CHƯƠNG II
MỘT TRÌNH BIÊN DỊCH ĐƠN GIẢN

Nội dung chính:
Chương này giới thiệu một *trình biên dịch cho các biểu thức số học đơn giản* (trình biên dịch đơn giản) gồm hai kỳ: Kỳ đầu (Front end) và kỳ sau (Back end). Nội dung chính của chương tập trung vào *kỳ đầu* gồm các giai đoạn: Phân tích từ vựng, phân tích cú pháp và sinh mã trung gian với mục đích chuyển một biểu thức số học đơn giản từ dạng trung tố sang hậu tố. Kỳ sau chuyển đổi biểu thức ở dạng hậu tố sang *mã máy ảo kiểu stack*, sau đó sẽ thực thi đoạn mã đó trên *máy ảo kiểu stack* để cho ra kết quả tính toán cuối cùng.

Mục tiêu cần đạt:
Sau khi học xong chương này, sinh viên phải nắm được:

- Các thành phần cấu tạo nên trình biên dịch đơn giản.
- Hoạt động và cách cài đặt các giai đoạn của kỳ trước của một trình biên dịch đơn giản.
- Cách sử dụng máy trừu tượng kiểu stack để chuyển đổi các biểu thức hậu tố sang mã máy ảo và cách thực thi các đoạn mã ảo này để có được kết quả cuối cùng.

Kiến thức cơ bản
Để tiếp nhận các nội dung được trình bày trong chương 2, sinh viên phải:

- Biết một ngôn ngữ lập trình nào đó: C, Pascal, v.v để hiểu cách cài đặt trình biên dịch.
- Có kiến thức về cấu trúc dữ liệu để hiểu cách tổ chức dữ liệu khi thực hiện cài đặt.

Tài liệu tham khảo:

[1] **Trình Biên Dịch** - Phan Thị Tươi (Trường Đại học kỹ thuật Tp.HCM) - NXB Giáo dục, 1998.

[2] **Compilers : Principles, Technique and Tools** - Alfred V.Aho, Jeffrey D.Ullman - Addison - Wesley Publishing Company, 1986.

I. ĐỊNH NGHĨA CÚ PHÁP

1. Văn phạm phi ngữ cảnh

Để xác định cú pháp của một ngôn ngữ, người ta dùng văn phạm phi ngữ cảnh CFG (Context Free Grammar) hay còn gọi là văn phạm BNF (Backers Naur Form)

Văn phạm phi ngữ cảnh bao gồm bốn thành phần:

1. Một tập hợp các token - các ký hiệu kết thúc (terminal symbols).

   Ví dụ: Các từ khóa, các chuỗi, dấu ngoặc đơn, ...

2. Một tập hợp các ký hiệu chưa kết thúc (nonterminal symbols), còn gọi là các biến (variables).

   Ví dụ: Câu lệnh, biểu thức, ...

3. Một tập hợp các luật sinh (productions) trong đó mỗi luật sinh bao gồm một ký hiệu chưa kết thúc - gọi là vế trái, một mũi tên và một chuỗi các token và / hoặc các ký hiệu chưa kết thúc gọi là vế phải.
4. Một trong các ký hiệu chưa kết thúc được dùng làm ký hiệu bắt đầu của văn phạm.

Chúng ta qui ước:

- Mô tả văn phạm bằng cách liệt kê các luật sinh.
- Luật sinh chứa ký hiệu bắt đầu sẽ được liệt kê đầu tiên.
- Nếu có nhiều luật sinh có cùng vế trái thì nhóm lại thành một luật sinh duy nhất, trong đó các vế phải cách nhau bởi ký hiệu "|"đọc là "hoặc".

Ví dụ 2.1: Xem biểu thức là một danh sách của các số phân biệt nhau bởi dấu + **và** dấu -. Ta có, văn phạm với các luật sinh sau sẽ xác định cú pháp của biểu thức.

list → list + digit
list → list - digit ⇔ list → list + digit | list - digit | digit
list → digit digit → 0 | 1 | 2 ...| 9
digit → 0 | 1 | 2 | ...| 9

Như vậy văn phạm phi ngữ cảnh ở đây là:

- Tập hợp các ký hiệu kết thúc: 0, 1, 2, ..., 9, +, -
- Tập hợp các ký hiệu chưa kết thúc: list, digit.
- Các luật sinh đã nêu trên.
- Ký hiệu chưa kết thúc bắt đầu: list.

Ví dụ 2.2:

Từ ví dụ 2.1 ta thấy: 9 - 5 + 2 là một list vì:

9 là một list vì nó là một digit.

9 - 5 là một list vì 9 là một list và 5 là một digit.

9 - 5 + 2 là một list vì 9 - 5 là một list và 2 là một digit.

Ví dụ 2.3:

Một list là một chuỗi các lệnh, phân cách bởi dấu ; của khối begin - end trong Pascal. Một danh sách rỗng các lệnh có thể có giữa begin và end.

Chúng ta xây dựng văn phạm bởi các luật sinh sau:

block → **begin** opt_stmts **end**

opt_stmts → stmt_list | **ε**

stmt_list → stmt_list **;** stmt | stmt

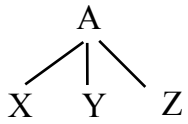
Trong đó opt_stmts (optional statements) là một danh sách các lệnh hoặc không có lệnh nào (ε).

Luật sinh cho stmt_list giống như luật sinh cho list trong ví dụ 2.1, bằng cách thay thế +, - bởi ; và stmt thay cho digit.

2. Cây phân tích cú pháp (Parse Tree)

Cây phân tích cú pháp minh họa ký hiệu ban đầu của một văn phạm dẫn đến một chuỗi trong ngôn ngữ.

Nếu ký hiệu chưa kết thúc A có luật sinh A → XYZ thì cây phân tích cú pháp có thể có một nút trong có nhãn A và có 3 nút con có nhãn tương ứng từ trái qua phải là X, Y, Z.



Một cách hình thức, cho một văn phạm phi ngữ cảnh thì cây phân tích cú pháp là một cây có các tính chất sau đây:

1. Nút gốc có nhãn là ký hiệu bắt đầu.
2. Mỗi một lá có nhãn là một ký hiệu kết thúc hoặc một ε.
3. Mỗi nút trong có nhãn là một ký hiệu chưa kết thúc.
4. Nếu A là một ký hiệu chưa kết thúc được dùng làm nhãn cho một nút trong nào đó và X1 ... Xn là nhãn của các con của nó theo thứ tự từ trái sang phải thì A → X1X2 ... Xn là một luật sinh. Ở đây X1, ..., Xn có thể là ký hiệu kết thúc hoặc chưa kết thúc. Đặc biệt, nếu A → ε thì nút có nhãn A có thể có một con có nhãn ε.

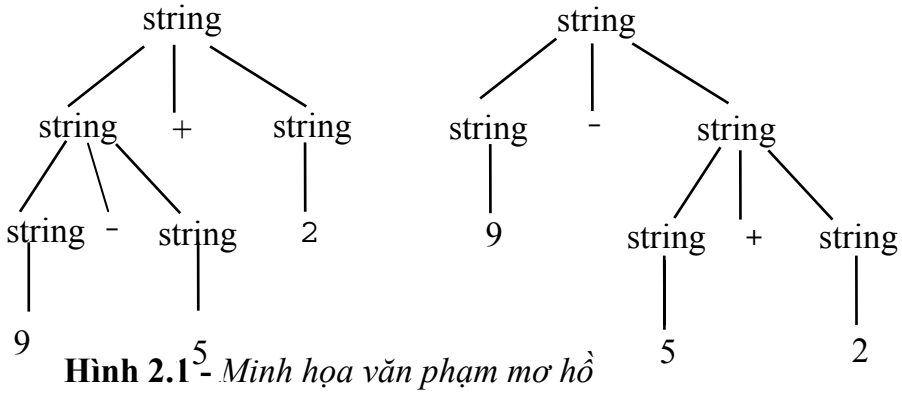
3. Sự mơ hồ của văn phạm

Một văn phạm có thể sinh ra nhiều hơn một cây phân tích cú pháp cho cùng một chuỗi nhập thì gọi là văn phạm mơ hồ.

Ví du 2.4: Giả sử chúng ta không phân biệt một list với một digit, xem chúng đều là một string ta có văn phạm:

string → string + string | string - string | 0 | 1 | ... | 9.

Với văn phạm này thì chuỗi biểu thức 9 - 5 + 2 có đến hai cây phân tích cú pháp như sau :



Hình 2.1 - *Minh họa văn phạm mơ hồ*

Tương tự với cách đặt dấu ngoặc vào biểu thức như sau :

(9 - 5) + 2 9 - (5 + 2)

Bởi vì một chuỗi với nhiều cây phân tích cú pháp thường sẽ có nhiều nghĩa, do đó khi biên dịch các chương trình ứng dụng, chúng ta cần thiết kế các văn phạm không có sự mơ hồ hoặc cần bổ sung thêm các qui tắc cần thiết để giải quyết sự mơ hồ cho văn phạm.

4. Sự kết hợp của các toán tử

Thông thường, theo quy ước ta có biểu thức 9 + 5 + 2 tương đương (9 + 5) + 2 và 9 - 5 - 2 tương đương với (9 - 5) - 2. Khi một toán hạng như 5 có hai toán tử ở trái và phải thì nó phải chọn một trong hai để xử lý trước. Nếu toán tử bên trái được thực hiện trước ta gọi là **kết hợp trái**. Ngược lại là **kết hợp phải**.

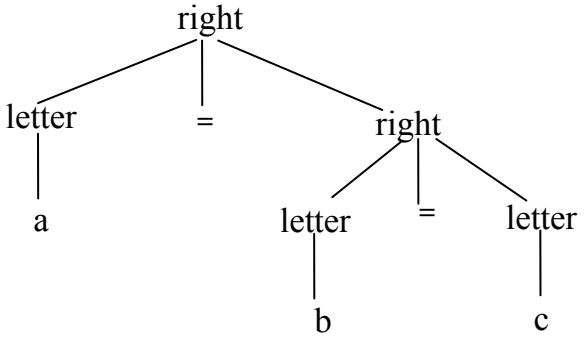
Thường thì bốn phép toán số học: +, -, *, / có tính kết hợp trái. Các phép toán như số mũ, phép gán bằng (=) có tính kết hợp phải.

Ví dụ 2.5: Trong ngôn ngữ C, biểu thức a = b = c tương đương a = (b = c) vì chuỗi a = b = c với toán tử kết hợp phải được sinh ra bởi văn phạm:

right $\rightarrow$ letter = right | letter

letter $\rightarrow$ a | b | ... | z

Ta có cây phân tích cú pháp có dạng như sau (chú ý hướng của cây nghiêng về bên phải trong khi cây cho các phép toán có kết hợp trái thường nghiêng về trái)



Hình 2.2 - *Minh họa cây phân tích cú pháp cho toán tử kết hợp phải*

5. Thứ tự ưu tiên của các toán tử

Xét biểu thức 9 + 5 * 2. Có 2 cách để diễn giải biểu thức này, đó là 9 + (5 * 2) hoặc (9 + 5) * 2. Tính kết hợp của phép + và * không giải quyết được sự mơ hồ này, vì vậy cần phải quy định một thứ tự ưu tiên giữa các loại toán tử khác nhau.

Thông thường trong toán học, các toán tử * và / có độ ưu tiên cao hơn + và -.

Cú pháp cho biểu thức :

Văn phạm cho các biểu thức số học có thể xây dựng từ bảng kết hợp và ưu tiên của các toán tử. Chúng ta có thể bắt đầu với bốn phép tính số học theo thứ bậc sau :

Kết hợp trái +, - | Thứ tự ưu tiên

Kết hợp trái *, / ↓ từ thấp đến cao

Chúng ta tạo hai ký hiệu chưa kết thúc expr và term cho hai mức ưu tiên và một ký hiệu chưa kết thúc factor làm đơn vị phát sinh cơ sở của biểu thức. Ta có đơn vị cơ bản trong biểu thức là số hoặc biểu thức trong dấu ngoặc.

factor → digit | (expr)

Phép nhân và chia có thứ tự ưu tiên cao hơn đồng thời chúng kết hợp trái nên luật sinh cho term tương tự như cho list :

term → term * factor | term / factor | factor

Tương tự, ta có luật sinh cho expr :

expr → expr + term | expr - term | term

Vậy, cuối cùng ta thu được văn phạm cho biểu thức như sau :

expr → expr + term | expr - term | term

term → term * factor | term / factor | factor

factor → digit | (expr)

Như vậy: Văn phạm này xem biểu thức như là một danh sách các term được phân cách nhau bởi dấu + hoặc -. Term là một list các factor phân cách nhau bởi * hoặc /. Chú ý rằng bất kỳ một biểu thức nào trong ngoặc đều là factor, vì thế với các dấu ngoặc chúng ta có thể xây dựng các biểu thức lồng sâu nhiều cấp tuỳ ý.

Cú pháp các câu lệnh:

Từ khóa (keyword) cho phép chúng ta nhận ra câu lệnh trong hầu hết các ngôn ngữ. Ví dụ trong Pascal, hầu hết các lệnh đều bắt đầu bởi một từ khóa ngoại trừ lệnh gán. Một số lệnh Pascal được định nghĩa bởi văn phạm (mơ hồ) sau, trong đó id chỉ một danh biểu (tên biến).

stmt → id := expr
| if expr then stmt
| if expr then stmt else stmt
| while expr do stmt
| begin opt_stmts end

Ký hiệu chưa kết thúc opt_stmts sinh ra một danh sách có thể rỗng các lệnh, phân cách nhau bởi dấu chấm phẩy (;)

II. DỊCH TRỰC TIẾP CÚ PHÁP (Syntax - Directed Translation)

Để dịch một kết cấu ngôn ngữ lập trình, trong quá trình dịch, bộ biên dịch cần lưu lại nhiều đại lượng khác cho việc sinh mã ngoài mã lệnh cần tạo ra cho kết cấu. Chẳng hạn nó cần biết kiểu (type) của kết cấu, địa chỉ của lệnh đầu tiên trong mã đích, số lệnh phát sinh,v.v Vì vậy ta nói một cách ảo về thuộc tính (attribute) đi kèm theo kết cấu. Một thuộc tính có thể biểu diễn cho một đại lượng bất kỳ như một kiểu, một chuỗi, một địa chỉ vùng nhớ, v.v

Chúng ta sử dụng **định nghĩa trực tiếp cú pháp (syntax - directed definition)** nhằm đặc tả việc phiên dịch các kết cấu ngôn ngữ lập trình theo các thuộc tính đi kèm

với thành phần cú pháp của nó. Chúng ta cũng sẽ sử dụng một thuật ngữ có tính thủ tục hơn là **lược đồ dịch (translation scheme)** để đặc tả quá trình dịch. Trong chương này, ta sử dụng lược đồ dịch để dịch một biểu thức trung tố thành dạng hậu tố.

1. Ký pháp hậu tố (Postfix Notation)

Ký pháp hậu tố của biểu thức E có thể được định nghĩa quy nạp như sau:

1. Nếu E là một biến hay hằng thì ký pháp hậu tố của **E** chính là **E**.

2. Nếu E là một biểu thức có dạng **E1 op E2** trong đó op là một toán tử hai ngôi thì ký pháp hậu tố của E là **E1' E2' op**. Trong đó E1', E2' tương ứng là ký pháp hậu tố của E1, E2.

3. Nếu E là một biểu thức dạng **(E1)** thì ký pháp hậu tố của E là ký pháp hậu tố của **E1**.

Trong dạng ký pháp hậu tố, dấu ngoặc là không cần thiết vì vị trí và số lượng các đối số chỉ cho phép xác định một sự giải mã duy nhất cho một biểu thức hậu tố.

Ví dụ 2.6: Dạng hậu tố của biểu thức (9 - 5) + 2 là 9 5 - 2 +

Dạng hậu tố của biểu thức 9 - (5 + 2) là 9 5 2 + -

2. Định nghĩa trực tiếp cú pháp (Syntax - Directed Definition)

Định nghĩa trực tiếp cú pháp sử dụng văn phạm phi ngữ cảnh để đặc tả cấu trúc cú pháp của dòng input nhập. Nó liên kết mỗi ký hiệu văn phạm với một tập các thuộc tính và mỗi luật sinh kết hợp với một tập các quy tắc ngữ nghĩa (semantic rule) để tính giá trị của thuộc tính đi kèm với những ký hiệu có trong luật sinh văn phạm. Văn phạm và tập các quy tắc ngữ nghĩa tạo nên định nghĩa trực tiếp cú pháp.

Phiên dịch (translation) là một ánh xạ giữa input - output (input - output mapping). Output cho mỗi input x được xác định theo cách sau. Trước hết xây dựng cây phân tích cú pháp cho x. Giả sử nút n trong cây phân tích cú pháp có nhãn là ký hiệu văn phạm X. Ta viết X.a để chỉ giá trị của thuộc tính a của X tại nút đó. Giá trị của X.a tại n được tính bằng cách sử dụng quy tắc ngữ nghĩa cho thuộc tính a kết hợp với luật sinh cho X tại nút n. Cây phân tích cú pháp có thể hiện rõ giá trị của thuộc tính tại mỗi nút gọi là cây phân tích cú pháp chú thích (annotated parse tree).

3. Thuộc tính tổng hợp (Synthesized Attributes)

Một thuộc tính được gọi là tổng hợp nếu giá trị của nó tại một nút trên cây cú pháp được xác định từ các giá trị của các thuộc tính tại các nút con của nút đó.

Ví dụ 2.7: Định nghĩa trực tiếp cú pháp cho việc dịch các biểu thức các số cách nhau bởi dấu + hoặc - thành ký pháp hậu tố như sau:

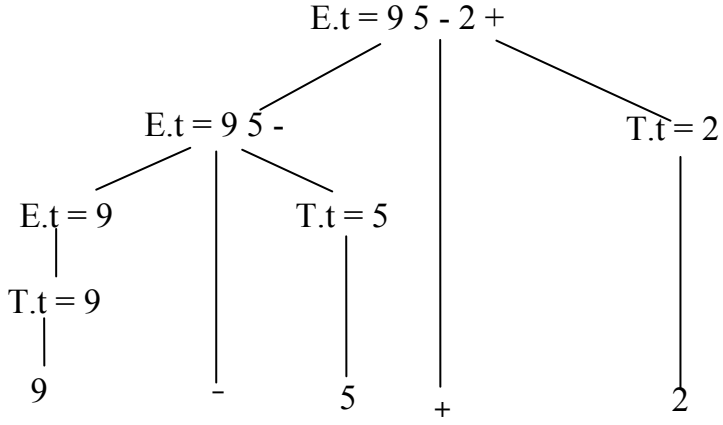
Luật sinh	**Quy tắc ngữ nghĩa**
$E \rightarrow E1 + T$	E.t := E1.t || T.t || '+'
$E \rightarrow E1 - T$	E.t := E1.t || T.t || '-'
$E \rightarrow T$	E.t := T.t
$T \rightarrow 0$	T.t := '0'
...	...

$T \rightarrow 9$	T.t := '9'

Hình 2.3 - *Ví dụ về định nghĩa trực tiếp cú pháp*

Chẳng hạn, một quy tắc ngữ nghĩa E.t := E1.t || T.t || '+' kết hợp với luật sinh xác định giá trị của thuộc tính E.t bằng cách ghép các ký pháp hậu tố của E1.t và T.t và dấu '+'. Dấu || có nghĩa như sự ghép các chuỗi.

Ta có cây phân tích cú pháp chú thích cho biểu thức 9 - 5 + 2 như sau :



Hình 2.4 - *Minh họa cây phân tích cú pháp chú thích*

Giá trị của thuộc tính t tại mỗi nút được tính bằng cách dùng quy tắc ngữ nghĩa kết hợp với luật sinh tại nút đó. Giá trị thuộc tính tại nút gốc là ký pháp hậu tố của chuỗi được sinh ra bởi cây phân tích cú pháp.

4. Duyệt theo chiều sâu (Depth - First Traversal)

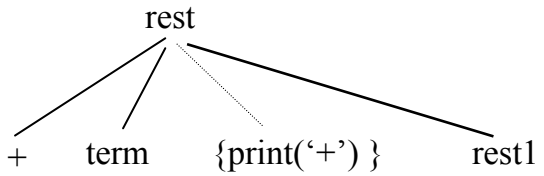
Quá trình dịch được cài đặt bằng cách đánh giá các luật ngữ nghĩa cho các thuộc tính trong cây phân tích cú pháp theo một thứ tự xác định trước. Ta dùng phép duyệt cây theo chiều sâu để đánh giá quy tắc ngữ nghĩa. Bắt đầu từ nút gốc, thăm lần lượt (đệ qui) các con của mỗi nút theo thứ tự từ trái sang phải.

```
Procedure   visit (n : node);
    begin
      for  với mỗi nút con m của n, từ trái sang phải   do
           visit (m);
      Đánh giá quy tắc ngữ nghĩa tại nút n;
    end
```

5. Lược đồ dịch (Translation Scheme)

Một lược đồ dịch là một văn phạm phi ngữ cảnh, trong đó các đoạn chương trình gọi là hành vi ngữ nghĩa (semantic actions) được gán vào vế phải của luật sinh. Lược đồ dịch cũng như định nghĩa trực tiếp cú pháp nhưng thứ tự đánh giá các quy tắc ngữ nghĩa được trình bày một cách rõ ràng. Vị trí mà tại đó một hành vi được thực hiện được trình bày trong cặp dấu ngoặc nhọn { } và viết vào vế phải luật sinh.

Ví dụ 2.8: rest $\rightarrow$ + term {print ('+')} rest1.



Hình 2.5 - *Một nút lá được xây dựng cho hành vi ngữ nghĩa*

Lược đồ dịch tạo ra một output cho mỗi câu nhập x sinh ra từ văn phạm đã cho bằng cách thực hiện các hành vi theo thứ tự mà chúng xuất hiện trong quá trình duyệt theo chiều sâu cây phân tích cú pháp của x. Chẳng hạn, xét cây phân tích cú pháp với một nút có nhãn rest biểu diễn luật sinh nói trên. Hành vi ngữ nghĩa { print('+') } được thực hiện sau khi cây con term được duyệt nhưng trước khi cây con rest1 được thăm.

6. Phát sinh bản dịch (Emitting a Translation)

Trong chương này, hành vi ngữ nghĩa trong lược đồ dịch sẽ ghi kết quả của quá trình phiên dịch vào một tập tin, mỗi lần một chuỗi hoặc một ký tự. Chẳng hạn, khi dịch 9 - 5 + 2 thành 9 5 - 2 + bằng cách ghi mỗi ký tự trong 9 - 5 + 2 đúng một lần mà không phải ghi lại quá trình dịch của các biểu thức con. Khi tạo ra output dần dần theo cách này, thứ tự in ra các ký tự sẽ rất quan trọng.

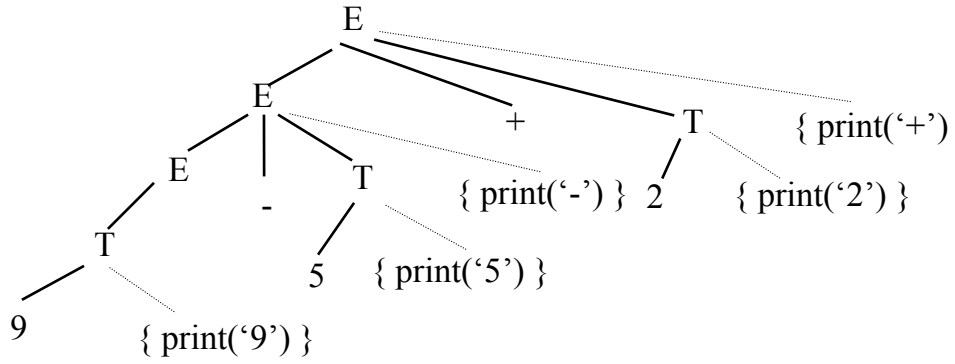
Chú ý rằng các định nghĩa trực tiếp cú pháp đều có đặc điểm sau: chuỗi biểu diễn cho bản dịch của ký hiệu chưa kết thúc ở vế trái của mỗi luật sinh là sự ghép nối của các bản dịch ở vế phải theo đúng thứ tự của chúng trong luật sinh và có thể thêm một số chuỗi khác xen vào giữa. Một định nghĩa trực tiếp cú pháp theo dạng này được xem là đơn giản.

Ví dụ 2.9: Với định nghĩa trực tiếp cú pháp như hình 2.3, ta xây dựng lược đồ dịch như sau :

$E \rightarrow E1 + T$	{ print ('+') }
$E \rightarrow E1 - T$	{ print ('-') }
$E \rightarrow T$	
$T \rightarrow 0$	{ print ('0') }
....	
$T \rightarrow 9$	{ print ('9') }

Hình 2.6 - *Lược đồ dịch biểu thức trung tố thành hậu tố*

Ta có các hành động dịch biểu thức 9 - 5 + 2 thành 9 5 - 2 + như sau :



Hình 2.7 - *Các hành động dịch biểu thức 9-5+2 thành 9 5- 2 +*

Xem như một quy tắc tổng quát, phần lớn các phương pháp phân tích cú pháp đều xử lý input của chúng từ trái sang phải, trong lược đồ dịch đơn giản (lược đồ dịch dẫn xuất từ một định nghĩa trực tiếp cú pháp đơn giản), các hành vi ngữ nghĩa cũng được thực hiện từ trái sang phải. Vì thế, để cài đặt một lược đồ dịch đơn giản, chúng ta có thể thực hiện các hành vi ngữ nghĩa trong lúc phân tích cú pháp mà không nhất thiết phải xây dựng cây phân tích cú pháp.

III. PHÂN TÍCH CÚ PHÁP (PARSING)

Phân tích cú pháp là quá trình xác định xem liệu một chuỗi ký hiệu kết thúc (token) có thể được sinh ra từ một văn phạm hay không ? Khi nói về vấn đề này, chúng ta xem như đang xây dựng một cây phân tích cú pháp, mặc dù một trình biên dịch có thể không xây dựng một cây như thế. Tuy nhiên, quá trình phân tích cú pháp (parse) phải có khả năng xây dựng nó, nếu không thì việc phiên dịch sẽ không bảo đảm được tính đúng đắn.

Phần lớn các phương pháp phân tích cú pháp đều rơi vào một trong 2 lớp: phương pháp phân tích từ trên xuống và phương pháp phân tích từ dưới lên. Những thuật ngữ này muốn đề cập đến thứ tự xây dựng các nút trong cây phân tích cú pháp. Trong phương pháp đầu, quá trình xây dựng bắt đầu từ gốc tiến hành hướng xuống các nút lá, còn trong phương pháp sau thì thực hiện từ các nút lá hướng về gốc. Phương pháp phân tích từ trên xuống thông dụng hơn nhờ vào tính hiệu quả của nó khi xây dựng theo lối thủ công. Ngược lại, phương pháp phân tích từ dưới lên lại có thể xử lý được một lớp văn phạm và lược đồ dịch phong phú hơn. Vì vậy, đa số các công cụ phần mềm giúp xây dựng thể phân tích cú pháp một cách trực tiếp từ văn phạm đều có xu hướng sử dụng phương pháp từ dưới lên.

1. Phân tích cú pháp từ trên xuống (Top - Down Parsing)

Xét văn phạm sinh ra một tập con các kiểu dữ liệu của Pascal

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

simple → **integer** | **char** | **num .. num**

Phân tích trên xuống bắt đầu bởi nút gốc, nhãn là ký hiệu chưa kết thúc bắt đầu và lặp lại việc thực hiện hai bước sau đây:

1. Tại nút n, nhãn là ký hiệu chưa kết thúc A, chọn một trong những luật sinh của A và xây dựng các con của n cho các ký hiệu trong vế phải của luật sinh.

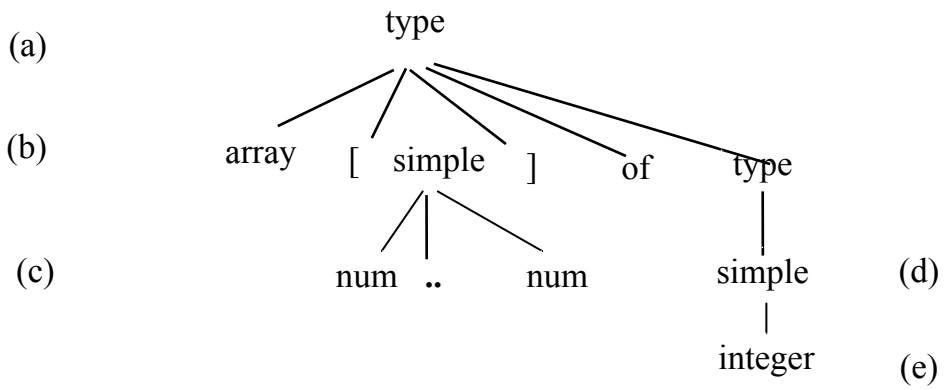
2. Tìm nút kế tiếp mà tại đó một cây con sẽ được xây dựng. Đối với một số văn phạm, các bước trên được cài đặt bằng một phép quét (scan) dòng nhập từ trái qua phải.

Ví dụ 2.10: Với các luật sinh của văn phạm trên, ta xây dựng cây cú pháp cho dòng nhập: **array [num .. num] of integer**

Mở đầu ta xây dựng nút gốc với nhãn **type**. Để xây dựng các nút con của type ta chọn luật sinh **type → array [simple] of type**. Các ký hiệu nằm bên phải của luật sinh này là **array, [, simple,], of, type** do đó nút gốc **type** có 6 con có nhãn tương ứng (áp dụng bước 1)

Trong các nút con của type, từ trái qua thì nút con có nhãn **simple** (một ký hiệu chưa kết thúc) do đó có thể xây dựng một cây con tại nút simple (bước 2)

Trong các luật sinh có vế trái là simple, ta chọn luật sinh **simple → num .. num** để xây dựng. Nói chung, việc chọn một luật sinh có thể được xem như một quá trình **thử và sai** (trial - and - error). Nghĩa là một luật sinh được chọn để thử và sau đó quay lại để thử một luật sinh khác nếu luật sinh ban đầu không phù hợp. Một luật sinh là không phù hợp nếu sau khi sử dụng luật sinh này chúng ta không thể xây dựng một cây hợp với dòng nhập. Để tránh việc lần ngược, người ta đưa ra một phương pháp gọi là phương pháp phân tích cú pháp dự đoán.



Hình 2.8 - *Minh họa quá trình phân tích cú pháp từ trên xuống*

2. Phân tích cú pháp dự đoán (Predictive Parsing)

Phương pháp phân tích cú pháp đệ qui xuống (recursive-descent parsing) là một phương pháp phân tích trên xuống, trong đó chúng ta thực hiện một loạt thủ tục đệ qui để xử lý chuỗi nhập. Mỗi một thủ tục kết hợp với một ký hiệu chưa kết thúc của văn phạm. Ở đây chúng ta xét một trường hợp đặc biệt của phương pháp đệ qui xuống là phương pháp phân tích dự đoán trong đó ký hiệu dò tìm sẽ xác định thủ tục được chọn đối với ký hiệu chưa kết thúc. Chuỗi các thủ tục được gọi trong quá trình xử lý chuỗi nhập sẽ tạo ra cây phân tích cú pháp.

Ví dụ 2.11: Xét văn phạm như trên, ta viết các thủ tục type và simple tương ứng với các ký hiệu chưa kết thúc type và simple trong văn phạm. Ta còn đưa thêm thủ tục match để đơn giản hóa đoạn mã cho hai thủ tục trên, nó sẽ dịch tới ký hiệu kế tiếp nếu tham số t của nó so khớp với ký hiệu dò tìm tại đầu đọc (lookahead).

```
procedure  match (t: token);
    begin
        if  lookahead = t  then
            lookahead := nexttoken
        else  error
    end;
procedure  type;
    begin
        if  lookahead  in [integer, char, num]    then
            simple
        else  if  lookahead = '↑'  then   begin
```

```
                match (‘↑‘);   match(id);
            end
        else if lookahead = array then begin
                match(array);   match(‘[‘);
                simple;
                match(‘]’);   match(of);
                type
            end
        else error;
    end;
procedure simple;
    begin
        if lookahead = integer then    match(integer)
        else if lookahead = char then    match(char)
        else if lookahead = num then
            begin
                match(num);  match(dotdot);  match(num);
            end
        else error
    end;
```

Hình 2.9 - *Đoạn mã giả minh họa phương pháp phân tích dự đoán*

Phân tích cú pháp bắt đầu bằng lời gọi tới thủ tục cho ký hiệu bắt đầu type. Với dòng nhập *array [num .. num] of integer* thì đầu đọc lookahead bắt đầu sẽ đọc token *array*. Thủ tục **type** sau đó sẽ thực hiện chuỗi lệnh: *match(array); match(‘[‘); simple; match(‘]’); match(of); type*. Sau khi đã đọc được **array** và **[** thì ký hiệu hiện tại là **num**. Tại điểm này thì thủ tục **simple** và các lệnh *match(num); match(dotdot); match(num)* được thực hiện.

Xét luật sinh **type** $\rightarrow$ **simple**. Luật sinh này có thể được dùng khi ký hiệu dò tìm sinh ra bởi **simple**, chẳng hạn ký hiệu dò tìm là *integer* mặc dù trong văn phạm không có luật sinh **type** $\rightarrow$ **integer**, nhưng có luật sinh **simple** $\rightarrow$ **integer**, do đó luật sinh **type** $\rightarrow$ **simple** được dùng bằng cách trong **type** gọi **simple**.

Phân tích dự đoán dựa vào thông tin về các ký hiệu đầu sinh ra bởi vế phải của một luật sinh. Nói chính xác hơn, giả sử ta có luật sinh $A \rightarrow \gamma$, ta định nghĩa tập hợp :

FIRST(γ) = { token | xuất hiện như các ký hiệu đầu của một hoặc nhiều chuỗi sinh ra bởi γ }. Nếu γ là ε hoặc có thể sinh ra ε thì $\varepsilon \in$ FIRST(γ).

Ví dụ 2.12: Xét văn phạm như trên, ta dễ dàng xác định:

FIRST(simple) = { integer, char, num }

FIRST(↑id) = { ↑ }

FIRST(array [simple] of type) = { array }

Nếu ta có $A \rightarrow \alpha$ và $A \rightarrow \beta$, phân tích đệ qui xuống sẽ không phải quay lui nếu $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$. Nếu ký hiệu dò tìm thuộc $FIRST(\alpha)$ thì $A \rightarrow \alpha$ được dùng. Ngược lại, nếu ký hiệu dò tìm thuộc $FIRST(\beta)$ thì $A \rightarrow \beta$ được dùng.

Trường hợp $\alpha = \varepsilon$ (**Luật sinh ε**)

Ví dụ 2.13: Xét văn phạm chứa các luật sinh sau :

stmt → **begin** opt_stmts **end**

opt_stmts → stmt_list | ε

Khi phân tích cú pháp cho opt_stmts, nếu ký hiệu dò tìm $\notin$ FIRST(stmt_list) thì sử dụng luật sinh: **opt_stmts → ε**. Chọn lựa này hòan tòan chính xác nếu ký hiệu dò tìm là **end**, mọi ký hiệu dò tìm khác **end** sẽ gây ra lỗi và được phát hiện trong khi phân tích stmt.

3. Thiết kế bộ phân tích cú pháp dự đoán

Bộ phân tích dự đoán là một chương trình bao gồm các thủ tục tương ứng với các ký hiệu chưa kết thúc. Mỗi thủ tục sẽ thực hiện hai công việc sau:

1. Luật sinh mà vế phải α của nó sẽ được dùng nếu ký hiệu dò tìm thuộc FIRST(α). Nếu có một sự đụng độ giữa hai vế phải đối với bất kỳ một ký hiệu dò tìm nào thì không thể dùng phương pháp này. Một luật sinh với ε nằm bên vế phải được dùng nếu ký hiệu dò tìm không thuộc tập hợp FIRST của bất kỳ vế phải nào khác.

2. Một ký hiệu chưa kết thúc tương ứng lời gọi thủ tục, một token phải phù hợp với ký hiệu dò tìm. Nếu token không phù hợp với ký hiệu dò tìm thì có lỗi.

4. Loại bỏ đệ qui trái

Một bộ phân tích cú pháp đệ quy xuống có thể sẽ dẫn đến một vòng lặp vô tận nếu gặp một luật sinh đệ qui trái dạng **E → E + T** bởi vì ký hiệu trái nhất bên vế phải cũng giống như ký hiệu chưa kết thúc bên vế trái của luật sinh.

Để giải quyết được vấn đề này chúng ta phải loại bỏ đệ qui trái bằng cách thêm vào một ký hiệu chưa kết thúc mới. Chẳng hạn với luật sinh dạng $A \rightarrow A\alpha \mid \beta$.Ta thêm vào một ký hiệu chưa kết thúc R để viết lại thành tập luật sinh như sau :

$A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \varepsilon$

Ví dụ 2.14: Xét luật sinh đệ quy trái : $E \rightarrow E + T \mid T$

Sử dụng quy tắc khử đệ quy trái nói trên với : $A \cong E$, $\alpha \cong + T$, $\beta \cong T$.

Luật sinh trên có thể biến đổi tương đương thành tập luật sinh :

$E \rightarrow T R$

$R \rightarrow + T R \mid \varepsilon$

IV. MỘT CHƯƠNG TRÌNH DỊCH BIỂU THỨC ĐƠN GIẢN

Sử dụng các kỹ thuật nêu trên, chúng ta xây dựng một bộ dịch trực tiếp cú pháp mà nó dịch một biểu thức số học đơn giản từ trung tố sang hậu tố. Ta bắt đầu với các biểu thức là các chữ số viết cách nhau bởi + hoặc -.

Xét lược đồ dịch cho dạng biểu thức này :

expr → expr + term { print ('+') }

expr → expr - term { print ('-') }

expr → term

term → 0 { print ('0') }

...

term → 9 { print ('9') }

Hình 2.10 - *Đặc tả lược đồ dịch khởi đầu*

Văn phạm nền tảng cho lược đồ dịch trên có chứa luật sinh đệ qui trái, bộ phân tích cú pháp dự đoán không xử lý được văn phạm dạng này, cho nên ta cần loại bỏ đệ quy trái bằng cách đưa vào một ký hiệu chưa kết thúc mới rest để được văn phạm thích hợp như sau:

expr → term rest

rest → + term { print('+') } rest | - term {print('-') rest | ε

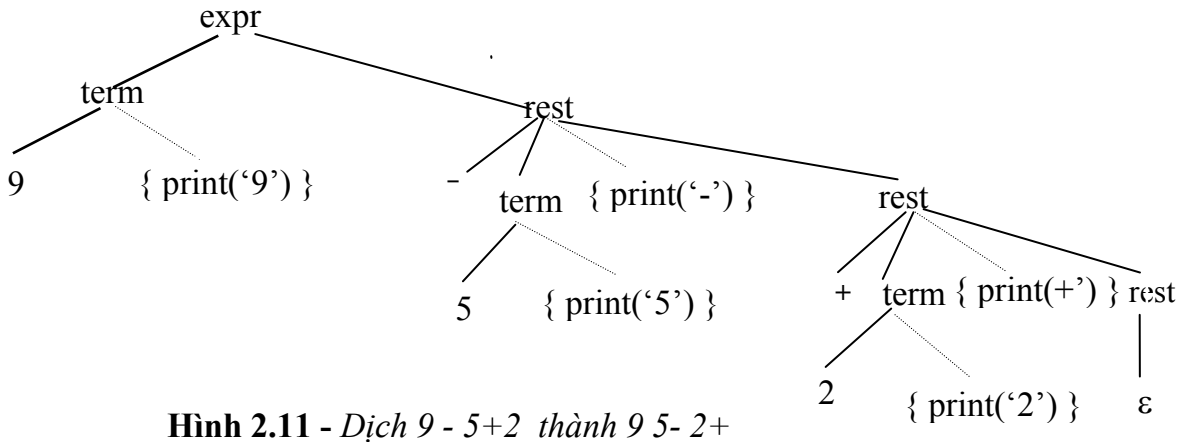
term → 0 { print('0') }

term → 1 { print('1') }

...

term → 9 { print('9') }

Hình sau đây mô tả quá trình dịch biểu thức 9 - 5 + 2 dựa vào lược đồ dịch trên:



Hình 2.11 - *Dịch 9 - 5+2 thành 9 5- 2+*

Bây giờ ta cài đặt chương trình dịch bằng C theo đặc tả như trên. Phần chính của chương trình này là các đoạn mã C cho các hàm expr, term và rest.

// Hàm expr( ) tương ứng với ký hiệu chưa kết thúc expr

expr( )

```
{
    term( ) ; rest( );
}

// Hàm expr( ) tương ứng với ký hiệu chưa kết thúc expr
rest( )
{
    if (lookahead = = '+' ) {
        match('+') ; term( ) ; putchar ('+ ') ; rest( );
    }
    else if (lookahead = = '-') {
        match('-') ; term( ) ; putchar ('-') ; rest( );
        }
      else ;
}

// Hàm expr( ) tương ứng với ký hiệu chưa kết thúc expr
term( )
{
    if (isdigit (lookahead) {
        putchar (lookahead); match (lookahead);
    }
    else error( );
}
```

Tối ưu hóa chương trình dịch

Một số lời gọi đệ quy có thể được thay thế bằng các vòng lặp để làm cho chương trình thực hiện nhanh hơn. Đoạn mã cho rest có thể được viết lại như sau :

```
rest( )
{
L : if (lookahead = = '+' ) {
        match('+') ; term( ) ; putchar ('+ ') ; goto L;
    }
    else if (lookahead = = '-') {
        match('-') ; term( ) ; putchar ('-') ; goto L;
```

```
            }
        else ;
    }
```

Nhờ sự thay thế này, hai hàm rest và expr có thể được tích hợp lại thành một.

Mặt khác, trong C, một câu lệnh stmt có thể được thực hiện lặp đi lặp lại bằng cách viết : while (1) stmt với 1 là điều kiện hằng đúng. Chúng ta cũng có thể thóat khỏi vòng lặp dễ dàng bằng lệnh break.

Đoạn chương trình có thể được viết lại như sau :

```
expr ( )
  {
     term ( )
     while (1)
        if (lookahead = = '+' ) {
           match('+') ; term( ) ; putchar ('+ ') ;
     }
     else if (lookahead = = '-') {
           match('-') ; term( ) ; putchar ('-') ;
     }
     else break;
  }
```

Chương trình C dịch biểu thức trung tố sang hậu tố

Chương trình nguồn C hoàn chỉnh cho chương trình dịch có mã như sau :

```
# include< ctype.h>                    /* nạp tập tin chứa isdigit vào*/
int   lookahead;

main ( )
{
   lookahead = getchar( );
   expr( ) ;  putchar(' \n');          /* thêm vào ký tự xuống hàng */
}

expr( )
{
 term( );
 while(1)
```

```
    if  (lookahead = = '+')
        {  match('+'); term( ); putchar('+ ');  }
    else  if   (lookahead = = '-' )
        {  match('-'); term( ); putchar('-');   }
    else    break;
 }

 term( )
{
   if  (isdigit(lookahead))
        {  putchar(lookahead);   match(lookahead);  }
   else    error( );
}

match ( int t)
{
   if   (lookahead = = t)
        lookahead = getchar();
   else   error( );
}

error( )
{
   printf ("syntax error \n");          /* in ra thông báo lỗi */
    exit(1);                            /* rồi kết thúc */
 }
```

V. PHÂN TÍCH TỪ VỰNG (Lexical Analysis)

Bây giờ chúng ta thêm vào phần trước trình biên dịch một bộ phân tích từ vựng để đọc và biến đổi dòng nhập thành một chuỗi các từ tố (token) mà bộ phân tích cú pháp có thể sử dụng được. Nhắc lại rằng một chuỗi các ký tự hợp thành một token gọi là trị từ vựng (lexeme) của token đó.

Trước hết ta trình bày một số chức năng cần thiết của bộ phân tích từ vựng.

1. Loại bỏ các khoảng trắng và các dòng chú thích

Quá trình dịch sẽ xem xét tất cả các ký tự trong dòng nhập nên những ký tự không có nghĩa (như khoảng trắng bao gồm ký tự trống (blanks), ký tự tabs, ký tự newlines)

hoặc các dòng chú thích (comment) phải bị bỏ qua. Khi bộ phân tích từ vựng đã bỏ qua các khoảng trắng này thì bộ phân tích cú pháp không bao giờ xem xét đến chúng nữa. Chọn lựa cách sửa đổi văn phạm để đưa cả khoảng trắng vào trong cú pháp thì hầu như rất khó cài đặt.

2. Xử lý các hằng

Bất cứ khi nào một ký tự số xuất hiện trong biểu thức thì nó được xem như là một hằng số. Bởi vì một hằng số nguyên là một dãy các chữ số nên nó có thể được cho bởi luật sinh văn phạm hoặc tạo ra một token cho hằng số đó. Bộ phân tích từ vựng có nhiệm vụ ghép các chữ số để được một số và sử dụng nó như một đơn vị trong suốt quá trình dịch.

Đặt **num** là một token biểu diễn cho một số nguyên. Khi một chuỗi các chữ số xuất hiện trong dòng nhập thì bộ phân tích từ vựng sẽ gửi **num** cho bộ phân tích cú pháp. Giá trị của số nguyên được chuyển cho bộ phân tích cú pháp như là một thuộc tính của token **num**. Về mặt logic, bộ phân tích từ vựng sẽ chuyển cả token và các thuộc tính cho bộ phân tích cú pháp. Nếu ta viết một token và thuộc tính thành một bộ nằm giữa < > thì dòng nhập 31 + 28 + 59 sẽ được chuyển thành một dãy các bộ :

<**num**, 31>, < +, >, <**num**, 28>, < +, >, <**num**, 59>.

Bộ <+, > cho thấy thuộc tính của + không có vai trò gì trong khi phân tích cú pháp nhưng nó cần thiết dùng đến trong quá trình dịch.

3. Nhận dạng các danh biểu và từ khóa

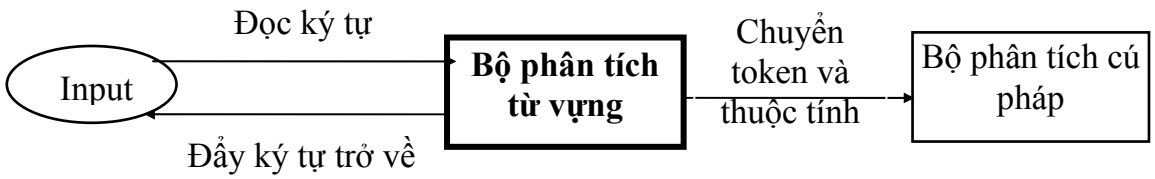
Ngôn ngữ dùng các danh biểu (identifier) như là tên biến, mảng, hàm và văn phạm xử lý các danh biểu này như là một token. Người ta dùng token id cho các danh biểu khác nhau do đó nếu ta có dòng nhập *count = count + increment*; thì bộ phân tích từ vựng sẽ chuyển cho bộ phân tích cú pháp chuỗi token: **id = id + id** (cần phân biệt token và trị từ vựng lexeme của nó: token id nhưng trị từ vựng (lexeme) có thể là *count* hoặc *increment*). Khi một lexeme thể hiện cho một danh biểu được tìm thấy trong dòng nhập cần phải có một cơ chế để xác định xem lexeme này đã được thấy trước đó chưa? Công việc này được thực hiện nhờ sự lưu trữ trợ giúp của *bảng ký hiệu* (symbol table) đã nêu ở chương trước. Trị từ vựng được lưu trong bảng ký hiệu và một con trỏ chỉ đến mục ghi trong bảng trở thành một thuộc tính của token **id**.

Nhiều ngôn ngữ cũng sử dụng các chuỗi ký tự cố định như **begin, end, if,** ... để xác định một số kết cấu. Các chuỗi ký tự này được gọi là từ khóa (keyword). Các từ khóa cũng thỏa mãn qui luật hình thành danh biểu, do vậy cần qui ước rằng một chuỗi ký tự được xác định là một danh biểu khi nó không phải là từ khóa.

Một vấn đề nữa cần quan tâm là vấn đề tách ra một token trong trường hợp một ký tự có thể xuất hiện trong trị từ vựng của nhiều token. Ví dụ một số các token là các toán tử quan hệ trong Pascal như : <, < =, < >.

4. Giao diện của bộ phân tích từ vựng

Bộ phân tích từ vựng được đặt xen giữa dòng nhập và bộ phân tích cú pháp nên giao diện với hai bộ này như sau:



Hình 2.12 - *Giao diện của bộ phân tích từ vựng*

Bộ phân tích từ vựng đọc từng ký tự từ dòng nhập, nhóm chúng lại thành các trị từ vựng và chuyển các token xác định bởi trị từ vựng này cùng với các thuộc tính của nó đến những giai đoạn sau của trình biên dịch. Trong một vài tình huống, bộ phân tích từ vựng phải đọc vượt trước một số ký tự mới xác định được một token để chuyển cho bộ phân tích cú pháp. Ví dụ, trong Pascal khi gặp ký tự >, phải đọc thêm một ký tự sau đó nữa; nếu ký tự sau là = thì token được xác định là "lớn hơn hoặc bằng", ngược lại thì token là "lớn hơn" và do đó một ký tự đã bị đọc quá. Trong trường hợp đó thì ký tự đọc quá này phải được đẩy trả về (push back) cho dòng nhập vì nó có thể là ký tự bắt đầu cho một trị từ vựng mới.

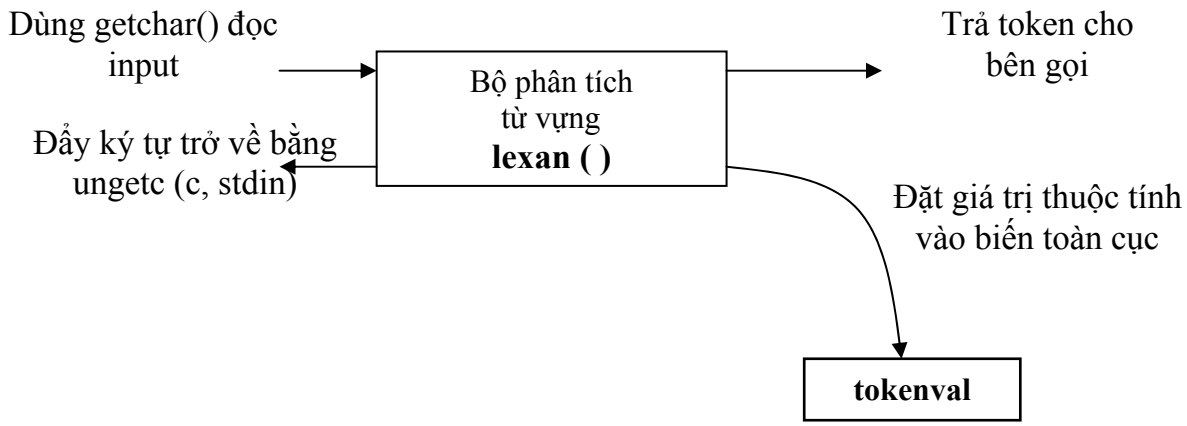
Bộ phân tích từ vựng và bộ phân tích cú pháp tạo thành một cặp "nhà sản xuất - người tiêu dùng" (producer - consumer). Bộ phân tích từ vựng sản sinh ra các token và bộ phân tích cú pháp tiêu thụ chúng. Các token được sản xuất bởi bộ phân tích từ vựng sẽ được lưu trong một bộ đệm (buffer) cho đến khi chúng được tiêu thụ bởi bộ phân tích cú pháp. Bộ phân tích từ vựng không thể hoạt động tiếp nếu buffer bị đầy và bộ phân tích cú pháp không thể hoạt động nữa nếu buffer rỗng. Thông thường, buffer chỉ lưu giữ một token. Để cài đặt tương tác dễ dàng, người ta tạo ra một thủ tục phân tích từ vựng được gọi từ trong thủ tục phân tích cú pháp, mỗi lần gọi trả về một token.

Việc đọc và quay lui ký tự cũng được cài đặt bằng cách dùng một bộ đệm nhập. Một khối các ký tự được đọc vào trong buffer nhập tại một thời điểm nào đó, một con trỏ sẽ giữ vị trí đã được phân tích. Quay lui ký tự được thực hiện bằng cách lùi con trỏ trở về. Các ký tự trong dòng nhập cũng có thể cần được lưu lại cho công việc ghi nhận lỗi bởi vì cần phải chỉ ra vị trí lỗi trong đoạn chương trình.

Để tránh việc phải quay lui, một số trình biên dịch sử dụng cơ chế đọc trước một ký tự rồi mới gọi đến bộ phân tích từ vựng. Bộ phân tích từ vựng sẽ đọc tiếp các ký tự cho đến khi đọc tới ký tự mở đầu cho một token khác. Trị từ vựng của token trước đó sẽ bao gồm các ký tự từ ký tự đọc trước đến ký tự vừa ngay ký tự vừa đọc được. Ký tự vừa đọc được sẽ là ký tự mở đầu cho trị từ vựng của token sau. Với cơ chế này thì mỗi ký tự chỉ được đọc duy nhất một lần.

5. Một bộ phân tích từ vựng

Bây giờ chúng ta xây dựng một bộ phân tích từ vựng cho chương trình dịch các biểu thức số học. Hình sau đây gợi ý một cách cài đặt giao diện của bộ phân tích từ vựng được viết bằng C dưới dạng hàm lexan. Lexan đọc và đẩy các ký tự trong input trở về bằng cách gọi thủ tục getchar va ungetc.



Hình 2.13 - *Cài đặt giao diện của bộ phân tích từ vựng*

Nếu ngôn ngữ cài đặt không cho phép trả về các cấu trúc dữ liệu từ các hàm thì token và các thuộc tính của nó phải được truyền riêng rẽ. Hàm lexan trả về một số nguyên mã hóa cho một token. Token cho một ký tự có thể là một số nguyên quy ước được dùng để mã hóa cho ký tự đó. Một token như num có thể được mã hóa bằng một số nguyên lớn hơn mọi số nguyên được dùng để mã hóa cho các ký tự, chẳng hạn là 256. Để dễ dàng thay đổi cách mã hóa, chúng ta dùng một hằng tượng trưng NUM thay cho số nguyên mã hóa của num. Hàm lexan trả về NUM khi một dãy chữ số được tìm thấy trong input. Biến toàn cục tokenval được đặt là giá trị của chuỗi số này.

Cài đặt của hàm lexan như sau :

```
# include<stdio.h>
# include<ctype.h>
int   lineno = 1;
int   tokenval = NONE;
int   lexan ( )
{
        int   t;
        while(1)  {
            t = getchar( );
            if  ( t = = ' ' || t = = '\t')   ;          /*  loại bỏ blank và tab */
            else  if  (t = = '\n')
                        lineno = lineno + 1;
            else  if  ( isdigit (t) )       {
                    tokenval = t  -  '0';
                    t = getchar( );
                    while ( isdigit (t) )    {
                        tokenval = tokenval * 10 + t - '0';
                        t = getchar( );
```

```
                }
                ungetc (t, stdin);
                return  NUM;
            }
            else  {
                tokenval = NONE;
                return  t;
            }
        }       /* while */
    }       /* lexan */
```

VI. SỰ HÌNH THÀNH BẢNG KÝ HIỆU

Một cấu trúc dữ liệu gọi là *bảng ký hiệu* (symbol table) thường được dùng để lưu giữ thông tin về các cấu trúc của ngôn ngữ nguồn. Các thông tin này được tập hợp từ các giai đoạn phân tích của trình biên dịch và được sử dụng bởi giai đoạn tổng hợp để sinh mã đích. Ví dụ trong quá trình phân tích từ vựng, các chuỗi ký tự tạo ra một token (trị từ vựng của token) sẽ được lưu vào một mục ghi trong bảng danh biểu. Các giai đoạn sau đó có thể bổ sung thêm các thông tin về kiểu của danh biểu, cách sử dụng nó và vị trí lưu trữ. Giai đoạn sinh mã sẽ dùng thông tin này để tạo ra mã phù hợp, cho phép lưu trữ và truy xuất biến đó.

1. Giao diện của bảng ký hiệu

Các thủ tục trên bảng ký hiệu chủ yếu liên quan đến việc lưu trữ và truy xuất các trị từ vựng. Khi một trị từ vựng được lưu trữ thì token kết hợp với nó cũng được lưu. Hai thao tác sau được thực hiện trên bảng ký hiệu.

Insert (s, t): Trả về chỉ mục của một ô mới cho chuỗi s, token t.

Lookup (s): Trả về chỉ mục của ô cho chuỗi s hoặc 0 nếu chuỗi s không tồn tại.

Bộ phân tích từ vựng sử dụng thao tác tìm kiếm **lookup** để xác định xem một ô cho một trị từ vựng của một token nào đó đã tồn tại trong bảng ký hiệu hay chưa? Nếu chưa thì dùng thao tác xen vào **insert** để tạo ra một ô mới cho nó.

2. Xử lý từ khóa dành riêng

Ta cũng có thể sử dụng bảng ký hiệu nói trên để xử lý các từ khóa dành riêng (reserved keyword). Ví dụ với hai token **div** và **mod** với hai trị từ vựng tương ứng là div và mod. Chúng ta có thể khởi tạo bảng ký hiệu bởi lời gọi:

insert (“div”, **div**);

insert (“mod”, **mod**);

Sau đó lời gọi lookup (“div”) sẽ trả về token **div**, do đó “div” không thể được dùng làm danh biểu.

Với phương pháp vừa trình bày thì tập các từ khóa được lưu trữ trong bảng ký hiệu trước khi việc phân tích từ vựng diễn ra. Ta cũng có thể lưu trữ các từ khóa bên ngoài

bảng ký hiệu như là một danh sách có thứ tự của các từ khóa. Trong quá trình phân tích từ vựng, khi một trị từ vựng được xác định thì ta phải tìm (nhị phân) trong danh sách các từ khóa xem có trị từ vựng này không. Nếu có, thì trị từ vựng đó là một từ khóa, ngược lại, đó là một danh biểu và sẽ được đưa vào bảng ký hiệu.

3. Cài đặt bảng ký hiệu

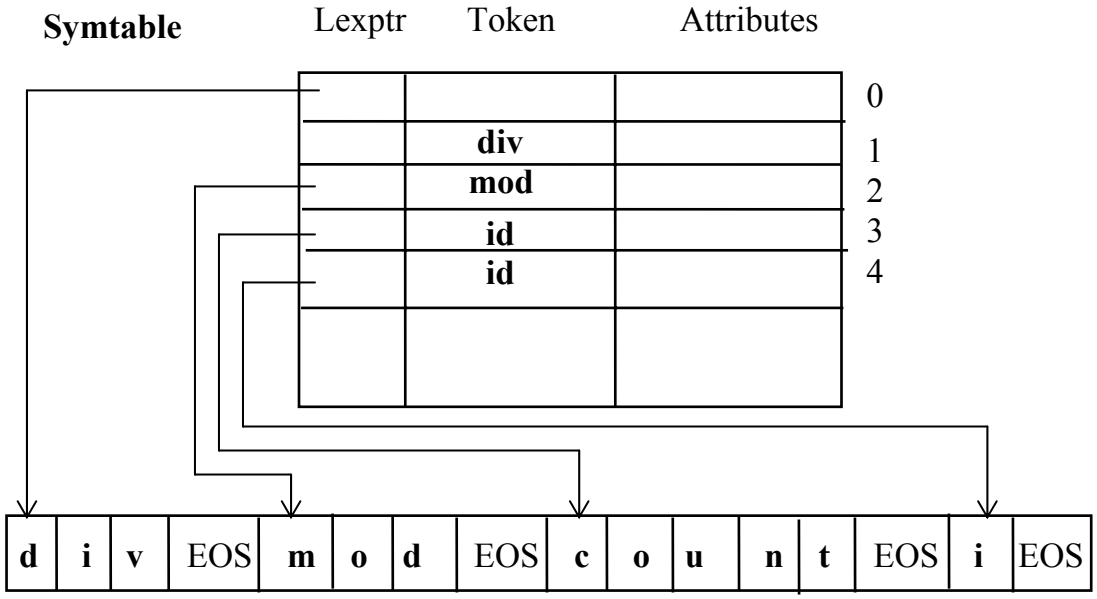
Cấu trúc dữ liệu cụ thể dùng cài đặt cho một bảng ký hiệu được trình bày trong hình dưới đây. Chúng ta không muốn dùng một lượng không gian nhớ nhất định để lưu các trị từ vựng tạo ra một danh biểu bởi vì một lượng không gian cố định có thể không đủ lớn để lưu các danh biểu rất dài và cũng rất lãng phí khi gặp một danh biểu ngắn.

Thông thường, một bảng ký hiệu gồm hai mảng :

1. Mảng **lexemes** (trị từ vựng) dùng để lưu trữ các chuỗi ký tự tạo ra một danh biểu, các chuỗi này ngăn cách nhau bởi các ký tự EOS (end - of - string).

2. Mảng **symtable** với mỗi phần tử là một mẩu tin (record) bao gồm hai trường, trường con trỏ **lexptr** trỏ tới đầu trị từ vựng và trường **token**. Cũng có thể dùng thêm các trường khác để lưu trữ giá trị các thuộc tính.

Mục ghi thứ zero trong mảng symtable phải được để trống bởi vì giá trị trả về của hàm lookup trong trường hợp không tìm thấy ô tương ứng cho chuỗi ký hiệu.



Hình 2.14 - *Bảng ký hiệu và mảng để lưu các chuỗi*

Trong hình trên, ô thứ nhất và thứ hai trong bảng ký hiệu dành cho các từ khóa **div** và **mod.** Ô thứ ba và thứ tư dành cho các danh biểu **count** và **i**.

Đoạn mã (ngôn ngữ giả) cho bộ phân tích từ vựng được dùng để xử lý các danh biểu như sau. Nó xử lý khoảng trắng và hằng số nguyên cũng giống như thủ tục đã nói ở phần trước. Khi bộ phân tích từ vựng đọc vào một chữ cái, nó bắt đầu lưu các chữ cái và chữ số vào trong vùng đệm lexbuf. Chuỗi được tập hợp trong lexbuf sau đó được tìm trong mảng symtable của bảng ký hiệu bằng cách dùng hàm **lookup**. Bởi vì bảng ký hiệu đã được khởi tạo với 2 ô cho div và mod (hình 2.14) nên nó sẽ tìm thấy

trị từ vựng này nếu lexbuf có chứa div hay mod, ngược lại nếu không có ô cho chuỗi đang chứa trong lexbuf thì hàm **lookup** sẽ trả về 0 và do đó hàm **insert** được gọi để tạo ra một ô mới trong symtable và p là chỉ số của ô trong bảng ký hiệu của chuỗi trong lexbuf. Chỉ số này được truyền tới bộ phân tích cú pháp bằng cách đặt tokenval := p và token nằm trong trường token được trả về.

Kết quả mặc nhiên là trả về số nguyên mã hóa cho ký tự dùng làm token.

```
Function   lexan: integer;
var   lexbuf: array[0..100] of char;
      c:  char
begin
   loop   begin
      đọc một ký tự vào c;
      if  c là một ký tự trống blank  hoặc ký tự tab  then
         không thực hiện điều gì ;
      else  if  c là ký tự  newline   then
         lineno = lineno + 1
      else   if  c là một ký tự số   then
         begin
            đặt tokenval là giá trị của ký số này và các ký số theo sau;
            return NUM;
         end
      else  if c là một chữ cái   then
         begin
            đặt c và các ký tự, ký số theo sau vào lexbuf;
            p := lookup (lexbuf);
            if   p = 0   then   p := insert (lexbuf, id);
            tokenval := p;
            return trường token của ô có chỉ mục p;
         end
      else
         begin       /* token là một ký tự đơn */
            đặt tokenval là NONE;       /* không có thuộc tính */
            return số nguyên mã hóa của ký tự  c;
         end;
   end;
```

end;

VII. MÁY ẢO KIỂU STACK

Ta đã biết rằng kết quả của giai đoạn phân tích là một biểu diễn trung gian của chương trình nguồn mà giai đoạn tổng hợp sử dụng nó để phát sinh mã đích. Một dạng phổ biến của biểu diễn trung gian là mã của một máy ảo kiểu Stack (abstact stack machine - ASM).

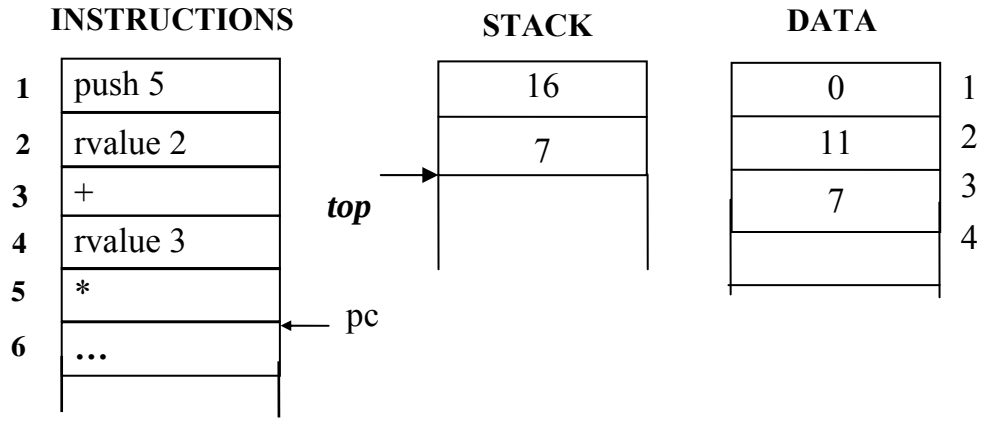
Trong phần này, chúng ta sẽ trình bày khái quát về một máy ảo kiểu Stack và chỉ ra cách sinh mã chương trình cho nó. Máy ảo này bao gồm 3 thành phần:

1. Vùng nhớ chỉ thị (instructions): là nơi chứa các chỉ thị. Các chỉ thị này rất hạn chế và được chia thành 3 nhóm chính: nhóm chỉ thị số học trên số nguyên, nhóm chỉ thị thao tác trên Stack và nhóm chỉ thị điều khiển trình tự.

2. Vùng Stack: là nơi thực hiện các chỉ thị trên các phép toán số học.

3. Vùng nhớ dữ liệu (data): là nơi lưu trữ riêng các dữ liệu.

Hình sau đây minh họa cho nguyên tắc thực hiện của dạng máy này, con trỏ pc (program counter) chỉ ra chỉ thị đang chờ để thực hiện tiếp theo. Các giá trị dùng trong quá trình tính toán được nạp vào đỉnh Stack. Sau khi tính toán xong, kết quả được lưu tại đỉnh Stack.



Hình 2.15 - *Minh họa hình ảnh một máy ảo kiểu Stack*

Ví dụ 2.15: Biểu thức (5 + b) * c với b = 11, c = 7 sẽ được thực hiện trên Stack dưới dạng biểu thức hậu tố 5 b + c *.

1. Các chỉ thị số học

Máy ảo phải cài đặt mỗi toán tử bằng một ngôn ngữ trung gian Khi gặp các chỉ thị số học đơn giản, máy sẽ thực hiện phép toán tương ứng với hai giá trị trên đỉnh Stack, kết quả cũng được lưu vào đỉnh STACK. Một phép toán phức tạp hơn có thể cần phải được cài đặt như một loạt chỉ thị của máy.

Mã chương trình máy ảo cho một biểu thức số học sẽ mô phỏng hành động ước lượng dạng hậu tố cho biểu thức đó bằng cách sử dụng Stack. Việc ước lượng được tiến hành bằng cách xử lý chuỗi hậu tố từ trái sang phải, đẩy mỗi toán hạng vào Stack khi gặp nó. Với một toán tử k - ngôi, đối số cận trái của nó nằm ở (k -1) vị trí bên dưới đỉnh Stack và đối số cận phải nằm tại đỉnh. Hành động ước lượng áp dụng toán tử cho k giá trị trên đỉnh của Stack, lấy toán hạng ra và đặt kết quả trở lại vào Stack.

Trong ngôn ngữ trung gian, mọi giá trị đều là số nguyên; số 0 tương ứng với false và các số khác 0 tương ứng với true. Toán tử logic **and** và **or** cần phải có cả 2 đối số.

2. Chỉ thị L- value và R-value

Ta cần phân biệt ý nghĩa của các danh biểu ở vế trái và vế phải của một phép gán. Trong mỗi phép gán sau :

i := 5;

i := i +1;

vế phải xác định một giá trị nguyên, còn vế trái xác định nơi giá trị được lưu. Tương tự, nếu p và q là những con trỏ đến các ký tự dạng :

p ↑ := q ↑;

thì vế phải q↑ xác định một ký tự, còn p↑ xác định vị trí ký tự được lưu. Các thuật ngữ L-value (giá trị trái) và R-value (giá trị phải) muốn nói đến các giá trị thích hợp tương ứng ở vế trái và vế phải của một phép gán. Nghĩa là, R-value có thể được xem là 'giá trị' còn L-value chính là các địa chỉ.

L-value l : Đẩy nội dung ở vị trí dữ liệu l vào Stack

R-value l : Đẩy địa chỉ của vị trí dữ liệu l vào Stack

3. Các chỉ thị thao tác trên STACK

Bên cạnh những chỉ thị cho thao tác đẩy một hằng số nguyên vào Stack và lấy một giá trị ra khỏi đỉnh Stack, còn có một số chỉ thị truy xuất vùng nhớ dữ liệu như sau:

push v	: Đẩy giá trị v vào đỉnh Stack (top := top +1)
pop	: Lấy giá trị ra khỏi đỉnh Stack (top := top +1)
:=	: R-value trên đỉnh Stack được lưu vào L-value ngay bên dưới nó và lấy cả hai ra khỏi Stack (top := top -2)
copy	: Sao chép giá trị tại đỉnh Stack (top := top +1)

4. Dịch các biểu thức

Đoạn mã chương trình dùng để ước lượng một biểu thức trên một máy ảo kiểu Stack có liên quan mật thiết với ký pháp hậu tố cho biểu thức đó.

Ví dụ 2.16: Dịch phép gán sau thành mã máy ảo kiểu Stack:

day := (1461 * y) div 4 + (153 * m + 2) div 5 + d

Ký pháp hậu tố của biểu thức như sau :

day 1461 y * 4 div 153 m * 2 + 5 div + d + :=

Đoạn mã máy có dạng :

L-value	day	push	2
push	1461	+	
R-value	y	push	5
*		div	
push	4	+	

div		R-value	d
push	153	+	
R- value	m	:=	
*			

5. Các chỉ thị điều khiển trình tự

Máy ảo kiểu Stack thực hiện các chỉ thị theo đúng thứ tự liệt kê trừ khi được yêu cầu thực hiện khác đi bằng các câu lệnh nhảy có điều kiện hoặc không điều kiện. Có một số các tùy chọn dùng để mô tả các đích nhảy :

1. Toán hạng làm chỉ thị cho biết vị trí đích.

2. Toán hạng làm chỉ thị mô tả khoảng cách tương đối cần nhảy theo chiều tới hoặc lui.

3. Đích nhảy đến được mô tả bằng các ký hiệu tượng trưng gọi là các nhãn.

Một số chỉ thị điều khiển trình tự cho máy là :

lable l : Gán đích của các lệnh nhảy đến là l, không có tác dụng khác.

goto l : Chỉ thị tiếp theo được lấy từ câu lệnh có lable l .

gofalse l : Lấy giá trị trên đỉnh Stack ra, nếu giá trị là 0 thì nhảy đến l, ngược lại, thực hiện lệnh kế tiếp.

gotrue l : Lấy giá trị trên đỉnh Stack ra, nếu giá trị khác 0 thì nhảy đến l, ngược lại, thực hiện lệnh kế tiếp.

halt : Ngưng thực hiện chương trình.

6. Dịch các câu lệnh

Sơ đồ phác thảo đoạn mã máy ảo cho một số lệnh cấu trúc được chỉ ra trong hình sau:

IF *expr* THEN *stmt*

Code for ***expr***
Gofalse out
Code for ***stmt 1***
Lable out

WHILE *expr* DO *stmt*

Label test
Code for ***expr***
Gofalse out
Code for ***stmt 1***
Goto test
Lable out

Hình 2.16 - *Sơ đồ đoạn mã cho một số lệnh cấu trúc*

Xét sơ đồ đoạn mã cho câu lệnh If . Giả sử rằng newlable là một thủ tục trả về một

nhãn mới cho mỗi lần gọi. Trong hành vi ngữ nghĩa sau đây, nhãn được trả về bởi một lời gọi đến newlabel được ghi lại bằng cách dùng một biến cục bộ out :

```
stmt → if  expr  then  stmt1        { out := newlable;
                                      stmt.t := expr.t  ||
                                      ' gofalse ' out  ||
                                      stmt1.t  ||
                                      ' lable ' out        }
```

Thay vì in ra các câu lệnh, ta có thể sử dụng thủ tục emit để che dấu các chi tiết in. Chẳng hạn như emit phải xem xét xem mỗi chỉ thị máy ảo có cần nằm trên một hàng riêng biệt hay không. Sử dụng thủ tục emit, ta có thể viết lại như sau :

```
stmt → if
        expr          { out := newlable;  emit (' gofalse ', out); }
        then
        stmt1         { emit (' lable ', out); }
```

Khi một hành vi ngữ nghĩa xuất hiện bên trong một luật sinh, ta xét các phần tử ở vế phải của luật sinh theo thứ tự từ trái sang phải. Đoạn mã (ngôn ngữ giả) cho phép dịch phép gán và câu lệnh điều kiện If tương ứng như sau :

```
procedure   stmt;
        var   test, out: integer;        /* dùng cho các nhãn */
begin
        if  lookahead = id  then
          begin
                emit ('lvalue', tokenval);   match (id);   match (':=');   expr;
          end
        else   if  lookahead = 'if'   then
          begin
                match ('if');   expr;   out := newlable;
                emit ('gofalse', out);   match('then');   stmt;
                emit ('lable', out);
          end
                /*  đoạn mã cho các lệnh còn lại */
        else   error;
end;
```

VIII. KẾT NỐI CÁC KỸ THUẬT

Trong các phần trên, chúng ta đã trình bày một số kỹ thuật phiên dịch trực tiếp cú pháp để xây dựng kỳ đầu của trình biên dịch. Phần này sẽ thực hiện việc kết nối chúng lại bằng cách giới thiệu một chương trình C có chức năng dịch trung tố - hậu tố cho một ngôn ngữ gồm dãy các biểu thức kết thúc bằng các dấu chấm phẩy. Các biểu thức gồm có các số, danh biểu, các toán tử +, -, *, /, div và mod. Output cho chương trình là dạng biểu diễn hậu tố cho mỗi biểu thức.

1. Mô tả chương trình dịch

Chương trình dịch được thiết kế bằng cách dùng lược đồ dịch trực tiếp cú pháp có dạng như sau :

start → list **eof**

list → expr **;** list
| ε

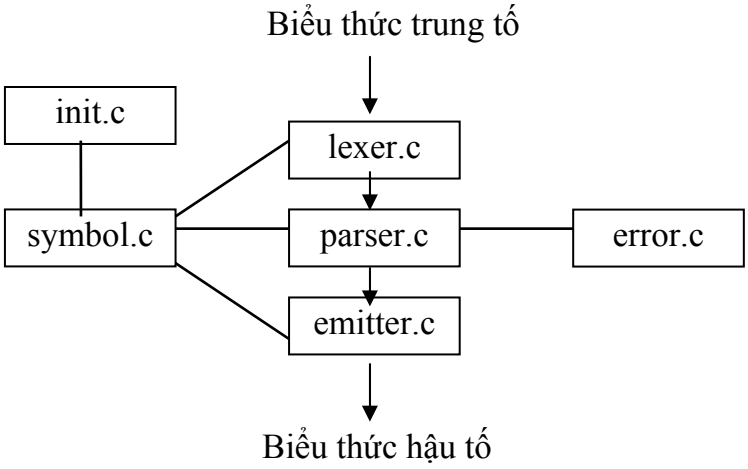
expr → expr + term { print (‘+ ’) }
| expr - term { print (‘- ’) }
| term

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

factor → (expr)
| **id** { print (**id**.lexeme) }
| **num** { print (**num**.value) }

Trong đó, token **id** biểu diễn một dãy không rỗng gồm các chữ cái và ký số bắt đầu bằng một chữ cái, **num** là dãy ký số, **eof** là ký tự cuối tập tin (end - of - file). Các token được phân cách bởi một dãy ký tự blank, tab và newline - gọi chung là các khoảng trắng (white space). Thuộc tính *lexeme* của token **id** là chuỗi ký tự tạo ra token đó, thuộc tính *value* của token **num** chứa số nguyên được biểu diễn bởi **num**.

Đoạn mã cho chương trình dịch bao gồm 7 thủ tục, mỗi thủ tục được lưu trong một tập tin riêng. Điểm bắt đầu thực thi chương trình nằm trong thủ tục chính **main.c** gồm có một lời gọi đến **init( )** để khởi gán, theo sau là một lời gọi đến **parse( )** để dịch. Các thủ tục còn lại được mô tả tổng quan như hình sau:



Hình 2.17 - *Sơ đồ các thủ tục cho chương trình dịch biểu thức*

Trước khi trình bày đoạn mã lệnh cho chương trình dịch, chúng ta mô tả sơ lược từng thủ tục và cách xây dựng chúng.

Thủ tục phân tích từ vựng lexer.c

Bộ phân tích từ vựng là một thủ tục có tên **lexan( )** được gọi từ bộ phân tích cú pháp khi cần tìm các token. Thủ tục này đọc từng ký tự trong dòng nhập, trả về token vừa xác định được cho bộ phân tích cú pháp. Giá trị của các thuộc tính đi kèm với token được gán cho biến toàn cục tokenval. Bộ phân tích cú pháp có thể nhận được các token sau : + - * / **DIV MOD ( ) ID NUM DONE**

Trị từ vựng	Token	Giá trị thuộc tính
Khoảng trắng		
Chuỗi các chữ số	NUM	Giá trị số
Div	DIV	
Mod	MOD	
Chuỗi mở đầu là chữ cái, theo sau là chữ cái hoặc chữ số	ID	Chỉ số trong symtable
Ký tự cuối tập tin - eof	DONE	
Các ký tự khác	Ký tự tương ứng	NONE

Trong đó ID biểu diễn cho một danh biểu, NUM biểu diễn cho một số và DONE là ký tự cuối tập tin eof. Các khoảng trắng đã được loại bỏ. Bảng sau trình bày các token và giá trị thuộc tính được sinh ra bởi bộ phân tích từ vựng cho mỗi token trong chương trình nguồn.

Thủ tục phân tích cú pháp parser.c

Bộ phân tích cú pháp được xây dựng theo phương pháp phân tích đệ quy xuống. Trước tiên, ta loại bỏ đệ quy trái ra khỏi lược đồ dịch bằng cách thêm vào 2 biến mới R1 cho expr và R2 cho factor, thu được lược đồ dịch mới như sau:

start → list **eof**

```
list  → expr ; list
      | ε
expr → term R1
R1   → + term   { print (' + ') }  R1
      | - term   { print (' - ') }  R1
      | ε
term → factor R2
R2   → *  factor    { print (' * ') }  R2
      | /  factor    { print (' / ') }  R2
      | DIV  factor  { print ('DIV') }  R2
      | MOD factor  { print ('MOD') }R2
      | ε
factor → ( expr )
      | id     { print  (id.lexeme) }
      | num  { print (num.value) }
```

Sau đó, chúng ta xây dựng các hàm cho các ký hiệu chưa kết thúc *expr, term* và *factor*. Hàm **parse( )** cài đặt ký hiệu bắt đầu start của văn phạm, nó gọi **lexan** mỗi khi cần một token mới. Bộ phân tích cú pháp ở giai đoạn này sử dụng hàm **emit** để sinh ra kết quả và hàm **error** để ghi nhận một lỗi cú pháp.

Thủ tục kết xuất emitter.c

Thủ tục này chỉ có một hàm **emit (t, tval)** sinh ra kết quả cho token *t* với giá trị thuộc tính *tval*.

Thủ tục quản lý bảng ký hiệu symbol.c và khởi tạo init.c

Thủ tục **symbol.c** cài đặt cấu trúc dữ liệu cho bảng danh biểu. Các ô trong mảng *symtable* là các cặp gồm một con trỏ chỉ đến mảng *lexemes* và một số nguyên biểu thị cho token được lưu tại vị trí đó.

Thủ tục **init.c** được dùng để khởi gán các từ khóa vào bảng danh biểu. Biểu diễn trị từ vựng và token cho tất cả các từ khóa được lưu trong mảng *keywords* cùng kiểu với mảng symtable. Hàm **init( )** duyệt lần lượt qua mảng *keyword*, sử dụng hàm insert để đặt các từ khóa vào bảng danh biểu.

Thủ tục lỗi error.c

Thủ tục này quản lý các ghi nhận lỗi và hết sức cần thiết. Khi gặp một lỗi cú pháp, trình biên dịch in ra một thông báo cho biết rằng một lỗi đã xảy ra trên dòng nhập hiện hành và dừng lại. Một kỹ thuật khắc phục lỗi tốt hơn có thể sẽ nhảy qua dấu chấm phẩy kế tiếp và tiếp tục phân tích câu lệnh sau đó.

2. Cài đặt chương trình nguồn

Chương trình nguồn C cài đặt chương trình dịch trên.

```
/ ****          global.h        ************************************** /

# include <stdio.h>          /* tải các thủ tục xuất nhập */
# include <ctype.h>          /* tải các thủ tục kiểm tra ký tự */

# define   BSIZE    128     /* buffer size kích thước vùng đệm */
# define   NONE     - 1
# define   EOS      ' \ 0 '

# define   NUM      256
# define   DIV      257
# define   MOD      258
# define   ID       259
# define   DONE     260

int      tokenval;           /* giá trị cuả thuôcü tính token */
int      lineno;

struct  entry   {            /* khuôn dạng cho ô trong bảng ký hiệu*/
        char    * lexptr;
        int     token;
}
struct  entry   symtable[ ]   /* bảng ký hiệu*/

/ ****          lexer.c         ************************************** /

# include "global.h"

char    lexbuf [BSIZE]
int   lineno = 1;
int   tokenval = NONE;

int   lexan ( )              /* bộ phân tích từ vựng */
{
        int   t;
        while(1)  {
           t = getchar ( );
           if  ( t = = ‘ ‘ || t = = ‘ \t’)   ;        /* xóa các khoảng trắng */
           else  if  (t = = ‘ \n ’)
                        lineno = lineno + 1;
                 else  if  ( isdigit (t) )   {            /* t là một ký số */
                        ungetc (t, stdin);
                        scanf (" %d", & tokenval);
                        return  NUM;
                  }
                else  if  ( isalpha (t) )   {            /* t là một chữ cái */
```

```
                int  p, b = 0;
                while ( isalnum (t) )    {      /* t thuộc loại chữ - số */
                        lexbuf[b] = t;
                        t = getchar ( );
                        b = b + 1;
                        if  (b > = BSIZE)
                                error("compiler error");
                    }
                lexbuf[b] = EOS;
                if  (t ! = EOF)
                      ungetc (t, stdin);
                p = lookup (lexbuf);
                if  (p = = 0)
                        p = insert (lexbuf, ID)
                tokenval = p;
                return  symtable[p].token;
            }
            else   if  (t = = EOF) {
                return  DONE;
            else   {
                tokenval  = NONE;
                return  t;
            }
        }
    }
/ ****          parser.c       ****************************************** /

# include "global.h"

int   lookahead;

parse ( )               /* phân tích cú pháp và dịch danh sách biểu thức */
{
        lookahead = lexan ( );
        while  (lookahead ! =  DONE)   {
                expr( ) ;  match (‘ ; ’);
        }
}

expr ( )
{
        int      t;
        term ( );
        while(1)
           switch (lookahead)     {
                case   ' + ' :    case   ' - ' :
                   t = lookahead;
```

```
                match (lookahead);  term ( );  emit (t, NONE);
                continue;
            default :
                return;
        }
}

term ( )
{
    int     t;
    factor ( );
    while(1)
      switch (lookahead)    {
          case  ' * ' :   case  ' / ' :    case  ' DIV ' :    case  'MOD ' :
            t = lookahead;
            match (lookahead);  factor ( );  emit (t, NONE);
            continue;
          default :
            return;
        }
}

factor ( )
{
    switch (lookahead)    {
          case  ' ( ' :
            match (' ( ');  expr ( );  match (' ) ');      break;
          case  NUM  :
            emit (NUM, tokenval) ;  match (' NUM ');      break;
          case  ID  :
            emit (ID, tokenval) ;  match (' ID ');     break;
          default :
            error ( "syntax error");
        }
}

match ( t )
    int  t;
{
      if  (lookahead = = t)
          lookahead = lexan( );
      else  error ("syntax error");
}

/ ****          emitter.c      ********************************************* /

# include "global.h"
```

```
emit (t, tval)                  /* tạo ra kết quả */
        int     t, tval;
{
        switch ( t )    {
            case  '+' :     case  '-' :    case  '*' :    case  '/ ' :
                printf (" %c \n", t);  break;
            case  DIV :
                printf (" DIV \n", t);          break;
            case  MOD :
                printf (" MOD \n", t);          break;
            case  NUM :
                printf (" %d \n", tval );       break;
            case  ID :
                printf (" %s  \n", symtable [tval]. lexptr);    break;
            default :
                printf (" token %d , tokenval %d \n ",   t,   tval );
        }
}

/ ****        symbol.c      ****************************************** /

# include "global.h"

# define        STRMAX    999             /* kích thước mảng lexemes */
# define        SYMMAX    100             /* kích thước mảng symtable */

char    lexemes [STRMAX];
int     lastchar = -1           /* vị trí được dùng cuối cùng trong lexemes */
struct  entry   symtable [SYMMAX];
int     lastentry = 0           /* vị trí được dùng cuối cùng trong symtable */

int     lookup (s)              /* trả về vị trí của ô cho s */
        char    s [ ];
{
        int     p;
        for (p = lastentry;   p > 0;  p = p - 1)
                if  (strcmp (symtable[p].lexptr, s ) = = 0)
                        return  p;
        return  0;
}
int     insert (s, tok)         /* trả về vị trí của ô cho s */
        char    s [ ];
        int     tok;
{
        int     len;
        len = strlen  (s)       /* strlen tính chiều dài của s */
```

```
        if   ( lastentry + 1 > =  SYMMAX)
                error ("symbol table full");
        if   ( lastchar + len  + 1  > =  SYMMAX)
                error ("lexemes array full");
        lastentry = lastentry + 1;
        symable [lastentry].token = tok;
        symable [lastentry].lexptr = &lexemes [lastchar + 1];
        lastchar = lastchar  + len + 1;
        strcpy (symable [lastentry].lexptr, s);
        return   lastentry;
}

/ ****          init.c          ************************************** /

# include "global.h"

struct  entry   keyword [ ] =  {
        "div", DIV
        "mod", MOD
        0,      0
}
init ( )                /* đưa các từ khóa vào symtable */
{
        struct entry    * p ;
        for (p = keywords;   p → token;  p ++ )
                if  (strcmp (symtable[p].lexptr, s ) = = 0)
                insert (p → lexptr, p → token) ;
}

/ ****          error.c             ************************************** /

# include "global.h"

eeror (m)               /* sinh ra tất cả các thông báo lỗi * /
        char    * m;
{
        fprintf   (stderr,   " line  % d : % s \n, lineno, m)
        exit ( 1 )      /* kết thúc không thàinh công * /
}


/ ****          main.c              ************************************** /

# include "global.h"

main ( )
{
```

```
        init ( );
        parse ( );
        exit (0);       /* kết thúc thàinh công * /
}

/ ********************************************************************** /
```

BÀI TẬP CHƯƠNG II

2.1. Cho văn phạm phi ngữ cảnh sau:

$$S \rightarrow S\ S\ + \mid S\ S\ * \mid \boldsymbol{a}$$

a) Viết các luật sinh dẫn ra câu nhập: **a a + a ***

b) Xây dựng một cây phân tích cú pháp cho câu nhập trên?

c) Văn phạm này sinh ra ngôn ngữ gì? Giải thích câu trả lời.

2.2. Ngôn ngữ gì sinh ra từ các văn phạm sau? Văn phạm nào là văn phạm mơ hồ?

a) $S \rightarrow \mathbf{0}\ S\ \mathbf{1} \mid \mathbf{0}\ \mathbf{1}$

b) $S \rightarrow +\ S\ S \mid -\ S\ S \mid \boldsymbol{a}$

c) $S \rightarrow S\ (\ S\ )\ S \mid \in$

d) $S \rightarrow \mathbf{a}\ S\ \mathbf{b}\ S \mid \mathbf{b}\ S\ \mathbf{a}\ S \mid \in$

e) $S \rightarrow \mathbf{a} \mid S + S \mid S\ S \mid S\ * \mid (\ S\ )$

2.3. Xây dựng văn phạm phi ngữ cảnh đơn nghĩa cho các ngôn ngữ sau đây:

a) Các biểu thức số học dưới dạng hậu tố.

b) Danh sách danh biểu có tính kết hợp trái được phân cách bởi dấu phẩy.

c) Danh sách danh biểu có tính kết hợp phải được phân cách bởi dấu phẩy.

d) Các biểu thức số học của số nguyên và danh biểu với 4 phép toán hai ngôi : +. -, *, /.

2.4. Viết chỉ thị máy ảo kiểu Stack cho quá trình dịch các biểu thức sau sang dạng hậu tố:

a) t : = (a mod b) * 1998 - (2000 * c +100) div 4 +1999

b) t : = a_1 mod c_2 + (b_3 -156 * d_4) div 7 / 3

c) y := x + 100 z^3 t

2.5. Xây dựng lược đồ dịch trực tiếp cú pháp để dịch một biểu thức số học từ dạng trung tố sang dạng hậu tố (cho các phép toán 2 ngôi).

a) Xây dựng chương trình đổi mã hậu tố sang mã máy ảo kiểu Stack .

b) Viết chương trình thực thi mã máy ảo .

2.6. Yêu cầu như bài 5 cho biểu thức số học ở dạng hậu tố sang dạng trung tố.

2.7. Xây dựng một lược đồ dịch trực tiếp cú pháp để xác định rằng các dấu ngoặc trong một chuỗi nhập là cân bằng.

2.8. Xây dựng lược đồ dịch trực tiếp cú pháp để dịch phát biểu FOR của ngôn ngữ C có dạng như sau: ***FOR (exp1; exp2; exp3) Stmt*** sang dạng mà máy ảo kiểu Stack. Viết chương trình thực thi mã máy ảo kiểu Stack .

2.9. Xét đoạn văn phạm sau đây cho các câu lệnh if-then và if-then-else:

$Stmt \rightarrow$ ***if*** expr ***then*** stmt

| ***if*** expr ***then*** stmt ***else*** stmt

| ***other***

a) Chứng tỏ văn phạm này là văn phạm mơ hồ.

b) Xây dựng một văn phạm không mơ hồ tương đương với quy tắc: mỗi else chưa được kết hợp sẽ được kết hợp với then chưa kết hợp gần nhất trước đó.

c) Xây dựng một lược đồ dịch trực tiếp cú pháp để dịch các câu lệnh điều kiện thành mã máy ảo kiểu Stack.

2.10. Xây dựng lược đồ dịch trực tiếp cú pháp để dịch các phát biểu của ngôn ngữ PASCAL có dạng như sau sang dạng mà máy ảo kiểu Stack. Viết chương trình thực thi mã máy ảo kiểu Stack:

a) **REPEAT** *Stmt* **UNTIL** *expr*

b) **IF** *expr* **THEN** *Stmt* **ELSE** *Stmt*

c) **WHILE** *expr* **DO** *Stmt*

d) **FOR** i := *expr1* **downto** *expr2* **DO** *Stmt*