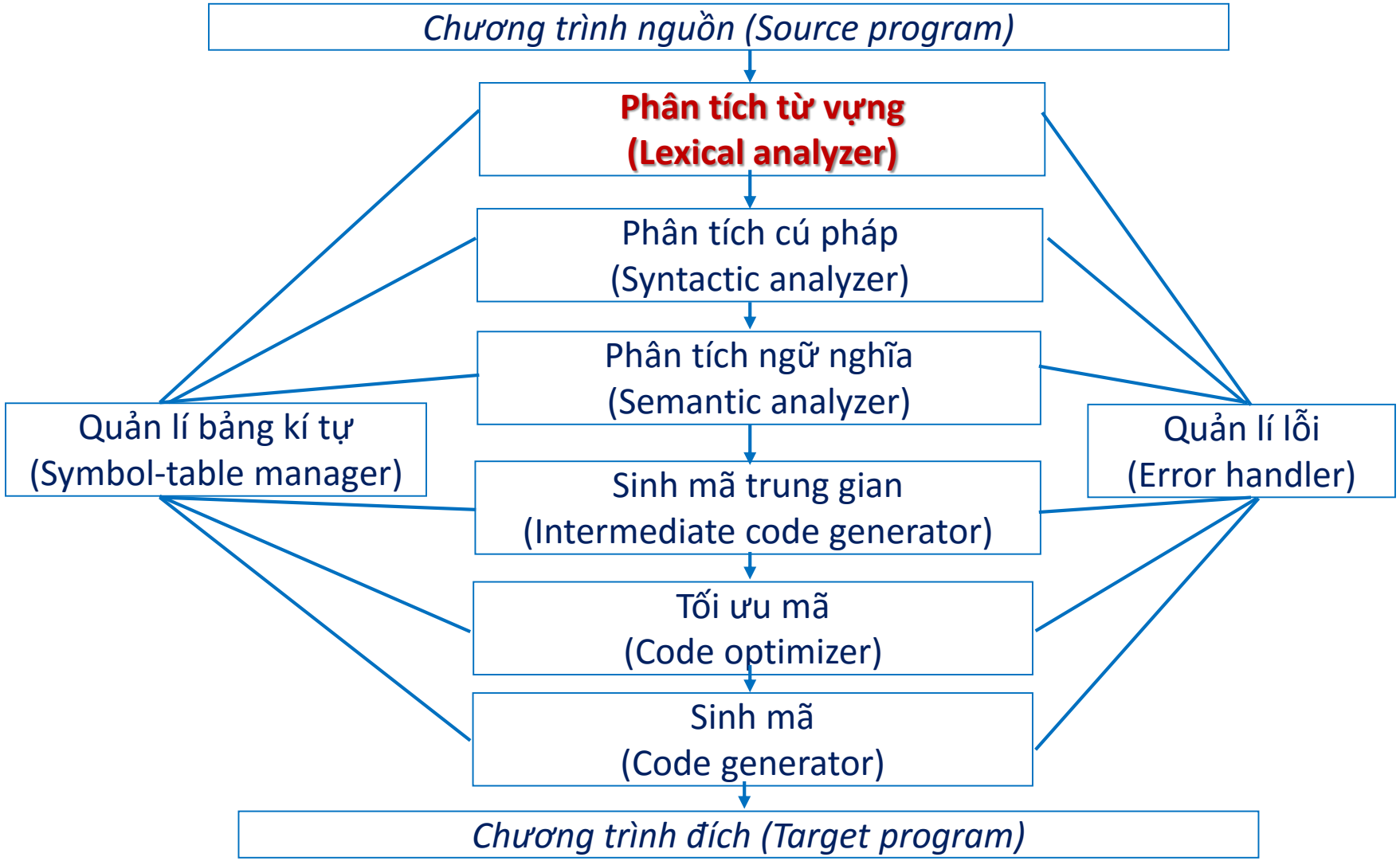
Lecture 6. Lexical Analysis (Phân tích từ vựng)

Hà Chí Trung, BM KHMT, KCNTT, HVKTQS

hct2009@yahoo.com

01685-582-102

Lexical analysis



Lexical analysis

- **LA** là giai đoạn đầu tiên của quá trình dịch, giúp cho các giai đoạn biên dịch tiếp theo dễ dàng hơn;
- Nhiệm vụ chính: phân tích chương trình nguồn thành các **token**:
  - Đọc từng ký tự một;
  - Loại bỏ các ký tự vô nghĩa (*dòng trắng, space, chú thích...*)
  - Xác định các từ tố (*token*) và thông tin thuộc tính của chúng;
  - Chuyển thông tin của các từ tố cho bộ parser (***SA***) và bảng quản lý ký hiệu (*symbol-table*)
  - Phát hiện các lỗi cấp độ từ vựng.

Lexical analysis

- **Lexeme**: Một nhóm các ký tự kề nhau có thể tuân theo một quy ước (mẫu hay luật) nào đó và tạo thành một từ vị.
- **Pattern**: Pattern là các qui tắc kết hợp các kí tự để miêu tả một nhóm từ vị nào đó. Pattern được biểu diễn dưới dạng RE.
- **String**: Là một chuỗi các kí tự từ một bảng chữ cái nào đó
- **Language**: Là tập hợp các string được xây dựng từ một bảng chữ cái cho trước.
- **Token**: Một token là một tập hợp các lexemes mang một nghĩa chung xác định.

Lexical analysis

- Các **tokens** khác nhau có các luật mô tả khác nhau. **Token** được mô tả bằng lời (**?**) bằng mẫu (**pattern**), hoặc các luật dưới dạng **CFG** (**BNF**) hoặc sơ đồ chuyển (**FA)**. Ví dụ:
  - Các từ khoá (keywords);
  - định danh (identifiers);
  - toán tử (operators);
  - hằng số (consts);
  - xâu kí tự (strings);
  - dấu phân cách - separator(ngoặc đơn, dấu phẩy, chấm phẩy...)
  - ...

Lexical analysis

Token	Lexeme	Thông tin mô tả của pattern
const	const	const
if	if	if
relop	<, <=, >, >=, =, <>	< hoặc <= hoặc > hoặc >= hoặc = hoặc <>
id	pi, count, d2	một ký tự, tiếp theo là các ký tự hoặc chữ số
num	3.1416, 0, 6.02E2	bất kì hằng số nào
literal	"computer”	các kí tự nằm giữa " và “, ngoại trừ “
….		

Định nghĩa chính quy

- **Regular definition (RD)**: Một định nghĩa chính quy (**RD**) là một dãy quy tắc có dạng

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

.....

$$d_n \rightarrow r_n$$

Trong đó d_i là các tên, r_i là các **RE** trên tập các kí hiệu $\Sigma \cup \{d_1, d_2, ...d_{i-1}\}$

- Như vậy, **RD** thực chất là các luật của văn phạm **CFG** (ở dang **EBNF**).

Định nghĩa chính quy

- **VD: RD** của các định danh (identifiers) trong pascal là

  letter $\rightarrow$ A | B | ...| Z | a | b |...| z

  digit $\rightarrow$ 0 | 1 | ...| 9

  identifier $\rightarrow$ letter (letter | digit)*

- **VD: RD** của các số không dấu trong pascal: 3254, 23.243E5, 16.264E-3...

  digit $\rightarrow$ 0 | 1 |...| 9

  digits $\rightarrow$ digit digit*

  optional_fraction $\rightarrow$. digits | ε

  optional_exponent $\rightarrow$ (E (+ | - | ε) digits) | ε

  num $\rightarrow$ digits optional_fraction optional_exponent

Định nghĩa chính quy

- **VD:** if, then, else, relop, id sinh ra tập các xâu kí tự theo các RD sau:

  if $\rightarrow$ if

  then $\rightarrow$ then

  else $\rightarrow$ else

  relop $\rightarrow$ < | <= | = | <> | > | >=

- **VD**: Các kí tự khoảng trắng (loại bỏ khi phân tích từ vựng) được định nghĩa bởi RD sau:

  delim $\rightarrow$ **blank** | **tab** | **newline**

  ws $\rightarrow$ **delim**$^{+}$

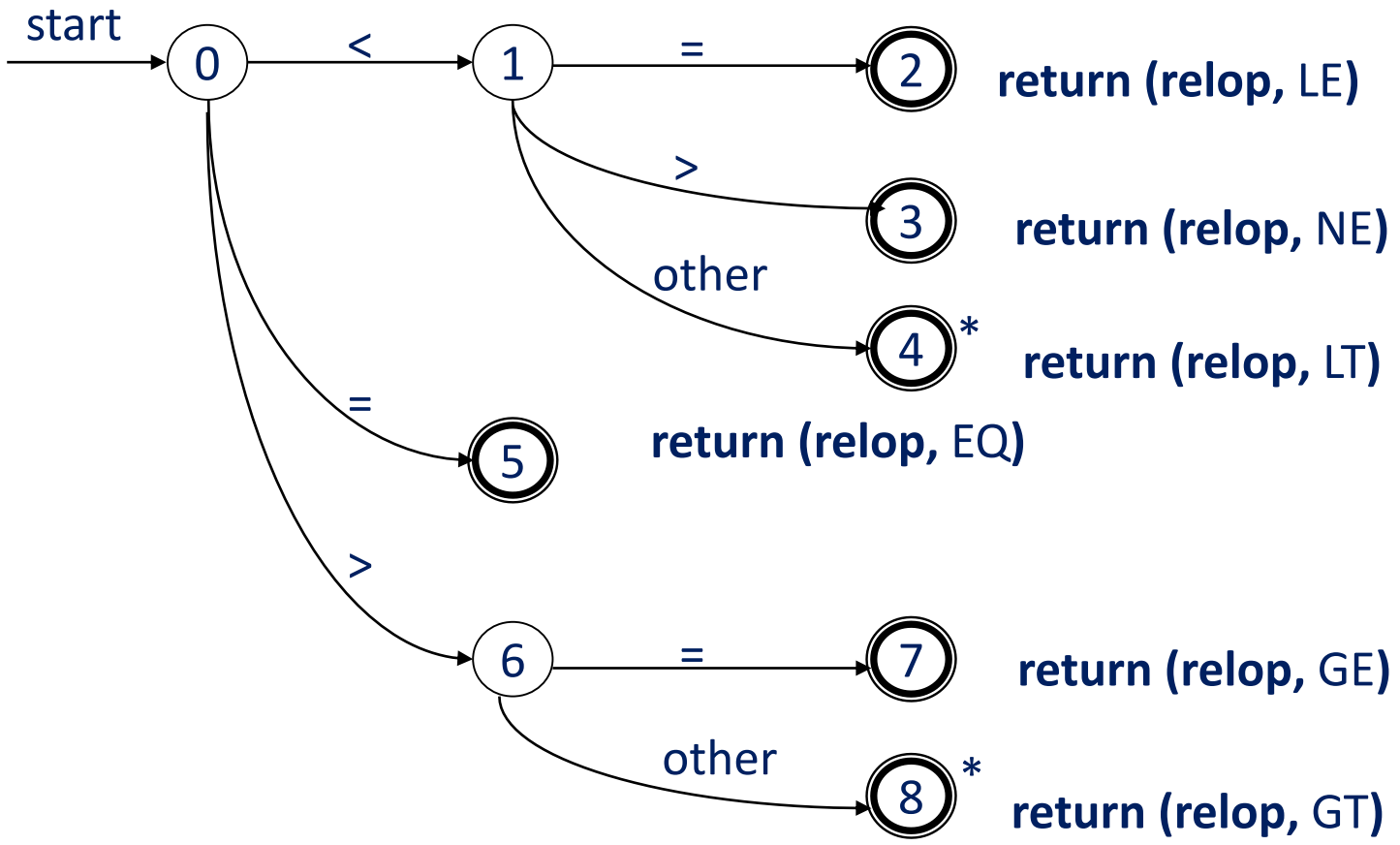
Bảng ký tự (Symbol-table)

- Kết quả của lexical analyzer tạo ra output là các cặp:
- *<token, attribute-value>*
- Các cặp này được lưu trữ, quản lý trong bảng ký tự.

RE	Token	Attribute-value
ws	-	-
if	**if**	-
for	**for**	-
else	**else**	-
i	**id**	pointer to table entry
num	**num**	pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

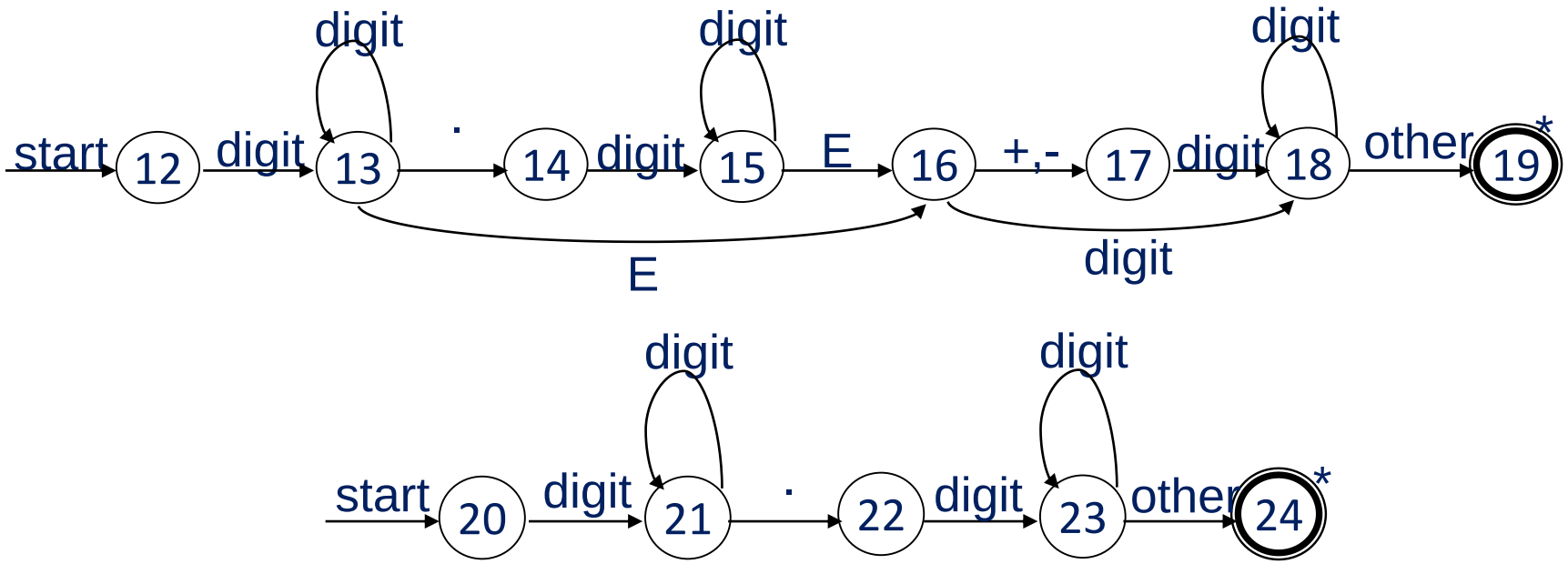
Nhận dạng token bằng DFA

- **DFA** đoán nhận token các toán tử quan hệ (relop - Pascal)



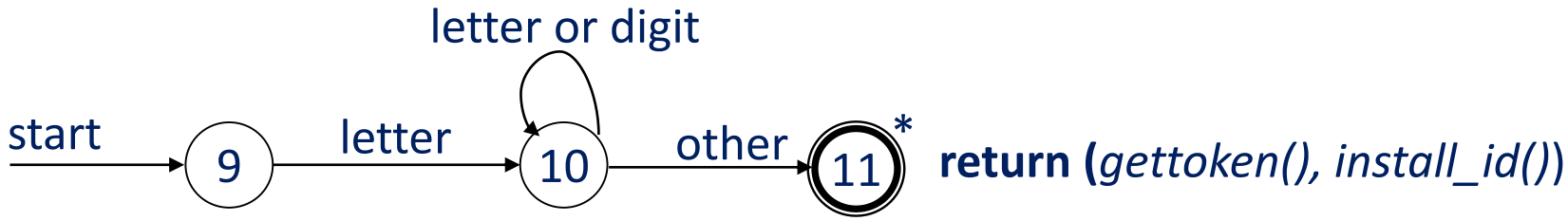
Nhận dạng token bằng DFA

- **VD 2.9**: FA đoán nhận các unsigned numbers trong pascal:
  - sốthựcmũ → (chữsố)$^+$ [.(chữsố)$^+$] ? [E [+|-] ? (chữsố)$^+$] ?
  - sốthực → chữsố+ . chữsố+
  - sốnguyên → chữsố+



Nhận dạng token bằng DFA

- Sơ đồ xác định identifier và keywords



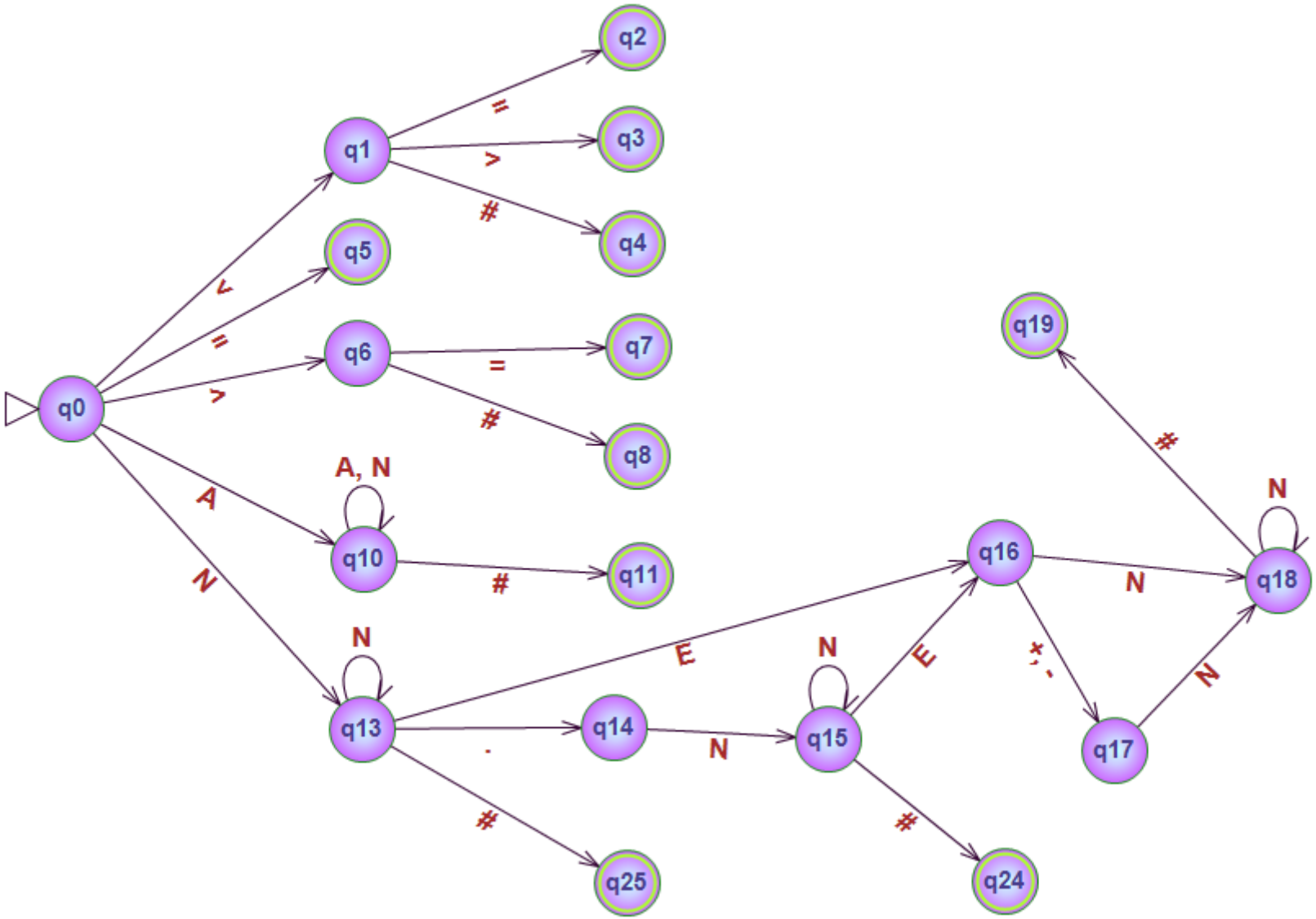
trong đó:

- Keywords là các từ khóa được lưu trữ sẵn trong symbol-table;
- gettoken() tra cứu lexeme trong symbol-table, nếu nó là keyword thì trả về token tương ứng, ngược lại trả về token id;
- install_id() tra cứu lexeme trong symbol-table, nếu là 1 keyword thì trả lại giá trị 0, nếu là một biến đã có thì trả lại một con trỏ tới vị trí trong symbol-table. Nếu lexeme không có thì tạo một phần tử mới trong symbol-table và trả về con trỏ tới nó.

Nhận dạng token bằng DFA

- **Các bước xây dựng LA nhận dạng tokens bằng DFA:**
  a) Tập hợp tất cả các mẫu của từ tố;
  b) Lập bộ phân tích từ vựng bằng phương pháp diễn giải đồ thị chuyển;
  c) Kết hợp các đồ thị chuyển thành một đồ thị chuyển duy nhất. Lập bộ phân tích từ vựng điều khiển bằng bảng chuyển (mô phỏng ôtômát hữu hạn đơn định).
- **Ưu điểm:** dễ hiểu, dễ viết.
- **Nhược điểm:** gắn kết cấu đồ thị chuyển vào trong chương trình. Khi thay đổi đồ thị thì phải viết lại toàn bộ chương trình.

Nhận dạng token bằng DFA



```
class Scanner {
        InputStream _in;
        char       _la; // The lookahead character
        char[]     _window; // lexeme window
        Token  nextToken() {
        startLexeme();    // reset window at start
                while(true) {
                        switch(_state) {
                                case 0: {
                                _la = getChar();
                                if (_la == '<') _state = 1;
                                        else if (_la == '=') _state = 5;
                                        else if (_la == '>') _state = 6;
                                        else failure(state);
                                }break;
                                case 6: {
                                        _la = getChar();
                                        if  (_la == '=') _state = 7;
                                        else _state = 8;
                                }break;
                        }
                }
        }
}
```

```
Token nextToken(){
char c ;
loop: c = getchar();
switch (c){
        case ` `:goto loop ;
        case `;`:  return SemiColumn;
        case `+`:  c = getchar() ;
            switch (c) {
              case `+': return PlusPlus ;
              case '=’  return PlusEqual;
              default:  ungetc(c);
                                return Plus;}
        case `<`:...
        case `w`:...
}
```

Một số vấn đề trong xây dựng LA

- **Xác định lỗi trong LA:** Rất ít lỗi được phát hiện trong lúc phân tích từ vựng, vì bộ phân tích từ vựng chỉ quan sát chương trình nguồn một cách rất cục bộ.
  - **VD**: Có thể phát hiện được lỗi dạng: =! (nhầm của != trong C)
  - **VD**: không thể phát hiện được lỗi dạng: fi a = b then ...
- **Dấu hiệu nhận biết token?** Ký tự nào?
  - Đọc chuỗi dài nhất có thể?
  - Dấu hiệu kết thúc token: là những ký tự không thể có mặt trong dạng token đó!!!

Một số vấn đề trong xây dựng LA

- **Các cách khắc phục lỗi:**
  1. Ngừng hoạt động và báo lỗi cho người sử dụng.
  2. Ghi ra các lỗi này và cố gắng bỏ qua chúng để tiếp tục làm việc, nhằm phát hiện đồng thời thêm nhiều lỗi khác trong chương trình
     1. Xoá hoặc nhẩy qua các kí tự mà bộ phân tích không tìm được từ tố;
     2. Thêm một kí tự bị thiếu;
     3. Thay một kí tự sai bằng một kí tự đúng;
     4. Tráo hai kí tự đứng cạnh nhau.

Một số vấn đề trong xây dựng LA

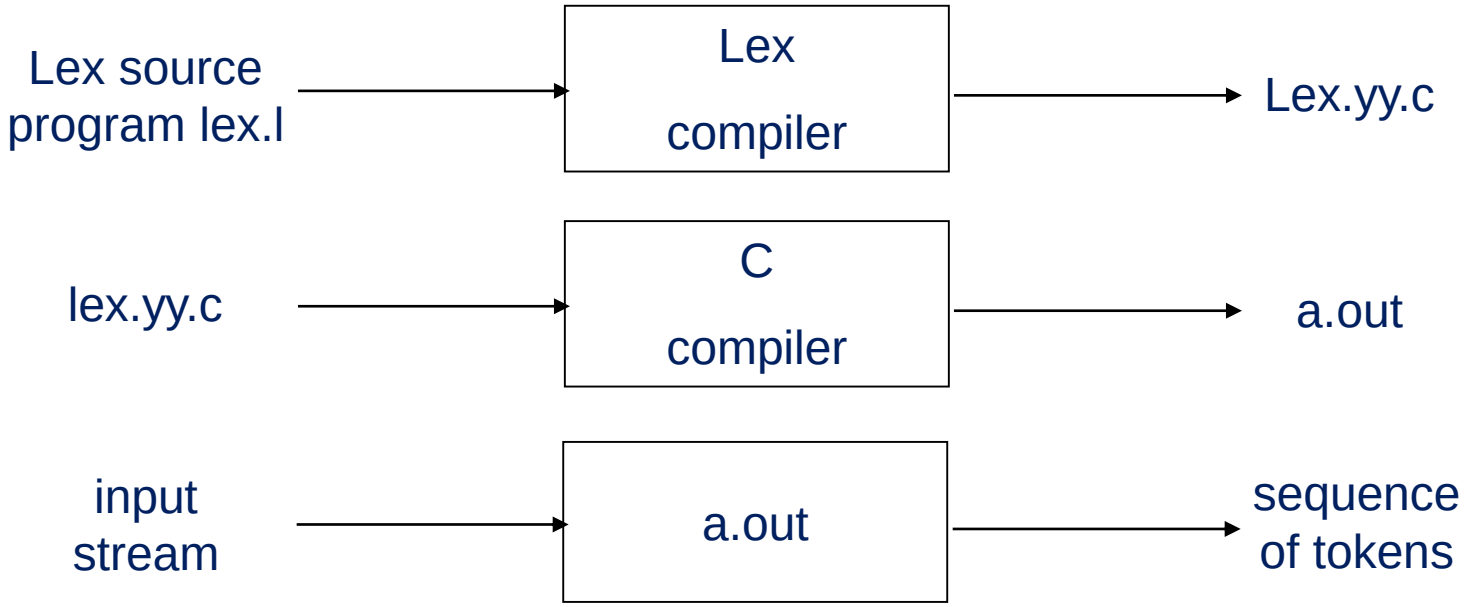
- **Bỏ qua chú thích (comment):**
  - Thông thường, ta không phân tích các comment thành token, bởi vậy LA trả về token tiếp theo mà không phải là chú thích cho SA;
  - Như vậy comments chỉ được xử lý trong bộ LA, và chúng không làm phức tạp thêm cú pháp của PL.
- **Symbol-table:**
  - symbol table lưu giữ thông tin về các token;
  - Làm thế nào để quản lý và sử dụng bảng ký tự? Những thuật toán nào được sử dụng? (Bảng băm (hash table), thêm token vào bảng băm, tìm vị trí của token theo lexeme).
  - Vị trí của token trong SP? (cho vấn đề error handling).

Các bước và công cụ xây dựng LA

- **Sơ đồ chung:**
  1. Sưu tầm tất cả các luật từ vựng, các luật này thường được mô tả bằng lời hoặc ở dạng **RD** hoặc **RE** để tiện theo dõi và chỉnh sửa, và làm dễ cho việc dựng đồ thị chuyển.
  2. Xây dựng đồ thị chuyển (hoặc bảng chuyển) cho từng luật một.
  3. Kết hợp các luật này thành một đồ thị chuyển duy nhất.
  4. Sinh mã từ đồ thị chuyển này.
  5. Bổ sung các thành phần chương trình để thành bộ phân tích hoạt động được.
  6. Thêm phần báo lỗi để thành bộ phân tích từ vựng hoàn chỉnh.

Các bước và công cụ xây dựng LA

- Một số công cụ có sẵn cho phép xây dựng một bộ phân tích từ vựng dựa trên các biểu thức chính qui như Lex, Flex, Tplex, Jlex...
- Lex là công cụ sinh bộ phân tích từ vựng, phát triển bởi Lesk và Schmidt tại AT&T Bell Lab, viết trên ngôn ngữ C trong UNIX.
- Cách tạo một bộ phân tích từ vựng bằng cách sử dụng **Lex**:



Đặc tả của Lex

Một chương trình Lex thông thường gồm 3 phần:

```
%{
    các khai báo bổ trợ
%}
    phần khai báo chính
%%
    các luật dịch
%%
    các hàm bổ trợ
```

Các khai báo bổ trợ trong Lex

- Định nghĩa các lớp ký tự và các biểu thức chính quy bổ trợ:

  [] ký hiệu giới hạn của lớp ký tứ tự

  - khoảng ký tự: **[xyz]** = = **[x-z]**

  **** ký tự định dạng giống trong C.

  ^ Phần bù của các ký tự (**Not**):

  [^xy] tất cả các ký tự ngoại trừ **x** và **y**.

  |, *, và **+** (hoặc, Kleene closure, và positive closure).

  () gom nhóm, điều khiển các biểu thức con.

  (expr)? = = **(expr)|λ,** có nghĩa là **expr** xuất hiện 0 hoặc 1 lần.

  [ab][cd] sẽ so khớp với ad, ac, bc, hoặc bd.

  begin = = “begin” = = [b][e][g][i][n]

Các khai báo chính

```
%{
        /*definitions of manifest constants
        LT,LE,EQ,GT,GE,IF,THEN,ELSE,ID*/
%}
        /*regular expression*/
       delim [\t\n]
       ws      {delim}+
       letter   [A-Za-z]
       digit    [0-9]
       id       {letter}({letter}|{digit})*
```

Các luật dịch

p_1 {action$_1$} /*p—pattern(Regular exp) */

…

p_n {action$_n$}

VD:

{if} {return(IF);}

{id} {yylval=install_id();return(ID);}

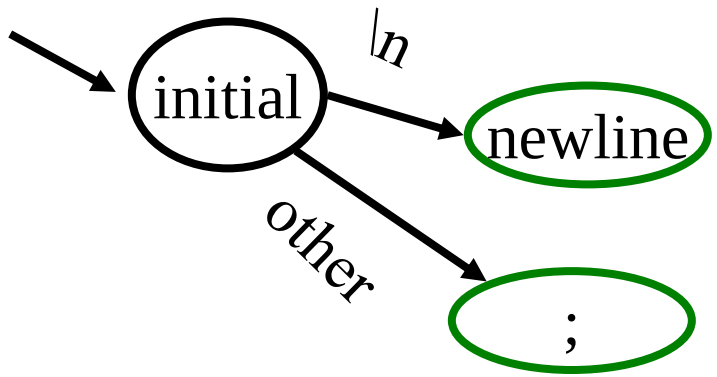
Các hàm bổ trợ

```
install_id() {
        /* tra cứu lexeme trong symbol-table, nếu là 1 keyword thì trả lại giá trị 0,
   nếu là một biến đã có thì trả lại một con trỏ tới vị trí trong symbol-table. Nếu
   lexeme không có thì tạo một phần tử mới trong symbol-table và trả về con trỏ tới
   nó.*/
}
```

Ví dụ về Flex

VD: Đếm số dòng của chương trình

```
int num_lines = 0;
%%
\n ++num_lines;
.  ;
%%
main(){
  yylex();
  printf( "# of lines = %d\n", num_lines);
}
```



Ví dụ về Flex

```
%{                          // auxiliary declarations (in C)
#define LT24
#define LE25
#define EQ26
  ...
%}                          // regular definitions
delim       [ \t\n]
ws          {delim}+
letter           [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
```

Ví dụ về Flex

// translation rules (actions are in C)

```
{ws}        { /* no action and no return */ }
if          {return (IF);}
then        {return (THEN);}
else        {return (ELSE);}
{id}        {yylval=install_id(); return (ID);}
{number}    {yylval=install_num(); return (NUMBER);}
"<"         {yylval=LT; return (RELOP);}
"<="        {yylval=LE; return (RELOP);}
  ...
```

// auxiliary procedures (in C)

```
%%
install_id() { … /* yytext to symbol table */  }
install_num() { ... /* yytext to symbol table */ }
```

Bài tập thực hành phân tích từ vựng

- Tìm hiểu chương trình LEX (Lex compiler) và thiết kế bộ phân tích từ vựng ngôn ngữ Pascal, C, C++, C#, Java… Lex là công cụ sinh bộ PTTV trên UNIX. Flex, TPlex là những công cụ sinh bộ PTTV trên nền Windows
  - http://dinosaur.compilertools.net/lex/index.html
  - http://flex.sourceforge.net/
  - http://gnuwin32.sourceforge.net/packages/flex.htm
  - http://www.gnu.org/software/bison/bison.html
  - http://stackoverflow.com/questions/5456011/how-to-compile-lex-yacc-files-on-windows
  - http://userpages.monmouth.com/~wstreett/lex-yacc/lex-yacc.html
  - …