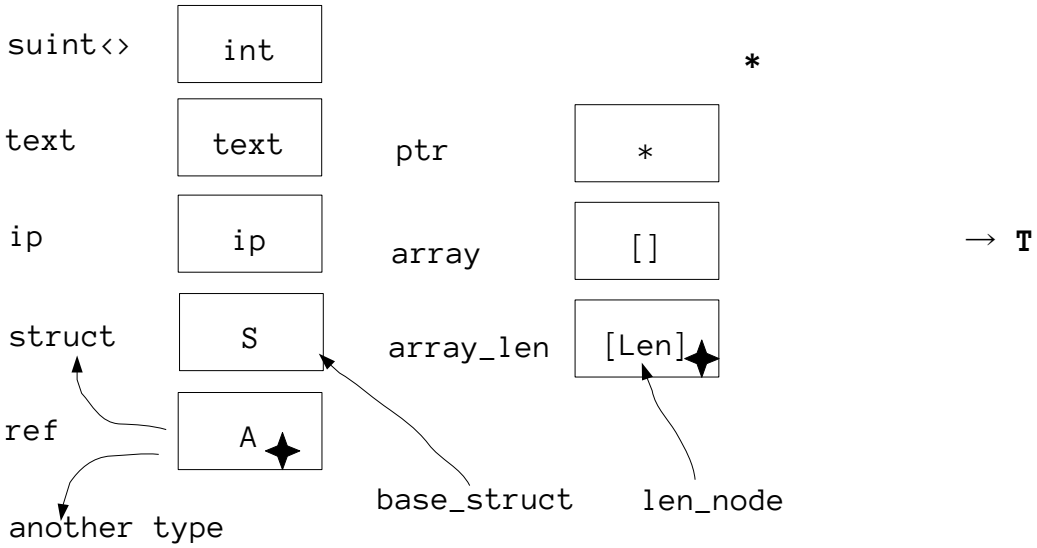
dev:(name/)*name .

Имена, обрабатываемы с учетом пути исходного файла:

(name/)*name .

Модель вычислений.

типы



недопустимые циклы в типах

T[] → T

T[Len] → T

T * → T

struct { T_1, ..., T_n } → T_1, ..., T_n

T[] → T

T[Len] → T

классы типов и специальные типы

int {**slen**, sint8, ..., uint64}

text

ip

struct {}

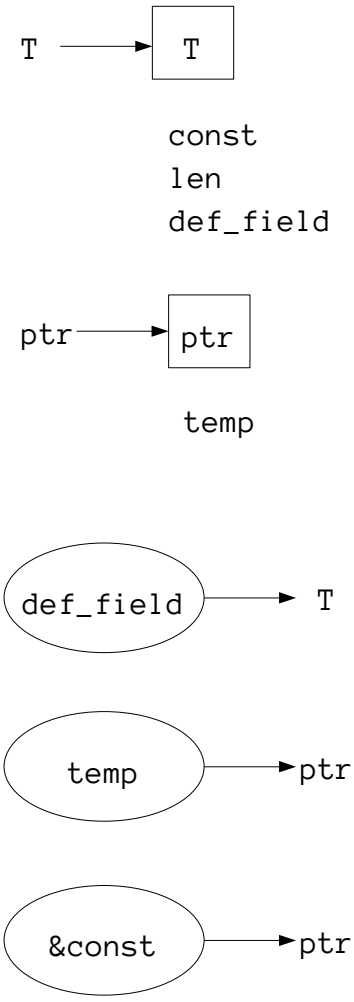
array {}

ptr {}

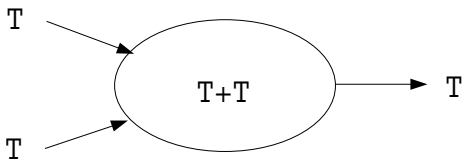
ptr*

LVptr

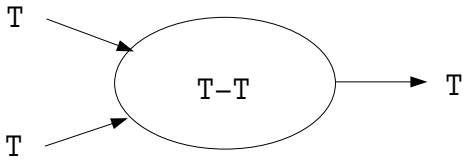
граф вычислений



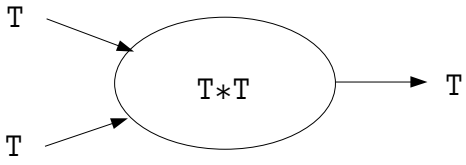
T : int



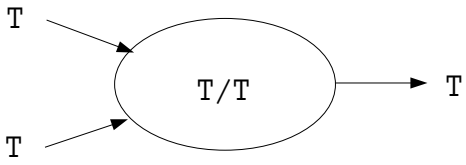
a+b



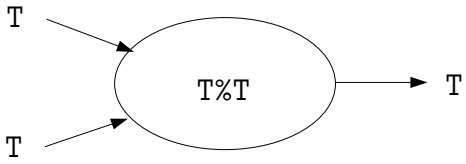
a−b



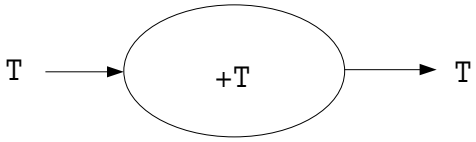
a∗b



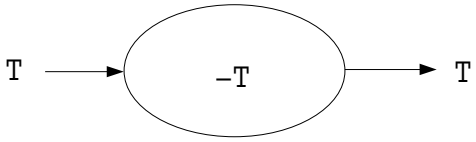
a/b



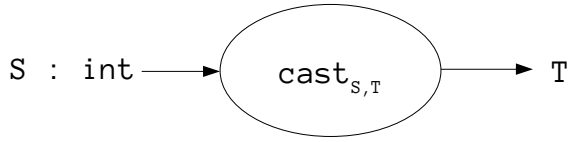
a%b

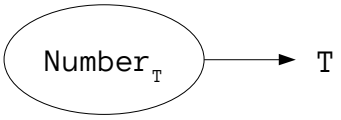


+a

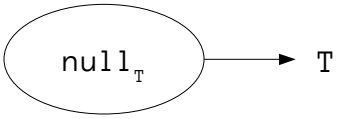


−a



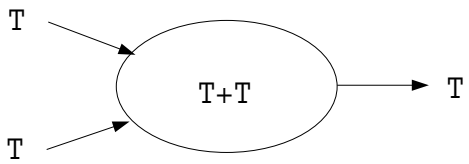


Number

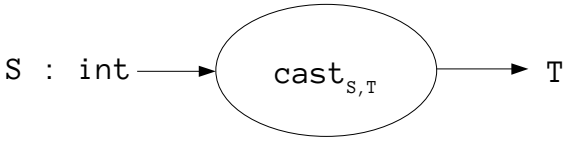


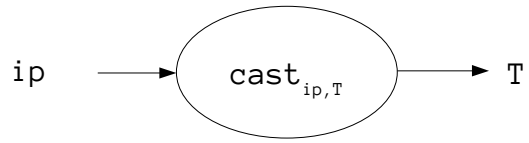
null, {}

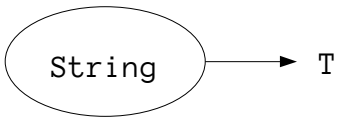
T = text



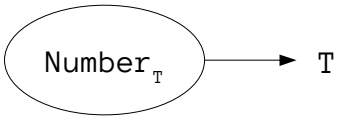
a+b



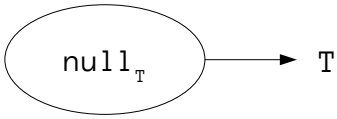




String

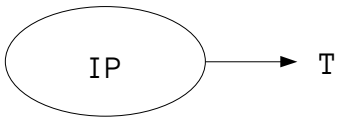


Number

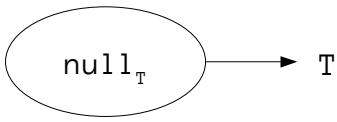


null, {}

T = ip

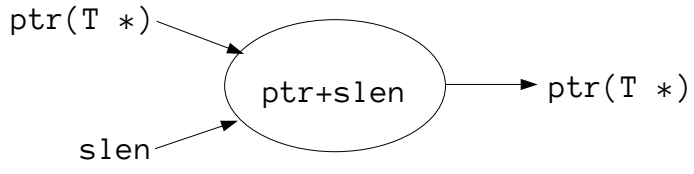


IP

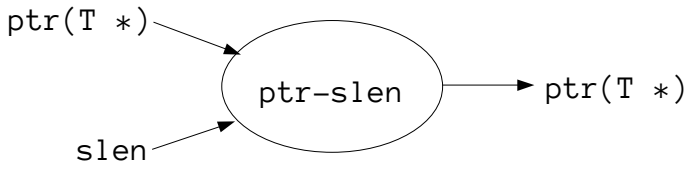


null, {}

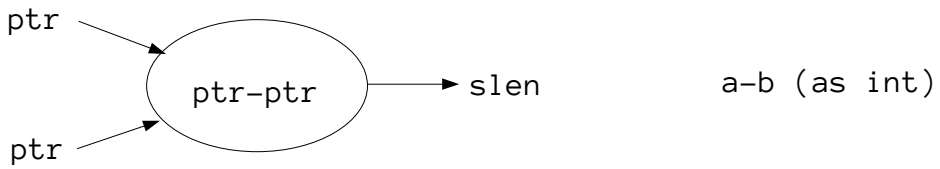
ptr

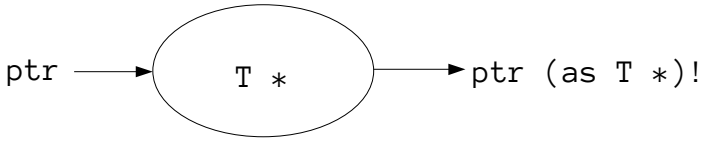


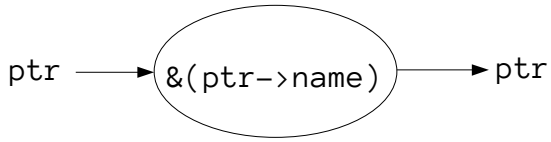
a+b (as ptr,T *)

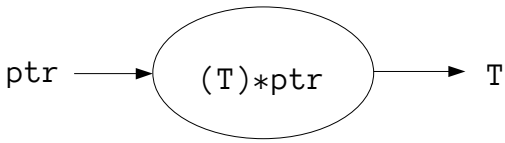


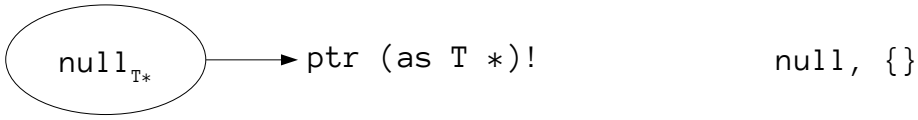
a-b (as ptr,T *)


ptr
ptr-ptr
slen
a-b (as int)
ptr


ptr
T *
ptr (as T *)!

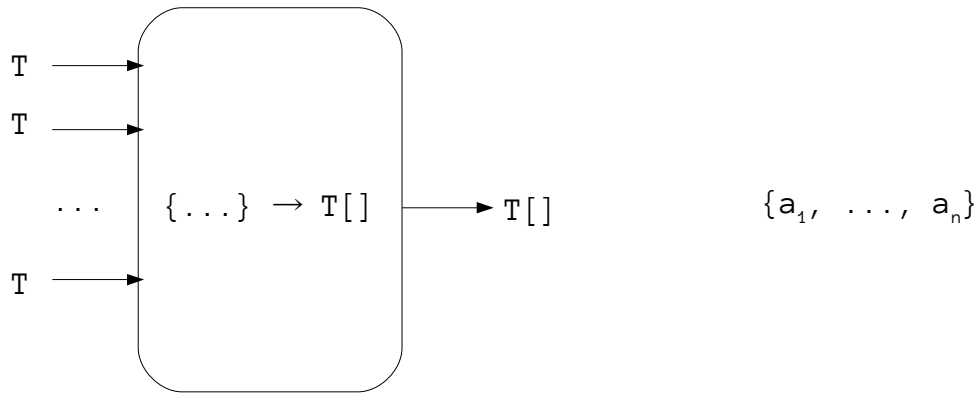

ptr
&(ptr->name)
ptr


ptr
(T)*ptr
T

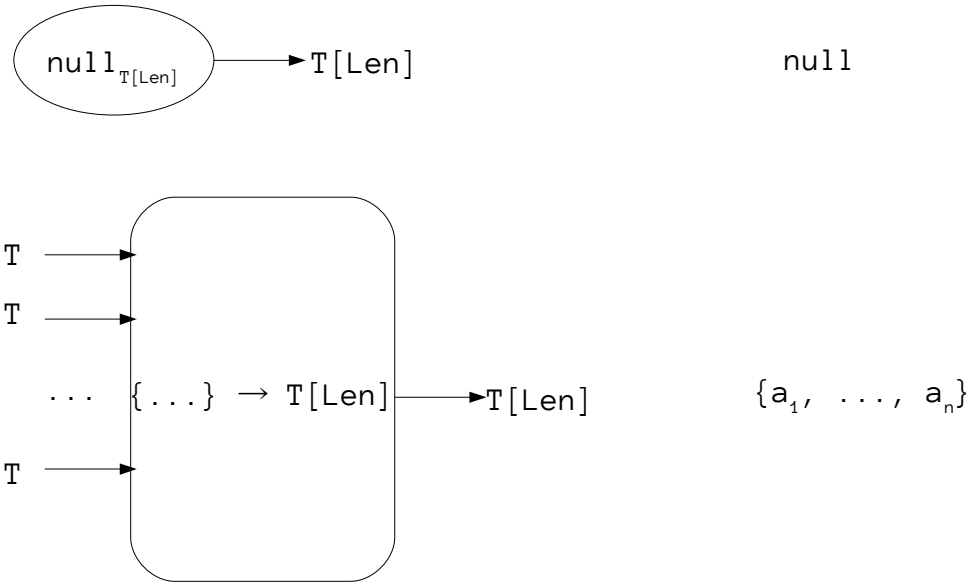

null T*
ptr (as T *)!
null, {}

T[]

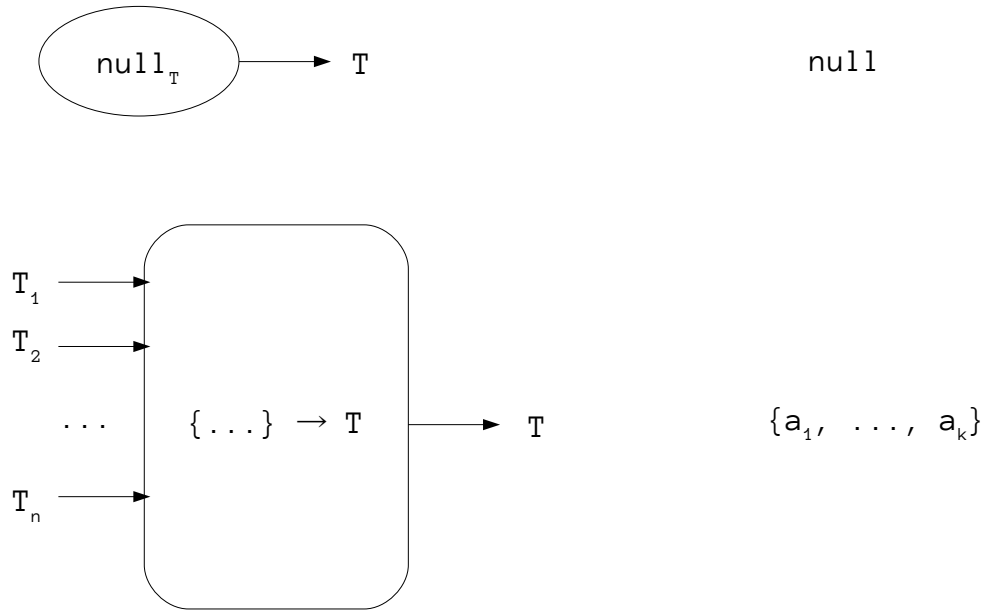

null T[]
T[]
null


T
T
...
T
{...} → T[]
T[]
{a1, ..., an}

T[Len]



T = struct { T_1, ..., T_n }



простые выражения

```
null
Number
String
IP
const           LV
```

составные выражения

```
a + b
a – b
a ∗ b
a / b
a % b
a [ b ]         // ∗(a+b)

+ a
– a
∗ a             LV
& a

domain( a )

a . name        // &a->name
a → name        LV

{ a₁, ..., aₙ }
{ .name₁=a₁, ..., .nameₙ=aₙ }
b { .name₁=a₁, ..., .nameₙ=aₙ }
```

вычисление выражений

(T, IP)

T = ip ip ← IP

T = text text ← ip ← IP

(T, String)

T = text text ← String

(T, Number)

T : int T ← Number

T = text text ← Number // copy source string

(T, a + b)

T : int (T,a) $+_{T}$ (T,b)

T = text (T,a) $+_{text}$ (T,b)

T : ptr (T,a) $+_{ptr*}$ (slen,b) или (T,b) $+_{ptr*}$ (slen,a)

T = ptr^{*} (ptr^{*},a) $+_{ptr*}$ (slen,b) или (ptr^{*},b) $+_{ptr*}$ (slen,a)

(T, a – b)

T : int (T,a) $-_{T}$ (T,b) или T ← (ptr^{*},a) $-_{ptr*}$ (ptr^{*},b)

T : ptr (T,a) $-_{ptr*}$ (slen,b)

T = ptr^{*} (ptr^{*},a) $-_{ptr*}$ (slen,b)

(T, a * b)

T : int (T,a) $*_T$ (T,b)

(T, a / b)

T : int (T,a) $/_T$ (T,b)

(T, a % b)

T : int (T,a) $\%_T$ (T,b)

(T, + a)

T : int $+_T$ (T,a)

(T, - a)

T : int $-_T$ (T,a)

(T, domain(a))

T : int T ← (domain,a)
T = text T ← (domain,a) // default decimal integer format

(T, null)

(T, {})

T : int	0_T
T = text	“”
T = ip	$0.0.0.0_{ip}$
T : ptr	$nothing_T$

(T, null)

T : struct

T	
{	{
T_1 f_1;	(T_1,null),
...	...
T_n f_n;	(T_n,null)
}	}

T : array

T = T´[L]	{ ((T´,null),)^L }
T = T´[]	{}

(T, { a_1, ..., a_n })

T : struct

T

{	{
T_1 f_1;	(T_1,a_1),
...	...
T_n f_n;	(T_n,a_n),
...	...
T_p f_p;	(T_p,{}),
...	...
T_q f_q = b_q ;	(T_q,b_q),
...	...
}	}

T : array

T = T´[L]	{ (T´,a_1), ..., (T´,a_n), ((T´,{}),)$^{L-n}$ }
T = T´[]	{ (T´,a_1), ..., (T´,a_n) }

(T, { .$name_1$=a_1, ..., .$name_n$=a_n })

T : struct

T

{	{
...	...
T_n f_n;	(T_n,a_k), // f_n == $name_k$
...	...
T_p f_p;	(T_p,{}),
...	...
T_q f_q = b_q ;	(T_q,b_q),
...	...
}	}

(T, b { .name1=a1, . . . , .namen=an })

T : struct

T	
{	{
. . .	. . .
T_n f_n;	(T_n,a_k), // f_n == $name_k$
. . .	. . .
T_p f_p;	(T,b).f_p ,
. . .	. . .
}	}

(T, &a)

T : ptr	T ← (LVPtr,a)
T = ptr*	(LVPtr,a)

(T, v)

T = LVPtr	&v
T != LVPtr	T ← * &v

(T, *a)

T = LVPtr	(ptr*,a)
T != LVPtr	T ← * (ptr*,a)

(T, a → name)

T = LVPtr	&(ptr*,a)→name
T != LVPtr	T ← * &(ptr*,a)→name

(T, a.name)

T = LVPtr &(LVPtr,a)→name

T != LVPtr T ← * &(LVPtr,a)→name

(T, a[b])

T = LVPtr (ptr*,a+b)

T != LVPtr T ← * (ptr*,a+b)

преобразования типов

int ← int´

text ← int ip

struct ← struct´

ptr* ← array