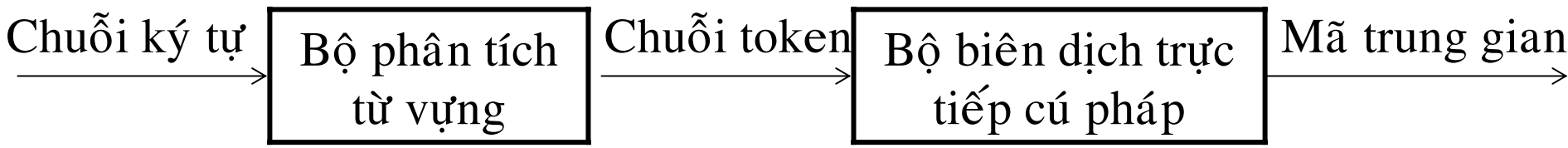
CHƯƠNG 2

TRÌNH BIÊN DỊCH ĐƠN GIẢN

2.1. Tổng quát



Hình 2.1. Cấu trúc trình biên dịch "front end"

2.2. Định nghĩa cú pháp

Văn phạm phi ngữ cảnh (PNC) được định nghĩa:

$$G2 = (V_t, V_n, S, P)$$

$$P : \quad A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots\ldots\ldots \mid \alpha_n$$

Thí dụ 2.1. Cho văn phạm G:

$$P: \quad list \rightarrow list + digit$$
$$\mid list - digit$$
$$\mid digit$$
$$digit \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

Thí dụ 2.2. Văn phạm miêu tả phát biểu hỗn hợp *begin end* của Pascal

P : block → **begin** opt_stmts **end**

opt_stmts → stmt_list |€

stmt_list → stmt_list ; stmt | stmt

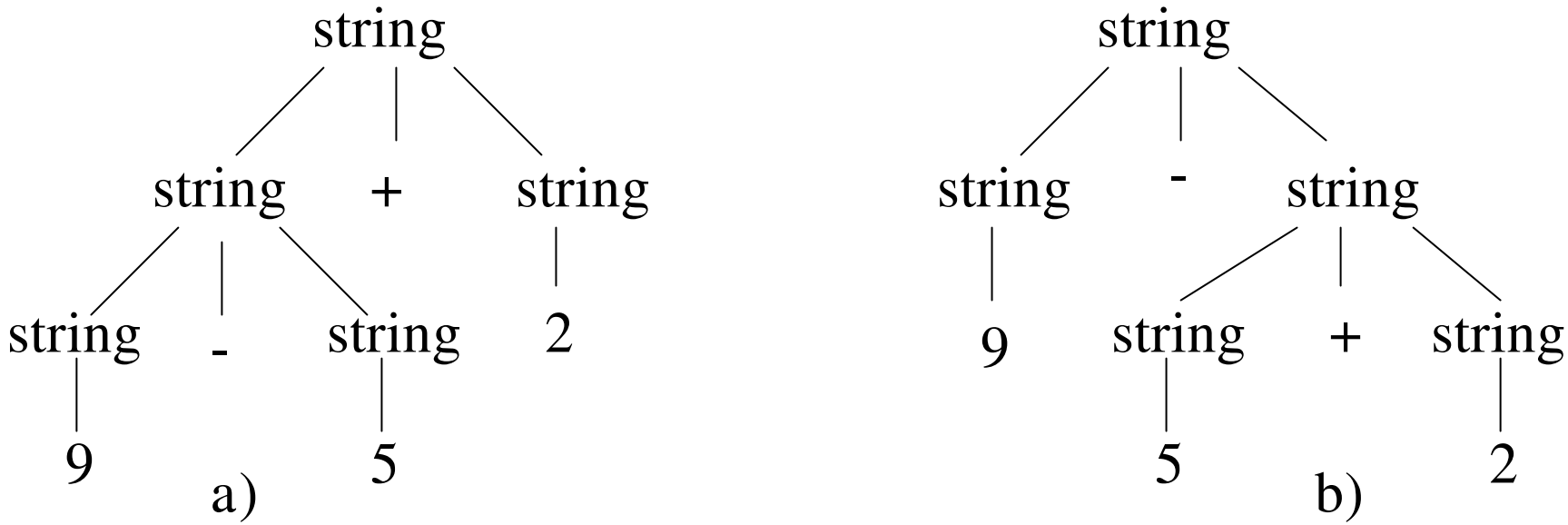
- Cây phân tích

Sự không tường minh

Thí dụ 2.3. Văn phạm G sau đây là không tường minh:

P : string → string + string | string – string | 0 | 1 | ... |9

Câu 9 – 5 + 2 cho hai cây phân tích:



Hình 2.2 Hai cây phân tích của câu 9 – 5 + 2

Sự kết hợp của các toán tử

Mức ưu tiên của các toán tử: * và / có mức ưu tiên hơn + , - . Dựa vào nguyên tắc trên chúng ta xây dựng cú pháp cho biểu thức số học:

exp → exp + term | exp – term | term
term → term * factor | term / factor | factor
factor → digit | (exp)

Lưu ý: phép toán lũy thừa và phép gán trong C là phép toán kết hợp phải. Văn phạm cho phép gán như sau:

right → letter = right | letter
letter → a | b | ... | z

2.3. Sự biên dịch trực tiếp cú pháp (Syntax-Directed Translation)

1. Ký hiệu hậu tố

1) Nếu E là biến hoặc hằng số thì ký hiệu hậu tố của E chính là E.
2) Nếu E là biểu thức có dạng E_1 *op* E_2 với *op* là toán tử hai ngôi thì ký hiệu hậu tố của E là E_1' E_2' *op*.
3) Nếu E là biểu thức có dạng (E_1) thì ký hiệu hậu tố của E_1 cũng là ký hiệu hậu tố của E.

Lưu ý: Không cần có dấu đóng, mở ngoặc trong ký hiệu hậu tố.

2. Định nghiã trực tiếp cú pháp (Syntax-directed definition)

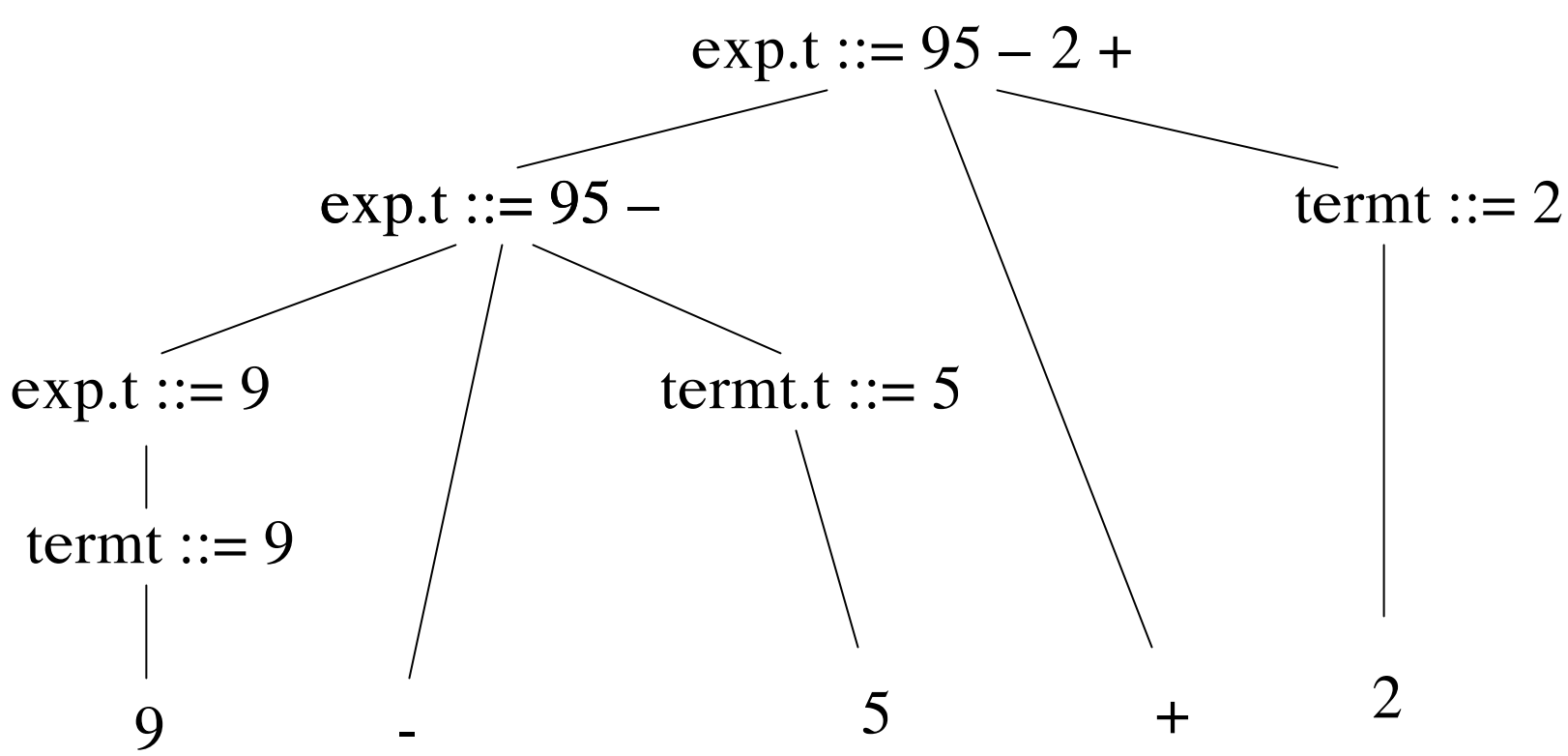
Văn phạm phi ngữ cảnh và tập luật ngữ nghiã sẽ thiết lập định nghĩa trực tiếp cú pháp. Biên dịch là phép ánh xạ từ nhập → xuất. Dạng xuất của chuỗi nhập x được xác định như sau:

1. Xây dựng cây phân tích cho chuỗi x.
2. Giả sử nút n của cây phân tích có tên cú pháp X, X.a là trị thuộc tính a của X, được tính nhờ luật ngữ nghĩa. Cây phân tích có chú thích các trị thuộc tính ở mỗi nút được gọi là cây phân tích chú thích

Tổng hợp thuộc tính (synthesized attributes)

Thí dụ 2.4. Cho văn phạm G có tập luật sinh P:

Tập luật sinh	Tập luật ngữ nghĩa
exp → exp + term	exp.t ::= exp.t || term.t || '+'
exp → exp – term	exp.t ::= exp.t || term.t || '-'
exp → term	exp.t ::= term.t
term → 0	term.t ::= '0'
...	...
term → 9	term.t ::= '9'



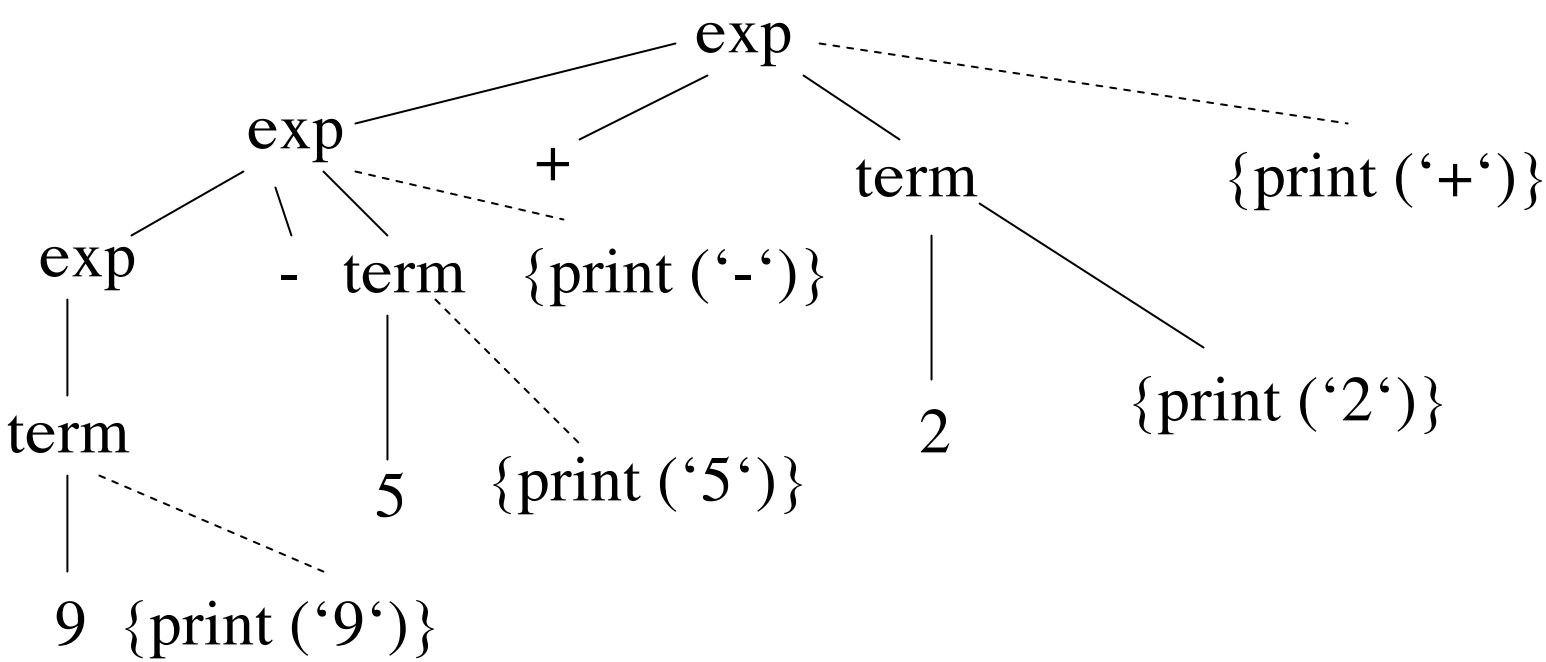
Hình 2.3. Cây phân tích chú thích cho định nghĩa trực tiếp cú pháp

Lược đồ dịch

Lược đồ dịch là văn phạm PNC, trong đó các đoạn chương trình gọi là hành vi ngữ nghiã được nhúng vào vế phải của luật sinh.

Thí dụ 2.5. Lược đồ dịch của văn phạm G:

Tập luật sinh	Tập luật ngữ nghĩa
exp → exp + term	exp → exp + term { print (‘+’)}
exp → exp – term	exp → exp – term {print (‘-’)}
exp → term	exp → term
term → 0	term → 0 {print (‘0’)}
.....	
term → 9	term → 9 {print {‘9’)}



Hình 2.4. Lược đồ dịch của câu 9 – 5 + 2

Mô phỏng 2.1. Giải thuật depth- first traversals của cây phân tích

```
Procedure visit (n: node);
    begin
        for với mỗi con m của n, từ trái sang phải do
     visit (m);
     tính trị ngữ nghiã tại nút n
    end;
```

2.4. Phân tích cú pháp

1. Phân tích cú pháp từ trên xuống

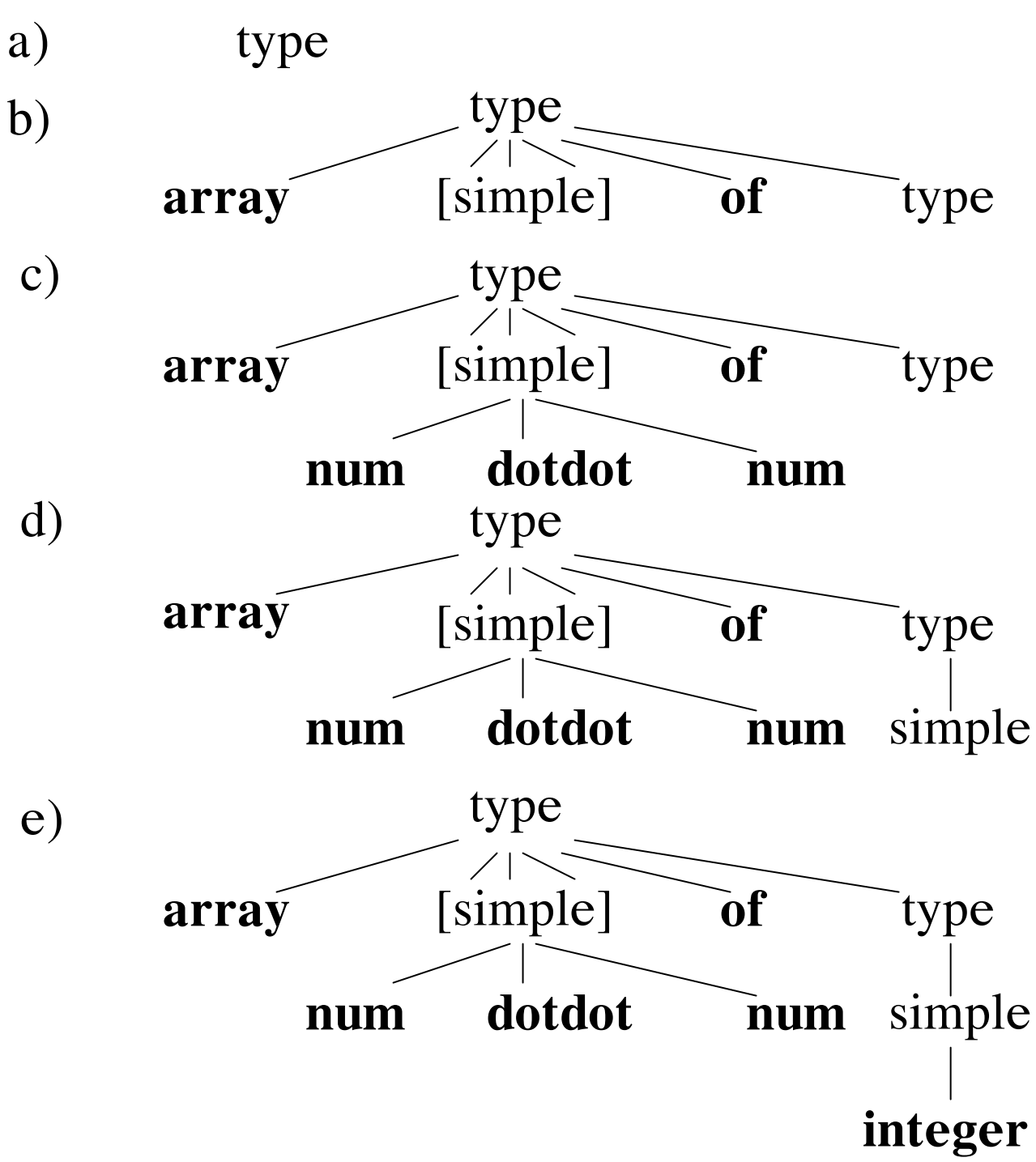
Thí dụ 2.6. Cho văn phạm G:

type → simple | ↑ **id** | array **[** simple**]** **of** type

simple → **integer** | **char** | **num dotdot num**

Hãy xây dựng cây phân tích cho câu:

array [num dotdot num] of integer



Hình 2.6.Các bước xây dựng cây phân tích theo phương pháp từ trên xuống cho câu: ***array [numdotdot num] of integer***

2. Sự phân tích cú pháp đoán nhận trước

Dạng đặc biệt của phân tích cú pháp từ trên xuống là phương pháp đoán nhận trước. Phương pháp này sẽ nhìn trước một ký hiệu nhập để quyết định chọn thủ tục cho ký hiệu không kết thúc tương ứng.

Thí dụ 2.8. Cho văn phạm G: P: $S \rightarrow xA$ $A \rightarrow z \mid yA$
Dùng văn phạm G để phân tích câu nhập xyyz

Bảng 2.1. *Các bước phân tích cú pháp của câu xyyz*

Luật áp dụng	Chuỗi nhập
S	xyyz
xA	xyyz
yA	yyz
A	yz
yA	yz
A	z
z	z
-	-

Thí dụ 2.9. Cho văn phạm với các luật sinh như sau :

$S \rightarrow A \mid B \qquad A \rightarrow xA \mid y \qquad B \rightarrow xB \mid z$

Bảng 2.2. *Phân tích cú pháp cho câu xxxz không thành công*

Luật áp dụng	Chuỗi nhập
S	xxxz
A	xxxz
xA	xxxz
A	xxz
xA	xxz
A	xz
xA	xz
A	z

- *Điều kiện 1* : $A \rightarrow \xi_1 \mid \xi_2 \mid ... \mid \xi_n$

- *Định nghĩa:*

first (ξ_i) = {s | s là ký hiệu kết thúc và $\xi \Rightarrow s...$}

Điều kiện 1 được phát biểu như sau :

$A \rightarrow \xi_1 \mid \xi_2 \mid ... \mid \xi_n$

first $(\xi_i) \cap$ first $(\xi_j) = \varnothing$ với $i \neq j$

Lưu ý: 1. first $(a\xi) = \{a\}$

2. Nếu $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$; thì

first $(A\xi)$ = first $(\alpha_1) \cup$ first (α_2) ... $\cup$ first (α_n)

Thí dụ 2.11. Cho văn phạm G có tập luật sinh:

$S \rightarrow Ax$ $A \rightarrow x \mid \in$ với $\in$ là chuỗi rỗng

Bảng 2.3. *Phân tích câu nhập : x*

Luật	Chuỗi nhập
A	x
xx	x
x	-

Sự phân tích thất bại

- *Điều kiện 2:* first (A) $\cap$ follow (A) = $\varnothing$

Với $A \rightarrow \xi_1 \mid \xi_2 \mid ... \mid \xi_n \mid \in$

Follow (A) được tính như sau: Với mỗi luật sinh P_i có dạng $X \rightarrow \xi A \eta$ thì follow (A) là first (η).

Ở thí dụ 2.11 first (A) $\cap$ follow (A) = {x}

Lưu ý văn phạm có đệ quy trái sẽ vi phạm điều kiện 1. Thí dụ:

$A \rightarrow B \mid AB$ (2.1)

Vậy first (A) = first (B) ; first (AB) = first (A) = first (B).

first (B) $\cap$ first (AB) $\neq \varnothing$ vi phạm điều kiện 1.

Nếu sửa luật (2.1) thành $A \rightarrow \in \mid AB$ thì sẽ vi phạm điều kiện 2.

Thí dụ 2.12. Cho văn phạm như ở thí dụ 2.6, chúng ta dùng phương pháp phân tích đoán nhận trước để phân tìch câu array[num dot dot num] of integer *(tự xem ở trang 41).*

Các thủ tục được gọi khi sinh cây phân tích cho các câu thuộc văn phạm ở thí dụ 2.12.

2.5. Trình biên dịch cho biểu thức đơn giản

Thí dụ:

exp → exp + term {print ('+')} (2.5)
exp → exp – term {print ('-')}
exp → term
term → 0 {print ('0'}
.....
term → 9 {print ('9'}

Loại bỏ đệ quy trái:

exp → term rest
exp.t ::= term.t || rest.t
rest → + exp
rest.t ::= exp.t || '+'
rest → - exp
rest.t ::= exp.t || '-'
rest → ∈
term → 0
term.t ::= '0'
rest → ∈

term → 0

term.t ::= ‘0’

term → 9

term.t ::= ‘9’

Văn phạm này không phù hợp cho biên dịch trực tiếp cú pháp.

Lược đồ dịch:

exp → exp + term {print (‘+’)}

exp → exp –term {print (‘-’)}

exp → term

term → 0 {print (‘0’)}

……..

term → 9 {print (‘9’)}

Loại bỏ đệ quy trái cho lược đồ dịch:

exp → term rest

rest→ + term {print (‘+’)} | - term {print (‘-’)} | ∈

term → 0 {print (‘0’) }

....

term → 9 {print (‘9’)}

Cây phân tích chú thích cho câu: 9-5 = 2 ở tr.44

Chương trình biên dịch biểu thức từ dạng trung tố sang dạng hậu tố:

```
procedure exp;
procedure match ( t : token );
        begin if lookahead = t then
                lookahead := nexttoken
        else error
        end;
procedure term ;
        begin
                if  lookahead = num   then  begin
                                            write ( num);
                                            match (lookahead);
                                            end
                else  error
        end;
procedure  rest;
        begin
```

```
                    if lookahead = ' +'  then   begin
                                                    match ('+'); term;
                                                    write ('+');
                                                        end
                    else if  lookahead = '-' then
                                                begin
                                                match ('-'); term; write('-');
                                                end;
            end;
    begin
            term; rest;
    end;
```

Tối ưu trình biên dịch:

Để tăng tốc độ biên dịch ta thực hiên gỡ đệ quy của thủ tục rest:

```
    procedure exp;
            procedure term;
    begin
```

```
:
end;
begin
    term;
        repeat
            if lookahead = '+' then
                begin
                        match ('+'); term; write('+');
                end
            else if lookahead = '-' then
                begin
                        match('-'); term; write('-')
                end;
        until (lookahead < > '+') and (lookahead < > '-');
end;
```

Hoàn chỉnh chương trình:

Chương trình này bao gồm cả chương trình đọc chuỗi nhập.

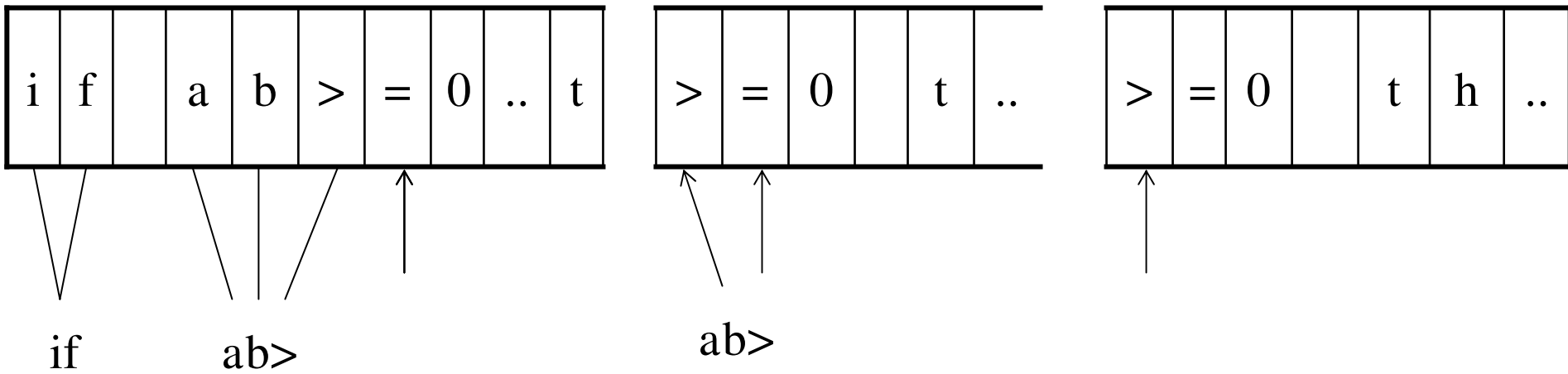
```
procedure exp;
    procedure match (t : char);
        begin
            if lookahead = t then  lookahead := readln (c);
                                   else error
        end;
procedure term;
    begin
        val (i,lookahead,e);
        if  e = 0 then begin
                                write (i);
                                match (lookahead );
                        end
                        else error;
    end;{term}
begin
```

```
    term;
    repeat
    if lookahead = '+' then
        begin
            match ('+'); term; write('+');
        end
    else  if  lookahead = '-'   then
        begin
        match ('-'); term; write('-');
        end;
    until (lookahead < > '+' ) and (lookahead < > '-');
end; {exp }
begin
    readln( c);
    lookahead := c;
    exp;
end;
```

2.6. Sự phân tích từ vựng

1. Loại bỏ khoảng trắng và chú thích
2. Nhận biết các hằng
3. Nhận biết danh biểu và từ khóa

Giao tiếp với bộ phân tích từ vựng



Hình 2.10. Nhận dạng token của bộ phân tích từ vựng

2.7. Sự hình thành bảng danh biểu

1. Giao tiếp với bảng danh biểu

Hai thao tác với bảng danh biểu: *insert* (s,t) và *lookup* (s).

2. Lưu giữ từ khóa

3. Hiện thực bảng danh biểu

Bảng danh biểu gồm có bảng **symtable** và dãy **lexemes.**

Bảng symtable

	lexptr	token	các thuộc tính khác
0			
1	1	div	
2	5	mod	
3	9	id	
4	15	id	

Dãy lexemes

d	i	v	EOS	m	o	d	EOS	c	o	u	n	t	EOS	i	EOS

Hình 2.11. Bảng danh biểu

Mô phỏng 2.2. *Giải thuật phân tích từ vựng*

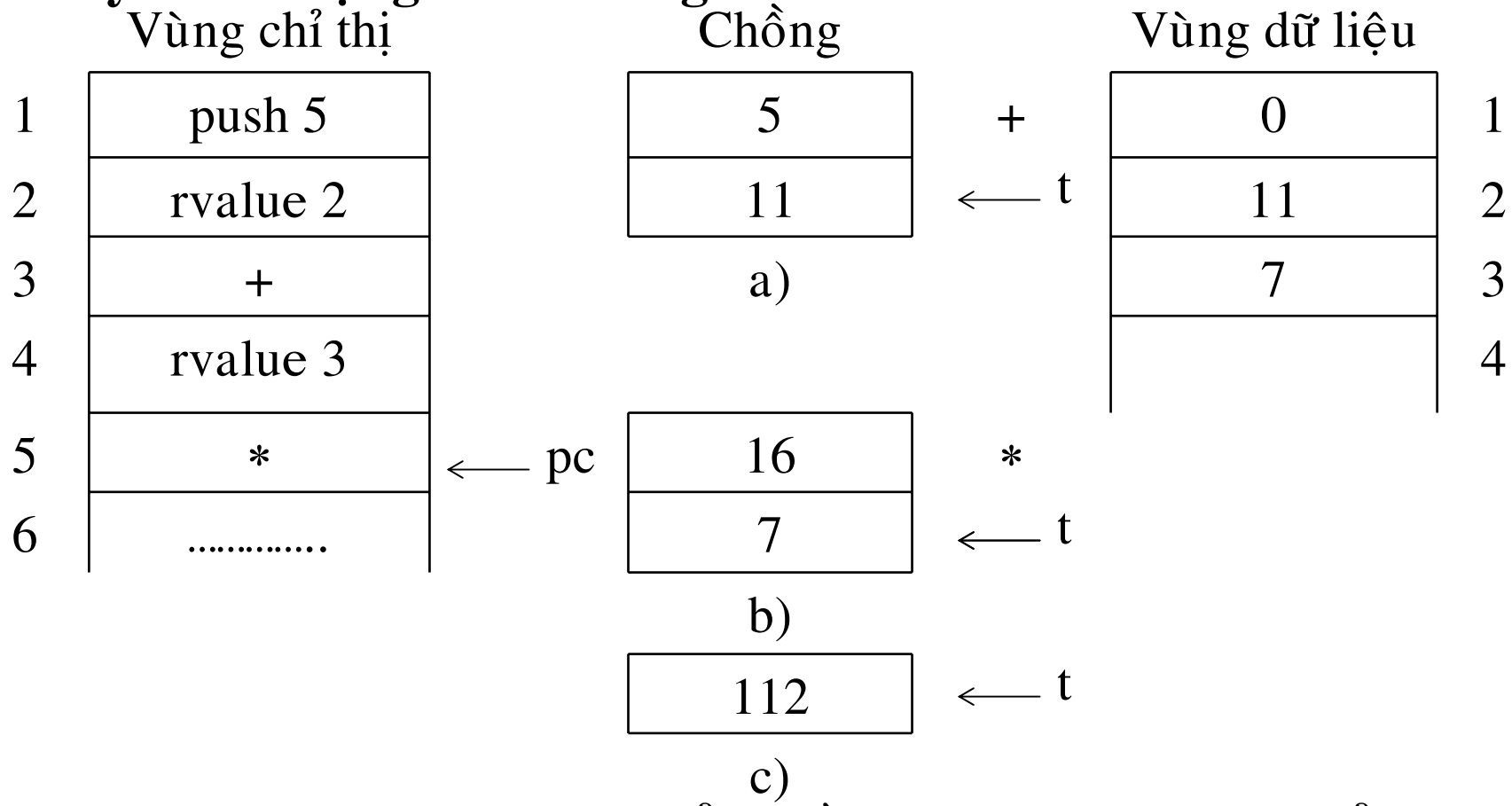
```
Procedure  lexan;
        var lexbuf array [0..100] of char;
            c : char;  ngưng : boolean;
        begin
            repeat
            read (c ); ngưng := true;
            if  (c = blank ) or (c = tab)  then   ngưng := false
            else  if  c = newline  then begin  line := lineno + 1
    ngưng := false;
        end
        else if c là chữ số then
```

```
    begin
    val (i, c, e);
    tokenval := 0;
    while e = 0 do begin
        tokenval := tokenval * 10 + i;
        read (c);
        val (i, c, e);
    end;
typetoken := num;
    end {là số}
else if  c là chữ then begin
    p := 0; b := 0;
    while c là chữ hoặc số do
        begin lexbuf [b] := c;
            read (c);
```

```
                    b := b + 1;
                    if b => b_size then error
                end; /* b size là kích thước tối đa của lexbuf*/
        lexbuf [b] := eos;
        p := lookup (lexbuf);
if  p = 0 then
                    p = insert (lexbuf, ID);
            tokenval := p;
            typetoken := symtable [p]. token; end
        else if c = eof then begin
                    tokenval := none;
                    typetoken := done; {hết chương trình nguồn}
                    end
                    else begin
                        tokenval := none; typetoken := c;
```

```
        end
    until ngưng;
end;
```

2.8. Máy trừu tượng kiểu chồng



*Hình 2.12. Máy trừu tượng kiểu chồng với việc thực thi biểu thức (5 + 11) * 7*

1. Chỉ thị số học

2. Lvalue và Rvalue

Thí dụ: i := i + 1

3. Thao tác với chồng

Các chỉ thị: Lvalue, Rvalue, push v, pop, copy, :=

4. Biên dịch cho biểu thức

Thí dụ: Biên dịch phát biểu gán:

day := (53*y) div 4 + (273 * m + 2) div 5 + d

chuyển sang ký hiệu hậu tố

day 53y * 4 div 273 m * 2 + 5 div + d + :=

dịch sang mã máy trừu tượng

5. Chỉ thị điều khiển trình tự

Các chỉ thị bao gồm: label l, goto l, gotofalse l, gototrue l, halt.

6. Sự biên dịch các phát biểu

Thí dụ: Phát biểu if:

stmt→ **if** exp **then** stmt $\xrightarrow{\text{ngữ nghĩa}}$

out := newlabel

stmt.t ::= exp.t || ‘gotofalse’ out || stmt.t || ‘label’ out

*Hình 2.13. Mã máy trừu tượng của phát biểu **if***

7. Giải thuật của trình biên dịch các phát biểu

```
procedure stmt;
    var out : integer;
    begin
        if  lookahead = id  then
            begin  emit ('lvalue', tokenval);
                match (id); match (' := '); exp; emit (':=', tokenval)
            end
```

```
    else if  lookahead = 'if'  then
        begin  match ('if'); exp;
               out := newlabel;
               emit ('gotofalse', out);
               match ('then'); stmt;
               emit ('label',out)
        end
    else error
end;
```

2.9. Thiết kế trình biên dịch đơn giản

1. Đặc tả trình biên dịch

start→ list **eof**

list→ exp **;** list | ∈

exp → exp + term {print ('+')}

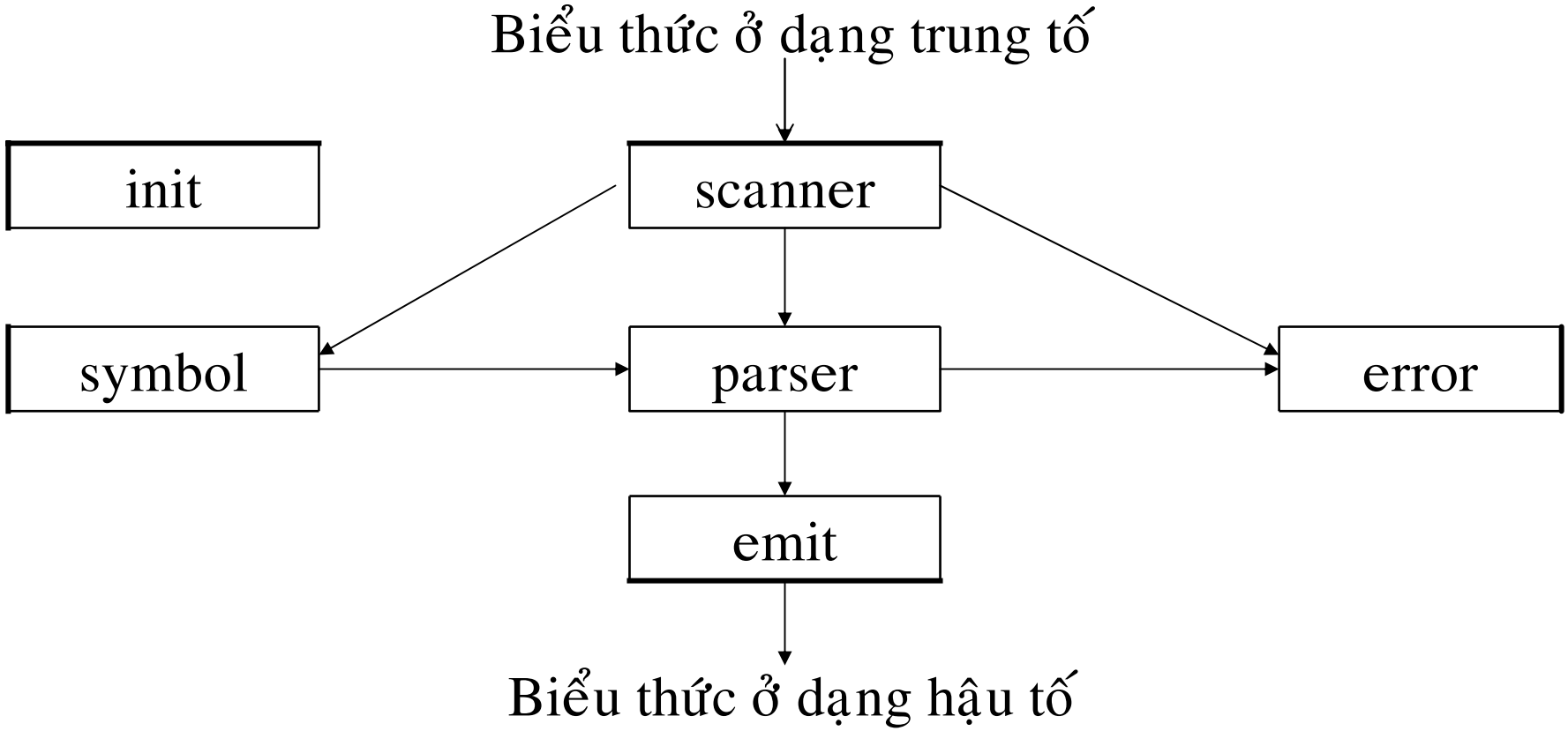
lexp – term {print ('-')}

| term

term → term * factor {print ('*')}

| term / factor {print(‘/’)}
| term **div** factor {print (‘div’)}
| term **mod** factor {print (‘mod’)}
| factor

factor → (exp) | **id** | **num**



Hình 2.14. Sơ đồ của trình biên dịch cho biểu thức từ dạng trung tố sang dạng hậu tố

2. Nhiệm vụ của các chương trình con của trình biên dịch

scanner: phân tích từ vựng; ***parser***: phân tích cú pháp; ***emit***: tạo dạng xuất của token; ***symbol***: xây dựng bảng danh biểu và thao tác với bảng danh biểu bằng *insert* và *lookup*; ***init***: cất các từ khóa vào bảng danh biểu; ***error***: thông báo lỗi.

Mô phỏng 2.3. *Lược đồ dịch trực tiếp cú pháp cuả G sau khi được bỏ đệ quy trái:*

start → list **eof**
list → exp ; list | ∈
exp → term $Rest_1$
$Rest_1$ → + term {print ('+')} $Rest_1$ | ∈
| - term {print ('-'-)} | ∈
term → factor $Rest_2$
$Rest_2$ →* factor {print ('*')} $Rest_2$
l/ factor {print ('/')} $Rest_2$
| **div** factor {print (div')} $Rest_2$ | ∈
| **mod** factor {print (mod')} $Rest_2$ | ∈
factor → (exp)
| **id** {print (id.lexeme)}
| num {print(num.value)}

3. Giải thuật của trình biên dịch

```
const  bsize = 128;     |para = 40;
       none = '#';      plus = 43;
       num = 256;       minus = 45;
       div = 257;       star = 42;
       mod = 258;       slash = 47;
       id = 259;
       done = 260;
       strmax = 999;
       symax = 100;
type   entry = record
                 lexptr : integer;
                 token : integer;
                 end;
       str = string;
var  tokenval : integer;
       lineno : integer;
       lookahead : char;
```

```
        symtable : array [1..100] of entry;
        lexbuf : string [bsize];
        typetoken : integer;
        lexemes: array[1..strmax] of char;
        lastentry : integer;
        lastchar : integer;
procedure  scanner;
    var  t: char;
            p, b, i: integer;
begin
                read (t);
    if (t = ‘ ‘ ) or (t = \t’)  then
                repeat   read (t);
                until  (t < > ‘ ‘) and (t < > ‘\t’);
    else  if  t = ‘\t’  then  begin
                lineno := lineno + 1;
                read ( t );
                end
```

```
else  if  t in [‘0’..’9’]  then  begin
                val ( i,t,e);
                tokenval := 0;
while  e = 0  do   begin
tokenval := tokenval *10 + I;
                read (t);
                val (i,t,e);
                end;
typetoken := num;
end
else  if  t  in [ ‘A’..’Z’,’a’..’z’]  then
                begin
                     p:= 0; b := 0;
                while t in [‘0’..’9’,’A’..’Z’,’a’..’z’] do
                begin lexbuf [b] := t;
                     read (t);
                          b := b + 1;
                if  (b > = bsize)  then
```

```
                    error
                end;
        lexbuf [b] := eos;
        p := lookup (lexbuf);
        if p = 0  then  p := insert ( lexbuf, id);
        tokenval := p;
        typetoken := symtable[p].token;
        end
    else   if  t = eof  then   typetoken := done
        else begin
                typetoken := ord (t);
        read (t)
        end;
        tokenval := none;
        end;
end; {scanner}
/*----------------------*/
procedure parser;
```

```
    procedure exp;
        var  t : integer;
    procedure term;
        var  t :  integer;
    procedure factor;
        begin
        case   lookahead   of
    |para :  begin  match ( lpara); exp;
                match(rpara); end;
    num :  begin  emit (num, tokenval);  match (num)
        end;
    id :  begin emit (id, tokenval );
        match (id)  end;
    else error (‘ lỗi cú pháp’, lineno);
                end; {case}
                end; {factor}
/*---------------------------*/
```

```
begin {term}
    factor;
        while  lookahead in [star, slash, div, mod]  do
            begin
            t := lookahead;
            match (lookahead);
            factor; emit (t, none);
            end;
        end; {term}
begin {exp}
    term;
    while (lookahead = plus) or (lookahead = minus)  do
        begin
            t := lookahead ; match (lookahead);
            term;  emit (t, none);
        end;  end;
begin  {parser}
    scanner;  lookahead := typetoken;
```

```
        while lookahead < > done do
            begin  exp; match (semicolon);  end;
            end; {parser}
    /*----------------------*/
    procedure match (t : integer);
        begin
        if lookahead = t then  begin
            scanner;
            lookahead := typetoken ;  end
        else  error (‘ lỗi cú pháp’, lineno);
    end;
procedure emit (t : integer; tval : integer);
    begin
        case  t of
    plus, minus, star, slash : writeln (chr (t ));
        div : writeln (‘div’);
        mod : writeln (‘mod’);
        num : writeln (tval);
```

```
            id : wrteln (symtable[tval].lexptr^);
            else writeln (chr (t). tval);
            end;
        end; {emit}
function strcmp (cp : integer; s: str) : integer;
        var i, l :  integer;
        begin  i := t; l := length (s);
        while ( I < = l ) and (s[i] = lexemes [cp] do
            begin
                i := i + 1;
                cp := cp + 1;
            end;
        if i > l  then   strcmp := 1
            else  strcmp := 0
        end; {strcmp}
procedure strcopy (cp : integer; t : str);
        var   i : integer;
        begin
```

```
        for i := 1 to length (t) do
            begin
                lexemes [cp] := t [i]
                cp := cp + 1;
            end;
    lexemes [cp] := eos;
    end; {Strcopy}
function lookup (s : string) : integer;
            var  I, p: integer;
            begin p := lastentry;
    while  (p > 0) and (Strcmp (symtable [p].lexptr ^ , s) = 0) do
    p := p – 1;
```

```
        lookup := p;
    end; {lookup}
/*------------------ */
function insert (s : str; typetoken : integer) : integer;
    var len: integer;
    begin
        len := length (s );
        if (lastentry + 1 > = symax )  then error ('bảng danh
                                                biểu đầy', lineno);
        if (lastchar + len + 1 > = strmax )  then
        error ('dãy lexemes đầy, lineno);
        lastentry := lastentry + 1;
        symtable [ lastentry].token := typyetoken;
        symtable [latsentry].lexptr := @lexemes[lastchar + 1];
        lastchar := lastchar + len + 1;
```

```
        strcopy (symtable [latsentry].lexptr ^, s)
        insert := lastentry;
    end; {insert}
/*-----------------*/
procedure init;
    var keyword : array[1.3] of
                    record
                    lexeme : string [10]
                    token : integer;
                    end;
                    r, i : integer;
    begin  keyword [i].lexeme := 'div';
        keyword [1].token := div;
        keyword [2].lexeme:= 'mod';
        keyword [2].token := mod;
        keyword [3].lexeme := '0';
        keyword [3].token := 0;
        r := 3;
```

```
    for i := 1 to r do
        p := insert (keyword [i].lexem, keyword [i].token);
    end;
/*----------------*/
procedure error (m : str; lineno : integer);
    begin  writeln (m, lineno);
    stop;
    end;
/*----------------*/
    begin {main}
        lastentry := 0;  lineno := 0;  tokenval := -1;
        lastchar := 0;
        init;
        parser;
    end; {main}
```