

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 2 - PARSER

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

Mục lục

1	Lý thuyết parser	2
2	Ví dụ parser	3
2.1	Expression và Value	3
2.2	declared	5
2.3	Statements	5
3	Cách run và test BTL Parser	7
3.1	Cách hoạt động của lệnh python3/python run.py test ParserSuite	7
4	Bài tập parser	8
5	Parser phần 1(Deadline cả 2 phần 23:59 19/1)	9
5.1	Literal 6.6 pdf	9
5.2	Expressions 6 pdf	10



1 Lý thuyết parser

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
Để hiểu nhất trong quá trình sản xuất bánh mì thì các chẽ bén mới vào nặng bột sau đó hấp sau đó nữa là bán các bước này đều vậy nguyên liệu từ các bước trước đó.

- Cần *tokens* nào thì sẽ ưu tiên 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 nhiều nếu bạn không rèn 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

```
// cách 1
list_ID: list | ;
list: ID COMMA list | ID;
```

```
// cách 2
list_ID: 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

6. loại sáu khai báo đối xứng số lượng id bên trái bằng với số lượng khởi tạo bên phải

```
variables: ID CM variables CM expression | ID all_types ASSIGNINIT expression;
```

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



2 Ví dụ parser

2.1 Expression và Value

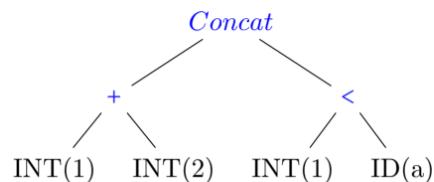
Cho bảng độ ưu tiên *precedence* toán tử và tính kết hợp *association* của chúng với nhau, *literal* sẽ gồm `in_literal` và `array_literal` có thể rỗng

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

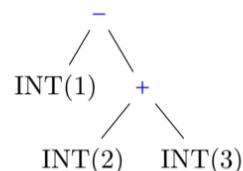
- Độ ưu tiên được xếp theo thứ tự trên xuống nghĩa là trong cùng một biểu thức thì toán tử có độ ưu tiên cao sẽ thực thi trước
- Tính kết hợp sẽ gồm 3 loại
 - Loại Left nghĩa là trong một biểu thức phía trái sẽ được tính trước, ví dụ `1 & 2 & 3` sẽ tương đương với `(1 & 2) & 3`
 - Loại Right nghĩa là trong một biểu thức phía phải sẽ được tính trước, ví dụ `1 + 2 - 3` sẽ tương đương với `1 + (2 - 3)`
 - Loại None nghĩa là trong một biểu thức không có 2 toán tử ngang hàng với nhau trên cùng biểu thức mà không có `()` hay toán tử cao hơn, ví dụ `1 concat 2 concat 3` là không tồn tại

```
// parse
program: expression EOF;
list_expression: expression CM list_expression | expression;
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 | literal | LPAREN expression RPAREN | ID LPAREN list_expression? RPAREN;
literal: INT | array_literal;
array_literal: LCB list_expression? RCB;
// lexer
ADD: '+';
SUB: '-';
LT: '<';
GT: '>';
AND: '&';
OR: '|';
CONCAT: 'concat';
LPAREN: '(';
RPAREN: ')';
INT: [0-9]*;
ID: [a-zA-Z_][a-zA-Z0-9_]*;
CM: ',';
LCB: '{';
RCB: '}';
```

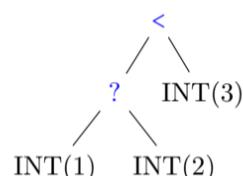
- Cây của biểu thức `1 + 2 concat 1 < a`



- Cây của biểu thức $1 - 2 + 3 = -4$



- Cây của biểu thức $1 > 2 < 3 = \text{true}$



- Cây của biểu thức $1 \text{ concat } 2 \text{ concat } 3 \rightarrow \text{lỗi}$

Bài tập vẽ các cây của phần *BTL* hiện tại với bảng *Expression*

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



2.2 declared

program sẽ gồm danh sách các khai báo *variables* và *Function*

1. **variables** tùy vào đặt tả của ngôn ngữ mà cách khai báo sẽ khác nhau

- Khai báo không khởi tạo có thể thành một danh sách (giống C++)

```
variables: all_types list_identifier CM;
list_identifier : ID CM list_identifier | ID;
// ví dụ : int a, b, c;
```

- Khai báo có khởi tạo mà phải đối xứng nhau

```
variables: all_types variables_init CM;
variables_init: ID CM variables_init CM expression | ID ASSIGNINIT expression;
// ví dụ : int a, b, c = 1, 2, 3;
```

- **all_types** thường là tất cả các tuy của ngôn ngữ (tùy theo thày mà có thể void không thuộc nhóm type khai báo biến)

- Tùy vào một số ngôn ngữ có cách khai báo biến đặt trưng mà quá trình bắt buộc có khởi tạo hay gì đó (Zcode BTL HK232)

```
implicit_var: VAR ID ASSIGNINIT expression;
keyword_var: all_type ID (ASSIGNINIT expression)?;
implicit_dynamic: DYNAMIC ID (ASSIGNINIT expression)?;
// ví dụ :
    ///! var id <- 1;
    ///! dynamic id;
    ///! string a <- 1
```

- **dimensions** của type array sẽ là danh sách integer, không được rỗng

```
atomic_type: INTEGER | FLOAT | BOOLEAN | STRING;
array_type: atomic_type LSB dimensions RSB;
dimensions: INT_LIT CM dimensions | INT_LIT;
// ví dụ: float id[1,2];
```

2. **Function** khai báo hàm tùy vào đặt tả ngôn ngữ mà cách hiện thực hàm sẽ khác nhau

- Type của hàm gồm tất cả các type kể cả Void

- Prama thì sẽ giống variables mà sẽ không có khởi tạo (tùy vào ngôn ngữ) (ví dụ Zcode BTL HK232)

```
// function declare
function: FUNC ID LP (parameters_list)? RP (return_statement | block_statement |);
//TODO parameters_list
parameters_list: parameter CM parameters_list | parameter;
parameter: primitive_decl | array_decl;
```

2.3 Statements

Các loại biểu thức thường gồm các dạng sau:

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, 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. **While statement** thực thi lặp đi lặp lại danh sách câu lệnh có thể rỗng trong một vòng lặp. Các câu lệnh while có dạng sau, trong đó <expression> ước tính thành giá trị boolean. Nếu giá trị là đúng, vòng lặp while sẽ thực thi <statement> lặp đi lặp lại cho đến khi biểu thức trở thành sai.

```
while (<expression>)
    <statement>
```

6. **Do-while**, giống như câu lệnh while, thực thi <block-statement> trong một vòng lặp (<block-statement> phải ở dạng câu lệnh khối trong 10). Không giống như câu lệnh while trong đó điều kiện vòng lặp được kiểm tra trước mỗi lần lặp, điều kiện của câu lệnh do-while được kiểm tra sau mỗi lần lặp. Do đó, vòng lặp do-while được thực hiện ít nhất một lần. Câu lệnh do-while có dạng sau:

```
do
    <block-statement>
while (<expression>);
```

7. **Break statement and Continue statement** chỉ cần dùng lại **keyword** trước đó thôi

8. Return statement phía sau có thể tùy chọn biểu thức có hoặc không

```
return <expression>?
```

9. **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 thẳng gọi hàm trong phần **expression**

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

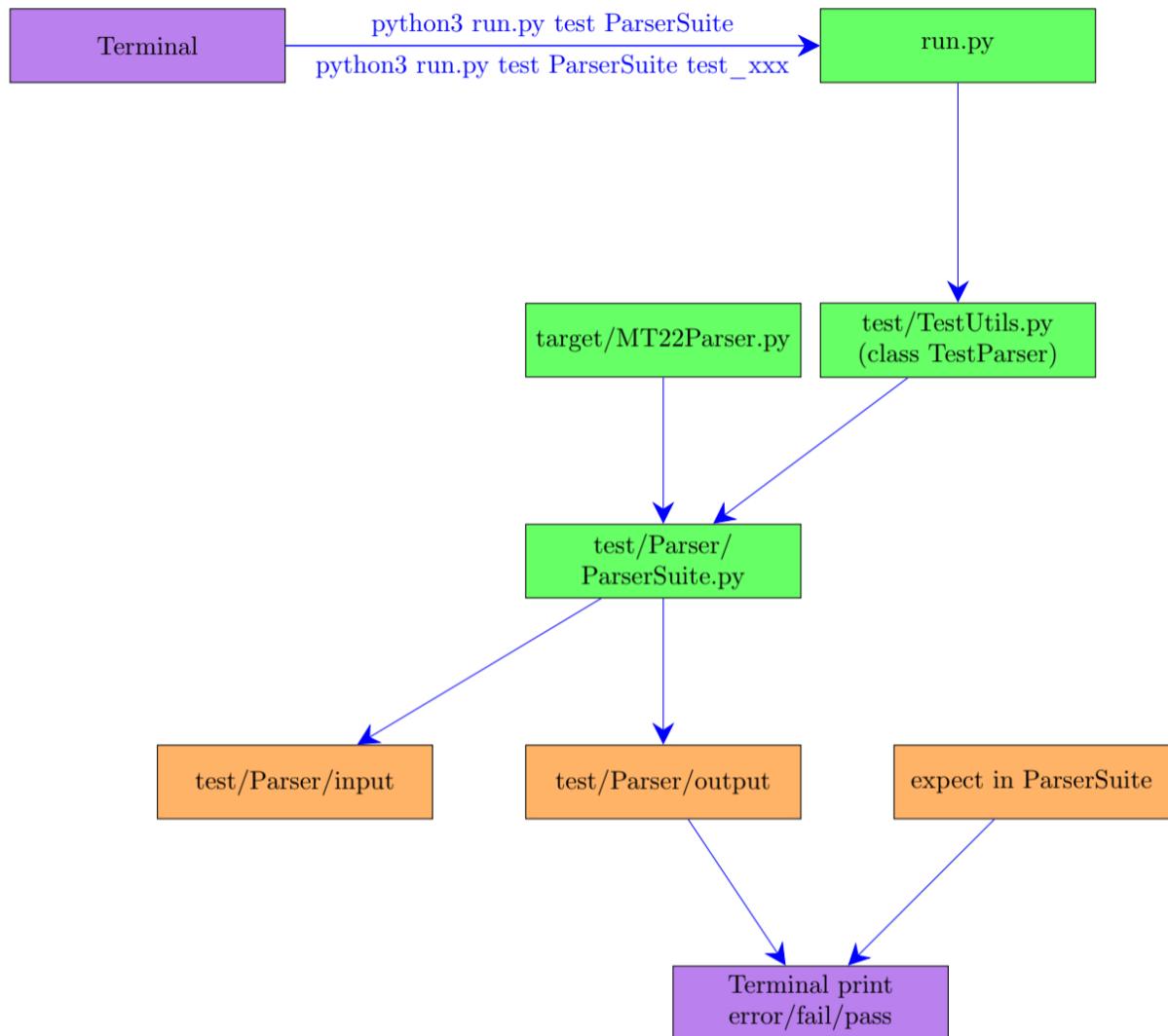
10. **Block statement** được bao quanh bởi **keywords** là **begin** và **end** bên trong là danh sách các **Statements**

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



3 Cách run và test BTL Parser

3.1 Cách hoạt động của lệnh python3/python run.py test ParserSuite





4 Bài tập parser

Code + test trong folder TASK02 Hãy biết biểu thức parser cho trường tình sẽ gồm 2 loại và luôn khác rỗng

1. Biểu thức khai báo

- Kiểu và danh sách ID

```
int a, b, c;
```

- Kiểu và danh sách ID và phép gán với danh sách biểu thức (mô tả ở sau) yêu cầu số lượng bên phải nhỏ hơn bằng bên trái, bên trái luôn lớn hơn 0

```
int a, b = 1; // PASS
int a, b = 1, 3; // PASS
```

```
int a, b = ; // FAIL
int a, b = 1, 3, 4; // FAIL
int = ; // FAIL
```

- **exp** gồm biểu thức tính toán của **INT_LIT** và **ID** với các toán tử

2. Biểu thức gán

số phần tử bên trái và bên phải bằng nhau

```
a, b = 1, 3;
```

Cho bảng độ ưu tiên *precedence* toán tử và tính kết hợp *association* của chúng với nhau

Operator Type	Operator	Arity	Position	Association
Adding	+,-	Binary	Infix	Right
MUL	*, /	Binary	Infix	Left
EXP	**	Binary	Infix	None

Hãy vẽ cây cho các biểu thức sau

- $1 + 2 - 3 / 3 * 4 ** 5$
- $1 + 2 ** 3 * 4$
- $1 + 2 ** 3 * 4 / 2$



5 Parser phần 1(Deadline cả 2 phần 23:59 19/1)

thêm COLON: ':';

Chạy lại test Lexer anh có thêm 1 test case về newline

5.1 Literal 6.6 pdf

1. Kiểu cơ bản

- Integer Literals
- Floating-point Literals
- String Literals
- Boolean Literals
- Nil Literal

2. Array Literal: Một biểu thức mảng bắt đầu bằng kiểu mảng, sau đó là danh sách các phần tử được bao quanh bởi dấu ngoặc nhọn.

`type_array {list_expression}`

- **type_array** Một biểu thức mảng bắt đầu bằng kiểu mảng, sau đó là danh sách các phần tử được bao quanh bởi dấu ngoặc nhọn.
 - Các mảng có thể có một chiều hoặc nhiều chiều.
 - Một khai báo kiểu mảng bắt đầu bằng danh sách các chiều, sau đó là một loại, có thể là bất kỳ loại nguyên thủy nào (int, float, boolean, string) hoặc các loại phức hợp như struct.
 - Một chiều là một hằng số nguyên hoặc hằng số được bao quanh bởi một cặp dấu ngoặc vuông [].

PASS

[5] int
[5] [0] string

FAIL

[5.] int
[5, 0 + 1] int

- **list_expression** danh sách các expression cách nhau bởi dấu phẩy

PASS

{2, {1 + 2}, "string"}
{1, 2, 3}, {4, 5, 6}

- **list_expression** có thể rỗng hay không ???

3. Struct Literal: Một khởi tạo struct bắt đầu bằng tên của struct, tiếp theo là một cặp dấu ngoặc nhọn chứa một danh sách các phần tử, phân tách bằng dấu phẩy, tùy chọn. Mỗi phần tử bao gồm một tên trường, dấu hai chấm và một biểu thức.

`ID {list_elements}`

list_elements bao gồm các ID : expression cách nhau dấu phẩy

PASS

Person{name: "Alice", age: 30}
Person{}



5.2 Expressions 6 pdf

Trong MiniGo, các toán tử được đánh giá dựa trên độ ưu tiên và tính kết hợp của chúng. Bảng sau tóm tắt độ ưu tiên và tính kết hợp của tất cả các toán tử:

Toán tử	Độ Ưu Tiên	Tính Kết Hợp
[], .	1 (Cao nhất)	Từ trái sang phải
!, - (đơn vị)	2	Từ phải sang trái
*, /, %	3	Từ trái sang phải
+, - (nhị phân)	4	Từ trái sang phải
==, !=, <, <=, >, >=	5	Từ trái sang phải
&&	6	Từ trái sang phải
	7 (Thấp nhất)	Từ trái sang phải

Bảng 1: Độ Ưu Tiên và Tính Kết Hợp của Các Toán Tử trong MiniGo

Biểu thức trong dấu ngoặc đơn có độ ưu tiên cao nhất, vì vậy dấu ngoặc đơn được sử dụng để thay đổi độ ưu tiên của các toán tử.

1. **Accessing array elements** Để truy cập một phần tử của mảng, MiniGo sử dụng toán tử chỉ mục [], trong đó biểu thức bên trong dấu ngoặc vuông phải được đánh giá thành một số nguyên.

```
a[2][3]
a["string"]
2[2]
```

```
//fail
a[]
```

2. **Accessing struct fields** Trong MiniGo, để truy cập các trường của một struct, ta sử dụng toán tử dấu chấm (.). Toán tử dấu chấm được sử dụng để tham chiếu một trường cụ thể của một struct.

```
person.name
person.age
person.address.name
(1+2).b
```

```
//fail
person.
person.(1+1)
```

3. **Function Call** Để gọi một hàm trong MiniGo, sử dụng tên hàm theo sau bởi một cặp dấu ngoặc đơn () chứa các đối số thực tế (nếu có). Các đối số trong dấu ngoặc đơn phải khớp với các tham số được khai báo trong chữ ký của hàm, cả về số lượng và kiểu dữ liệu.

```
add(3 + 2 / 3, "string", zero())
reset()
```

```
//fail
2(3, 4)
```

4. **Method Call** Gọi phương thức trong MiniGo yêu cầu bạn sử dụng cú pháp instance.methodName(arguments), nơi instance là thể hiện của kiểu dữ liệu (thường là struct), methodName là tên phương thức, và arguments là các đối số thực tế. Loại nhận được truyền một cách ngầm định, cho phép phương thức truy cập và thao tác với dữ liệu của instance mà không cần phải truyền nó như một tham số.

```
calculator.add(3, 4)
calculator.reset()
```



```
//fail  
int.a()
```

