

# VÔ TIẾN

Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)  
<https://www.facebook.com/groups/khmt.ktmt.cse.bku>



---

## Nguyên Lý Ngôn Ngữ Lập Trình (PPL)

---

**PPL1 - HK242**

---

### Task 3 - AST

---

Thảo luận kiến thức CNTT trường BK  
về KHMT(CScience), KTMT(CEngineering)  
<https://www.facebook.com/groups/khmt.ktmt.cse.bku>

# Mục lục

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Lý thuyết AST</b>   | <b>2</b>  |
| 1.1      | Kiến thức Python cơ bản . . . . .  | 2         |
| 1.2      | Kiến thức Class AST . . . . .  | 3         |
| 1.3      | Class của variable ctx . . . . .   | 5         |
| 1.3.1    | class RuleContext (org.antlr.v4.runtimeRuleContext) . . . . .                      | 5         |
| 1.3.2    | class ParserRuleContext(org.antlr.v4.runtime.ParserRuleContext) . . . . .          | 6         |
| 1.3.3    | class FunctionContext(ví dụ cụ thể) . . . . .                                      | 6         |
| 1.4      | Class ASTGeneration . . . . .  | 7         |
| 1.4.1    | Class ParseTreeVisitor . . . . .   | 7         |
| 1.4.2    | Class MiniGoVisitor . . . . .  | 7         |
| 1.4.3    | Class ASTGeneration . . . . .  | 7         |
| <b>2</b> | <b>Ví dụ AST</b>   | <b>8</b>  |
| <b>3</b> | <b>Một số phong cách code</b>  | <b>10</b> |
| <b>4</b> | <b>BTL2 AST phần 1 (không có deadline nên làm trước tết vì liên quan tới BTL1)</b> | <b>11</b> |



# 1 Lý thuyết AST

## 1.1 Kiến thức Python cơ bản

- Lệnh if else trong python

```
# lệnh if else cơ bản
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")

# if else tối giản giống toán tử 3 ngôi ? trong c++
print("A") if a > b else print("B")
```

- Dệ quy mảng. Ví dụ cho mảng array hãy nhân các phần tử trong mảng lênh 2 lần

```
def rec(array):
    if len(array) == 0: return []
    return [array[0] * 2] + rec(array[1:])
```

- Chuyển *string* về *int* dùng lệnh *int("123")*
- viết vòng lặp for ngay bên trong dấu ngoặc vuông để tạo list comprehension

```
squares = [x*x for x in range(10)]
evens = [x for x in range(10) if x % 2 == 0]
pairs = [(i, j) for i in [1, 2, 3] for j in [4, 5, 6]]
```

- Tuple trong Python là một cấu trúc dữ liệu cho phép bạn lưu trữ nhiều giá trị trong một biến, tương tự như list, nhưng tuple là bất biến (immutable) – tức bạn không thể sửa đổi, thêm hoặc xóa các phần tử sau khi đã tạo.

- constructor* trong python và cách khởi tạo đối tượng đó

```
@dataclass ## @dataclass là một decorator được cung cấp trong module dataclasses
class VarDecl(Decl, Stmt):
    name : Id
    varType : Type = None # None if there is no type
    modifier: str = None # None if there is no modifier
    varInit : Expr = None # None if there is no initial

    # khởi tạo đối tượng các phần tử không khởi tạo thì mặc định là None
    person1 = VarDecl(Id("VoTien"), BoolType(), None, None)
    person1 = VarDecl(Id("VoTien"), BoolType())
    person1 = VarDecl(Id("VoTien"), BoolType(), None, NumberLiteral(1))
```

- Đa hình (polymorphism) trong Python (hay trong lập trình hướng đối tượng nói chung) là khả năng một phương thức (hay hàm) có cùng tên nhưng hành vi (cách thực thi) khác nhau tuỳ thuộc vào lớp hoặc kiểu dữ liệu cụ thể.



## 1.2 Kiến thức Class AST

```
AST (Abstract Base Class)
| Expr (AST)           <-- Biểu diễn các biểu thức
|   | LHS (Expr)       <-- Biểu diễn các LHS
|   |   | Id(LHS)        <-- Tên biến, kiểu Id
|   |   | ArrayCell(LHS) <-- Phần tử mảng (arr, idx)
|   |   | FieldAccess(LHS) <-- Truy cập trường của một đối tượng (obj, fieldname)
|   | BinaryOp(Expr)    <-- Biểu thức nhị phân: phép tính giữa 2 vế (op, left, right)
|   | UnaryOp(Expr)     <-- Biểu thức đơn ngôi: phép tính trên 1 vế (op, body)
|   | CallExpr(Expr)    <-- Gọi hàm (obj, method, param)
|   | Literal(Expr)     <-- Lớp cơ sở của các literal (hằng)
|   |   | IntLiteral      <-- Hằng số nguyên (value)
|   |   | FloatLiteral    <-- Hằng số thực (value)
|   |   | StringLiteral   <-- Hằng chuỗi (value)
|   |   | BooleanLiteral  <-- Hằng boolean (value)
|   |   | ArrayLiteral    <-- Hằng mảng (typ, dimensions, value)
|   |   | StructLiteral   <-- Hằng kiểu struct (name, value là list cặp)
|   |   | NilLiteral      <-- Hằng nil (trống)

Type (AST)           <-- Biểu diễn các kiểu dữ liệu
| IntType
| FloatType
| StringType
| BooleanType
| StructType          <-- StructType(name: Id)

Program (AST)
| decl : List[Expr]    <-- TEST PHẦN 1
```

### Cách Khởi tạo

```
# 1. Ví dụ BinaryOp đơn giản: 1 + 2
expr1 = BinaryOp("+", IntLiteral(1), IntLiteral(2))
print("expr1 =", expr1)
# Kết quả chuỗi: BinaryOp("+", IntLiteral(1), IntLiteral(2))

# 2. Ví dụ UnaryOp: - ( 3.14 )
expr2 = UnaryOp("-", FloatLiteral(3.14))
print("expr2 =", expr2)
# Kết quả chuỗi: UnaryOp("-",FloatLiteral(3.14))

# 3. Ví dụ BinaryOp lồng nhau: (1 + 2) * 3
expr3 = BinaryOp("*", BinaryOp("+", IntLiteral(1), IntLiteral(2)), IntLiteral(3))
print("expr3 =", expr3)
# Kết quả chuỗi:
# BinaryOp("*",BinaryOp("+",IntLiteral(1),IntLiteral(2)),IntLiteral(3))

# 4. Gọi hàm (CallExpr): foo(1,2)
expr4 = CallExpr(None, Id("foo"), [IntLiteral(1), IntLiteral(2)])
print("expr4 =", expr4)
# Kết quả chuỗi:
# CallExpr(None,Id("foo"),[IntLiteral(1),IntLiteral(2)])

# 5. Gọi hàm với obj (ví dụ obj.bar("string", true)):
expr5 = CallExpr(Id("obj"), Id("bar"), [StringLiteral("Hello"), BooleanLiteral(True)])
```



```
print("expr5 =", expr5)
# Kết quả chuỗi:
# CallExpr(Id("obj"), Id("bar"), [StringLiteral("Hello"), BooleanLiteral(True)])

# 6. Truy cập phần tử mảng: arr[5]
expr6 = ArrayCell(Id("arr"), IntLiteral(5))
print("expr6 =", expr6)
# Kết quả chuỗi:
# ArrayCell(Id("arr"), IntLiteral(5))

# 7. Truy cập trường (structObj.fieldname)
expr7 = FieldAccess(Id("structObj"), Id("fieldname"))
print("expr7 =", expr7)
# Kết quả chuỗi:
# FieldAccess(Id("structObj"), Id("fieldname"))

# 8. ArrayLiteral: int[2][3], giá trị [1,2,3,4,5,6]
expr8 = ArrayLiteral(
    typ=IntType(),
    dimensions=[2, 3],
    value=[
        IntLiteral(1), IntLiteral(2), IntLiteral(3),
        IntLiteral(4), IntLiteral(5), IntLiteral(6)
    ]
)
print("expr8 =", expr8)
# Kết quả chuỗi:
# ArrayLiteral(IntType(), [2, 3], value=[IntLiteral(1), IntLiteral(2), ... IntLiteral(6)])

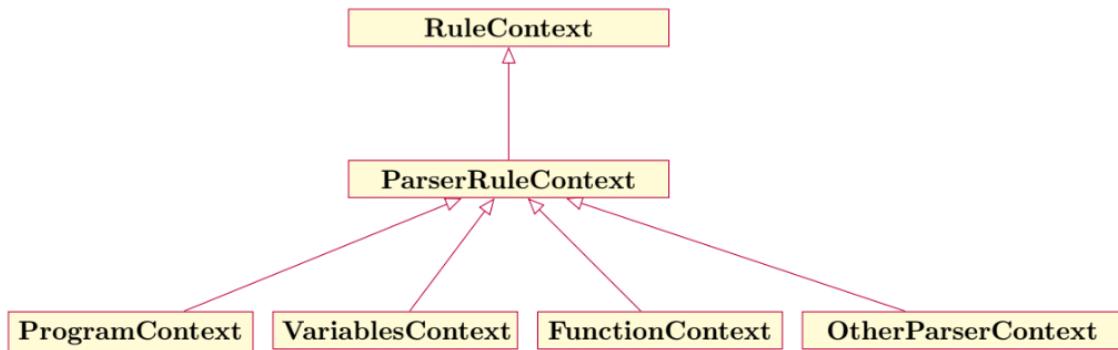
# 9. StructLiteral: struct MyStruct { x = 10, y = 2.5 }
expr9 = StructLiteral(
    name=Id("MyStruct"),
    value=[(Id("x"), IntLiteral(10)), (Id("y"), FloatLiteral(2.5))]
)
print("expr9 =", expr9)
# Kết quả chuỗi:
# StructLiteral(Id("MyStruct"), [(Id("x"), IntLiteral(10)), (Id("y"), FloatLiteral(2.5))])

# 10. NilLiteral
expr10 = NilLiteral()
print("expr10 =", expr10)
# Kết quả chuỗi:
# NilLiteral()

# ---- Thủ gộp nhiều biểu thức trên vào Program ----
program_example = Program([expr1, expr2, expr3, expr4, expr8, expr9])
print("program_example =", program_example)
# Ví dụ chuỗi in ra:
# Program([
#     BinaryOp("+", IntLiteral(1), IntLiteral(2)),
#     UnaryOp("-", FloatLiteral(3.14)),
#     BinaryOp("*", BinaryOp("+", IntLiteral(1), IntLiteral(2)), IntLiteral(3)),
#     CallExpr(None, Id("foo"), [IntLiteral(1), IntLiteral(2)]),
#     ArrayLiteral(IntType(), [2, 3], value=[IntLiteral(1), ... IntLiteral(6)]),
#     StructLiteral(Id("MyStruct"), [(Id("x"), IntLiteral(10)), (Id("y"), FloatLiteral(2.5))])
# ])
```



### 1.3 Class của variable ctx



#### 1.3.1 class RuleContext (org.antlr.v4.runtimeRuleContext)

Document : <https://www.antlr.org/api/JavaTool/org/antlr/v4/runtime/RuleContext.html>  
API function : thường dùng

1. `getChild(int i)` lấy ra con trong tree parser, sau là ví dụ khi ctx đang là function

```
function: ID COL FUNCTION (all_types | VOID) LP parameters_list? RP (INHERIT ID)?  
block_statement;  


- getChild(0) : object ID
- getChild(1) : object COL
- getChild(2) : object FUNCTION
- getChild(3) : object all_types | object VOID
- getChild(4) : object LP
- getChild(5) : object parameters_list (nếu có parameters_list) | object RP (nếu không có parameters_list)
- getChild(6) : object RP (nếu có parameters_list) | object INHERIT (nếu không có parameters_list và có (INHERIT ID)) | object block_statement (nếu không có cả (INHERIT ID) và parameters_list)
- ...

```

2. `getChildCount()` lấy ra số lượng con trong tree parser

```
function: ID COL FUNCTION (all_types | VOID) LP parameters_list? RP (INHERIT ID)?  
block_statement;  


- getChildCount() : 10 nếu tồn tại parameters_list và INHERIT ID
- getChildCount() : 9 nếu tồn tại INHERIT ID và không tồn tại parameters_list
- ....

```

3. `getText()` lấy ra string của node leaf tree parser

```
function: ID COL FUNCTION (all_types | VOID) LP parameters_list? RP (INHERIT ID)?  
block_statement;  


- ctx.ID().getText() : string ra chuỗi tên bắt ở lexer
- ctx.all_types().getText : error lỗi gì này là parser nên không phải là node leaf nên không tồn tại

```



### 1.3.2 class ParserRuleContext(org.antlr.v4.runtime.ParserRuleContext)

Document : <https://www.antlr.org/api/JavaTool/org/antlr/v4/runtime/ParserRuleContext>  
API function : thường dùng thừa kế từ RuleContext

### 1.3.3 class FunctionContext(ví dụ cụ thể)

Document : folder MinigoParser các bạn nộp ở TASK 2 BTL hiện tại là MinigoParser  
Code :

```
function: ID COL FUNCTION (all_types | VOID) LP parameters_list? RP (INHERIT ID)?  
        block_statement;
```

API function :

```
class FunctionContext(ParserRuleContext):  
    def ID(self, i:int=None):  
        if i is None:  
            return self.getTokens(MinigoParser.ID)  
        else:  
            return self.getToken(MinigoParser.ID, i)  
  
    def COL(self):  
        return self.getToken(MinigoParser.COL, 0)  
  
    def FUNCTION(self):  
        return self.getToken(MinigoParser.FUNCTION, 0)  
  
    def LP(self):  
        return self.getToken(MinigoParser.LP, 0)  
  
    def RP(self):  
        return self.getToken(MinigoParser.RP, 0)  
  
    def block_statement(self):  
        return self.getTypedRuleContext(MinigoParser.Block_statementContext, 0)  
  
    def all_types(self):  
        return self.getTypedRuleContext(MinigoParser.All_typesContext, 0)  
  
    def VOID(self):  
        return self.getToken(MinigoParser.VOID, 0)  
  
    def parameters_list(self):  
        return self.getTypedRuleContext(MinigoParser.Parameters_listContext, 0)  
  
    def INHERIT(self):  
        return self.getToken(MinigoParser.INHERIT, 0)
```

- Nếu parameters\_list không có gọi ctx.parameters\_list tương đương False trong python
- Có thể gọi các hàm từ class tổ tiên của.
- self.getToken này đang là node leaf nghĩa là lexer
- self.getTypedRuleContext này đang là node trung gian nghĩa là parser



## 1.4 Class ASTGeneration



### 1.4.1 Class ParseTreeVisitor

Document : <https://www.antlr.org/api/Java/org/antlr/v4/runtime/tree/ParseTreeVisitor.html>

API function : thường dùng

1. `visit(ParseTree tree)` gọi hàm visit với param tương ứng (có thể gọi đến chính nó bằng tên hàm trực tiếp hoặc có thể gọi qua param truyền vào)
2. `visitChildren(RuleNode node)` gọi xuống hàm con của biểu thức luôn, ví dụ 9 phần dưới

### 1.4.2 Class MiniGoVisitor

Document : target/MiniGoVisitor.py

### 1.4.3 Class ASTGeneration

Các bước Thực hiện trước khi code

**Bước 1:** Chạy lệnh `python run.py gen` thì 2 file `MinigoParser` và `MiniGoVisitor` nằm trong folder `target` có thể chạy lệnh `python run.py test LexerSuite` thì 2 file `MinigoParser` và `MiniGoVisitor` nằm trong folder `parse`

**Bước 2:** Tìm file `MiniGoVisitor.py` copy tất cả các hàm trong `class MiniGoVisitor` sang cho `ASTGeneration` đang viết, chỉ các hàm thôi nha

**Bước 3:** cập nhật file test case `test/ASTGen/ASTGenSuite.py` anh gửi trong discord

**Bước 4:** Có thể chỉnh file `MiniGo.g4` để code cho dễ nếu cần

**Bước 5:** Yêu cầu comment đúng chuẩn theo dạng phía trên là biểu thức parser

```
/* program: list_declared EOF;
def visitProgram(self, ctx:MinigoParser.ProgramContext):
    return Program(self.visit(ctx.list_declared()))
```



## 2 Ví dụ AST

- Biểu thức dạng trả về danh sách

```
list_declared: declared list_declared | declared;
```

Ta muốn lấy ra một mảng list\_declared dùng đệ quy

```
def visitList_declared(self, ctx:ZCodeParser.List_declaredContext):
    if ctx.list_declared():
        return [self.visit(ctx.declared())] + self.visit(ctx.list_declared())
    return [self.visit(ctx.declared())]
```

- Biểu thức dạng trả về một đối tượng

```
implicit_var: VAR ID ASSIGNINIT expression;
```

Xem danh sách các tham số trong file *AST* của class *ImplicitVarDecl* trong hàm *init* để truyền vào

```
def visitImplicit_var(self, ctx:ZCodeParser.Implicit_varContext):
    return VarDecl(Id(ctx.ID().getText()), None, None, self.visit(ctx.expression()))
```

- Biểu thức dạng trả về nhiều kiểu

```
type_prime: BOOL | NUMBER | STRING;
```

Thực hiện if else để kiểm tra thuộc loại nào rồi trả về *Type* của loại đó

```
def visitType_prime(self, ctx:ZCodeParser.Type_primeContext):
    if ctx.BOOL():
        return BoolType()
    elif ctx.NUMBER():
        return NumberType()
    return StringType()
```

- Biểu thức dạng expr

```
expression: expression1 CONCAT expression1 | expression1;
expression2: expression2 (AND | OR) expression3 | expression3
```

nếu trong một biểu thức có nhiều *expression* chung tên thì sẽ là một mảng nên cần lấy từng phần tử, nếu một trong hai bên của | có nhiều phân tử thì bên còn lại tuy có một vẫn tử nhưng vẫn mặc định là mảng

```
def visitExpression(self, ctx:ZCodeParser.ExpressionContext):
    if ctx.getChildCount() == 1:
        return self.visit(ctx.expression1()[0])

    op = ctx.CONCAT().getText()
    left = self.visit(ctx.expression1()[0])
    right = self.visit(ctx.expression1()[1])
    return BinaryOp(op, left, right)

def visitExpression2(self, ctx:ZCodeParser.Expression2Context):
    if ctx.getChildCount() == 1:
        return self.visit(ctx.expression3())

    op = ''
    if ctx.AND():
        op = ctx.AND().getText()
    elif ctx.OR():
        op = ctx.OR().getText()
```



```
    left = self.visit(ctx.expression2())
    right = self.visit(ctx.expression3())
    return BinaryOp(op, left, right)
```

## 5. Phần xử lí hơi rắc rồi hơn

```
## expression7: (ID / ID LPAREN index_operators? RPAREN)
##   LBRACKET index_operators RBRACKET / expression8;
if ctx.getChildCount() == 1:
    return self.visit(ctx.expression8())
elif ctx.getChildCount() == 4:
    return ArrayCell(Id(ctx.ID().getText()), self.visit(ctx.index_operators()[0]))
elif len(ctx.index_operators()) == 2:
    return ArrayCell(CallExpr(Id(ctx.ID().getText()),
        self.visit(ctx.index_operators()[0])), self.visit(ctx.index_operators()[1]))
return ArrayCell(CallExpr(Id(ctx.ID().getText()), []),
    self.visit(ctx.index_operators()[0]))
```

6. dùng *tuple* trong python xử lí với return a, b thì trả về một tuple (a,b) ta có thể gán c, b = (a,b) cũng được hoặc có thể phân rã nó ra dùng toán tử \*

```
## callStmt: func -> statement
def visitCallStmt(self, ctx:ZCodeParser.CallStmtContext):
    return CallStmt(*self.visit(ctx.func()))

## funcCall: func -> expression
def visitFuncCall(self, ctx:ZCodeParser.FuncCallContext):
    return FuncCall(*self.visit(ctx.func()))

## func: ID LPAREN index_operators? RPAREN
def visitFunc(self, ctx:ZCodeParser.FuncContext):
    if ctx.index_operators():
        return Id(ctx.ID().getText()), self.visit(ctx.index_operators())
    return Id(ctx.ID().getText()), []
```

7. xử lí ở biểu thức *IF*

```
## if_statement : IF expression statement elif_list (ELSE statement)?;
def visitIf_statement(self, ctx:ZCodeParser.If_statementContext):
    if not ctx.ELSE():
        return If(self.visit(ctx.expression()),
            self.visit(ctx.statement()[0]),
            self.visit(ctx.elif_list()), None)
    return If(self.visit(ctx.expression()),
        self.visit(ctx.statement()[0]),
        self.visit(ctx.elif_list()), self.visit(ctx.statement()[1]))

## elif_list: ELIF expression ignore? statement elif_list / ;
def visitElif_list(self, ctx:ZCodeParser.Elif_listContext):
    if ctx.getChildCount() == 0: return []
    return [(self.visit(ctx.expression()), self.visit(ctx.statement())),
        + self.visit(ctx.elif_list())]
```

## 8. Biểu thức statement

```
# Visit a parse tree produced by MT22Parser#statement.
## statement: ( assignment_statement
##             / if_statement / for_statement
```



```
##           / break_statement | continue_statement
##           / return_statement | call_statement
##           / block_statement | while_statement | do_while_statement);
def visitStatement(self, ctx:MT22Parser.StatementContext):
    return self.visitChildren(ctx) #! return self.visit(ctx.getChild(0))
```

#### 9. Biểu thức list\_statement

```
## list_statement: (statement | declaration_statement) list_statement
##                   | statement | declaration_statement;
def visitList_statement(self, ctx:MT22Parser.List_statementContext):
    ## statement : object | declaration_statement : list<object>
    stmt = self.visit(ctx.getChild(0))
    if ctx.getChildCount() == 1:
        if ctx.statement() : return [stmt]
        return stmt
    if ctx.statement() : return [stmt] + self.visit(ctx.list_statement())
    return stmt + self.visit(ctx.list_statement())
```

### 3 Một số phong cách code

- Nếu chắc chắn chỉ có 1 con

```
# a: b / c / d;
c1 => self.visitChildren(ctx)
c2 => self.visit(ctx.getChild(0)))
c3 => if else check ctx.a(), ctx.b(), ctx.c()
```

- biểu thức Expression

```
# a: b (+ / -) c;
c1 => BinaryOp(ctx.getChild(1).getText(), self.visit(ctx.c()), self.visit(ctx.c()))
c2 => dùng cơ bản if else để lấy + và - ra
```

- không dùng visit gọi thẳng luôn hàm

```
# a: e;
c1 => self.visitE(ctx.e())
c2 => self.visit(ctx.e())
```



#### 4 BTL2 AST phần 1 (không có deadline nên làm trước tết vì liên quan tới BTL1)

1. Tải Code về
2. Cập nhật code **MiniGo.g4** của bạn ở BTL1 của phần 1 Task 2 và chỉnh **program: list\_expression;**
3. **File ASTGeneration** với class được thừa kế từ **MiniGoVisitor** nên thừa kế lại tất cả các hàm **target/main/MiniGoVisitor.py**
4. **Code File ASTGeneration** với các hàm được thừa kế từ **MiniGoVisitor** từ các phần lý thuyết trên

