Cú pháp chương trình

Cú pháp và ngữ nghĩa

- Cú pháp của ngôn ngữ lập trình: cấu trúc hình thức của chương trình như thế nào?
  - Cú pháp được định nghĩa bằng một văn phạm hình thức
- Ngữ nghĩa của ngôn ngữ lập trình: chương trình làm gì, hoạt động như thế nào?
  - Ngữ nghĩa chương trình khó định nghĩa hơn cú pháp.

Nội dung

- Văn phạm và ví dụ về cây cú pháp
- BNF và định nghĩa cây cú pháp
- Xây dựng văn phạm
- Cấu trúc câu và từ vựng
- Các dạng văn phạm khác

Văn phạm tiếng Anh

Một câu là một cấu trúc: danh ngữ + động từ + danh ngữ.	<*S*> ::= <*NP*> <*V*> <*NP*>
Một danh ngữ là ...	<*NP*> ::= <*A*> <*N*>
Một động từ là …	<*V*> ::= **`loves`** | **`hates`** | **`eats`**
Một mạo từ là …	<*A*> ::= **`a`** | **`the`**
Một danh từ là ...	<*N*> ::= **`dog`** | **`cat`** | **`rat`**

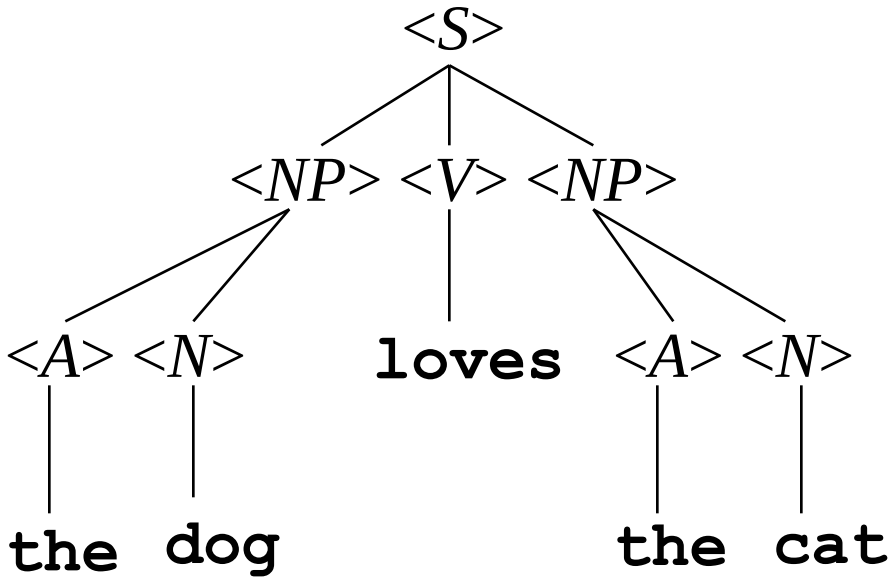
Văn phạm hoạt động như thế nào

- Văn phạm là một tập hợp các qui tắc chỉ ra cách thức xác định một *cây cú pháp*
- Cho <*S*> là gốc của cây
- Văn phạm xác định cách thức mà các nút con có thể được thêm vào cây
- Như vậy, qui tắc

  <*S*> ::= <*NP*> <*V*> <*NP*>

  cho biết có thể thêm <*NP*>, <*V*>, và <*NP*>, theo trật tự trên làm các nút con của <*S*>

Cây cú pháp



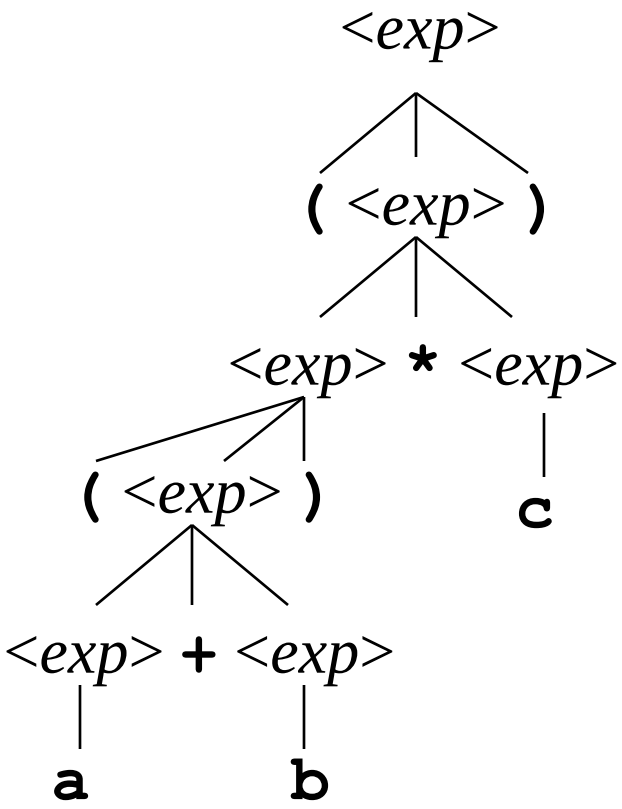
Văn phạm ngôn ngữ lập trình

<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**

- Một biểu thức có thể là tổng của hai biểu thức, hoặc tích của hai biểu thức, hoặc một biểu thức trong cặp ngoặc đơn
- Hoặc có thể là một trong các biến **a**, **b** hay **c**

Cây cú pháp

((a+b)*c)



Tổng quan

- Văn phạm và ví dụ cây cú pháp
- BNF và định nghĩa cây cú pháp
- Xây dựng văn phạm
- Cấu trúc câu và từ vựng
- Các dạng văn phạm khác

CFG – Context Free Grammars (1950)

- Được phát triển bởi Noam Chomsky giữa thập niên 1950
- Language generators : miêu tả cú pháp của ngôn ngữ tự nhiên
- CFG là một văn phạm hình thức mà trong đó các luật là ở dạng $V \rightarrow w$

- A CFG consists of the following components:
- a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.
- a set of *nonterminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
- a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
- a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

- **A CFG for Arithmetic Expressions**
- An example grammar that generates strings representing arithmetic expressions with the four operators +, -, *, /, and numbers as operands is:

- <expression> --> number
- <expression> --> (<expression>)
- <expression> --> <expression> + <expression>
- <expression> --> <expression> - <expression>
- <expression> --> <expression> * <expression>
- <expression> --> <expression> / <expression>

- The only nonterminal symbol in this grammar is <expression>, which is also the start symbol. The terminal symbols are {+,-,*,/,(,),number}.

BNF - Backus Normal Form (1959)

- Được tạo ra bởi John Backus để miêu tả cho Algol 58
- BNF tương đương với CFG (context-free grammars)
- Meta language là một dạng ngôn ngữ dùng để miêu tả cho một ngôn ngữ khác

Ví dụ :mô tả địa chỉ bưu điện ở mỹ :

<postal-addr> ::= <name-part> <street-addr> <zip-part>

(Một địa chỉ gồm 3 phần :Tên, địa chỉ và mã vùng)

<name-part> ::=

<personal-part> <name-part> |

<personal-part> <last-name> <suffix-part>

(Tên gồm 2 phần :phần riêng ,tên hoặc phần riêng, tên và xưng hô)

<personal-part> ::= <first-name> | <initial> "."

(Phần riêng :tên hoặc ký tự đầu)

<street-addr> ::= <house-num> <street-name>

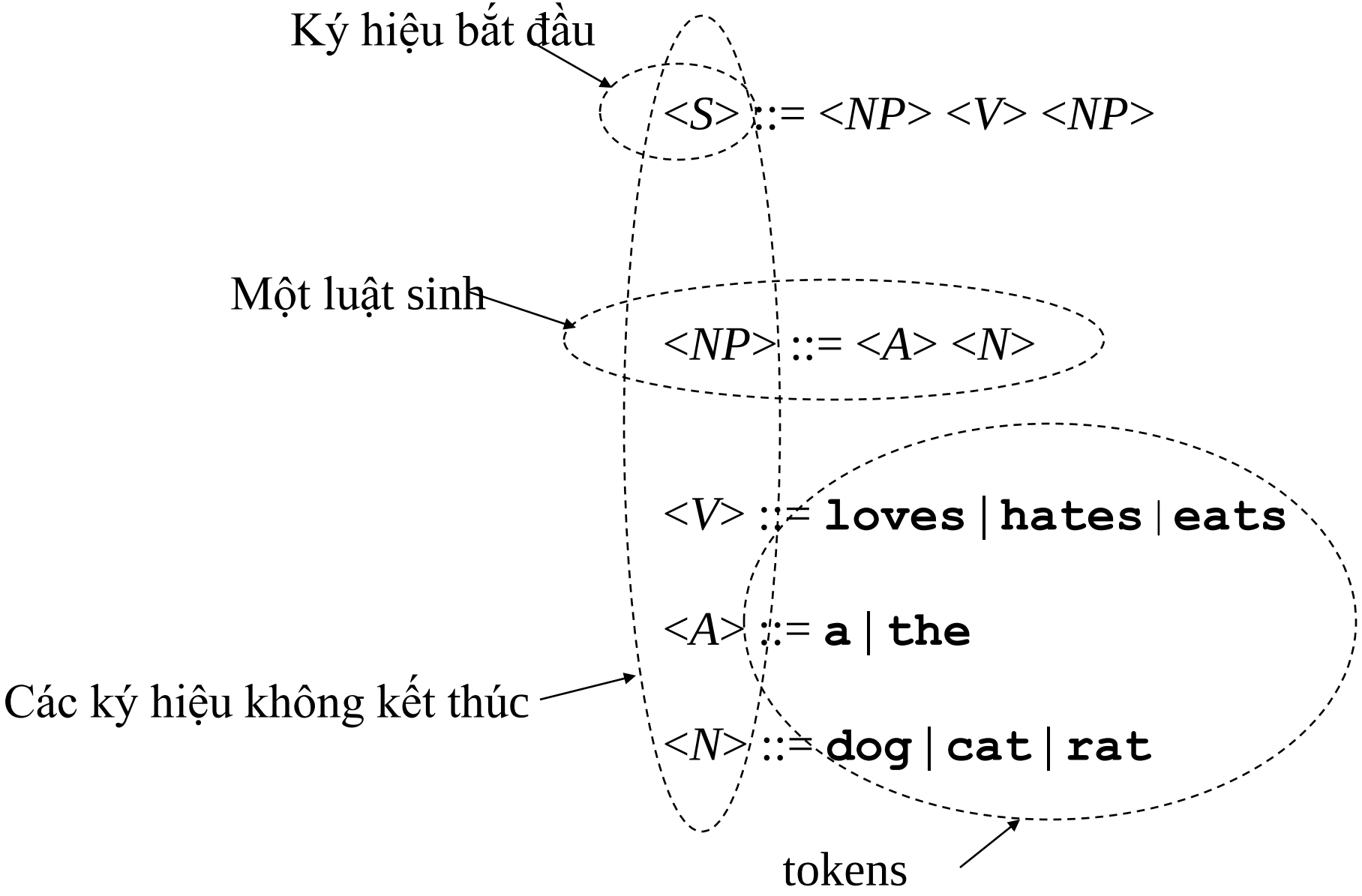
(Địa chỉ :số nhà và tên đường)

<zip-part> ::=

<town-name> "," <state-code> <ZIP-code>

(Zip-part :tên thành phố,state-code, zipcode

<suffix-part> ::= "Sr." | "Jr." | <roman-numeral>


Ký hiệu bắt đầu
<S> ::= <NP> <V> <NP>
Một luật sinh
<NP> ::= <A> <N>
<V> ::= loves | hates | eats
<A> ::= a | the
Các ký hiệu không kết thúc
<N> ::= dog | cat | rat
tokens

Định nghĩa văn phạm BNF

- Một văn phạm BNF gồm có 4 phần:
  - Mộ tập hợp các *tokens*
  - Một tập hợp các *ký hiệu không kết thúc*
  - Một *ký hiệu bắt đầu*
  - Một tập các *luật sinh (productions)*

Định nghĩa văn phạm BNF

- Các *tokens* là những đơn vị nhỏ nhất của cú pháp:
  - Token có thể là một chuỗi gồm một hay nhiều ký tự
  - Token là nguyên tố: không phân tách nhỏ hơn

- Các *ký hiệu không kết thúc*:
  - Các ký hiệu không kết thúc là các chuỗi trong cặp ngoặc <>, chẳng hạn <*NP*>
  - Văn phạm xác định cách thức mở rộng các ký hiệu không kết thúc thành các token

Định nghĩa văn phạm BNF

- *Ký hiệu bắt đầu* là một ký hiệu không kết thúc đặc biệt, dùng làm nút gốc của cây phân tích

- Các *luật sinh là các luật xây dựng cây cú pháp*

Định nghĩa văn phạm BNF

- Mỗi luật sinh có một phần bên trái, một bộ tách `::=`, và một phần bên phải
  - Phần bên trái là một đơn ký hiệu không kết thúc
  - Phần bên phải là một dãy, mỗi thành phần của dãy này có thể là một token hay một ký hiệu không kết thúc

- Mỗi luật sinh xác định một khả năng để xây dựng cây cú pháp.

Định nghĩa văn phạm BNF

- Khi có nhiều hơn một luật sinh có cùng một phần bên trái giống nhau thì có thể dùng dạng thu gọn

- Văn phạm BNF có thể đặt phần bên trái, bộ tách **::=**, và một danh sách các phần bên phải – mỗi thành phần trong danh sách đó cách nhau bởi ký hiệu |

Ví dụ

<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**

Chú ý rằng có 6 luật sinh trong văn phạm trên
Văn phạm trên tương đương với:

<*exp*> ::= <*exp*> **+** <*exp*>
<*exp*> ::= <*exp*> ***** <*exp*>
<*exp*> ::= **(** <*exp*> **)**
<*exp*> ::= **a**
<*exp*> ::= **b**
<*exp*> ::= **c**

Empty

- Ký hiệu <*empty*> được dùng để chỉ ra là văn phạm không sản sinh bất cứ gì ở vị trí nó xuất hiện
- Ví dụ, văn phạm sau định nghĩa một cấu trúc if-then với một phần else tùy chọn:

<*if-stmt*> **::=** **if** <*expr*> **then** <*stmt*> <*else-part*>
<*else-part*> **::=** **else** <*stmt*> | <*empty*>

Cây cú pháp

- Để xây dựng một cây cú pháp, thiết lập ký hiệu bắt đầu ở nút gốc
- Thêm các ký hiệu không kết thúc vào, *dựa trên các luật sinh trong văn phạm*
- Kết thúc khi tất cả các nút lá là token
- Đọc các nút lá từ trái sang phải, đây chính là chuỗi phát sinh từ cây cú pháp

Ứng dụng

<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**

Vẽ cây phân tích cú pháp cho các chuỗi sau:

a+b
a*b+c
(a+b)
(a+(b))

Định nghĩa ngôn ngữ

- Văn phạm được dùng để định nghĩa cú pháp của các ngôn ngữ lập trình

- Ngôn ngữ được định nghĩa bằng một văn phạm là một tập các chuỗi có thể phát sinh từ một cây cú pháp nào đó của văn phạm

Nội dung

- Văn phạm và ví dụ cây cú pháp
- BNF và định nghĩa cây cú pháp
- Xây dựng văn phạm
- Cấu trúc câu và từ vựng
- Các dạng văn phạm khác

Xây dựng văn phạm

- Ví dụ: các khai báo trong ngôn ngữ Java: tên kiểu, danh sách các biến cách nhau bởi dấu phẩy, và một dấu chấm phẩy
- Mỗi biến có thể được theo sau bởi một bộ khởi tạo:

```
float a;
boolean a,b,c;
int a=1, b, c=1+2;
```

Ví dụ (tiếp theo)

- Nếu chưa kể đến bộ khởi tạo:

<var-dec> **::=** *<type-name>* *<declarator-list>* **;**

- Các kiểu dữ liệu cơ bản:

<type-name> **::= boolean** | **byte** | **short** | **int**
| **long** | **char** | **float** | **double**

<declarator-list> **::=** *<declarator>*
| *<declarator>* **,** *<declarator-list>*

<declarator> **::=** *<variable-name>*
| *<variable-name>* **=** *<expr>*

Nội dung

- Văn phạm và ví dụ cây cú pháp
- BNF và định nghĩa cây cú pháp
- Xây dựng văn phạm
- Cấu trúc câu và từ vựng
- Các dạng văn phạm khác

Token

- Các token là những mẫu văn bản trong chương trình mà chúng không thể được chia nhỏ hơn nữa
- Định danh (**`count`**), từ khóa (**`if`**), toán tử (**`==`**), hằng số (**`123.4`**), etc.
- Chương trình máy tính là một dãy các ký tự
- Làm thế nào một dãy các ký tự có thể phân tích thành các token?

Cấu trúc từ vựng và cấu trúc câu

- Văn phạm định nghĩa *cấu trúc từ vựng*: một tập tin văn bản được phân tách thành các token như thế nào?

- Văn phạm định nghĩa *cấu trúc câu*: chương trình được xây dựng như thế nào từ một chuỗi các token ?

Một văn phạm cho cấu trúc từ vựng và cấu trúc câu

- Theo quan điểm này, sử dụng một văn phạm chung cho cấu trúc câu và cấu trúc từ
- Không hiệu quả trong thực tế: các khoảng trắng hay các ghi chú làm cho văn phạm khó đọc và rườm rà

<*if-stmt*> ::= **`if`** <*white-space*> <*expr*> <*white-space*>
`then` <*white-space*>
<*stmt*> <*white-space*> <*else-part*>
<*else-part*> ::= **`else`** <*white-space*> <*stmt*> | <*empty*>

Hai văn phạm tách biệt

- Thông thường, các ngôn ngữ sử dụng hai văn phạm tách biệt
  - Một văn phạm chỉ ra cách thức cấu trúc các token từ một văn bản các ký tự
  - Một văn phạm chỉ ra cách xây dựng cây cú pháp từ một dãy các token

<program-file> ::= *<end-of-file>* | *<element>* *<program-file>*
<element> ::= *<token>* | *<one-white-space>* | *<comment>*
<one-white-space> ::= *<space>* | *<tab>* | *<end-of-line>*
<token> ::= *<identifier>* | *<operator>* | *<constant>* | …

Lưu ý

- Các ngôn ngữ lập trình trước đây thường không tách bạch cấu trúc từ vựng với cấu trúc câu
  - Trong các thế hệ Fortran và Algol đầu tiên, khoảng trắng có thể ở bất kỳ vị trí nào, ngay cả ở giữa một từ khóa
  - Các ngôn ngữ khác như PL/I cho phép các từ khóa được dùng làm các định danh

Lưu ý

- Một số ngôn ngữ có định dạng cố định cho cấu trúc từ vựng – theo các vị trí cột
  - Mỗi dòng một phát biểu
  - Một vài cột đầu tiên dành cho nhãn của phát biểu
- Ví dụ: một vài thế hệ đầu của Fortran, Cobol, và Basic
- Hầu hết các ngôn ngữ lập trình hiện nay có định dạng tự do cho cấu trúc từ vựng

Nội dung

- Văn phạm và ví dụ cây cú pháp
- BNF và định nghĩa cây cú pháp
- Xây dựng văn phạm
- Cấu trúc câu và từ vựng
- Các dạng văn phạm khác

Các dạng văn phạm khác

- Các biến thể của BNF
- Các biến thể của EBNF
- Các sơ đồ cú pháp

Các biến thể của BNF

- Một số dùng $\rightarrow$ hay = thay vì ::=
- Một số cho phép dùng cặp dấu nháy đơn bao các tokens, ví dụ phân biệt ‘ | ’ như là một token với | như là meta-symbol

Các biến thể của BNF

- Cú pháp bổ sung cho BNF:
  - {x} nghĩa là không hay lặp lại x nhiều lần
  - [x] có nghĩa x là tùy chọn (i.e. x | <*empty*>)
  - () cho nhóm
  - | chọn một trong số các chọn lựa
  - Cặp dấu ngoặc kép bao ký hiệu để phân biệt với các meta-symbols

EBNF Examples

<*if-stmt*> **::=** **if** <*expr*> **then** <*stmt*> [**else** <*stmt*>]

<*stmt-list*> **::=** {<*stmt*> **;**}

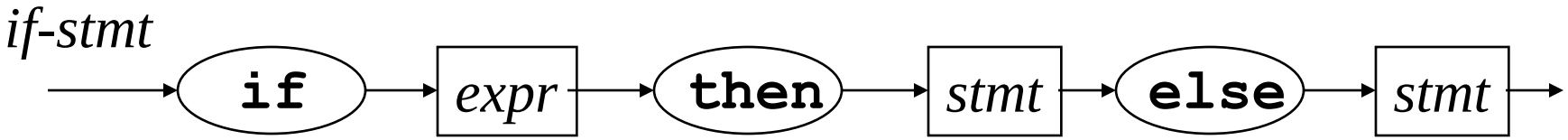
<*thing-list*> **::=** { (<*stmt*> | <*declaration*>) **;**}

- Các mở rộng của BNF được gọi là Extended BNF: EBNF
- Có nhiều biến thể của BNF

Các sơ đồ cú pháp

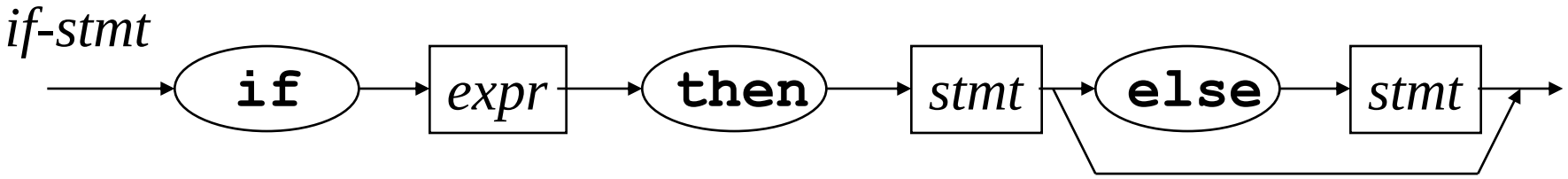
- Văn phạm EBNF
- Một luật sinh đơn giản là một chuỗi các ký hiệu kết thúc và không kết thúc:

<*if-stmt*> **::=** **if** <*expr*> **then** <*stmt*> **else** <*stmt*>



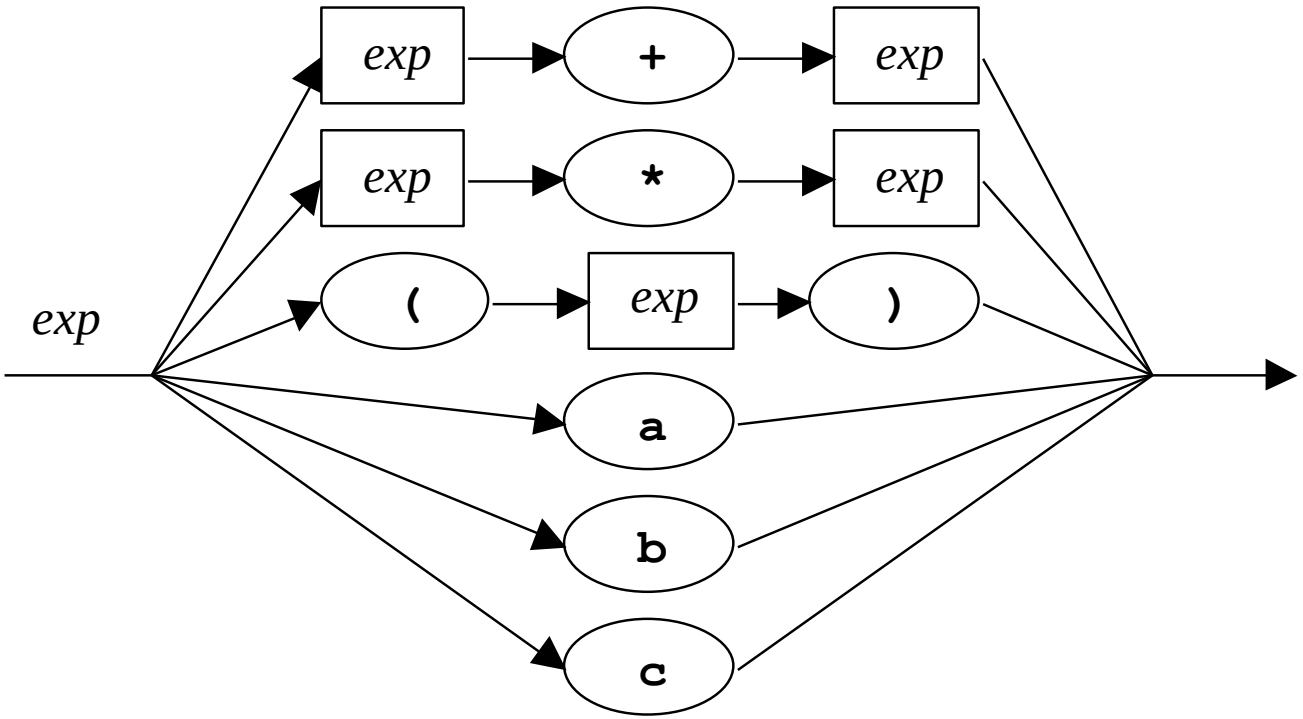
Đường vòng

<*if-stmt*> **::=** **if** <*expr*> **then** <*stmt*> [**else** <*stmt*>]



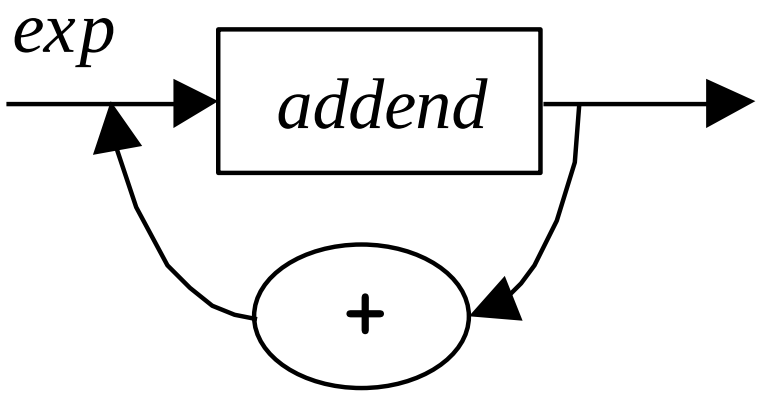
Rẽ nhánh

<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**



Vòng lặp

<*exp*> **::=** <*addend*> { **+** <*addend*> }



Ví dụ

WhileStatement:

while (*Expression*) *Statement*

DoStatement:

do *Statement* while (*Expression*) ;

ForStatement:

for ($ForInit_{opt}$; $Expression_{opt}$; $ForUpdate_{opt}$)

Statement

Cho luật sinh sau

<assign> → <id> = <expr>

<id> → A | B | C

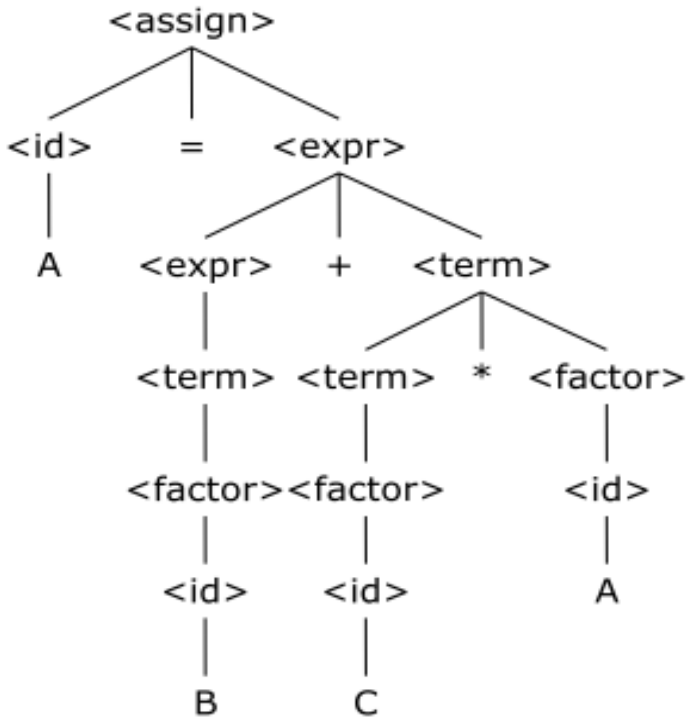
<expr> → <expr> + <term> | <term>

<term> → <term> * <factor> | <factor>

<factor> → (<expr>) | <id>

Example: Derivation and parser tree of sentence A = B + C * A

<assign>
⇒ <id> = <expr>
⇒ A = <expr>
⇒ A = <expr> + <term>
⇒ A = <term> + <term>
⇒ A = <factor> + <term>
⇒ A = <id> + <term>
⇒ A = B + <term>
⇒ A = B + <term> * <factor>
⇒ A = B + <factor> * <factor>
⇒ A = B + <id> * <factor>
⇒ A = B + C * <factor>
⇒ A = B + C * <id>
⇒ A = B + C * A



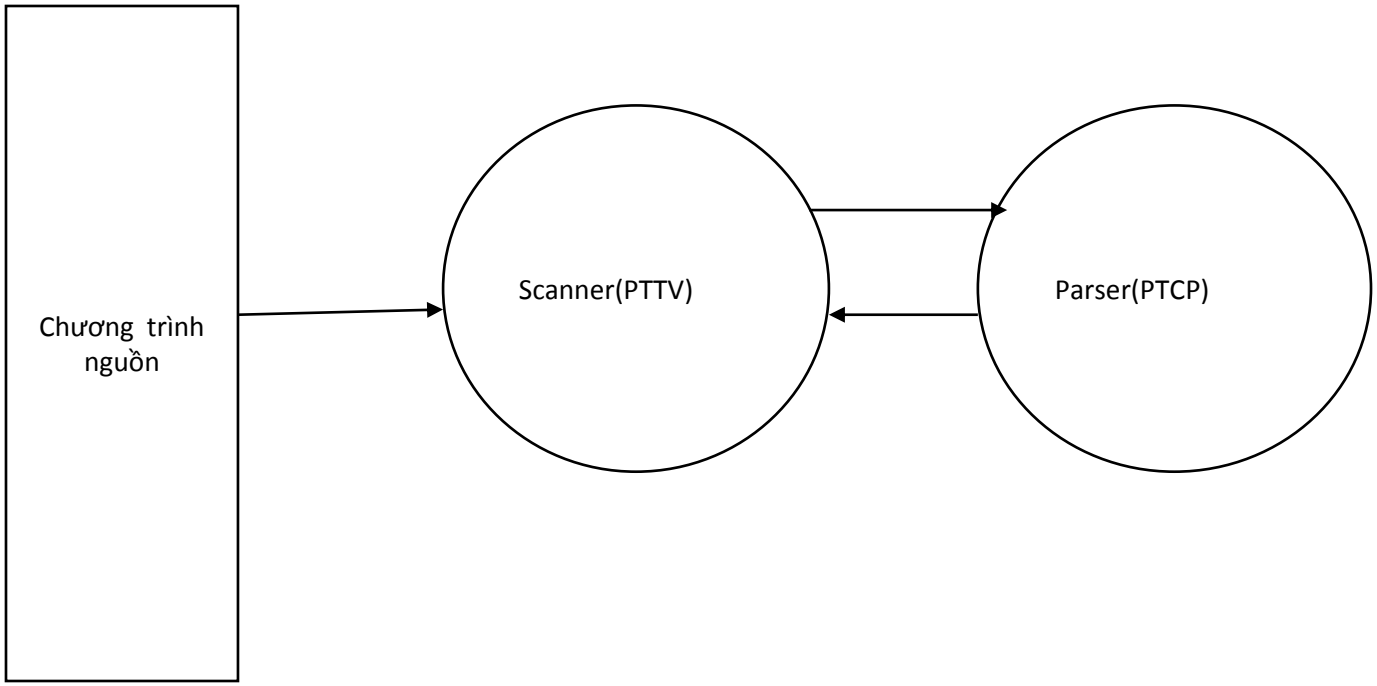
1-35

- **Ba cách xây dựng bộ phân tích từ vựng (Scanner)**
  - Dùng phần mềm phát sinh tự động(đầu vào là một văn phạm chính quy)
  - Xây dựng sơ đồ trạng thái DFA
  - Xây dựng bảng thực thi dòng, cột
- Luật tìm lexeme dài nhất
  - Scanner chỉ trả về token khi đọc đến ký tự kết thúc token(thường là khoàng trắng)
  - Trong một số trường hợp Scanner phải đọc nhiều hơn một ký tự để nhận biết đã kết thúc token

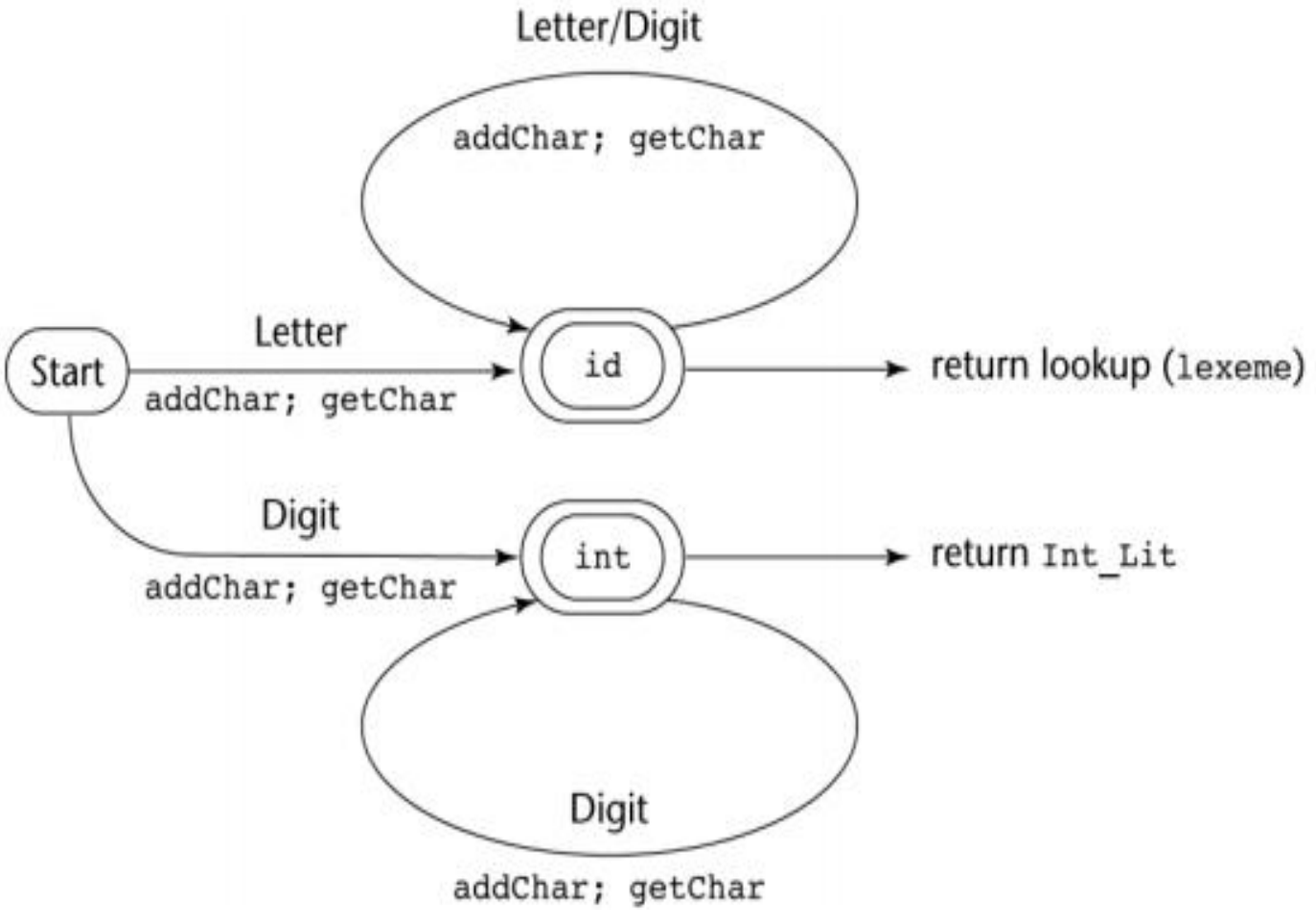
Example

`sum = oldsum - value / 100;`

Token	*Lexeme*
IDENT	`sum`
ASSIGN_OP	`=`
IDENT	`oldsum`
SUBSTRACT_OP	`-`
IDENT	`value`
DIVISION_OP	`/`
INT_LIT	`100`
SEMICOLON	`;`


Chương trình nguồn
Scanner(PTTV)
Parser(PTCP)

State Diagram



1-15

Lexical Analysis – Implementation

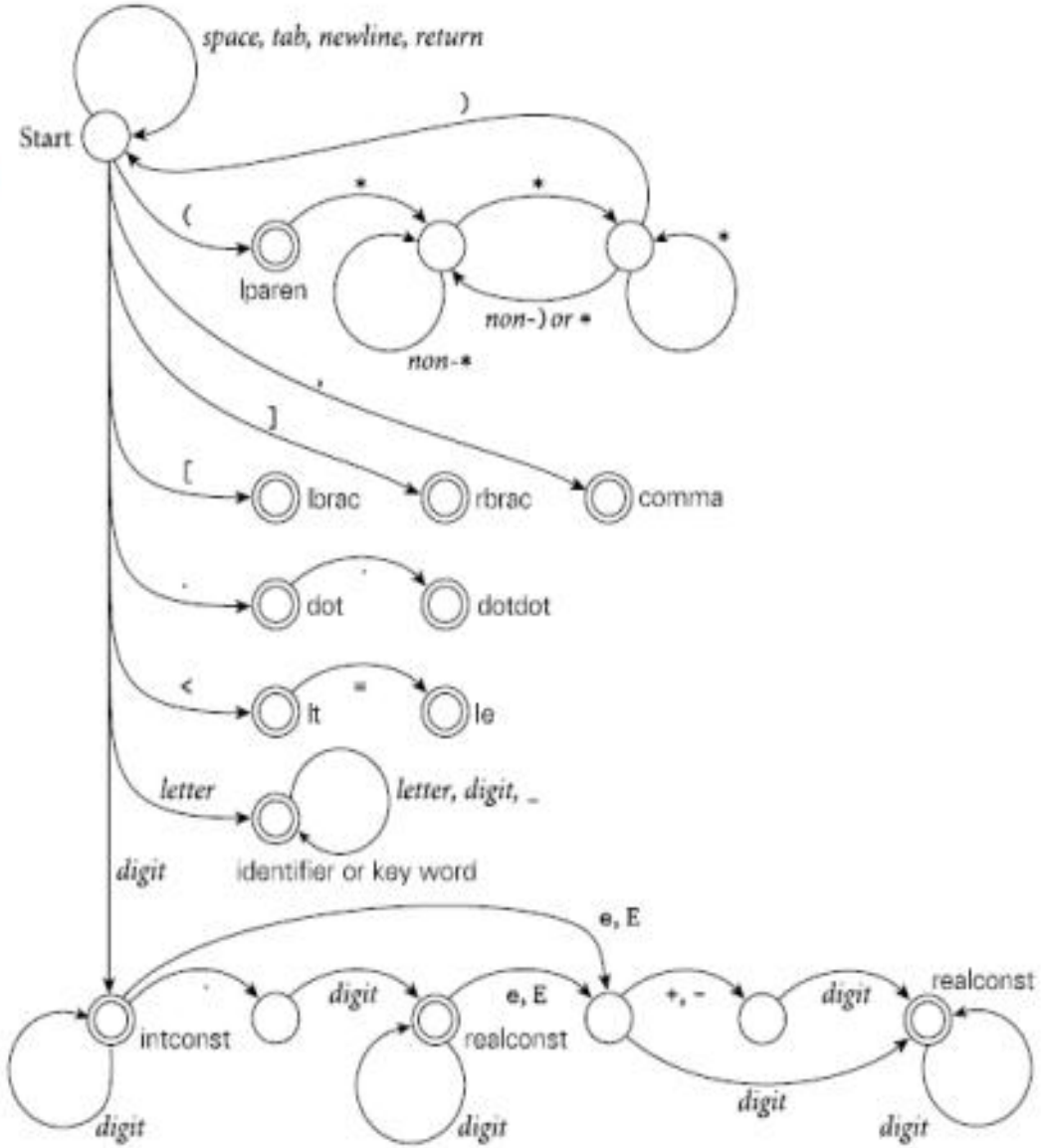
```
int lex() {
  getChar();
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT) {
        addChar();
        getChar();
      }
      return lookup(lexeme);
      break;
      ...
```


1-16

Lexical Analysis – Implementation

```
        case DIGIT:
            addChar();
            getChar();
            while (charClass == DIGIT) {
                addChar();
                getChar();
            }
            return INT_LIT;
    }  /* End of switch */
}  /* End of function lex() */
```

State Diagram (Pascal)



- **Phân tích cú pháp**

I.Nhiệm vụ

- Tìm tất cả các lỗi cú pháp , đưa ra các thông báo lỗi
- Xây dựng cây phân tích cú pháp (lưu vết) cho chương trình

II.Có hai kỹ thuật phân tích

- Top-down : xây dựng cây phân tích cú pháp bắt đầu từ gốc
- Bottom up : xây dựng cây phân tích cú pháp bắt đầu từ lá

Top-down and bottom-up parsing of the input string

A, B, C;