VÕ 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 1 - LEXER

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

TP. HỒ CHÍ MINH, THÁNG 1/2025

Mục lục

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
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

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

// cách 2
list_ID: list?;
list: ID COMMA list | ID;
```

list_ID: Đây là một danh sách các ID (tên biến hoặc tên token).
list_ID có thể là một danh sách (list) hoặc rỗng (;).
Ví dụ: a, b, c hoặc không có gì (rỗng).
list: Cách định nghĩa danh sách:
Nếu có nhiều ID, chúng được nối với nhau bằng dấu phẩy (,).
Nếu chỉ có một ID, không cần dấu phẩy.
Cách hoạt động đệ quy:
list: ID COMMA list → Nếu có nhiều ID (ví dụ: a, b, c).
list: ID → Nếu chỉ có một ID (ví dụ: a).

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 ?, |

1. Lý thuyết về Parser
Parser là thành phần phân tích cú pháp trong trình biên dịch. Nhiệm vụ chính của nó là:

Phân tích cú pháp của chuỗi các tokens (đầu vào từ Lexer) để kiểm tra thứ tự và cấu trúc có tuân theo ngữ pháp đã định nghĩa hay không.
Xây dựng một cây cú pháp (AST - Abstract Syntax Tree) để biểu diễn cấu trúc của mã nguồn.
Chức năng chính của Parser:
Nhận các tokens từ Lexer: Parser sẽ yêu cầu Lexer cung cấp các tokens cần thiết.
Kiểm tra thứ tự cú pháp: Đảm bảo các tokens xuất hiện đúng vị trí và tuân theo các quy tắc ngữ pháp (ví dụ: vị trí của chủ ngữ, vị ngữ trong câu lệnh).
Sinh ra cây AST: AST là một cấu trúc cây biểu diễn logic của mã nguồn, có thể loại bỏ các tokens không cần thiết để tập trung vào phần quan trọng.
Ví dụ: Trong quá trình làm bánh mì:

Lexer như bước lấy nguyên liệu thô (bột, men, nước, v.v.).
Parser kiểm tra và đảm bảo các bước như nhào bột, nướng bánh, và bán bánh diễn ra đúng thứ tự.

2. Các loại ngữ pháp Parser
BNF (Backus-Naur Form):
Đây là dạng ngữ pháp chuẩn, không cho phép sử dụng biểu thức chính quy (regular expressions).
Mọi quy tắc được định nghĩa một cách nghiêm ngặt và tuần tự.
EBNF (Extended BNF):
Là phiên bản mở rộng của BNF, cho phép sử dụng các biểu thức chính quy như ?, *, và +.
Cách này linh hoạt hơn nhưng có thể gây ra khó khăn trong lập trình nếu không quen với các toán tử như * và +.
Lưu ý: Trong bài tập lớn (BTL2), tránh sử dụng các toán tử * và + vì dễ gây chậm và khó xử lý nếu chưa quen với lập trình hàm (functional programming). Tuy nhiên, toán tử ? (đại diện cho "có hoặc không có") vẫn có thể sử dụng.

3. Một số loại cấu trúc thường gặp
Loại 1: Các tokens ID cách nhau bởi dấu phẩy (COMMA), có thể rỗng
Cách viết 1:
bnf
list_ID: list | ;
list: ID COMMA list | ID;
Cách viết 2 (sử dụng EBNF):
ebnf
list_ID: list?;
list: ID COMMA list | ID;
Loại 2: Các tokens ID cách nhau bởi dấu phẩy (COMMA), không thể rỗng
bnf
list_ID: ID COMMA list_ID | ID;
Loại 3: Các tokens ID không cách nhau bởi dấu gì, có thể rỗng
bnf
list_ID: ID list_ID | ;
Loại 4: Các tokens ID không cách nhau bởi dấu gì, không thể rỗng
bnf
list_ID: ID list_ID | ID;
Loại 5: Biểu thức Expression
Biểu thức này dùng để xử lý các phép toán hoặc cấu trúc phức tạp.
Không có ví dụ cụ thể trong nội dung gốc.
Loại 6: Khai báo đối xứng
Số lượng ID bên trái phải bằng số lượng biểu thức bên phải:
bnf
variables: ID CM variables CM expression | ID all_types ASSIGNINIT expression;
4. Lưu ý trong quá trình code
Không sử dụng toán tử + và *: Dễ gây ra hiệu suất chậm hoặc lỗi nếu chưa quen.
Có thể sử dụng toán tử ? và |: ? đại diện cho "có hoặc không có," còn | đại diện cho "hoặc."

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 thức 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
  1. 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
  2. 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)
  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: '}';
```

program: Một chương trình hợp lệ phải bao gồm một biểu thức (expression) và kết thúc với EOF (end of file).
EOF: Đảm bảo Parser biết khi nào chương trình kết thúc.
Ví dụ:

Đầu vào: 1 + 2 * 3
Parse: program → expression EOF.

list_expression: Một danh sách các biểu thức được phân tách bởi dấu phẩy.
Trường hợp 1: Một biểu thức duy nhất (expression).
Trường hợp 2: Một biểu thức, dấu phẩy, và danh sách các biểu thức còn lại (expression CM list_expression).
Ví dụ:

Đầu vào: 1, 2, 3
Parse: list_expression → expression CM list_expression → expression CM expression CM expression.

Đầu vào: 42
Parse: list_expression → expression.

expression: Biểu thức cấp cao nhất.
Trường hợp 1: Biểu thức nối chuỗi (expression1 CONCAT expression1).
Trường hợp 2: Một biểu thức cấp thấp hơn (expression1).
Ví dụ:

Đầu vào: "hello" concat "world"
Parse: expression → expression1 CONCAT expression1.

Đầu vào: 1 + 2
Parse: expression → expression1.

- Cây của biểu thức 1 + 2 *concat* 3 < 4

expression1: expression1 (LT | GT) expression2 | expression2;
expression2: expression3 (ADD | SUB) expression2 | expression3;
expression3: expression3 (AND | OR) expression4 | expression4;
Ý nghĩa:
Cách này chia biểu thức thành nhiều cấp độ theo độ ưu tiên toán tử.
Cấp độ cao hơn gọi đệ quy xuống cấp độ thấp hơn.
Giải thích từng cấp:
expression1: Biểu thức với toán tử so sánh < hoặc >.
Ví dụ: 1 < 2 → expression1 → expression1 LT expression2.
expression2: Biểu thức với toán tử cộng/trừ +, -.
Ví dụ: 1 + 2 - 3 → expression2 → expression3 ADD expression2.
expression3: Biểu thức với toán tử logic &, |.
Ví dụ: true & false → expression3 → expression3 AND expression4.

expression4: ID | literal | LPAREN expression RPAREN | ID LPAREN list_expression? RPAREN;
literal: INT | array_literal;
array_literal: LCB list_expression? RCB;
expression4: Biểu thức đơn giản nhất, có thể là:

Một ID (ví dụ: a).
Một literal (số hoặc mảng).
Một biểu thức trong ngoặc đơn ((expression)).
Một hàm với danh sách tham số (ID LPAREN list_expression? RPAREN).
literal: Giá trị cơ bản, gồm:

Số nguyên (INT).
Mảng (array_literal): Danh sách các biểu thức nằm trong {}.
Đầu vào: (1 + 2)

Parse: expression4 → LPAREN expression RPAREN.
Đầu vào: {1, 2, 3}

Parse: array_literal → LCB list_expression RCB.

3. Tổng kết với ví dụ đầy đủ
Đầu vào: 1 + 2 * 3 < 4 & true concat "hello"
Phân tích các cấp:

1 + (2 * 3) → Ưu tiên * trước +.
(1 + (2 * 3)) < 4 → Ưu tiên + trước <.
((1 + (2 * 3)) < 4) & true → Ưu tiên < trước &.
(((1 + (2 * 3)) < 4) & true) concat "hello" → Ưu tiên concat.
Parse từng phần:

expression → expression1 CONCAT expression1.
expression1 → expression1 AND expression2.
expression2 → expression3 ADD expression2.

1. Các khái niệm cần hiểu

1.1. Độ ưu tiên (Precedence):

Độ ưu tiên xác định thứ tự thực thi các toán tử trong một biểu thức.

Toán tử có độ ưu tiên cao sẽ được thực thi trước toán tử có độ ưu tiên thấp hơn.

Ví dụ:

Trong biểu thức 1 + 2 * 3, phép * có độ ưu tiên cao hơn phép +, nên kết quả sẽ là 1 + (2 * 3).

1.2. Tính kết hợp (Association):

Khi có các toán tử có cùng độ ưu tiên, tính kết hợp xác định thứ tự thực thi.

Có 3 loại:

Left (trái): Toán tử bên trái được tính trước.

Ví dụ: 1 & 2 & 3 tương đương với (1 & 2) & 3.

Right (phải): Toán tử bên phải được tính trước.

Ví dụ: 1 + 2 - 3 tương đương với 1 + (2 - 3).

None (không xác định): Không thể có hai toán tử ngang hàng trong cùng một biểu thức mà không có dấu ngoặc hoặc toán tử ưu tiên cao hơn.

Ví dụ: 1 concat 2 concat 3 không hợp lệ nếu không có dấu ngoặc hoặc quy định rõ hơn.

2. Cấu trúc của Parser

Parser được định nghĩa qua các quy tắc ngữ pháp (grammar rules) để phân tích cú pháp. Mỗi quy tắc xác định cách kết hợp các tokens từ Lexer thành một cấu trúc logic.

2.1. Quy tắc Parser cho biểu thức

Chương trình chính (program):

bnf

```
program: expression EOF;
```

Một chương trình chỉ gồm một biểu thức kết thúc bởi EOF (end of file).

Danh sách biểu thức (list_expression):

bnf

```
list_expression: expression CM list_expression | expression;
```

Một danh sách các biểu thức được phân tách bởi dấu phẩy ,.

Biểu thức chính (expression):

bnf

```
expression: expression1 CONCAT expression1 |
expression1;
```

Biểu thức có thể là phép nối chuỗi (concat) hoặc một biểu thức cấp thấp hơn (expression1).

Biểu thức cấp thấp hơn:

bnf

```
expression1: expression1 (LT | GT) expression2 |
expression2;
expression2: expression3 (ADD | SUB) expression2 |
expression3;
expression3: expression3 (AND | OR) expression4 |
expression4;
```

expression1: Xử lý toán tử so sánh (<, >).

expression2: Xử lý toán tử cộng/trừ (+, -).

expression3: Xử lý toán tử logic (&, |).

expression4: Xử lý cơ bản nhất (như literal, ID, hoặc nhóm ngoặc đơn).

Biểu thức cơ bản (expression4):

bnf

```
expression4: ID
        | literal
        | LPAREN expression RPAREN
        | ID LPAREN list_expression? RPAREN;
```

Có thể là một ID, một literal, hoặc một biểu thức trong ngoặc đơn.

Ngoài ra, có thể là một hàm với danh sách biểu thức làm tham số.

Literal:

bnf

```
literal: INT | array_literal;
```

Literal có thể là một số nguyên (INT) hoặc một mảng (array_literal).

Mảng (array_literal):

bnf

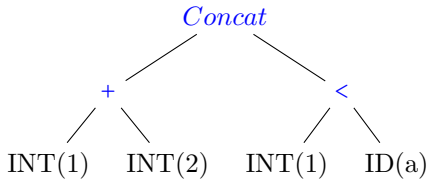
```
array_literal: LCB list_expression? RCB;
```

Một mảng được biểu diễn trong dấu {} và chứa danh sách các biểu thức (hoặc rỗng).

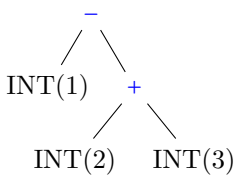
2.2. Lexer (Bộ phân tích từ vựng)
Lexer xác định các tokens cơ bản từ mã nguồn. Dưới đây là danh sách tokens cùng định nghĩa:

Token	Ký hiệu	Mô tả
ADD	+	Toán tử cộng.
SUB	-	Toán tử trừ.
LT	<	Toán tử so sánh nhỏ hơn.
GT	>	Toán tử so sánh lớn hơn.
AND	&	Toán tử logic "và".
OR	`	`
CONCAT	concat	Toán tử nối chuỗi.
LPAREN	(	Dấu ngoặc đơn mở.
RPAREN	)	Dấu ngoặc đơn đóng.
INT	[0-9]*	Literal số nguyên.
ID	[a-zA-Z_][a-zA-Z0-9_]*	Tên biến hoặc ID.
CM	,	Dấu phẩy, dùng để phân tách danh sách.
LCB	{	Dấu ngoặc nhọn mở (mảng).
RCB	}	Dấu ngoặc nhọn đóng (mảng).

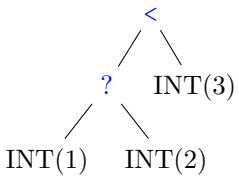
3. Độ ưu tiên và tính kết hợp trong Parser
Độ ưu tiên toán tử (cao nhất đến thấp nhất):
Logical (&, |): Thực thi sau cùng.
Adding (+, -): Ưu tiên cao hơn logic.
Relational (<, >): Ưu tiên cao hơn phép cộng/trừ.
String concat (concat): Thực thi đầu tiên.
Tính kết hợp của các toán tử:
Toán tử &, |, <, > có tính kết hợp Left.
Toán tử +, - có tính kết hợp Right.
Toán tử concat không cho phép tính kết hợp ngang hàng.



- Cây của biểu thức 1 - 2 + 3 = -4



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



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

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 thầy 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 vả 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 (prameters_list)? RP (return_statement | block_statement |);
     //TODO prameters_list
     prameters_list: prameter CM prameters_list | prameter;
     prameter: 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
   ```

Variables - Khai báo biến
1.1. Khai báo không khởi tạo
variables: all_types list_identifier CM;
list_identifier: ID CM list_identifier | ID;
Giải thích:
Biến có thể được khai báo mà không cần khởi tạo giá trị ban đầu.
Nếu có nhiều biến, chúng được ngăn cách bởi dấu phẩy ,.

Ví dụ:
int a, b, c; // Khai báo 3 biến kiểu int, chưa khởi tạo
Cách parse:
variables → all_types list_identifier CM
list_identifier → ID CM list_identifier → ID CM ID CM ID

1.2 Khai báo có khởi tạo
variables: all_types variables_init CM;
variables_init: ID CM variables_init CM expression | ID ASSIGNINIT expression;
Giải thích:
Các biến vừa được khai báo, vừa được khởi tạo giá trị.
Lưu ý quan trọng: Số lượng biến và số lượng giá trị khởi tạo phải đối xứng nhau.

Ví dụ:
int a, b, c = 1, 2, 3; // Khai báo và khởi tạo a = 1, b = 2, c = 3
Cách parse:
variables → all_types variables_init CM
variables_init → ID CM variables_init CM expression

1.3 Các loại khai báo đặc biệt
implicit_var: VAR ID ASSIGNINIT expression;
keyword_var: all_type ID (ASSIGNINIT expression)?;
implicit_dynamic: DYNAMIC ID (ASSIGNINIT expression)?;
Giải thích:
Một số ngôn ngữ có cú pháp khai báo đặc biệt:
VAR để khai báo biến với kiểu suy diễn tự động.
DYNAMIC để khai báo biến kiểu động.
ASSIGNINIT là toán tử gán (ví dụ: = hoặc <-).
Biến có thể được khởi tạo hoặc không.
Ví dụ:
var x <- 10; // Khai báo kiểu VAR
dynamic y; // Khai báo kiểu DYNAMIC
string z <- "hello"; // Khai báo kiểu STRING

1.4. Khai báo mảng
atomic_type: INTEGER | FLOAT | BOOLEAN | STRING;
array_type: atomic_type LSB dimensions RSB;
dimensions: INT_LIT CM dimensions | INT_LIT;
Giải thích:
Mảng được định nghĩa với kiểu dữ liệu cơ bản (atomic_type) và danh sách kích thước (dimensions).
Kích thước mảng (dimensions) không được rỗng và được ngăn cách bởi dấu phẩy ,.
Ví dụ:
float arr[1, 2, 3]; // Mảng 3 chiều kiểu float
Cách parse:
array_type → atomic_type LSB dimensions RSB
dimensions → INT_LIT CM dimensions → INT_LIT CM INT_LIT CM INT_LIT

2. Function - Khai báo hàm
function: FUNC ID LP (parameters_list)? RP (return_statement | block_statement |);
parameters_list: parameter CM parameters_list | parameter;
parameter: primitive_decl | array_decl;
Giải thích:
Hàm được định nghĩa với tên (ID), danh sách tham số (có hoặc không), và phần thân (return_statement hoặc block_statement).
Danh sách tham số (parameters_list): Có thể là:
Khai báo kiểu cơ bản (primitive_decl).
Khai báo kiểu mảng (array_decl).
Ví dụ:
func sum(int a, int b) return a + b; // Hàm tính tổng
func process(float arr[10]) begin end // Hàm xử lý mảng
Cách parse:
function → FUNC ID LP parameters_list RP return_statement
parameters_list → parameter CM parameter → primitive_decl CM primitive_decl

3. Statements - Các loại câu lệnh
3.1. Khai báo biến
Khai báo biến tương tự phần variables ở trên.
3.2. Gán giá trị
Cú pháp:
id = expression; // Biến scalar
array[expression, ...] = value; // Mảng nhiều chiều
Không cho phép:
Gán vào kết quả của biểu thức hoặc hàm.
Ví dụ không hợp lệ:
id + 1 = 2;
fun() = 1;
3.3. If Statement
if (expression) <statement>
[elif (expression) <statement>]*
[else <statement>]?
Giải thích:
Câu lệnh if có thể đi kèm nhiều elif và một else.
Chỉ if là bắt buộc.
Ví dụ:
if (x > 0) return 1;
elif (x < 0) return -1;
else return 0;
3.4. For Loop
for <number-variable> until <condition-expression> by <update-expression> <statement>
Giải thích:
Vòng lặp for thực thi khi điều kiện until còn đúng.
Biến đếm phải là biến scalar.
Ví dụ:
for i until i >= 10 by i + 1 return i;
3.5. While Loop
while (<expression>) <statement>
Giải thích:
Vòng lặp thực thi khi expression là true.
Ví dụ:
while (x > 0) x = x - 1;
3.6. Do-While Loop
do <block-statement> while (<expression>);
Giải thích:
Thực thi ít nhất một lần trước khi kiểm tra điều kiện.
Ví dụ:

do begin print(x); x = x - 1; end while (x > 0);
3.7. Các loại khác
Break và Continue: Dừng hoặc bỏ qua vòng lặp hiện tại.
Return: Có thể có hoặc không có giá trị trả về.
Function Call: Gọi hàm với danh sách tham số hoặc rỗng.
Block Statement: Một khối lệnh được bao quanh bởi begin và end.
Ví dụ:
begin
 var a <- 10;
 return a + 1;
end

```
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 thằng gọi hàm trong phần *expression*

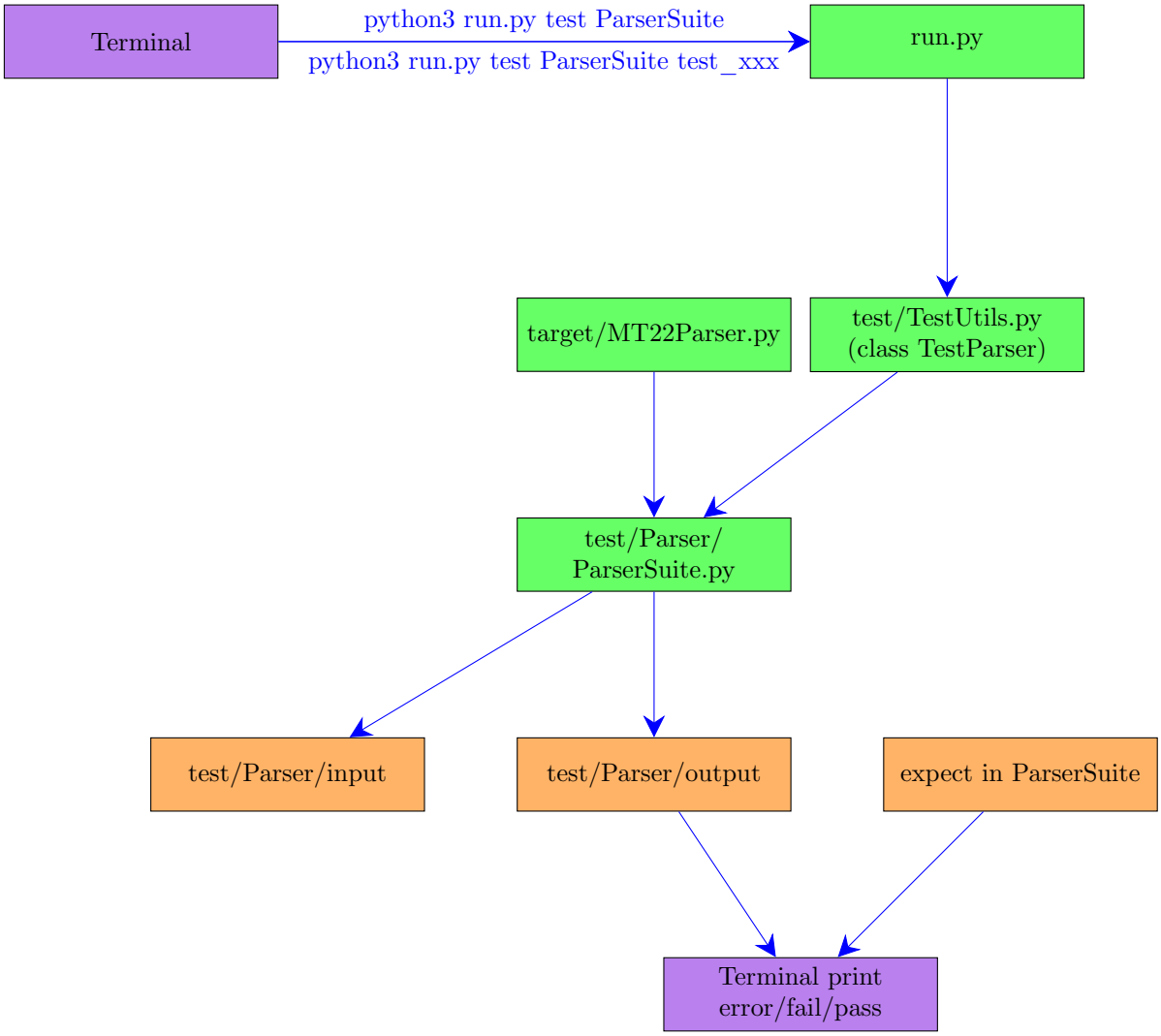
```
fun();
fun(1+2, 1);
```

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 trương 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