Nguyên Lý Ngôn Ngữ Lập Trình

Bài Tập Lớn

ZCODE

nhóm thảo luận CSE
https://www.facebook.com/groups/211867931379013

Tp. Hồ Chí Minh, Tháng 1/2024

Mục lục

1 Kế hoạch làm bài

Phần Làm	Bắt đầu	Thời gian	Kết thúc	Kiến thức	Nộp lại discord anh
BTL 2	29/1	6 ngày	23:59 5/2	AST+OOP+FP	BTL2 + Trắc nghiệm

Một số quy định của nhóm

- Nếu mấy bạn vô trễ thì thời gian làm có thể kéo dài theo ngày với thời gian Bắt đầu sẽ là ngày hôm đó
- Sau *deadline* thì anh không chỉ BTL phần đó nữa 
- Hỏi lý thuyết thì hỏi trong nhóm cho tiện
- Hỏi code thì nhắn riêng anh để bảo mật cho mấy bạn
- Nộp bài thì gửi file code *zcode.g4* đối btl1 và *ASTGeneration.py* đối btl2
- Trắc nghiệm thì hỏi lý thuyết *GK* hoặc *harmony*
- **Phần test case sẽ không cung cấp hết mà giữ lại để các bạn nộp anh check test case nếu sai anh gửi test sai cho mấy bạn fix**
- yêu cầu BTL2 là nộp file *ID_BTL2.py*(ASTGeneration.py) + *ID_BTL1.g4* trên mỗi hàm thì cần comment dòng sytax đã viết vào như ví dụ

```
#* program: (COMMENTS NEWLINE | NEWLINE)* list_declared EOF;
def visitProgram(self, ctx:ZCodeParser.ProgramContext):
    return Program(self.visit(ctx.list_declared()))
```

2 Lý thuyết Lexical structure

Zcode: Tokens

Tokens là khối xây dựng cơ bản của chương trình, chuỗi ký tự ngắn nhất có ý nghĩa riêng, Chúng là các thành phần cơ bản hay nguyên tố để xây dựng lên chương trình
Dễ hiểu nhất trong quá trình sản xuất bánh mì thì các nguyên vật liệu như **bột, hương vị, nước ...** là các *Tokens* mỗi thứ đều có ý nghĩa riêng của nó.

- *keywords* là các *Tokens* đã mặc định trước.
- *Operators* là các toán tử cho phép hiện thực trong chương trình
- *Separators* các kí hiệu phân cách của chúng
- *Identifiers* tên của biến, thuộc tính, hàm, class loại này là vô số *Tokens* được tạo ra
- *Literal* là chuỗi trong ngôn ngữ để thể hiện giá trị số nguyên, một chuỗi
- các lỗi (*lexical errors*) gồm 3 loại
  - Unclosed string không có kí tự để đóng chuỗi lại khi lập trình thì các phần sau hiện xanh lá trong *vscode*
  - Illegal escape in string một kí tự không cho phép trong ngôn ngữ
  - Error token không có *token* nào thõa mãn

Chú ý

- *Tokens* luôn bắt được chuỗi dài nhất có thể
- *Tokens* chuỗi bằng nhau thì ưu tiên chuỗi đầu tiên gặp được
- Nên viết tất cả các kí tự hoa (nếu không thì bắt buộc kí tự đầu viết hoa ràng buộc của ngôn ngữ rồi)

Một số ví dụ

1. Một chuỗi thời gian ngày/tháng/năm với ngày có thể 1 hoặc 2 kí tự số, tháng thì luôn là 2 kí tự số, năm thì có 4 kí tự số
   Kết quả:

```
DATE: [0-9][0-9]? '/' [0-9][0-9] '/' [0-9][0-9][0-9][0-9]
```

2. *INT*: số nguyên, là một chuỗi các chữ số bắt đầu bằng một chữ số khác 0 hoặc chỉ bằng 0, có thể được chỉ định ở dạng thập phân (cơ số 10). Tập hợp ký hiệu thập phân là (0-9) và không có chữ số 0 đứng trước. Các chữ số nguyên có thể chứa dấu gạch dưới (_) giữa các chữ số, để dễ đọc hơn các chữ số. Những dấu gạch dưới này sẽ được máy quét của MT22 (BTL kì k222) loại bỏ. Ví dụ
   1234 123
   1_72 (considered as 172 by scanner)
   1_234_567 (considered as 1234567 by scanner)
   Kết quả:

```
INT: [0-9] | [1-9] [0-9_]* [0-9] {self.text = "".join(self.text.split("_"))};
```

3. *STRING*: Một chuỗi ký tự bao gồm 0 hoặc nhiều ký tự được bao quanh bởi dấu ngoặc kép ("). Sử dụng chuỗi thoát (được liệt kê bên dưới) để biểu thị các ký tự đặc biệt trong một chuỗi. Hãy nhớ rằng dấu ngoặc kép không phải là một phần của chuỗi. Nó là một chuỗi biên dịch. -lỗi thời gian để một dòng mới hoặc ký tự EOF xuất hiện sau phần mở đầu (") và trước phần khớp đóng ("). Tất cả các chuỗi thoát được hỗ trợ
   \b backspace
   \f form feed
   \r carriage return
   \n newline

$\backslash t$ horizontal tab
$\backslash'$ single quote
$\backslash\backslash$ backslash
Đối với dấu ngoặc kép (") bên trong chuỗi, dấu gạch chéo ngược phải được viết trước chuỗi đó:$\backslash$".
Ví dụ:
"This is a string containing tab $\backslash t$"
"He asked me: $\backslash$"Where is John?$\backslash$""
Kết quả:

```
STRING_LIT: '"' (~[\r\n\f\\'"] | '\\' [bfrnt'"\\])* '"'{self.text = self.text[1:-1];};
```

Bảng kí hiệu trong *antlr*

Từ khóa	ý nghĩa	ví dụ
'kí tự'	Kí tự được bắt bảng ascii	z: 'z' -> kq z: 'a'
A B	sau kí tự A bắt buộc phải có B	z: 'a' 'b' -> kq z: 'ab'
A | B	chọn một kí tự trong 2 kí tự A và B	z: 'a' | 'b' -> kq z: 'a' or z: 'b'
'text'	này giống 'kí tự' mà này bắt luôn một chuỗi liên tiếp	z: 'VoTien' -> kq z: 'VoTien' (tưng đương A B)
A?	giống như A | rỗng	z: 'a'? -> kq z: 'VoTien' -> kq z: 'a' or z: ''
A*	rỗng hoặc nhiều hơn rỗng, A, AA, AAA, ...	z: 'a'* -> kq z: 'a' or z: 'aaaa'
A+	nhiều hơn A, AA, AAA, ... không có rỗng	z: 'a'+ -> kq z: 'a' or z: 'aaaa'
[a-z]	chọn một trong các kí tự từ a đến z	z: [a-z] -> kq z: 'a' or z: 'b'
[A-C]	chọn một trong các kí tự từ A đến C	z: [A-C] -> kq z: 'A' or z: 'Z'
[0-9]	chọn một trong các số từ 0 đến 9	z: [0-9] -> kq z: '5' or z: '6'
[a-z0-9]	chọn một trong các số từ 0 đến 9 hoặc từ a đến z	z: [a-z0-9] -> kq z: 'a' or z: '6'
[a-zA-Z0-9]	chọn một trong các số từ 0 đến 9 hoặc từ a đến z hoặc từ A đến Z	z: [a-zA-Z0-9] -> kq z: 'B' or z: '6'
[$\backslash n$]	kí tự xuống hàng	
[$\backslash r \backslash f \backslash \backslash$]	các kí tự hay dùng	
.(chấm)	tất cả các kí tự trong ascii	z: . -> kq z: '?' or z: '6'
[0-9]	tất cả các kí tự trong ascii ngoại trừ từ 0 đến 9	z: . -> kq z: '?' or z: 'a'
[a] -> skip	từ khóa skip bỏ qua khi bắt được kí từ a không nhận tokens này	z: [a] -> skip -> kq z: không hiện ra bỏ qua rồi
fragment INT: [0-9]+;	fragment Các đoạn là các phần có thể tái sử dụng của quy tắc lexer và không thể tự khớp với nhau - chúng cần được tham chiếu từ quy tắc lexer. giống $define$ trong $c++$	fragment z: [0-9]+ -> skip -> kq z: không hiện ra vì được tái sử dụng cho lexer khác
Biểu thức{self.text = self.text[1:-1];}	phần bên trong {} dùng để code python với $self.text$ là kí tự vừa bắt được với ví dụ này là lấy ra chuỗi bỏ đi kí tự đầu và cuối	https://www.w3schools.com/python/python_strings_slicing.asp

3 Lexical structure trong *Zcode*

3.1 Keywords, Operators và Separators

Phần này ghi hết rồi nên dễ dàng code thôi đọc phần *Keywords*, *Operators*, *Separators* đặt tên dễ nhớ, phần này hiện thực trước vì *antlr* bắt các kí tự đầu tiên thấy được tránh bắt nhầm xuống *Identifier*.

Zcode: Keywords, Operators, Separators

- *Keywords* từ khóa trong ngôn ngữ *Zcode* (không được đặt tên biến) : **true false number bool string return var dynamic func for until by break continue if else elif begin end not and or**
- *Operators* các toán tử trong ngôn ngữ *Zcode* : + - * / % **not and or** = <- != < <= > >= ... ==
- *Separators* các kí hiệu phân cách trong ngôn ngữ *Zcode* **:** ( ) [ ] ,
- **Chú ý:** Nếu *Keywords* trùng với *Operators* thị chọn 1 trong 2 thôi (đừng tham chọn cả 2 nếu dùng cả 2 thì nó chọn thằng đầu tiên khi code)

```
// KeyWord
IF: 'if';
ELSE: 'else';
FOR: 'for';
...
// Operator
ADD: '+';
SUB: '-';
MUL: '*';
GE: '>=';
...
// Separator
LPAREN: '(';
RPAREN: ')';
...
```

3.2 Identifier và Literal

Identifier là tên của biến, class, hàm, thuộc tính gì đó nào ...

```
ID: [a-zA-Z_][a-zA-Z0-9_]*;
```

- [] dùng để liệt kê các kí tự rồi còn 1 trong các kí tự đó ví dụ 'a' || 'b' sẽ tương đương với [ab] đối này chỉ dùng 1 kí tự nên nếu 'aa' || 'b' sẽ không dùng được, **Thường dùng gôm các kí tự thường, hoa, số lại**
- * này biểu thức chính quy *all* lấy kí

Literal giá trị mặt định gồm **Number, Boolean(này hiện thực lúc Keywords), String**

Zcode: Number

Số *ZCode* bao gồm phần nguyên buộc phải có, phần thập phân tùy chọn và phần mũ tùy chọn.

- Phần nguyên là một dãy gồm một hoặc nhiều chữ số 0 01 20 ... -> **chỉ cần 1 phần số nguyên**
- Phần thập phân là dấu thập phân, theo sau tùy chọn là một số chữ số . .0 .15 -> **gồm 2 phần nhỏ là dấu . và phần số nguyên với phần dấu chấm thì bắt buộc phải có còn số nguyên thì tùy chọn**
- Phần số mũ bắt đầu bằng ký tự 'e' hoặc 'E', theo sau là dấu '+' hoặc '-' tùy chọn, sau đó là một hoặc nhiều chữ số $xe+1$ $xe-1$ $xE-1$ $xe1$ -> **gồm 3 phần là chữ e và E thì bắt buộc phải có, phần $+$ và $-$ thì tuy chọn có thể không cần và cuối là phần số nguyên bắt buộc phải có**
- số đúng: 0 -0 199 001 012. 12. 0. 12.3 12.3e3 12.3e-30 2.e3 0.e-30 31e+3 31e-3 0e+3 0e-3
- số sai .12e-3 1.1h-2 1.12e-+3 1.12e-3.2 0.0.1

Zcode: String

chuỗi *ZCode* bao gồm tất cả kí tự và một số kí thực đặt biệt nằm trong dấu "

- khi lấy chuỗi thì không dùng dấu ": cần dùng *python* để xử lí $self.text = self.text[1:-1];;$ nghĩa là cắt chuỗi mới với lấy bỏ đi kí tự cuối và đầu
- Nếu một chuỗi mà xuống hàng thì sẽ bị lỗi nén phần sau.
- các kí tự cho phép ngoại trừ $\backslash r, \backslash n, \backslash f, \backslash\backslash, ', "$ tuy không cho phép $\backslash\backslash$ nhưng vẫn cho phép $\backslash\backslash$ kèm theo các kí tự $bfrnt$ và $\backslash\backslash$ theo kế bên luôn và thêm phần 2 kí tự liên tiếp để lấy dấu ngoặc kép " là $'"$
- Chuỗi có thể rỗng
- Phần biệt $'\backslash n'$(dấu enter khi gõ code) và $'\backslash\backslash n'$ (kí tự $'\backslash n'$ khi gõ code)

```
STRING_LIT: '"' (Kí tự cho phép)* '"' {self.text = self.text[1:-1];};
// với kí tự cho phép là
allow: ~[\r\n\f\\"] | '\\' [bfrnt'\\] | ['] ["]
```

3.3 Comments

```
NEWLINE: '\n'; // newline
COMEMENTS: '##' ~[\n\r\f]* -> skip; // Comments
WS : [ \t\r]+ -> skip ; // skip spaces, tabs
```

- *Zcode* có điểm đặt biệt là lấy luôn *NEWLINE* và *COMEMENTS* để xử lí phần sau vì có một số lỗi liên quan 2 này phần sau nên không *skip*
- *COMEMENTS* bắt đầu là cặp kí tự *##* sau đó một đoạn chuỗi tới kí tự xuống dòng cập nhật mới nhất này *skip*

3.4 Lỗi

```
ERROR_CHAR: .{raise ErrorToken(self.text)};
UNCLOSE_STRING: '"' (Kí tự cho phép)* ('\r\n' | '\n' | EOF)
{
    code python
};
```

```
ILLEGAL_ESCAPE: '"' (Kí tự cho phép)* ILLEGAL
{
    raise IllegalEscape(self.text[1:])
};
```

```
các bạn hay sai phần STRING_LIT, UNCLOSE_STRING, ILLEGAL_ESCAPE
-> kí tự không cho phép [\r\f\\] | '\\' ~[bfrnt'\\]
-> phần code python
    if self.text[-1] == '\n' and self.text[-2] == '\r':
        raise UncloseString(self.text[1:-2])
    elif self.text[-1] == '\n':
        raise UncloseString(self.text[1:-1])
    else:
        raise UncloseString(self.text[1:])
};
```

- *ERROR_CHAR* khi lexer phát hiện một ký tự không được nhận dạng
- *UNCLOSE_STRING* khi lexer phát hiện một chuỗi không kết thúc không bao gồm dấu ngoặc kép mở đầu
- *ILLEGAL_ESCAPE* khi lexer phát hiện một lối thoát bất hợp pháp trong chuỗi. Chuỗi sai là từ đầu chuỗi (không có dấu mở đầu) đến chỗ thoát trái phép. này là trường hợp các kí tự đặt biệt trong phần chuỗi
- *ILLEGAL* là các kí tự không được phép xử lí là ngược với phần kí tự cho phép
- các đưa code python vào thì chỉ cần dùng $\{PYTHON\}$ sử dùng chuỗi vừa bắt được là $self.text$
- $self.text$ lấy ra chuỗi vừa mới bắt được này là code *python*
- lấy kí tự chuỗi của kí tự $self.text[-1]$
- lấy chuỗi không lấy kì tự cuối $self.text[:-1]$
- lấy chuỗi không lấy kì tự đầu $self.text[1:]$
- lấy chuỗi không lấy kì tự cuối, đầu $self.text[1:-1]$
- *ErrorToken*, *IllegalEscape*, *UncloseString* dùng để nén ra lỗi nhận đầu vào là một chuỗi, *raise* nén ra lỗi

3.5 Vị trí file cần đưa vào và cách nộp bài

- *ZCode.g4* thay thế file *ZCode.g4* của thầy trong phần *src/main/zcode/parser/main/zcode/parser/*
- *LexerSuite.py* thay cho file *LexerSuite.py* của thầy trong file *src/test/LexerSuite.py*, phần này test một phần nên các bạn gửi code anh test trong máy anh đọc thử sai gì ko nha. Không lộ code đâu yên tâm 
- Nộp file pdf này + file code *ZCode.g4* nhớ đổi tên theo *ID* là $<ID>.pdf$ **và** $<ID>.g4$, *ID* trong phần anh gửi
- Làm phần TN rồi gửi nha
- **Deadline 23:59 ngày 16/1** 

3.6 Đọc lỗi

```
PS C:\Code\Programming-Course\PPL\BTL_232\BTL1\src> python run.py test LexerSuite
C:\Code\Programming-Course\PPL\BTL_232\BTL1\src\run.py:85: DeprecationWarning: unit
TestLoader.loadTestsFromTestCase() instead.
  suite = unittest.makeSuite(cls)
-------- Start ----------------------------------------------------------------
Tests run  1
Errors  []
[]
Test output
 .
----------------------------------------------------------------------
Ran 1 test in 0.025s

OK
```

Như này là pass qua hết rồi nha ăn mừng thôi 🙂🙂🙂

```
PS C:\Code\Programming-Course\PPL\BTL_232\BTL1\src> python run.py test LexerSuite
C:\Code\Programming-Course\PPL\BTL_232\BTL1\src\run.py:85: DeprecationWarning: unittest.makeSuite() is deprecate
TestLoader.loadTestsFromTestCase() instead.
  suite = unittest.makeSuite(cls)
-------- Start ----------------------------------------------------------------
Tests run  1
Errors  []
[(<LexerSuite.LexerSuite testMethod=test_simple_string>,
  'Traceback (most recent call last):\n'
  '  File '
  '"C:\\Code\\Programming-Course\\PPL\\BTL_232\\BTL1\\src\\./test\\LexerSuite.py", '
  'line 8, in test_simple_string\n'
  '    self.assertTrue(TestLexer.test("01","0,<EOF>",101))\n'
  'AssertionError: False is not true\n')]
Test output
 F
======================================================================
FAIL: test_simple_string (LexerSuite.LexerSuite.test_simple_string)
test simple string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Code\Programming-Course\PPL\BTL_232\BTL1\src\./test\LexerSuite.py", line 8, in test_simple_string
    self.assertTrue(TestLexer.test("01","0,<EOF>",101))
AssertionError: False is not true

----------------------------------------------------------------------
Ran 1 test in 0.005s

FAILED (failures=1)
```

Như này là fail rồi 🙂🙂🙂

- Chú ý ô hình màu đỏ thể hiện vị trí test đang kiểm tra đang nằm trong hàng số 8 của file *lexerSuite.py* trong đường dẫn *src/test/LexerSuite.py*
- hàm *TestLexer.test* phần đầu là *input* phần sau đáp án *expect* và 101 là vị trí file
- *src/test/testcases/101.txt* thì nảy chúng ta nhập *input* trong hàm *TestLexer.test* phần này ko cần bận tâm lắm
- *src/test/solutions/101.txt* **phần này chúng chạy** *input* **ra** *ouput* **của bạn hãy so với** *expect* **bạn đã sai gì**

4 syntax analyzer *Zcode*

4.1 Lý thuyết *syntax*

Zcode: Syntax Analysis

Syntax bước này sẽ phân tích cú pháp vị trí của các *tokens* bắt ở phần trước lấy ra xử lí xem thứ tự trước sau có đúng hay không, chủ ngữ vị ngữ có vị trí phù hợp hay không
Dễ hiểu nhất trong quá trình sản xuất bánh mì thì các chế bến mới vào nặng bột sau đó hấp sau đó nữa là bán các bước này đầu vấy nguyên liệu từ các bước trước đó.

- Cần *tokens* nào thì sẽ ưu cầu bước *lexer* lấy lên
- Sau đó kiểm tra vị trí từng *tokens*
- cuối cùng là sinh ra một cây AST từ các *tokens* trước đó có thể bỏ qua 1 số *tokens* không cần thiết
- Loại đầu *BNF* không cho phép xử dụng biểu thức chính quy
- Loại sau *EBNF* mở rộng của thằng trên nên cho xử dụng

Chú ý: vì phần này ảnh hưởng đến BTL2 nên các bạn không nên xử dụng các toán tử như * và + vì tới phần sau code sẽ chậm đi khá hiểu nếu bạn không rành về lập trình hàm(function programming) ở môn LTNC vẫn sử dụng ? như bình thường

Một số loại hay dùng:

1. Loại một là các *Tokens ID* cách nhau bởi dấu *COMMA*, có thể rỗng

```
list_ID: list | ; // có thể là list?
list: ID COMMA list | ID;
```

2. Loại hai là các *Tokens ID* cách nhau bởi dấu *COMMA*, không thể rỗng

```
list_ID: ID COMMA list_ID | ID;
```

3. Loại ba là các *Tokens ID* không cách nhau bởi dấu gì, có thể rỗng

```
list_ID: ID list_ID | ;
```

4. Loại bốn là các *Tokens ID* không cách nhau bởi dấu gì, không thể rỗng

```
list_ID: ID list_ID | ID;
```

5. Loại năm phần *Expression* ví dụ bên dưới

KHÔNG SỬ DỤNG TOÁN TỬ + VÀ * TRONG QUÁ TRÌNH CODE CÓ THỂ DÙNG ?, |

4.2 Expression và Value

4.2.1 Value (Literals)

1. Các giá trị của các kiểu nguyên thủy gồm *number*, *boolean*, *string*
   - *number* phần *NUMBER_LIT* được viết trước đó
   - *string* phần *STRING_LIT* được viết trước đó
   - *boolean* gồm *true* và *false* trong *keywords* hoặc *operator*
2. giá trị *array* danh sách các chữ được phân tách bằng dấu phẩy được đặt trong '[' và ']'. Các phần tử *Expression*. Ví Dụ $[1,5,7,12]$ *or* $[[1,2],[4,5],[3,5]]$.. Có thể lấy phần *list_expression* bên dưới. *array – lit* dùng để khai báo giá trị khác với *Index – operator* dùng để truy cập phần tử của mảng

```
# cho phép
[1, "1", [1,2], true, false]
[]
[1+1,2,3] -> 1+1
```

4.2.2 Expression

1. đầu tiên ta xét độ ưu tiên *precedence* của từng toán hạng có trong bảng sau (cao đến thấp) có nghĩ là chia từ ưu tiên thấp nhất là lên vị trí *expression* sau đó tới *expression1* *expression2* ... *expressionN* thì dừng

Operator Type	Operator	Arity	Position	Association
Index operator	[,]	Unary	Postfix	Left
Sign	-	Unary	Prefix	Right
Logical	not	Unary	Prefix	Right
Multiplying	*, /, %	Binary	Infix	Left
Adding	+, -	Binary	Infix	Left
Logical	and, or	Binary	Infix	Left
Relational	=, ==, !=, <, >, <=, >=	Binary	Infix	None
String	...	Binary	Infix	None

2. Loại *Association* = *None* ngôn ngữ này bao gồm $...,=,==,!=,<,>,<=,>=$ có nghĩa là không tồn tại 2 kí tự này trong một đoạn biểu thức mà toán tử này có độ ưu tiên thấp nhất
3. Loại *Association* = *Left* có nghĩa là trong một biểu thức cùng một toán từ thì bên trái được tính trước. nằm gần dưới gốc của cây hơn, sẽ đệ quy phía lên trái
4. Loại *Association* = *right* ngược lại với *left*
5. Cuối cùng độ ưu tiên cao nhất là các loại *ID*, *literal*, biểu thức trong dấu (*expression*) và gọi hàm.
6. Phần toán tử *index* thì bên trong là danh sách các *expression* dùng để truy cập một phần tử trong mảng và không có tính kết hợp, với giá trị truy cập sẽ là ID hoặc là hàm

```
# cho phép
a[1,"true", 1+1*3, [1,2]]
fun()[1,2]
# không cho phép
a[1][2]
"string"[1]
```

```
[1,2][1]
(1+1)[2]
```

7. Ngôn ngữ này thì thứ tự *expression* (toán tử ...), *expression1* (toán tử $=, ==, != , <, >, <=, >=$, *expression2* (toán tử *and*, *or*), ..., *expression7* (toán tử $[,]$) cuối cùng là *expression8* sẽ là các giá trị không phải toán tử là các loại *ID*, *literal*, biểu thức trong dấu (*expression*) và gọi hàm.
8. Tham số trong *Index* và *prameters* giống nên dùng chung là *list_expression* các *expression* cách nhau bởi dấu *COMMA*

4.2.3 Ví Dụ

1. Danh sách các *ID* cách nhau bởi dấu , (*COMMA*) và danh sách khác rỗng

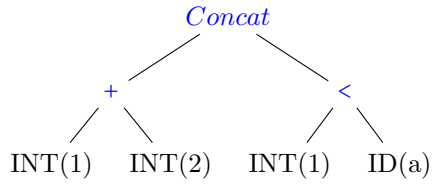
```
list_ID: ID COMMA list_ID | ID;
```

2. Ví dụ cho bảng độ ưu tiên *precedence* toán tử và tính kết hợp *association* của chúng với nhau

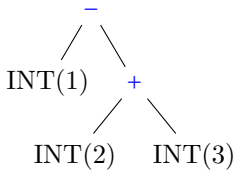
Operator Type	Operator	Arity	Position	Association
Logical	&,|	Binary	Infix	Left
Adding	+,-	Binary	Infix	Right
Relational	>,<	Binary	Infix	Left
String	concat	Binary	Infix	None

```
// parse
program: expression EOF;
expression: expression1 CONCAT expression1 | expression1;
expression1: expression1 (LT | GT) expression2 | expression2;
expression2: expression3 (ADD | SUB) expression2 | expression3;
expression3: expression3 (AND | OR) expression4 | expression4;
expression4: ID | INT | LPAREN expression RPAREN;
// lexer
ADD: '+';
SUB: '-';
LT: '<';
GT: '>';
AND: '&';
OR: '|';
CONCAT: 'concat';
LPAREN: '(';
RPAREN: ')';
INT: [0-9]*;
ID: [a-zA-Z_][a-zA-Z0-9_]*;
```

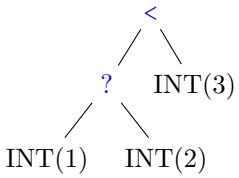
- Cây của biểu thức $1 \quad + \quad 2 \quad concat \quad 1 \quad < \quad a$



- Cây của biểu thức $1 \quad - \quad 2 \quad + \quad 3 = -4$



- Cây của biểu thức 1 > 2 < 3 = *true*



- Cây của biểu thức 1 *concat* 2 *concat* 3 -> lỗi

4.2.4 Bài tập vẽ các cây của phần *BTL* hiện tại với bảng *Expression* nằm ở phần 2.1.2

1. Cây của biểu thức **a + b * 2 - 1 % 2**
2. Cây của biểu thức **a = b and c = d**
3. Cây của biểu thức **(a = b) and (c = d)**
4. Cây của biểu thức **a = b ... c = d**
5. Cây của biểu thức **1 + a = b * c / 2**
6. Cây của biểu thức **not - 1 + 2**
7. Cây của biểu thức **fun(1) + 2 * - a[2,3][2]**

4.3 declared

program sẽ gồm danh sách các khai báo *variables* và *function* và *ignore* (này là *COMMENT*, *NEWLINE*)

1. *variables* chia thành 3 loại là

   - bắt đầu là *var* thì việc khởi tạo giá trị *expression* là bắt buộc. Nó bắt đầu bằng dấu gán (< −) và một biểu thức.

     ```
     # cho phép
     var Votien <- 2 + 1
     # không cho phép
     var Votien
     var Votien[1] <- 1
     ```

   - Khai báo *keyword* với các loại *type* có sẵn như *BOOL* | *NUMBER* | *STRING* và *array* khởi tạo luôn dấu gán (< −) hoặc có thể không gán. đối với *array* có dạng là *type ID*[*list_NUMBER_LIT*] với *list_NUMBER_LIT* là danh sách các các *NUMBER_LIT* cách nhau bởi dấu *COMMA*

     ```
     # cho phép
     number Votien
     number Votien <- 2 + 1
     number Votien[2,3] <- 1
     # không cho phép
     number Votien[]
     number Votien[1+2]
     number Votien["string"]
     ```

   - bắt đầu là *DYNAMIC* không được khai báo mảng *array* giống như trên khởi tạo luôn dấu gán (< −) hoặc có thể không gán

     ```
     # cho phép
     dynamic Votien
     dynamic Votien <- 1
     # không cho phép
     dynamic Votien[1] <- 2 + 1
     ```

2. *function* tùy chọn *statement* hoặc và có danh sách các *prameters* sẽ bao gồm Khai báo *keyword* với các loại *type* và mảng

   ```
   function: FUNC ID LPAREN prameters_list? RPAREN endl  (statement |);
   ```

   ```
   # cho phép
   func main()
   func main(number f1, bool x[5,2,3])
   # không cho phép
   func main(var c)
   func main(dynamic c)
   func main(number f1 <- 1)
   func main(number f1, bool x[true])
   func main(number f1, bool x[1+1])
   ```

3. *newline* thì gộp chung vào *ignore* trường hợp comment được khi phía trước nó là *newline* còn không thì sẽ ném ra lỗi

   ```
   # cho phép
   func main()

       return 1
   func main() return 1
   ```

4.4 Statements

Các loại biểu thức:

1. Variable declaration statement này giống phần khai báo biến trên này.
2. Assignment Statement với biểu thức *lhs < –expression* với *lhs* sẽ bao gồm *ID* là các biến *scalar* và *array* là các biến mảng có thể nhiều chiều *array* dùng toán tử *index* mà chỉ có ID không bao gồm hàm

```
# cho phép
id = 1;  // scalar
array[1+2, 2] = 2;  // array
# không cho phép
id + 1 = 2
fun() = 1
fun()[1] = 1
array[1][2] = 1
(array)[1+2, 2] = 2;
```

3. If statement với if là bắt buộc phải có còn khác thì tùy chọn. **viết các biểu thức *elif* một biến gọi tới**

```
if (expression-1) <statement-1>
[elif (expression-2) <statement-2>]?
[elif (expression-3) <statement-3>]?
[elif (expression-4) <statement-4>]?
...
[else <else-statement>]?
```

4. For statement biểu thức for thôi làm giống thầy thôi, *number – variable* là biến *ID*.

```
for <number-variable> until <condition expression> by <update-expression>
<statement>
```

```
# cho phép
for i until i >= 10 by 1 + 1 return 1
# không cho phép
for i[1] until i >= 10 by 1 + 1 return 1
```

5. Break statement và Continue statement chỉ cần dùng lại *keyword* trước đó thôi
6. Return statement phía sau có thể tùy chọn biểu thức có hoặc không

```
return <expression>?
```

7. Function call statement phía trước là *ID* sau đó là cặp dấu () bên trong nó có thể rỗng hoặc danh sách các *expression* cách nhau bởi dấu *COMMA*. giống thằng gọi hàm trong phần *expression*

```
fun();
fun(1+2, 1);
```

8. Block statement được bao quanh bởi *keywords* là *begin* và *end* bên trong là danh sách các *Statements*

```
begin
    var a <- 1
    break
    continue
    return 1+1
    begin end
end
```

9. *ignore* nằm ở cuối của các biểu thức *declaration*, *assignment*, *break*, *continue*, *return*, *callFunc*, *block*, đối với biểu thức trong *if*, *for*, *block* có thể có *ignore?* phía trước các *Statements* con của chúng, nhưng sau *begin* của block là bắt buộc *ignore*

5 Cập nhật mới nhất Ngày 23/1

- *array_literal* bên trong nhóm sẽ là các phần tử dạng *literal* không còn *expr* nữa và *array_literal* có thể rỗng

```
literal: NUMBER_LIT | STRING_LIT | TRUE | FALSE | array_literal;
array_literal: LBRACKET list_literal?  RBRACKET;
list_literal: literal COMMA list_literal | literal;
```

- *ignore* trong phần sau *begin* sẽ không phải tùy chọn nữa mà là bắt buộc

```
block_statement : BEGIN ignore statement_list END ignore;
```

- cập nhật *func* cuối không phải biểu thức nữa mà *block* hay *return*

```
function: FUNC ID LPAREN prameters_list? RPAREN
        (ignore? return_statement | ignore? block_statement | ignore);
```

6 Cập nhật mới nhất Ngày 25/1

- Comment ở lexer bị skip
- toán tử index của array bị thay đổi không còn có Association = Left mà là None chỉ cho phép biến và hàm

```
expression7: (ID | ID LPAREN index_operators? RPAREN)
LBRACKET index_operators  RBRACKET  | expression8;
```

- *prameters* không còn kiểu *DYNAMIC* nữa
- *return* có thể rỗng

7 Cập nhật mới nhất Ngày 30/1

- *STRING_LIT*, *UNCLOSE_STRING*, *ILLEGAL_ESCAPE* đối với kí tự cho phép thì cho phép thêm TH ′ đứng một mình, đối với kí tự không cho phép loại bỏ như sau

```
// kí tự cho phép
ALLOW: (~[\r\n\f\\"] | '\\' [bfrnt'\\] | '\'"' )
// kí tự không cho phép
NOT_ALLOW: [\r\f] | '\\' ~[bfrnt'\\]
```

- *array_literal* các giá trị nó là danh sách *expr* không phải danh sách *lit* nữa và không được rỗng

```
literal: NUMBER_LIT | STRING_LIT | TRUE | FALSE | array_literal;
array_literal: LBRACKET list_expr  RBRACKET;
list_expr: expression COMMA index_operators | expression;
```

- trong biểu thức *IF*, *ELSE*, *ELIF* thêm dấu ngoặc (*expr*)
- *expression6* ngoài toán tử *SUB* thêm *ADD* này thầy không ghi nhưng làm cho chắc ăn :<

```
expression6: (SUB | ADD) expression6 | expression7;
```

- Anh đưa test case lexer và parse gần đầy đủ rồi nha trong 2 file *LexerSuite.py* và *ParserSuite.py*
- Mấy bạn nộp rồi khỏi nộp lại Làm BTL2 đi
- Các bạn nộp file cho anh là dạng *ID_BTL1.g4* để anh đọc xem có phù hợp *BTL2* không, anh có đọc qua một số bài mà mấy bạn thích code đưa phần dư vào, phần dư tạo test kiểm tra rất khó nên thường đọc rồi thêm test case mới vào bộ test

8 Các Khóa Học HK232

nhóm thảo luận CSE

https://www.facebook.com/groups/211867931379013

- Lớp BTL1 + GK + LAB + Lý thuyết + Harmony của môn DSA HK232
- Lớp BTL2 + CK + LAB + Lý thuyết + Harmony của môn DSA HK232
- Lớp BTL1 + LAB + Lý thuyết + Harmony của môn KTLT HK232
- Lớp BTL2 + LAB + Lý thuyết + Harmony của môn KTLT HK232
- Lớp BTL1 + BTL2 + GK + Harmony của môn PPL HK232
- Lớp BTL3 + BTL4 + CK + Harmony của môn PPL HK232

CHÚC CÁC EM HỌC TỐT