Прокачиваем lldb

lldb formatters

C++ Senior Developer
PVS-Studio

Лысый Олег

std::string_view

```
leaf: {_M_len:9, _M_str:0x00000000024f537b}
    _M_len: 9
  > _M_str: "increment * 10000000;\n\n  return
```

std::string_view

```
∨ leaf: {_M_len:9, _M_str:0x00000000024f537b}
    _M_len: 9
  > _M_str: "increment * 10000000;\n\n  return
```

```
  > nonleaf: {child:0x00000000
  > leaf: increment
  m_what: 258
```

std::variant<….>

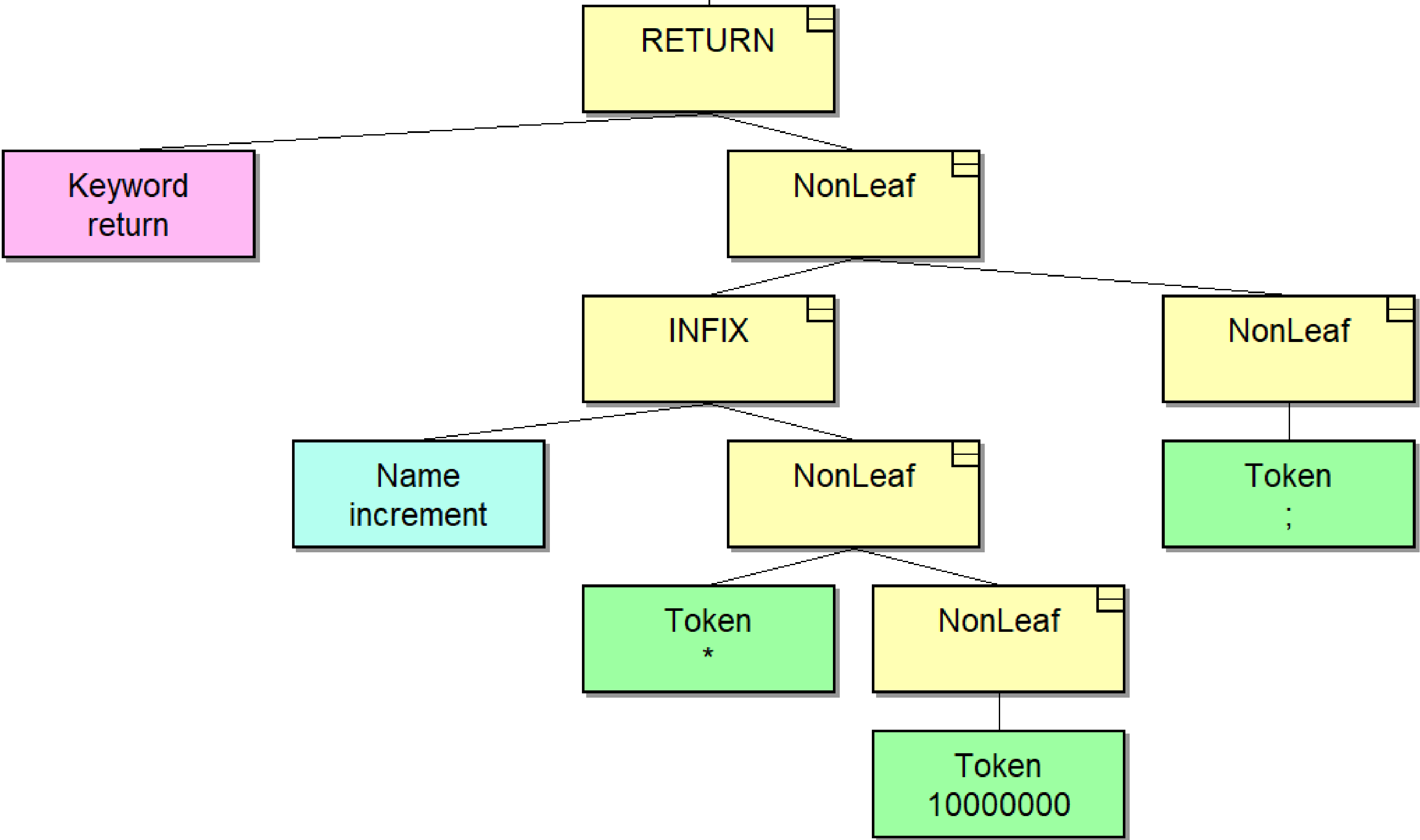
std::variant<....>

```
∨ [raw]: std::variant<DataFlow::IntegerInterval, ArrayView<const DataFlow::IntegerInterval> >
 ∨ std::__detail::__variant::_Variant_base<DataFlow::IntegerInterval, ArrayView<const DataFlow::Integ
  ∨ std::__detail::__variant::_Move_assign_alias<DataFlow::IntegerInterval, ArrayView<const DataFlow:
   ∨ std::__detail::__variant::_Copy_assign_alias<DataFlow::IntegerInterval, ArrayView<const DataFlow
    ∨ std::__detail::__variant::_Move_ctor_alias<DataFlow::IntegerInterval, ArrayView<const DataFlow:
     ∨ std::__detail::__variant::_Copy_ctor_alias<DataFlow::IntegerInterval, ArrayView<const DataFlow
      ∨ std::__detail::__variant::_Variant_storage_alias<DataFlow::IntegerInterval, ArrayView<const D
       ∨ _M_u: {...}
        ∨ _M_first: {...}
         ∨ _M_storage: Single interval [686047232:686047232]
          ∨ min: {...}
           > m_data: {s1:686047232}
          ∨ max: {...}
           > m_data: {s1:686047232}
        > _M_rest: {...}
         M_index: '\0'
```

std::variant<….>

```
⌄ m_intervals: {0}
    [Index]: 0
  ⌄ [Value]: Single interval [686047232:686047232]
    > [Min]: {686047232}
    > [Max]: {686047232}
    > [raw]: DataFlow::IntegerInterval
```

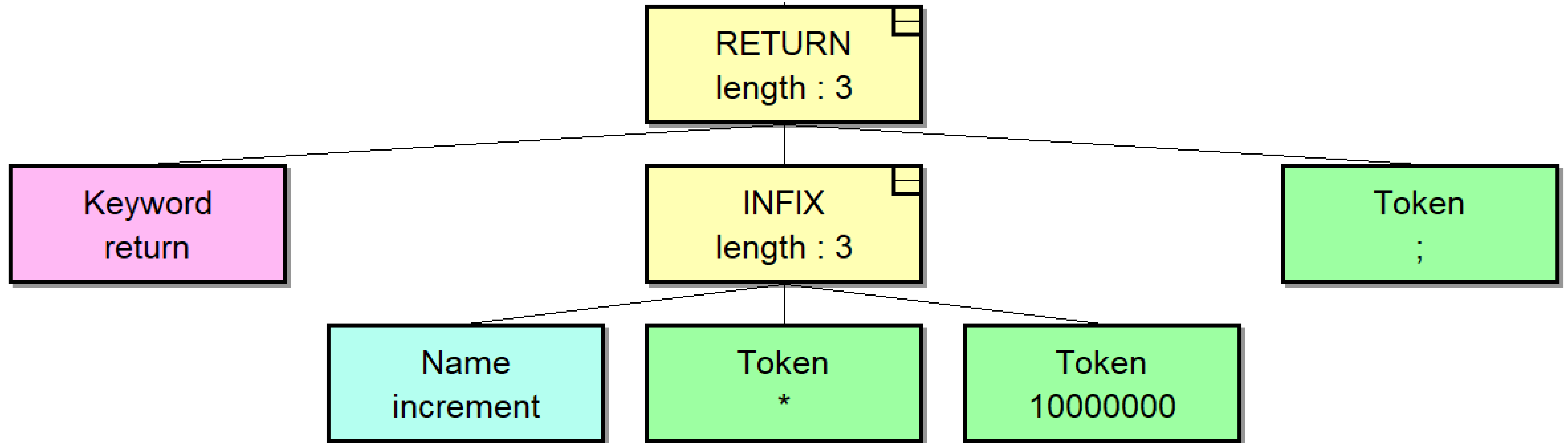
Syntax Tree



Syntax Tree

```
∨ expr: {m_parent:0x0000000002d0c430, m_what:553, ...}
  > m_parent: {m_parent:0x0000000002d0c408, m_what:280, ...}
  ∨ data: {...}
    ∨ nonleaf: {child:0x0000000002663bf0, next:0x0000000002d0c390}
      ∨ child: {m_parent:0x0000000002c7e810, m_what:258, ...}
        > m_parent: {m_parent:0x0000000002d0c430, m_what:553, ...}
        ∨ data: {...}
          > nonleaf: {child:0x0000000000000009, next:0x00000000024f537b}
          ∨ leaf: {_M_len:9, _M_str:0x00000000024f537b}
              _M_len: 9
            > _M_str: "increment * 10000000;\n\n  return 0;\n}\n\nint TestSignedOve…
          m_what: 258
          m_isLeaf: 1
          m_isArtificial: 0
          m_parentInited: 1
      ∨ next: {m_parent:0x0000000002c7e810, m_what:280, ...}
        > m_parent: {m_parent:0x0000000002d0c430, m_what:553, ...}
        ∨ data: {...}
          ∨ nonleaf: {child:0x0000000002d0c368, next:0x0000000002d0c3b8}
            ∨ child: {m_parent:0x0000000002d0c390, m_what:280, ...}
```

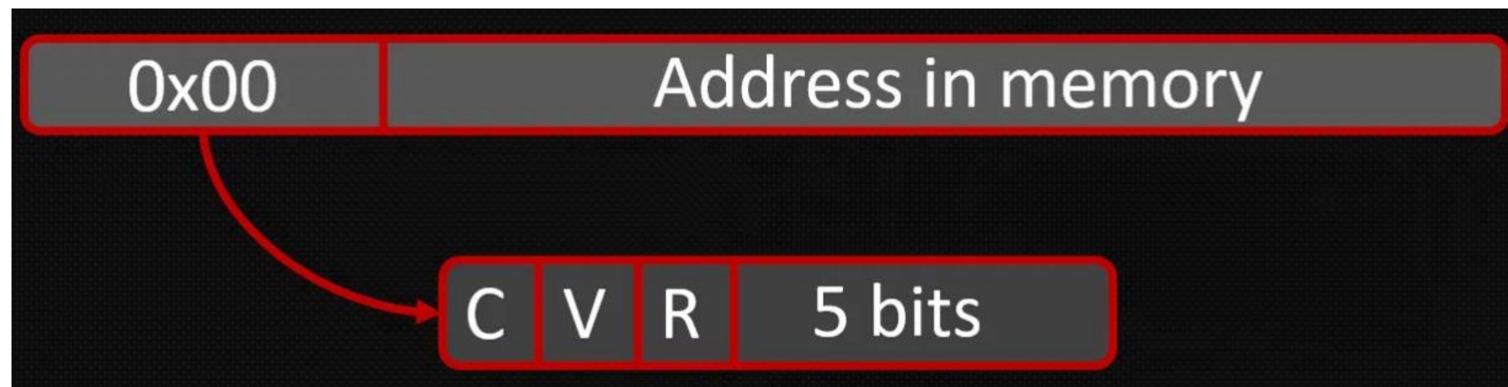
Syntax Tree



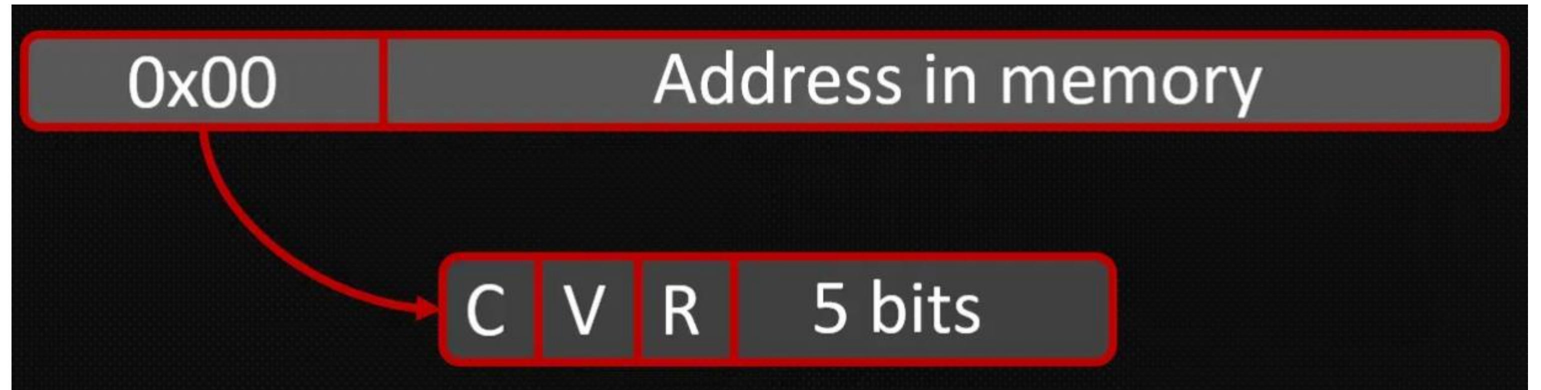
Syntax Tree

```
expr: ntInfixExpr
  [Type]: ntInfixExpr
> [ToString]: "increment * 10000000"
> [WiseType]: {m_simpleType:ST_INT, m_firstTemplateArgType:ST_UNKNOWN, ...}
  [IsLeaf]: false
> [Parent]: BadToken
> [QualType]: Alias
v [First]: tkIdentifier
    [Type]: tkIdentifier
  > [ToString]: "increment"
  > [WiseType]: {m_simpleType:ST_INT, m_firstTemplateArgType:ST_UNKNOWN, ...}
    [IsLeaf]: true
  > [Parent]: ntInfixExpr
  > [QualType]: Alias
  > [First]: None
  > [Second]: None
  > [Third]: "nullptr"
  > [Nth(3)]: "nullptr"
  > [Nth(4)]: "nullptr"
  > [raw]: VivaCore::Ptree *const
> [Second]: BadToken
> [Third]: tkConstant
```

QualType



QualType




```
[QualType]: {m_type:40285104}
    m_type: 40285104
```

QualType

```
[QualType]: Alias
  > [ToString]: "int32_t"
  v [Type]: {...}
    > m_canonical: Builtin
      m_hash: 17808521697069060345
      m_id: Alias
    [IsConst]: false
    [IsVolatile]: false
    [IsRestrict]: false
    [TypeId]: Alias
  v [raw]: VivaCore::PtreeTyped::QualType
      m_type: 40285104
```

Посмотрим на примере

Принтер для std::string_view

Принтер для std::string_view

```
∨ leaf: {_M_len:9, _M_str:0x000000000024f537b}
    _M_len: 9
  > _M_str: "increment * 10000000;\n\n  return
```

Принтер для std::string_view

```
⌄ leaf: {_M_len:9, _M_str:0x00000000024f537b}
    _M_len: 9
  › _M_str: "increment * 10000000;\n\n  return
```

```python
def StdStringViewPrinter(value: lldb.SBValue, internal_dict):
```

Принтер для std::string_view

```
∨ leaf: {_M_len:9, _M_str:0x00000000024f537b}
    _M_len: 9
  > _M_str: "increment * 10000000;\n\n   return
```

```
def StdStringViewPrinter(value: lldb.SBValue, internal_dict):
  len        : int     = int(value.children[0].GetValueAsSigned())
```

Принтер для std::string_view

```
⌄ leaf: {_M_len:9, _M_str:0x00000000024f537b}
    _M_len: 9
  › _M_str: "increment * 10000000;\n\n  return
```

```python
def StdStringViewPrinter(value: lldb.SBValue, internal_dict):
  len       : int    = int(value.children[0].GetValueAsSigned())
  strValue : string = str(value.children[1].GetSummary())
```

Принтер для std::string_view

```
leaf: {_M_len:9, _M_str:0x000000000024f537b}
    _M_len: 9
  > _M_str: "increment * 10000000;\n\n   return
```

```
def StdStringViewPrinter(value: lldb.SBValue, internal_dict):
  len       : int    = int(value.children[0].GetValueAsSigned())
  strValue : string = str(value.children[1].GetSummary())

  pos : int = int(strValue.find('"'))
  return strValue[0 : pos] + strValue[pos : pos + len + 1] + '"'
```

Принтер для std::string_view

> nonleaf: {child:0x00000000

> leaf: increment

m_what: 258

```
def S                                          rnal_dict):
  len                                          ueAsSigned())
  strValue : string = str(value.children[1].GetSummary())

  pos : int = int(strValue.find('"'))
  return strValue[0 : pos] + strValue[pos : pos + len + 1] + '"'
```

Принтер для std::string_view – как подключить

```
lldb % "type summary add -F ModuleName.FunctionName TypeName"
```

Принтер для std::string_view – как подключить

```
lldb % "type summary add -F ModuleName.FunctionName TypeName"
```

```
def __lldb_init_module(debugger, internal_dict):
  ....
```

Принтер для std::string_view – как подключить

```
lldb % "type summary add -F ModuleName.FunctionName TypeName"
```

```
def __lldb_init_module(debugger, internal_dict):
  ....
```

```
def RegisterTypeSummary(debugger: lldb.SBDebugger, typePrinter, typeName: string):
  debugger.HandleCommand("type summary add -F %s.%s %s" %
                                      (__name__, typePrinter.__name__, typeName))
```

Принтер для std::string_view – как подключить

```
lldb % type synthetic add TypeName --python-class ModuleName.ClassName"
```

Принтер для std::string_view – как подключить

```
lldb % type synthetic add TypeName --python-class ModuleName.ClassName"
```

```
def RegisterTypeChildrenProvider(debugger: lldb.SBDebugger, childrenProvider,
                                 typeName: string):
  debugger.HandleCommand("type synthetic add %s --python-class %s.%s" %
                                (typeName, __name__, childrenProvider.__name__))
```

Принтер для std::string_view – как подключить

```
def RegisterTypeSummary(debugger: lldb.SBDebugger, typePrinter, typeName: string):
  debugger.HandleCommand("type summary add -F %s.%s %s" %
                                         (__name__, typePrinter.__name__, typeName))


def RegisterTypeChildrenProvider(debugger: lldb.SBDebugger, childrenProvider,
typeName: string):
  debugger.HandleCommand("type synthetic add %s --python-class %s.%s" %
                                 (typeName, __name__, childrenProvider.__name__))
```

Принтер для std::string_view – как подключить

```
def RegisterTypeSummary(debugger: lldb.SBDebugger, typePrinter, typeName: string):
  debugger.HandleCommand("type summary add -F %s.%s %s" %
                                        (__name__, typePrinter.__name__, typeName))


def RegisterTypeChildrenProvider(debugger: lldb.SBDebugger, childrenProvider,
typeName: string):
  debugger.HandleCommand("type synthetic add %s --python-class %s.%s" %
                                 (typeName, __name__, childrenProvider.__name__))


def RegisterPrinter(debugger: lldb.SBDebugger, summary, children, typeName: string):
  RegisterTypeSummary(debugger, summary, typeName)
  RegisterTypeChildrenProvider(debugger, children, typeName)
```

Принтер для std::string_view – как подключить

```
lldb % type summary add -F ModuleName.ClassName -x TypeName
```

Принтер для std::string_view – как подключить

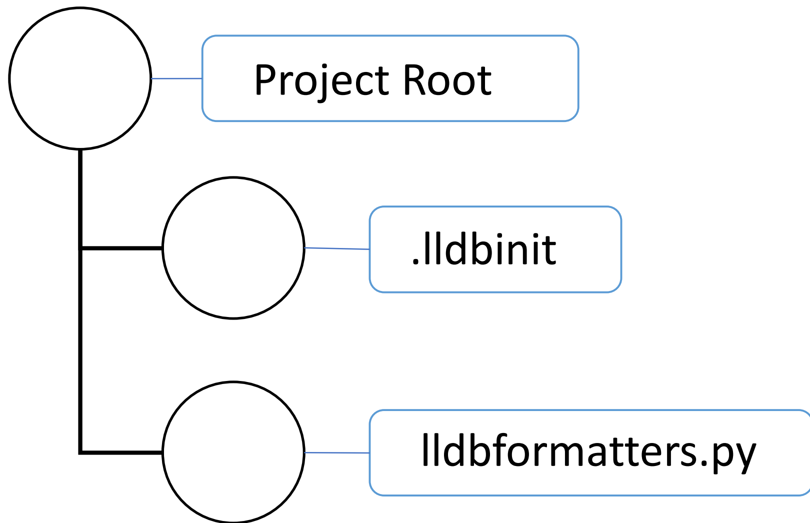
```
lldb % type summary add -F ModuleName.ClassName -x TypeName
```

```
def RegisterTypeSummaryTemplate(debugger: lldb.SBDebugger, typePrinter, typeName: string):
  debugger.HandleCommand("type summary add -F %s.%s -x %s" % (__name__, typePrinter.__name__, typeName))
```

Принтер для std::string_view – как подключить

```
def __lldb_init_module(debugger, internal_dict):
  lldb.formatters.Logger._lldb_formatters_debug_level = 2
  logger = lldb.formatters.Logger.Logger()


  # std
  ....
  RegisterPrinterTemplate(debugger, StdStringViewPrinter, StdStringViewProvider,
                          "^std::basic_string_view<.*>")


  logger >> "lldb formatters loaded"
```

Принтер для std::string_view – как подключить

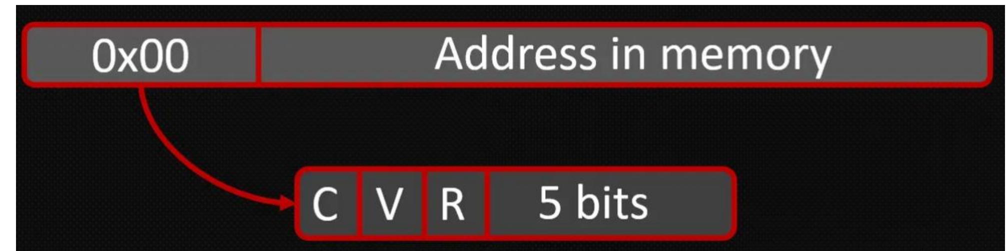


.lldbinit

```
command script import lldbformatters
```

Теперь подробнее

QualType

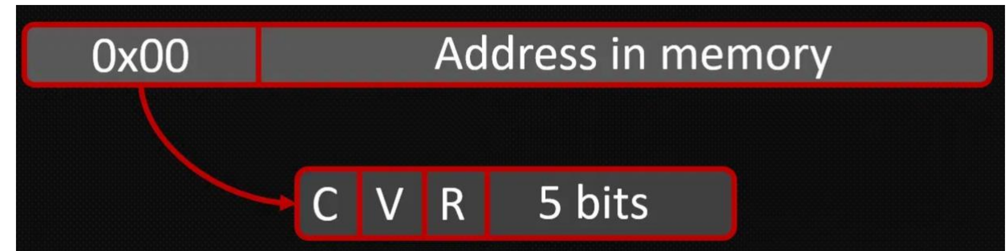


QualType

```
class QualType
{
  using ptr_t = TypeQualifiers::ptr_t;
  using realptr_t = TypeQualifiers::realptr_t;
  ....
public:
    realptr_t GetTypePtr() const noexcept;
private:
    ptr_t m_type{};
};
```

QualType

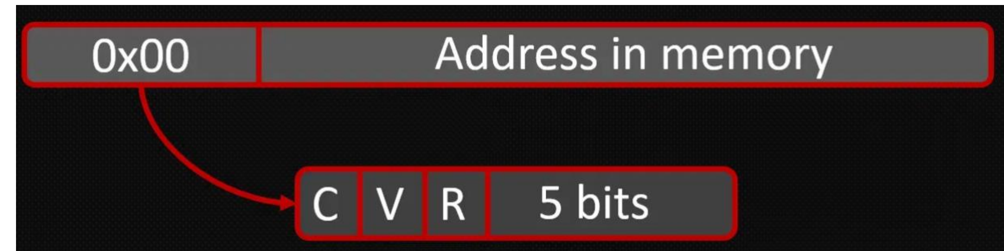
```
class TypeQualifiers
{
....
public:
  using ptr_t     = uintptr_t;
  using realptr_t = const Type*;
};

class QualType
{
  using ptr_t = TypeQualifiers::ptr_t;
  using realptr_t = TypeQualifiers::realptr_t;
  ....
public:
    realptr_t GetTypePtr() const noexcept;
private:
    ptr_t m_type{};
};
```



QualType

```
class TypeQualifiers
{
....
public:
  using ptr_t     = uintptr_t;
  using realptr_t = const Type*;
};

class QualType
{
  using ptr_t = TypeQualifiers::ptr_t;
  using realptr_t = TypeQualifiers::realptr_t;
  ....
public:
    realptr_t GetTypePtr() const noexcept;
private:
    ptr_t m_type{};
};
```

```
class Type
{
  ....

private:
  Id m_id;
};
```



QualType

```
class QualType
{
  ....
public:
    realptr_t GetTypePtr() const noexcept;
private:
    ptr_t m_type{};
};
```

QualType

```cpp
class QualType
{
  ....
public:
    realptr_t GetTypePtr() const noexcept;
private:
    ptr_t m_type{};
};
```

```python
def QualTypePrinter(value: lldb.SBValue, internal_dict):
  opt = lldb.SBExpressionOptions()
  if (not IsUnresolvedQualType(value, opt)):
    type : lldb.SBValue = value.EvaluateExpression("GetTypePtr();", opt)
    id : lldb.SBValue = type.deref.GetChildMemberWithName('m_id')
    return id.GetValue()
  return "[Unresolved]"
```

QualType

```
class QualType
{
  ....
public:
    realptr_t GetTypePtr() const noexcept;
private:
    ptr_t m_type{};
};
```

```
def IsUnresolvedQualType(qt : lldb.SBValue, opt : lldb.SBExpressionOptions) -> bool:
  return qt.EvaluateExpression("GetTypePtr()->IsUnresolved();", opt).GetValueAsSigned() != 0
```

```
def QualTypePrinter(value: lldb.SBValue, internal_dict):
  opt = lldb.SBExpressionOptions()
  if (not IsUnresolvedQualType(value, opt)):
    type : lldb.SBValue = value.EvaluateExpression("GetTypePtr();", opt)
    id : lldb.SBValue = type.deref.GetChildMemberWithName('m_id')
    return id.GetValue()
  return "[Unresolved]"
```

QualType

```
class QualType
{
  ....
public:
    realptr_t GetTypePtr() const noexcept;
private:
    ptr_t m_type{};
};
```

```
> [QualType]: Alias
```

```
def IsUnresolvedQualType(qt : lldb.SBValue, opt : lldb.SBExpressionOptions) -> bool:
  return qt.EvaluateExpression("GetTypePtr()->IsUnresolved();", opt).GetValueAsSigned() != 0

def QualTypePrinter(value: lldb.SBValue, internal_dict):
  opt = lldb.SBExpressionOptions()
  if (not IsUnresolvedQualType(value, opt)):
    type : lldb.SBValue = value.EvaluateExpression("GetTypePtr();", opt)
    id : lldb.SBValue = type.deref.GetChildMemberWithName('m_id')
    return id.GetValue()
  return "[Unresolved]"
```

QualType Children Provider

```
class QualTypeChildrenProvider:
  def __init__(self, valobj, internal_dict):
    ....
  def num_children(self):
    ....
  def get_child_index(self, name):
    ....
  def get_child_at_index(self, index):
    ....
```

QualType Children Provider

```
def __init__(self, valobj, internal_dict):
  self.value : lldb.SBValue = valobj
  self.frame : lldb.SBFrame = self.value.frame
  self.internal_dict = internal_dict
  self.opt = lldb.SBExpressionOptions()
  self.counter = 0
  self.isUnresolved = IsUnresolvedQualType(self.value, self.opt)
  self.opt.SetAllowJIT(True)
```

QualType Children Provider

```
def __init__(self, valobj, internal_dict):
  self.value : lldb.SBValue = valobj
  self.frame : lldb.SBFrame = self.value.frame
  self.internal_dict = internal_dict
  self.opt = lldb.SBExpressionOptions()
  self.counter = 0
  self.isUnresolved = IsUnresolvedQualType(self.value, self.opt)
  self.opt.SetAllowJIT(True)
```

```
def num_children(self):
  if self.isUnresolved:
    return 1
  return 6
```

```
def get_child_index(self, name):
  tmp = self.counter
  self.counter += 1
  return tmp
```

QualType Children Provider

```
def get_child_at_index(self, index):
```

QualType Children Provider

```
def get_child_at_index(self, index):
    if index == 0:
        asString : lldb.SBValue = self.value.EvaluateExpression("ToString();", self.opt)
        return asString.Clone("[ToString]")
```

QualType Children Provider

```
def get_child_at_index(self, index):
      if index == 0:
         asString : lldb.SBValue = self.value.EvaluateExpression("ToString();", self.opt)
         return asString.Clone("[ToString]")
      if index == 1 and not self.isUnresolved:
         type : lldb.SBValue = self.value.EvaluateExpression("GetTypePtr();", self.opt)
         return type.deref.Clone("[Type]")
```

QualType Children Provider

```
def get_child_at_index(self, index):
      if index == 0:
         asString : lldb.SBValue = self.value.EvaluateExpression("ToString();", self.opt)
         return asString.Clone("[ToString]")
      if index == 1 and not self.isUnresolved:
         type : lldb.SBValue = self.value.EvaluateExpression("GetTypePtr();", self.opt)
         return type.deref.Clone("[Type]")
      if index == 2:
         isConst : lldb.SBValue = self.value.EvaluateExpression("GetQualifiers().IsConst();", self.opt)
         return isConst.Clone("[IsConst]")
```

QualType Children Provider

```
def get_child_at_index(self, index):
      if index == 0:
         asString : lldb.SBValue = self.value.EvaluateExpression("ToString();", self.opt)
         return asString.Clone("[ToString]")
      if index == 1 and not self.isUnresolved:
         type : lldb.SBValue = self.value.EvaluateExpression("GetTypePtr();", self.opt)
         return type.deref.Clone("[Type]")
      if index == 2:
         isConst : lldb.SBValue = self.value.EvaluateExpression("GetQualifiers().IsConst();", self.opt)
         return isConst.Clone("[IsConst]")
      if index == 3:
         isVolatile : lldb.SBValue = self.value.EvaluateExpression("GetQualifiers().IsVolatile();", self.opt)
         return isVolatile.Clone("[IsVolatile]")
```

QualType Children Provider

```
def get_child_at_index(self, index):
      if index == 0:
         asString : lldb.SBValue = self.value.EvaluateExpression("ToString();", self.opt)
         return asString.Clone("[ToString]")
      if index == 1 and not self.isUnresolved:
         type : lldb.SBValue = self.value.EvaluateExpression("GetTypePtr();", self.opt)
         return type.deref.Clone("[Type]")
      if index == 2:
         isConst : lldb.SBValue = self.value.EvaluateExpression("GetQualifiers().IsConst();", self.opt)
         return isConst.Clone("[IsConst]")
      if index == 3:
         isVolatile : lldb.SBValue = self.value.EvaluateExpression("GetQualifiers().IsVolatile();", self.opt)
         return isVolatile.Clone("[IsVolatile]")
      if index == 4:
         isRestrict : lldb.SBValue = self.value.EvaluateExpression("GetQualifiers().IsRestrict();", self.opt)
         return isRestrict.Clone("[IsRestrict]")
```

QualType Children Provider

```python
def get_child_at_index(self, index):
      if index == 0:
         asString : lldb.SBValue = self.value.EvaluateExpression("ToString();", self.opt)
         return asString.Clone("[ToString]")
      if index == 1 and not self.isUnresolved:
         type : lldb.SBValue = self.value.EvaluateExpression("GetTypePtr();", self.opt)
         return type.deref.Clone("[Type]")
      if index == 2:
         isConst : lldb.SBValue = self.value.EvaluateExpression("GetQualifiers().IsConst();", self.opt)
         return isConst.Clone("[IsConst]")
      if index == 3:
         isVolatile : lldb.SBValue = self.value.EvaluateExpression("GetQualifiers().IsVolatile();", self.opt)
         return isVolatile.Clone("[IsVolatile]")
      if index == 4:
         isRestrict : lldb.SBValue = self.value.EvaluateExpression("GetQualifiers().IsRestrict();", self.opt)
         return isRestrict.Clone("[IsRestrict]")
      if index == 5:
         type : lldb.SBValue = self.value.EvaluateExpression("GetTypePtr();", self.opt)
         id : lldb.SBValue = type.deref.GetChildMemberWithName('m_id')
         return id.Clone("[TypeId]")
```

QualTyp[e Children Provider]

```
def get_child_at_index(self, index):
    if index == 0:
        asString : lldb.SBValue = self.v
        return asString.Clone("[ToString
    if index == 1 and not self.isUnreso
        type : lldb.SBValue = self.value
        return type.deref.Clone("[Type]"
    if index == 2:
        isConst : lldb.SBValue = self.va                                              )
        return isConst.Clone("[IsConst]"
    if index == 3:
        isVolatile : lldb.SBValue = self                                              lf.opt)
        return isVolatile.Clone("[IsVola
    if index == 4:
        isRestrict : lldb.SBValue = self                                              lf.opt)
        return isRestrict.Clone("[IsRest
    if index == 5:
        type : lldb.SBValue = self.value
        id : lldb.SBValue = type.deref.G
        return id.Clone("[TypeId]")
```

```
[QualType]: Alias
  [ToString]: "int32_t"
  [Type]: {...}
    m_canonical: Builtin
    m_hash: 17808521697069060345
    m_id: Alias
  [IsConst]: false
  [IsVolatile]: false
  [IsRestrict]: false
  [TypeId]: Alias
  [raw]: VivaCore::PtreeTyped::QualType
    m_type: 40285104
```

std::variant

```
template<typename... _Types>
    class variant
    : _Variant_storage<....> // схематически

{
};
```

std::variant

```
template<typename... _Types>
    class variant
    : _Variant_storage<....> // схематически

{
};


template<typename... _Types>
   struct _Variant_storage<false, _Types...>
{
  _Variadic_union<_Types...> _M_u;
  using __index_type = __select_index<_Types...>;
 __index_type _M_index;

}
```

std::variant

```
template<typename... _Types>
    class variant
    : _Variant_storage<....> // схематически

{
};


template<typename... _Types>
   struct _Variant_storage<false, _Types...>
{
  _Variant_union<_Types...> _M_u;
  using __index_type = __select_index<_Types...>;
 __index_type _M_index;

}
```

```
template<typename _First,
         typename... _Rest>
union _Variadic_union<_First, _Rest...>
{
  _Uninitialized<_First> _M_first;
  _Variadic_union<_Rest...> _M_rest
}
```

std::variant

```
∨ [raw]: std::variant<DataFlow::IntegerInterval, ArrayView<const DataFlow::IntegerInterval> >
  ∨ std::__detail::__variant::_Variant_base<DataFlow::IntegerInterval, ArrayView<const DataFlow::Integ
    ∨ std::__detail::__variant::_Move_assign_alias<DataFlow::IntegerInterval, ArrayView<const DataFlow:
      ∨ std::__detail::__variant::_Copy_assign_alias<DataFlow::IntegerInterval, ArrayView<const DataFlow
        ∨ std::__detail::__variant::_Move_ctor_alias<DataFlow::IntegerInterval, ArrayView<const DataFlow:
          ∨ std::__detail::__variant::_Copy_ctor_alias<DataFlow::IntegerInterval, ArrayView<const DataFlow
            ∨ std::__detail::__variant::_Variant_storage_alias<DataFlow::IntegerInterval, ArrayView<const
              ∨ _M_u: {...}
                ∨ _M_first: {...}
                  ∨ _M_storage: Single interval [686047232:686047232]
                    ∨ min: {...}
                      > m_data: {s1:686047232}
                    ∨ max: {...}
                      > m_data: {s1:686047232}
                > _M_rest: {...}
                _M_index: '\0'
```

```
_M_index
_M_first
_M_rest
```

std::variant

```
_M_index
_M_first
_M_rest
```

```
def StdVariantPrinter(value: lldb.SBValue, internal_dict):
  variantIndexValue = GetVariantIndex(value)
  variantIndex = variantIndexValue.GetValueAsSigned()
  return "std::variant:[index: %d]" % variantIndex
```

std::variant

```
def GetVariantIndex(variantValue : lldb.SBValue):
  return variantValue.children[0].children[0].children[0] \
                     .children[0].children[0].children[0] \
                     .children[1] # (7 раз)
```

```
_M_index
_M_first
_M_rest
```

```
def StdVariantPrinter(value: lldb.SBValue, internal_dict):
  variantIndexValue = GetVariantIndex(value)
  variantIndex = variantIndexValue.GetValueAsSigned()
  return "std::variant:[index: %d]" % variantIndex
```

std::variant SyntheticChildern

```
class StdVariantChildrenProvider:
    def __init__(self, valobj, internal_dict):
      self.value : lldb.SBValue = valobj
      self.frame : lldb.SBFrame = self.value.frame
      self.internal_dict = internal_dict
      self.opt = lldb.SBExpressionOptions()
      self.frame : lldb.SBFrame = self.value.GetFrame()
      self.module : lldb.SBModule = self.frame.GetModule()
      self.opt.SetAllowJIT(True)
```

```
_M_index
_M_first
_M_rest
```

std::variant SyntheticChildern

```python
class StdVariantChildrenProvider:
    def __init__(self, valobj, internal_dict):
      self.value : lldb.SBValue = valobj
      self.frame : lldb.SBFrame = self.value.frame
      self.internal_dict = internal_dict
      self.opt = lldb.SBExpressionOptions()
      self.frame : lldb.SBFrame = self.value.GetFrame()
      self.module : lldb.SBModule = self.frame.GetModule()
      self.opt.SetAllowJIT(True)

      self.variantIndexValue = GetVariantIndex(self.value)
      self.variantIndex = self.variantIndexValue.GetValueAsSigned()
```

```
_M_index
_M_first
_M_rest
```

std::variant SyntheticChildern

```
class StdVariantChildrenProvider:
    def __init__(self, valobj, internal_dict):
      self.value : lldb.SBValue = valobj
      self.frame : lldb.SBFrame = self.value.frame
      self.internal_dict = internal_dict
      self.opt = lldb.SBExpressionOptions()
      self.frame : lldb.SBFrame = self.value.GetFrame()
      self.module : lldb.SBModule = self.frame.GetModule()
      self.opt.SetAllowJIT(True)

      self.variantIndexValue = GetVariantIndex(self.value)
      self.variantIndex = self.variantIndexValue.GetValueAsSigned()

    def num_children(self):
      return 2

    def get_child_index(self, name):
      tmp = self.counter
      self.counter += 1
      return tmp
```

```
_M_index
_M_first
_M_rest
```

std::variant SyntheticChildern

```
class StdVariantChildrenProvider:
  ....
  def get_child_at_index(self, index):
    if index == 0:
      return MakeInt(self.value, self.variantIndex, "[Index]")
```

```
_M_index
_M_first
_M_rest
```

std::variant SyntheticChildern

```
class StdVariantChildrenProvider:
  ....
  def get_child_at_index(self, index):
    if index == 0:
      return MakeInt(self.value, self.variantIndex, "[Index]")

    if index == 1:
      firstNode : lldb.SBValue = GetVariantStorage(self.value)
      if self.variantIndex == None:
        return firstNode.Clone("[Value storage]")

      __M_storage : lldb.SBValue = GetVariantChildAtIndex(self.variantIndex, firstNode)
      return __M_storage.Clone("[Value]")
```

```
_M_index
_M_first
_M_rest
```

std::variant SyntheticChildern

```
def GetVariantStorage(variantValue : lldb.SBValue):
  return variantValue.children[0].children[0].children[0] \
                     .children[0].children[0].children[0] \
                     .children[0] # (7 раз)
```

```
_M_index
_M_first
_M_rest
```

std::variant SyntheticChildern

```python
def GetVariantStorage(variantValue : lldb.SBValue):
  return variantValue.children[0].children[0].children[0] \
                     .children[0].children[0].children[0] \
                     .children[0] # (7 раз)


def GetVariantChildAtIndex(index : int, begin : lldb.SBValue):
  result : lldb.SBValue = begin

  while index > 0:
    result = result.children[1] # rest
    index -= 1

  return result.children[0].children[0] # _M_first._M_storage
```

```
_M_index
_M_first
_M_rest
```

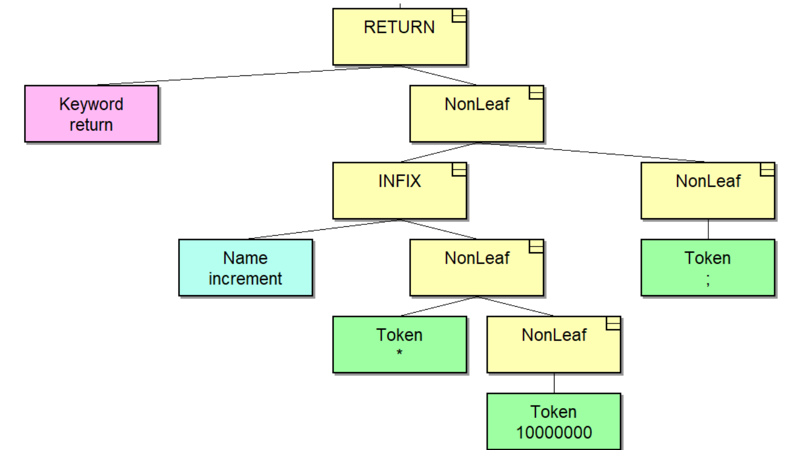
std::variant SyntheticChildern

```
def GetVariantStorage(variantValue : lldb.SBValue):
  retur                                            ren[0].
variant
childre


def Get                                            BValue):
  resul

  while index > 0:
    result = result.children[1] # rest
    index -= 1

  return result.children[0].children[0] # _M_first._M_storage
```

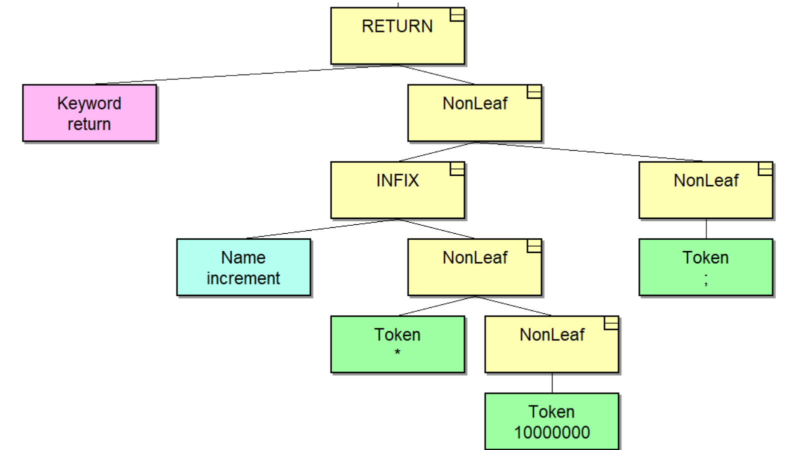
```
m_intervals: {0}
  [Index]: 0
  [Value]: Single interval [686047232:686047232]
    > [Min]: {686047232}
    > [Max]: {686047232}
    > [raw]: DataFlow::IntegerInterval
```

```
_M_index
_M_first
_M_rest
```

Syntax Tree

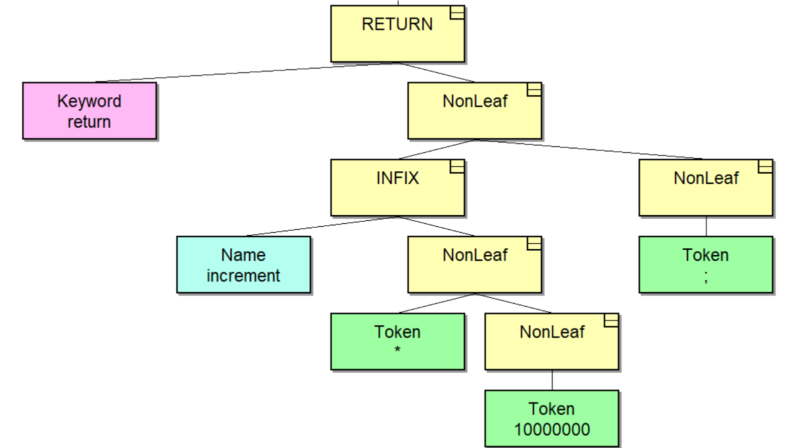


Syntax Tree



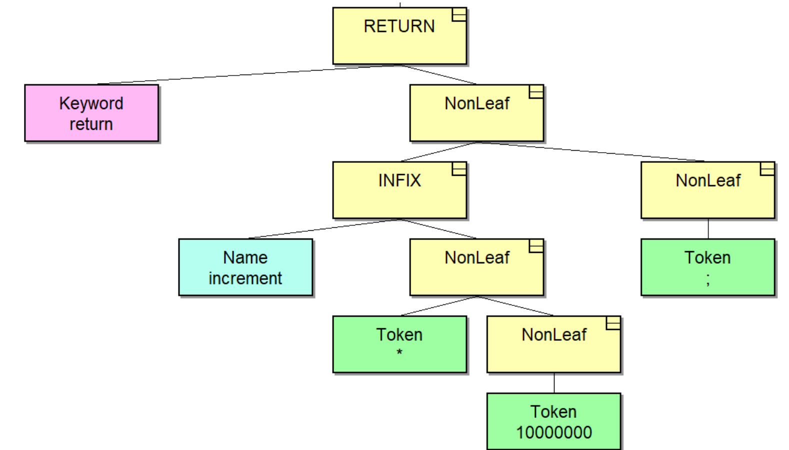

```
class SyntaxTree
{
public:
  SyntaxTree& operator =(const SyntaxTree&) = delete;
  SyntaxTree(const SyntaxTree&) = delete;

  // constructors
  SyntaxTree(SyntaxTree *p, SyntaxTree *q, const unsigned short what = TokenNames::BadToken)
noexcept;
  SyntaxTree(const char *ptr, ptrdiff_t len, const unsigned short what = TokenNames::BadToken)
noexcept;
  SyntaxTree(const Token &tk, const unsigned short what = TokenNames::BadToken) noexcept;
  explicit SyntaxTree(std::string_view str) noexcept;

  ....
 };
```

Syntax Tree



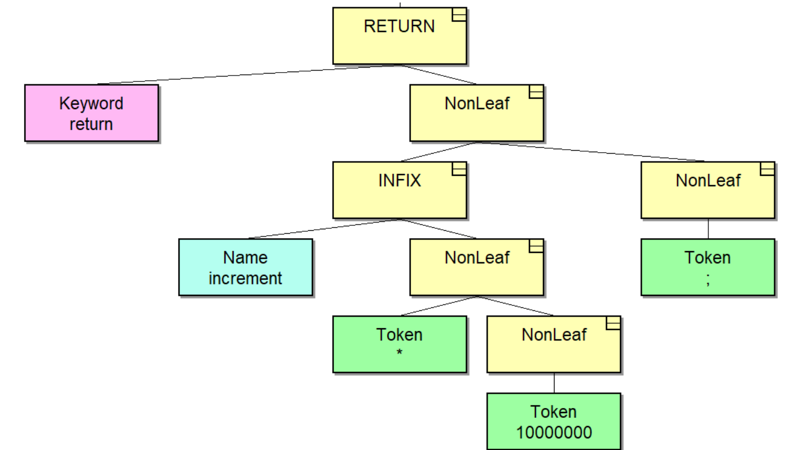

```
class SyntaxTree
{
public:
  ....
  union SyntaxTreeData
  {
    SyntaxTreeData(SyntaxTree *c, SyntaxTree *n) : nonleaf { c, n } { }
    SyntaxTreeData(std::string_view s) : leaf { s } { }

    struct _nonleaf
    {
      SyntaxTree *child;
      SyntaxTree *next;
    } nonleaf;

    std::string_view leaf;
  } data;
};
```

Syntax Tree




```
class SyntaxTree
{
public:
  ....
  [[nodiscard]] std::string ToString() const;
  unsigned short What() const noexcept { return m_what; }

private:
  unsigned short m_what : 13;
  unsigned short m_isLeaf : 1;
};
```

Syntax Tree




```
def SyntaxTreePrinter(value: lldb.SBValue, internal_dict):
  tokenString : lldb.SBValue = value.frame.EvaluateExpression("%s.What()" % GetName(value))
  type = value.frame.module.FindFirstType("VivaCore::TokenNames")
  return tokenString.Cast(type).value
```

Syntax Tree

```
class SyntaxTreeChildrenProvider :
  def __init__(self, valobj, internal_dict):

    childCountVal : lldb.SBValue = ExecuteFunction(self.value, "Util::Length", self.valName)

    self.childCount : int = childCountVal.GetValueAsSigned()
    self.fieldsCount = self.childCount + 1
    self.isValid = True
```

Syntax Tree

```
def ExecuteFunction(value : lldb.SBValue, functionName : string, valName : string = None):
  if valName is None:
    valName = value.GetName()

  if value.TypeIsPointerType():
    return value.frame.EvaluateExpression("%s(%s)" % (functionName, valName),  \
                                                                        lldb.eDynamicCanRunTarget)
  else:
    return value.frame.EvaluateExpression("%s(&%s)" % (functionName, valName), \
                                                                        lldb.eDynamicCanRunTarget)


class SyntaxTreeChildrenProvider :
  def __init__(self, valobj, internal_dict):

    childCountVal : lldb.SBValue = ExecuteFunction(self.value, "Util::Length", self.valName)

    self.childCount : int = childCountVal.GetValueAsSigned()
    self.fieldsCount = self.childCount + 1
    self.isValid = True
```

Syntax Tree

```
class SyntaxTreeChildrenProvider :
  ....
  def num_children(self):
      if not self.isValid:
        return 0

      # Обычные поля
      num = self.fieldsCount

      # потомки дерева
      if self.childCount > 0:
        num += self.childCount
      return num

    def get_child_index(self, name):
        tmp = self.counter
        self.counter += 1
        return tmp
```

Syntax Tree

```
class SyntaxTreeChildrenProvider :
  ....
  def get_child_at_index(self, index):
    if index == 0: # Type
      token : lldb.SBValue = self.value.GetChildMemberWithName("m_what", \
                                                                lldb.eDynamicCanRunTarget)
      type = self.module.FindFirstType("VivaCore::TokenNames")
      return MakeInt(token, token.GetValueAsSigned(), "[Type]").Cast(type)
```

Syntax Tree

```
def MakeInt(value : lldb.SBValue, intValue : int, varName : string) -> lldb.SBValue:
  data : lldb.SBData = lldb.SBData.CreateDataFromUInt64Array(  \
                                                   value.target.byte_order,  \
                                                   value.target.addr_size, [intValue])

  intType : lldb.SBType = value.target.GetBasicType(lldb.eBasicTypeInt)
  return value.CreateValueFromData(varName, data, intType)


class SyntaxTreeChildrenProvider :
  ....
  def get_child_at_index(self, index):
    if index == 0: # Type
      token : lldb.SBValue = self.value.GetChildMemberWithName("m_what", \
                                                          lldb.eDynamicCanRunTarget)
      type = self.module.FindFirstType("VivaCore::TokenNames")
      return MakeInt(token, token.GetValueAsSigned(), "[Type]").Cast(type)
```

Syntax Tree

```
class SyntaxTreeChildrenProvider :
  ....
  def get_child_at_index(self, index):
    ....
    if index == 1: # ToString
        ptreeToStringValue = ExecuteMemberFunction(self.value, self.valName, "ToString")
        type = self.module.FindFirstType("std::string")
        return PrepareValue(self.value, ptreeToStringValue.Cast(type), "[ToString]")
```

Syntax Tree

```
def ExecuteMemberFunction(value : lldb.SBValue, valName : string, functionName : string,\
                                                                              args : string = ""):
  if valName is None:
    valName = value.GetName()

  if value.TypeIsPointerType():
    return value.frame.EvaluateExpression("%s.%s(%s)" % (valName, functionName, args), \
                                                                       lldb.eDynamicCanRunTarget)
  else:
    return value.frame.EvaluateExpression("%s->%s(%s)" % (valName, functionName, args), \
                                                                       lldb.eDynamicCanRunTarget)

class SyntaxTreeChildrenProvider :
  ....
  def get_child_at_index(self, index):
    ....
    if index == 1: # ToString
        ptreeToStringValue = ExecuteMemberFunction(self.value, self.valName, "ToString")
        type = self.module.FindFirstType("std::string")
        return PrepareValue(self.value, ptreeToStringValue.Cast(type), "[ToString]")
```

Syntax Tree

```
class SyntaxTreeChildrenProvider :
  ....
  def get_child_at_index(self, index):
    ....
    if index == 4: # Parent
        rawValue : lldb.SBValue = GetMember(self.value, self.valName, "m_parent;")
        parent = PrepareValue(self.value, \
                   CloneValue(rawValue.GetDynamicValue(lldb.eDynamicCanRunTarget), "[Parent]"))

        if parent is None or IsNullPointer(parent):
          return MakeString(self.value, "nullptr", "[Parent]")

        return parent
```

Syntax Tree

```
parentDict : dict = dict()

def CloneValue(oldValue : lldb.SBValue, newName : string, newType : lldb.SBType = None) ->
lldb.SBValue:
  newName += ":\n" + oldValue.name
  parentDict[newName] = oldValue.name

  if newType is not None:
    oldValue = oldValue.Cast(newType)

  return oldValue.Clone(newName)
```

Syntax Tree

```
parentDict : dict = dict()

def CloneValue(oldValue : lldb.SBValue, newName : string, newType : lldb.SBType = None) ->
lldb.SBValue:
  newName += ":\n" + oldValue.name
  parentDict[newName] = oldValue.name

  if newType is not None:
    oldValue = oldValue.Cast(newType)

  return oldValue.Clone(newName)


class SyntaxTreeChildrenProvider :
    def __init__(self, valobj, internal_dict):

     if len(parentDict) > 5000:
         parentDict.clear()
```

Syntax Tree

```
def PrepareValue(parentValue : lldb.SBValue, value : lldb.SBValue, \
                 newName : string = None, typename : string = None) -> lldb.SBValue:
  if IsValid(value, typename):
    if not newName is None:
      return value.Clone(newName)
    return value

  if newName is None:
    newName = value.name

  return MakeString(parentValue, "Printer error: Can't evaluate", newName)
```

Syntax Tree

```
def IsValid(value : lldb.SBValue, typename : string = None) -> bool:
  if not value.IsValid():
    return False

  if typename is None:
    return True

  return value.type.name == typename

def PrepareValue(parentValue : lldb.SBValue, value : lldb.SBValue, \
                 newName : string = None, typename : string = None) -> lldb.SBValue:
  if IsValid(value, typename):
    if not newName is None:
      return value.Clone(newName)
    return value

  if newName is None:
    newName = value.name

  return MakeString(parentValue, "Printer error: Can't evaluate", newName)
```

Syntax Tree

```
class SyntaxTreeChildrenProvider :
  ....
  def get_child_at_index(self, index):
    ....
    if index == 5: # QualType
      ptreeValue = ExecuteFunction(self.value, "Util::GetPtreeValue", self.valName)

    if IsNullPointer(ptreeValue):
        return MakeString(self.value, "Not a Value", "[QualType]")

    qt : lldb.SBValue = ptreeValue.GetChildMemberWithName("m_type", \
                                                          lldb.eDynamicCanRunTarget)
    return PrepareValue(self.value, qt, "[QualType]", "VivaCore::PtreeTyped::QualType")
```

Syntax Tree

```
class SyntaxTreeChildrenProvider :
  ....
  def get_child_at_index(self, index):
    ....
    if index >  5: # Nth
        childrenNumber : string = index - self.fieldsCount
        currentChild : lldb.SBValue = PtreeNth(self.value, self.valName, childrenNumber)

        name : string = None
        if index == 6:
          name = "[First]"
        elif index == 7:
          name = "[Second]"
        elif index == 8:
          name = "[Third]"
        else:
          name = "[Nth(%s)]" % str(childrenNumber)

        if IsNullPointer(currentChild):
          return MakeString(self.value, "nullptr", name)

        return PrepareValue(self.value, CloneValue(currentChild, name))
```

Syntax Tree

```
def PtreeNth(ptree : lldb.SBValue, valName : string, index : int) -> lldb.SBValue:
  query : string = str()
  for _ in range(0, index):
    query += "data.nonleaf.next."
  query += "data.nonleaf.child"

  return GetMember(ptree, valName, query)
```

```
def PtreeNth(ptree : lldb.SBVa
  query : string = str()
  for _ in range(0, index):
    query += "data.nonleaf.nex
  query += "data.nonleaf.child

  return GetMember(ptree, valNa
```

```
⌄ expr: ntInfixExpr
    [Type]: ntInfixExpr
  > [ToString]: "increment * 10000000"
  > [WiseType]: {m_simpleType:ST_INT, m_firstTemplateArgType:ST_UNKNOWN, ...}
    [IsLeaf]: false
  > [Parent]: BadToken
  > [QualType]: Alias
  ⌄ [First]: tkIdentifier
      [Type]: tkIdentifier
    > [ToString]: "increment"
    > [WiseType]: {m_simpleType:ST_INT, m_firstTemplateArgType:ST_UNKNOWN, ...}
      [IsLeaf]: true
    > [Parent]: ntInfixExpr
    > [QualType]: Alias
    > [First]: None
    > [Second]: None
    > [Third]: "nullptr"
    > [Nth(3)]: "nullptr"
    > [Nth(4)]: "nullptr"
    > [raw]: VivaCore::Ptree *const
  > [Second]: BadToken
  > [Third]: tkConstant
```

Спасибо за внимание