

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI  
---★★★---

## Luận văn tốt nghiệp kỹ sư II

Chuyên ngành **Công nghệ thông tin**

Đề tài

# PHÂN TÍCH BIỂU THỨC SỐ HỌC

(Trình biên dịch)

Sinh viên: **Trần Tuấn Lâm**

Thầy hướng dẫn: **Phạm Đăng Hải**

Thầy phản biện :

Vũng tàu 2001

# LỜI NÓI ĐẦU

Các ngôn ngữ lập trình và trình biên dịch cho chúng là một trong những bộ phận cấu thành quan trọng nhất của công nghệ thông tin. Ta có thể ví trình biên dịch như những chiếc máy cái trong ngành sản xuất phần mềm, thông qua chúng người ta sản xuất tất cả những phần mềm khác từ hệ điều hành cho tới các trình ứng dụng, các hệ quản trị cơ sở dữ liệu lớn cũng như các trò chơi.

Lý thuyết trình biên dịch đặt cơ sở trên lý thuyết ngôn ngữ hình thức, là một lý thuyết có chiều sâu toán học sâu sắc và liên quan chặt chẽ tới logic toán. Trình biên dịch là kết quả cài đặt các thuật giải được trình bày trong lý thuyết. Việc cài đặt trình biên dịch, dù nhỏ cũng đòi hỏi một lượng kiến thức tương đối lớn, tính chính xác và tỷ mỉ cao.

Nhiệm vụ của trình biên dịch là chuyển một văn bản chương trình nguồn thành một mã đích nào đó, thường là các mã đích thực thi được để chạy trên một hệ điều hành (ví dụ như chuyển văn bản Pascal thành chương trình .exe để chạy trên DOS, Windows), hay chạy trên một máy ảo nào đó (như trong Java). Tuy nhiên đó cũng có thể là một hệ thống dịch từ ngôn ngữ này sang ngôn ngữ khác hay tạo ra một cấu trúc dữ liệu cần thiết nào đó như trong trường hợp của luận văn này - tạo ra cây phân tích biểu thức số học dùng để vẽ (biểu diễn) biểu thức dưới dạng toán học.

Luận văn này bao gồm hai phần: phần lý thuyết phân tích các thuật giải sử dụng trong trình biên dịch ngôn ngữ PL/0 được sửa đổi và tăng cường để phù hợp với nội dung bài toán được đặt ra là "Phân tích biểu thức số học để vẽ các biểu thức số học, tính toán các giá trị của chúng, vẽ đồ thị và giải phương trình"; phần cài đặt là một chương trình thực thi cùng toàn bộ chương trình nguồn được viết trong Delphi 5. Phần lõi biên dịch được cài đặt ở đây dựa trên trình biên dịch cho ngôn ngữ PL/0 trong cuốn "Thực hành kỹ thuật biên dịch - Nguyễn Văn Ba". Để giải quyết bài toán được đặt ra thì trình biên dịch này đã được sửa đổi khá nhiều.

Trong điều kiện hạn chế về tài liệu, thời gian nghiên cứu, việc thiếu chính xác trong các thuật ngữ cũng như hạn chế của chính tác giả nên luận văn này không tránh khỏi những thiếu sót. Vì vậy tôi rất mong muốn nhận được các ý kiến đóng góp, chỉ bảo để hoàn thiện hơn luận văn này.

Tôi xin bày tỏ lòng biết ơn chân thành tới thầy Phạm Đăng Hải cũng như các thầy cô giáo trong khoa Công nghệ thông tin trường Đại học Bách khoa Hà nội, Trung tâm ngoại ngữ-tin học Vũng tàu đã tận tình hướng dẫn và giúp đỡ, động viên, tạo điều kiện thuận lợi cho tôi hoàn thành tốt luận văn tốt nghiệp này.

# Đề tài

Phân tích biểu thức số học để:

1. Viết dưới dạng thường gặp trong các ngôn ngữ lập trình rồi biểu diễn chúng thành dạng toán học. Ví dụ

$$(x+y)/2*5 \Rightarrow \frac{x+y}{2}*5$$

$$(x+y)/(x+y+z) \Rightarrow \frac{x+y}{x+y+z}$$

$$\sin((x+y)/2+1)/(x+y+z) \Rightarrow \sin\left(\frac{\frac{x+y}{2}+1}{x+y+z}\right)$$

2. Tính toán giá trị các biểu thức
3. Vẽ đồ thị các hàm cho bởi các biểu thức trên
4. Giải phương trình trên đoạn do người dùng đưa vào
5. Tính giá trị hàm số tại điểm cho trước

# Chương 1

## Mở đầu

### 1.1. Biên dịch biểu thức số học để làm gì?

---

Biểu thức số học là phần được nghiên cứu rất kỹ, là một ví dụ rất phổ biến trong các giáo trình, tài liệu về ngôn ngữ hình thức và trình biên dịch. Phạm vi ứng dụng của nó rất rộng. Tất cả các ngôn ngữ lập trình đều có biểu thức số học và như vậy trong trình biên dịch tương ứng phải có phần phân tích biểu thức số học. Các trình bảng tính như Quattro Pro, Excel... đều có phần phân tích biểu thức số học bên trong để tính toán giá trị các công thức. Các chương trình toán học như MathCAD, Mathematica cũng có phần phân tích biểu thức số học để tính toán và vẽ các biểu thức.

Trong luận văn này chúng ta quan tâm tới phân tích Biểu thức số học với mục đích :

- Vẽ biểu thức dưới dạng thường thấy trong toán học. Dạng này dễ đọc, dễ hiểu hơn;
- Giải phương trình trên đoạn do người dùng đưa vào;
- Tính giá trị hàm số tại điểm cho trước;
- Vẽ đồ thị các hàm số do người dùng đưa vào (sau khi chương trình đã được biên dịch);
- Lập bảng giá trị các hàm số.

Chương trình ""Phân tích BTSH"" được cài đặt hiện tại giải quyết được toàn bộ các vấn đề nêu trên. Ngoài ra nó còn là trình biên dịch PL/0 mở rộng làm việc được với các số thực.

Ngoài ra còn có thể mở rộng để:

- Lưu trữ các công thức dưới dạng văn bản chứ không phải đồ họa. Không gian lưu trữ công thức sẽ gọn hơn, dễ dàng lưu trữ trong các cơ sở dữ liệu;
- Dùng để trình bày các lời giải (trong các chương trình toán học);
- Có khả năng soạn thảo công thức dưới dạng đồ họa;
- Tính đạo hàm, tích phân dưới dạng ký hiệu của mộ công thức;
- Dùng activeX để hiển thị trong các browser;
- ...

Các phần mở rộng nằm ngoài phạm vi của luận văn này do chúng đòi hỏi nhiều công sức lập trình.

## 1.2. Nội dung của luận văn

---

### 1.2.1. Phần lý thuyết:

- Trình bày phương pháp phân tích một biểu thức số học để có được thông tin dùng để vẽ, tính toán giá trị của biểu thức.
- Trình bày các cấu trúc dữ liệu dùng để lưu trữ kết quả khi biên dịch biểu thức
- Trình bày thuật toán vẽ biểu thức.

### 1.2.2 Phần cài đặt:

- Chương trình thực thi (Chương trình được gọi là **"Phân tích BTSH"**- file BTSH.EXE)
- Mã nguồn của chương trình được viết dựa vào những phần lý thuyết đã nêu trên.

## 1.3. Các điểm chính về cài đặt chương trình "Phân tích BTSH"

---

"Phân tích BTSH" là một trình biên dịch ngôn ngữ PL/0. Nhiều thành phần liên quan tới biên dịch, sinh mã được lấy từ các ví dụ trong cuốn Thực hành kỹ thuật biên dịch - Nguyễn Văn Ba. Sau đây là một số điểm chính mà ta *sửa đổi* hoặc *thêm vào*.

1. Chương trình mẫu trên ở dạng nguyên bản không dịch được bằng các bộ dịch ngôn ngữ lập trình Pascal của Borland do có câu lệnh `goto` vượt ra ngoài thủ tục/hàm. Để không phải sửa lại nhiều những câu lệnh `goto` được thay bằng lệnh `abort` (sử dụng ngoại lệ).
2. Có một số tên lấy trùng với các từ khóa (như `object`, `procedure`) được sửa lại;
3. Sửa lại thủ tục thông báo lỗi để định vị ngay tới chỗ gây lỗi trong màn hình soạn thảo.
4. Chương trình dừng ngay khi phát hiện lỗi đầu tiên (khác với chương trình ở mức 2 và 3 trong nguyên bản).
5. Khi chạy chương trình PL/0 có đưa ra tập tin 'CrtOut.\$\$\$' *quá trình thực hiện từng lệnh*. Khi kết thúc chương trình tập tin này được nạp vào tab CRT để dễ theo dõi. Thông tin này giúp cho theo dõi logic của chương trình dễ dàng hơn rất nhiều. Trong nguyên bản không có công cụ này.
6. Tách phần phân tích từ vựng, biên dịch và thực thi mã thành 3 phần riêng biệt tách rời nhau: Phần phân tích từ vựng (tập tin `Tuvung.pas`), phần biên dịch sinh mã (`PL0Paser.pas`) và phần máy ảo PL/0 (`Interpreter.pas` mà thủ tục trung tâm là `interpret`). Điều này không những giúp cho chương trình sáng sửa hơn về cấu trúc mà còn giúp tính toán nhanh mà không cần biên dịch lại chương trình

PL/0 trong trường hợp phải tính giá trị một mảng số (khi dùng tính toán các giá trị để vẽ đồ thị cho hàm số).

7. Viết thêm nhiều đơn vị chương trình liên quan tới cây phân tích biểu thức số học, vẽ các biểu thức, vẽ đồ thị, giải phương trình...

Ngoài ra còn thực hiện những sửa đổi sau liên quan tới ngôn ngữ:

- Chương trình mẫu chỉ tính toán với số nguyên, nay sửa lại để tính toán được thực hiện trên số thực (dấu phẩy động). Ngoài ra cũng sửa lại cài đặt các phép toán  $\leq$ ,  $\geq$  mà trong chương trình mẫu chưa thực sự có cài đặt.
- Hiểu, tính toán được một số hàm toán học "dựng sẵn bên trong" (intrinsic function) như `sin`, `cos`, `ln`, `exp`, `abs`, `sqr`, `sqrt`...
- Xây dựng được cấu trúc dữ liệu các biểu thức dưới dạng cây để có thể vẽ các biểu thức dưới dạng toán học.

#### 1.4. Tại sao ta lại không làm việc biên dịch biểu thức riêng mà lại lồng vào trong trình biên dịch PL/0?

- Nếu trình biên dịch quá đơn giản ta không thể khai báo hằng làm cho công thức không được đẹp, không viết được công thức có chứa hằng số dưới dạng ký hiệu.
- Vì để tính toán được giá trị biểu thức ta phải lập một máy ảo nào đó. Máy ảo PL/0 tương đối đơn giản nên việc mở rộng sử dụng nó không phức tạp hơn việc tạo ra máy ảo mới.
- Ta có thể dễ dàng mở rộng các tính năng sau này.

#### 1.5. Làm thế nào mở rộng "Phân tích BTSH" để tránh việc phải viết biểu thức trong ngôn ngữ PL/0.

Giả sử ta chỉ có 1 dòng chứa biểu thức số học cần tính toán (tương tự như các công thức trong Excel). Công thức này chưa biên dịch được bằng chương trình "Phân tích BTSH" vì các biểu thức phải là các biểu thức nằm trong chương trình nguồn PL/0.

Để giải quyết vấn đề trên với nỗ lực tối thiểu ta có thể dùng kỹ thuật sau.

1. Ta sẽ dùng cách tự động khai báo - Phần phân tích từ vựng được sửa đổi lại một chút để có thể ghi nhận được tất cả các từ tố được dùng trong công thức.
2. Tự động tạo một chương trình PL/0 có các khai báo những từ tố trên và một biểu thức duy nhất chứa công thức phía bên phải và một biến tự động khai báo khác ở bên trái
3. Truyền chương trình này cho "Phân tích BTSH" dịch.

Cài đặt này được mô tả trong chương 2.

## Chương 2

# Phân tích biểu thức số học

### 2.1. Sơ đồ tổng quát của chương trình

---

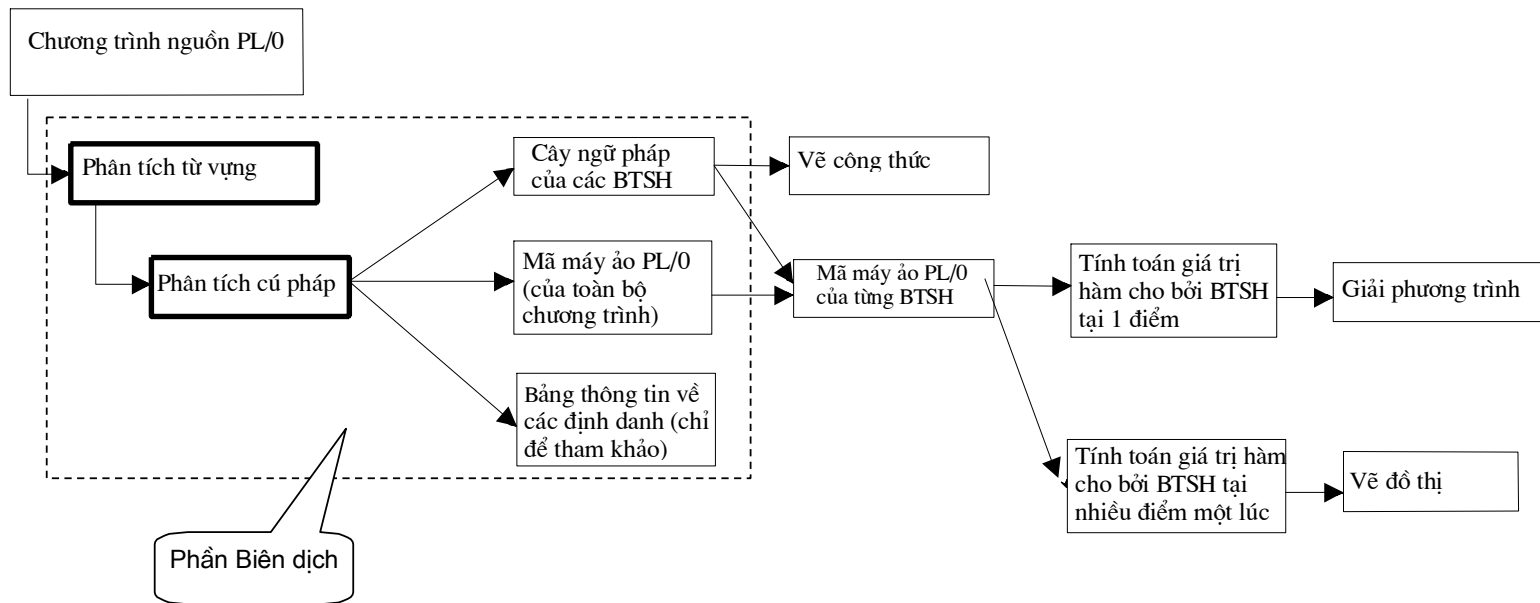
Rõ ràng để có thể vẽ được biểu thức số học ta cần phải hiểu được nó:

- Biểu thức viết có đúng văn phạm?
- Đây là tử số, đây là mẫu số, đây là đối số của hàm? Chú ý là các biểu thức có thể lồng nhau: trong biểu thức sẽ có thể có biểu thức.
- Xây dựng được cây văn phạm của biểu thức. Từ cây văn phạm này, ta có thể thực hiện việc sinh mã tính toán và vẽ biểu thức toán học.

Để giải quyết các vấn đề trên ta sử dụng các kiến thức về Ngôn ngữ hình thức và Trình biên dịch. Ta có thể tóm tắt sơ đồ chương trình được cài đặt như sau:

- I. Bộ phân tích từ vựng sẽ đọc dòng ký tự từ chương trình nguồn PL/0 và phân tích thành các từ tố cũng như một số thông tin khác về định danh biến, giá trị số.
- II. Những thông tin này sẽ được đưa vào bộ phân tích cú pháp và sau khi làm việc nếu không có sai sót sẽ đưa ra kết quả là:
  1. Cây ngữ nghĩa của các biểu thức số học cần quan tâm (các biểu thức này phải được viết theo một số qui tắc đặc biệt). Thông tin này sẽ dùng để vẽ các biểu thức số học.
  2. Mã máy ảo PL/0 (dùng để chạy các chương trình PL/0 một cách bình thường)
  3. Bảng thông tin về các định danh. Các thông tin này do phần phân tích ngữ nghĩa sử dụng, sau khi phân phân tích ngữ nghĩa làm việc xong thì nó chỉ còn ý nghĩa tham khảo. Tuy nhiên nếu phát triển thêm chương trình thì chúng có thể dùng như các thông tin debug ...

Cây văn phạm của biểu thức số học dùng để vẽ biểu thức. Nó cũng chứa thông tin về địa chỉ bắt đầu và kết thúc của mã sinh ra cho biểu thức, thông tin này giúp ta có thể "trích ra" phần mã của biểu thức và chạy nó trên một máy ảo khác dùng để tính giá trị hàm tại 1 điểm (coi mỗi biểu thức là một hàm), giải phương trình và tính toán giá trị tại hàng loạt điểm để vẽ đồ thị. Sơ đồ làm việc tổng quát của chương trình được trình bày tại hình sau:



Sơ đồ tổng quát chương trình "Phân tích biểu thức số học"



## 2.2. Văn phạm của biểu thức số học

Văn phạm của biểu thức số học mà "Phân tích BTSH" có thể hiểu được như sau

Expression  $\rightarrow$  Sign Term A

A  $\rightarrow$   $\epsilon$  | Addopr Term A

Addopr  $\rightarrow$  + | -

Sign  $\rightarrow$  Addopr |  $\epsilon$

Term  $\rightarrow$  Factor B

B  $\rightarrow$   $\epsilon$  | Mulopr Factor B

Mulopr  $\rightarrow$  \* | /

Factor  $\rightarrow$  ident | number | Function(Expression) | (Expresion)

Function  $\rightarrow$  sin | cos | tan | arctan | ln | exp |  
abs | sqrt | sqr

Trong đó các ký hiệu chưa kết thúc là

A, B, Expression, Sign, Term, Addopr, Mulopr, Factor, Function

Biểu đồ cú pháp của Factor cũng như của ngôn ngữ PL/0 được trình bày trong phần 3.2. Biểu đồ cú pháp của PL/0 ở trang 22.

## 2.3. Phân tích biểu thức số học trong quá trình biên dịch

Việc phân tích ngữ pháp của trình biên dịch thực ra là quá trình xây dựng lại cây văn phạm (cây phân tích) từ chương trình nguồn. Trong quá trình đó thì mã máy tương ứng cũng được sinh ra khi cần thiết. Sau đây ta sẽ xem xét kỹ hơn quá trình sinh mã cho biểu thức số học.

## 2.4. Về sinh mã tính toán cho các biểu thức số học.

Rất nhiều trình biên dịch sinh mã tính toán cho biểu thức số học bằng các sử dụng ngăn xếp. Trong lúc tính toán chương trình chỉ làm việc với các phần tử trên đỉnh ngăn xếp mà không cần để ý tới các phần tử phía dưới. Không phải ngẫu nhiên mà co-processor x87 của Intel có các thanh ghi được tổ chức dưới dạng ngăn xếp - điều này nhằm giúp cho các trình biên dịch sinh mã tính toán các biểu thức số học dễ dàng hơn.

Xét biểu thức

$a+b*c+d$

các bước tính toán tiến hành như sau

1. Nạp a lên đỉnh ngăn xếp;
2. Nạp b lên đỉnh ngăn xếp;

3. Nạp c lên đỉnh ngăn xếp. Phép cộng chưa được thực hiện do nó có mức ưu tiên thấp hơn phép nhân;
4. Thực hiện phép nhân với hai phần tử trên đỉnh ngăn xếp. Loại bỏ hai phần tử này khỏi ngăn xếp và lưu kết quả phép nhân lên đỉnh ngăn xếp;
5. Thực hiện phép cộng với hai phần tử trên đỉnh ngăn xếp. Loại bỏ hai phần tử này khỏi ngăn xếp và lưu kết quả phép cộng lên đỉnh ngăn xếp;
6. Nạp d lên đỉnh ngăn xếp;
7. Thực hiện lại bước 5;

Như vậy cuối cùng trên đỉnh ngăn xếp chỉ còn lại một phần tử duy nhất là kết quả tính toán biểu thức.

- Các phép toán  $+$ ,  $-$ ,  $*$ ,  $/$  làm việc với 2 phần tử trên đỉnh ngăn xếp. Hai phần tử này sau phép toán sẽ bị loại ra khỏi ngăn xếp còn kết quả tính toán sẽ được lưu trở lại vào ngăn xếp;
- Các hàm như  $\sin$ ,  $\cos$ ... và phép toán đổi dấu chỉ làm việc với 1 phần tử trên đỉnh ngăn xếp. Sau khi tính toán, phần tử trên đỉnh ngăn xếp sẽ bị loại ra và kết quả tính toán sẽ được đưa trở lại vào ngăn xếp.

Sau đây là sơ đồ biến đổi của ngăn xếp khi  $a=1$ ,  $b=2$ ,  $c=3$ ,  $d=4$ . Ngăn xếp phát triển lên phía trên

1

↑  
nạp số 1 lên đỉnh ngăn xếp

2
1

↑  
nạp số 2 lên đỉnh ngăn xếp

3
2
1

↑  
nạp số 3 lên đỉnh ngăn xếp

6
1

↑  
thực hiện phép nhân  $3*2$ . Loại 3, 2 ra khỏi ngăn xếp, lưu tích của chúng là 6 trở lại ngăn xếp

7

↑  
thực hiện phép cộng  $6+1$ . Loại 6, 1 ra khỏi ngăn xếp, lưu tổng của chúng là 7 trở lại ngăn xếp

4
7



nạp số 4 lên đỉnh ngăn xếp

11

↑ thực hiện phép cộng  $7+4$ . Loại 7, 4 ra khỏi ngăn xếp, lưu tổng của chúng là 11 trở lại ngăn xếp.  
Đây chính là kết quả tính toán biểu thức

Như vậy mã máy được sinh ra để tính toán một biểu thức số học sẽ gồm một loạt các lệnh làm việc với 1 hay 2 phần tử trên đỉnh ngăn xếp. Điều này làm đơn giản hóa quá trình sinh mã rất nhiều.

## 2.5. Xây dựng cây văn phạm của biểu thức số học.

Trình biên dịch PL/0 cũng sinh mã để tính toán biểu thức số học dùng ngăn xếp theo sơ đồ như trên.

Từ quá trình sinh mã cho biểu thức số học như trên ta có thể dựng lại cây văn phạm cho biểu thức số học như sau:

- ứng với việc nạp hằng số ta nạp 1 nút mới chứa các thông tin cần thiết vào ngăn xếp;
- ứng với phép toán 2 ngôi ta thực hiện
  1. tạo nút mới tương ứng với phép toán
  2. gán các phần tử trái, phải của nút tương ứng là phần tử trước đỉnh và phần tử đỉnh của ngăn xếp
  3. Loại bỏ hai phần tử trên đỉnh ngăn xếp
  4. Nạp vào đỉnh ngăn xếp nút mới được tạo ra.
- ứng với phép toán 1 ngôi và gọi hàm dựng sẵn ta thực hiện tương tự
  1. tạo nút mới tương ứng với phép toán
  2. gán các phần tử trái của nút là phần tử của đỉnh ngăn xếp
  3. Loại bỏ phần tử trên đỉnh ngăn xếp
  4. Nạp vào đỉnh ngăn xếp nút mới được tạo ra.

Ta thấy quá trình trên hoàn toàn tương ứng với quá trình thực hiện phép toán số học. Như vậy chỉ với một số lượng dòng mã tương đối ít ta có thể sinh ra cây văn phạm *song song* với quá trình sinh mã của biểu thức số học.

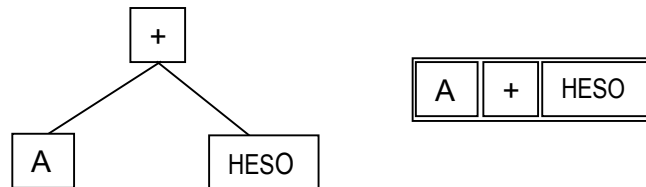
Cấu trúc dữ liệu để thực hiện thuật toán này được trình bày ở các phần sau (lớp `TNodeStack`).

## 2.6. Làm thế nào để vẽ được biểu thức toán học

Đây chính là một trong những phần chính mà luận văn cần giải quyết.

Giả sử có cây văn phạm của biểu thức thì từ cây này ta có thể vẽ được biểu thức dưới dạng toán học như sau:

Xét một nút cây



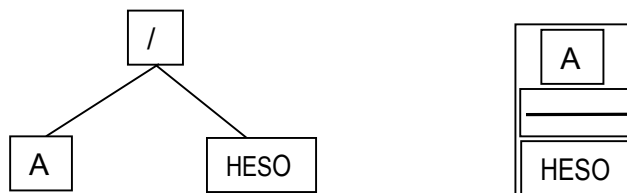
Giả sử ta đã tính được hình chữ nhật chứa biểu thức bên trái và bên phải (tức là của các nút con), khi đó ta có thể tính được hình chữ nhật bao quanh các biểu thức +, -, \*, /, hàm số ... của chính nút đang xét.

- Với phép toán +, -, \*: Hình chữ nhật này sẽ là hình chữ nhật nhỏ nhất bao quanh hình chữ nhật chứa biểu thức bên trái, hình chữ nhật chứa phép toán, hình chữ nhật bên phải được xếp cạnh nhau theo hàng ngang.
- Với phép toán / (phép chia): Hình chữ nhật này sẽ là hình chữ nhật nhỏ nhất bao quanh hình chữ nhật chứa biểu thức bên trái, hình chữ nhật chứa phép chia, hình chữ nhật bên phải được xếp cạnh nhau theo hàng dọc.

Quá trình thực hiện từ dưới lên trên (từ các nút lá lên gốc) bằng cách duyệt cây. Sau khi duyệt xong cây thì ta cũng có thể biết được hình chữ nhật bao quanh biểu thức. Hình chữ nhật bao quanh các biểu thức lá không có gì phức tạp. Ta có thể tính chúng bằng cách gọi hàm `GetTextExtentPoint32` của windows.

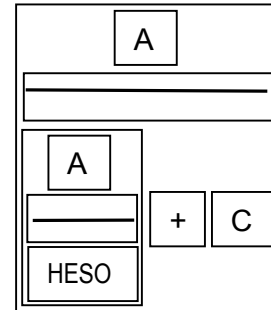
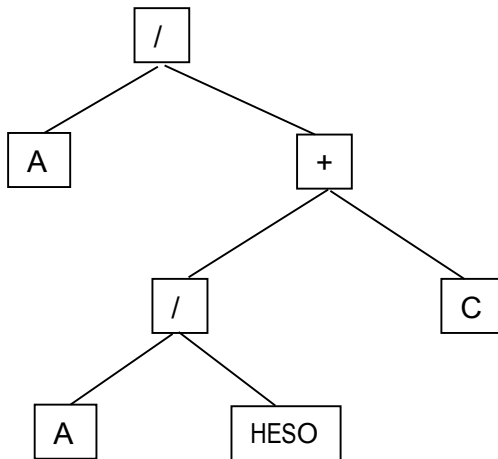
- đối với các phép toán cộng, trừ, nhân thì không gặp khó khăn gì lớn, nhưng với phép chia vấn đề sẽ phức tạp hơn rất nhiều. Hãy xem các ví dụ sau:

Ví dụ: Công thức  $A/HESO$

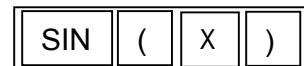
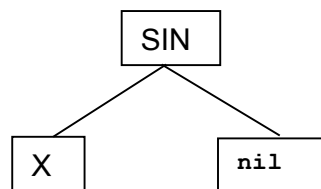


Ví dụ khác : Công thức  $A/(A/HESO+C)$

Công thức  $A / (A / HESO + C)$

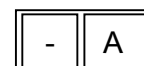
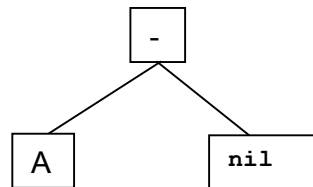


- Với phép gọi hàm (ví dụ như hàm `sin` trong hình vẽ dưới đây) ta chỉ sử dụng lá bên trái, còn lá bên phải sẽ bỏ qua (gán bằng `nil`). Chú ý là không có nút lá để chứa dấu ngoặc. Tuy nhiên dấu ngoặc sẽ vẫn được vẽ. Chú ý là dấu ngoặc sẽ được vẽ hay không phụ thuộc vào "ngữ nghĩa" của cây.



- Với phép đảo dấu

Tương tự như các phép cộng, trừ khi không có toán tử thứ nhất. Tuy nhiên cây bên phải sẽ là `nil` giống như phép gọi hàm.



Sau khi biết được các hình chữ nhật bao quanh từng phần tử của biểu thức ta có thể thực hiện vẽ biểu thức dựa vào các thông tin này.

Đó là ý tưởng chính dùng để vẽ một biểu thức số học.

Sau đây là các cấu trúc dữ liệu và thuật giải để thực thi thuật giải như trên.

### 2.6.1. Cấu trúc dữ liệu thể hiện nút

Để thực hiện thuật toán trên ta cần cấu trúc dữ liệu thích hợp.

Mặc dù trông bề ngoài có vẻ đơn giản nhưng khi lập trình cài đặt ta có thể sẽ gặp khó khăn đáng kể do có rất nhiều chi tiết. Ta sẽ sử dụng kiểu lập trình hướng đối tượng để giải quyết bài toán của chúng ta. Lập trình hướng đối tượng cũng sẽ giúp chúng ta dễ dàng hơn trong việc mở rộng bài toán sau này.

Trong đơn vị chương trình `ParseTree.pas` chứa hai lớp đối tượng để thực thi ý tưởng xây dựng cây văn phạm cho biểu thức số học nêu trên, đó là

1. `TNode`: lớp thể hiện một nút cây văn phạm của biểu thức số học. Cây văn phạm cũng chính là một nút có chứa nhiều nút con trái, phải bên trong (Xem tiếp phần `FLeftNode`, `FRightNode`)
2. `TNodeStack`: lớp chứa *tất cả* các cây văn phạm của các biểu thức số học cần quan tâm. Mỗi phần tử bên trong là một `TNode`. Như vậy ta có thể cùng một lúc lưu được *nhiều* cây biểu thức số học (ứng với các công thức khác nhau). `TNodeStack` có các phương thức để dựng cây văn phạm song song với việc sinh mã tính toán.

#### Các trường của nút (TNode):

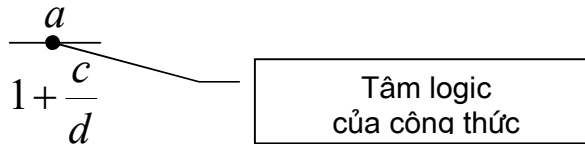
Mỗi nút sẽ cần các thông tin sau:

- kiểu nút `FSymbol`. Đây là các ký hiệu tương ứng với các từ tố khi phân tích từ vựng. Tuy nhiên nó không hoàn toàn tương đương, ví dụ `symNeg`, `symFunc` (đạo dấu) không có từ tố tương đương. Ta không dùng các từ tố được khai báo trong `TuVung.pas` mà khai báo kiểu mới để có thể mở rộng dễ dàng hơn.
- Nhãn của nút: `FSymbolLabel` có kiểu là `string` ('+', '-', 'sin', 'cos', 'mass'...) đây chính là dòng chữ sẽ xuất hiện trên màn hình khi vẽ nút.
- Các nút con bên trái, bên phải `FLeftNode`, `FRightNode`.

Ngoài ra ta cần các trường để lưu thông tin về nút trong quá trình tính toán

- `FFormulaRect`: hình chữ nhật bao toàn bộ công thức của nút (bao gồm hai nút con trái-phải và chính nút đang xét)
- `FLabelRect`: hình chữ nhật bao quanh nhãn của nút. Thông tin này giúp cho vẽ nhãn được dễ dàng hơn

- `FFormulaCenter`: tâm của công thức. Đây không phải là tâm hình học mà là tâm logic của công thức. Tại sao phải có tâm này? Đó là vì trong phép chia tâm logic và tâm hình học không trùng nhau. Ví dụ có công thức



thì tâm logic của công thức nằm trên đường thẳng của phép chia có tử số là  $a$  chứ không phải là tâm hình học của công thức. Tâm hình học có thể dễ dàng tính được từ `FFormulaRect` bằng cách lấy trung bình cộng các tọa độ  $X$  và tọa độ  $Y$ .

- `FNeedPar`: xác định xem có cần dấu ngoặc bao quanh biểu thức hay không. Chú ý là cây biểu thức của ta hoàn toàn xác định về ngữ nghĩa: các phép tính phải được thực hiện từ gốc lên ngọn. Do đó trong cây không có dấu đóng mở ngoặc. Tuy nhiên khi viết ta không thể bỏ dấu ngoặc khi viết công thức  $a * (b + c)$  được. Những trường hợp như trên hay trong lời gọi hàm khi vẽ ta phải để dấu ngoặc xung quanh biểu thức con. Những nút như vậy sẽ cần có trường `FNeedPar` là `True`. Trường này được cập nhật khi ta gọi thủ tục `UpdateNeedPar`.

Khi vẽ công thức, nếu `FNeedPar` là `true` thì ta vẽ dấu ngoặc xung quanh biểu thức con, nếu `FNeedPar` là `False` thì không vẽ dấu ngoặc.

- Cũng cần nói thêm rằng cây được sinh ra trong quá trình biên dịch và phản ánh ngữ nghĩa (ý nghĩa) của biểu thức số học, do vậy tất cả các dấu ngoặc (...) sẽ bị loại bỏ trong quá trình dịch. Khi vẽ lại công thức ta được công thức có *ý nghĩa* như công thức đã nhập chứ không phải công thức y nguyên như công thức đã nhập. Ví dụ :

( $x$ ) khi biểu diễn sẽ là  $x$ , dấu ngoặc bị bỏ đi  
 $x + (n + m)$  khi biểu diễn sẽ là  $x + m + n$ , dấu ngoặc bị bỏ đi  
 $x * (n + m)$  khi biểu diễn sẽ là  $x * (m + n)$ , dấu ngoặc *không* bị bỏ đi

Ngoài ra còn có hàng loạt các phương thức để thao tác với các nút. Vì các nút được định nghĩa đệ qui nên các thủ tục thường cũng là các thủ tục đệ qui. Ta sẽ xét sơ qua ý nghĩa một số phương thức quan trọng nhất để thực hiện ý tưởng tính các hình chữ nhật bao quanh công thức nói trên.

**procedure** `UpdateNeedPar`;

**procedure** `UpdateRect` (`DC`: `hDC`) ;

**procedure** `Draw` (`DC`: `hDC`) ;

- `UpdateNeedPar`: Duyệt cây để xác định xem nút cây nào cần vẽ dấu ngoặc đơn xung quanh. Cập nhật trường `FNeedPar` của từng nút.

- `UpdateRect`: Duyệt cây để nhật trường `FFormulaCenter`, `FFormulaRect`, `FLabelRect`. Các thông tin này sẽ được dùng để vẽ công thức. Đây là một thủ tục quan trọng và phức tạp nhất của lớp `TNode`. Phép duyệt được thực hiện đệ qui. Trước tiên gọi thủ tục `UpdateRect` cho cây bên trái và cây bên phải, sau đó ta thực hiện các tính toán cho chính nút cây đang xét. Tùy theo nút đang xét chứa văn bản bình thường, các phép toán cộng-trừ-nhân, phép chia, lời gọi hàm hay phép đảo dấu mà sẽ có những xử lý tương ứng khác nhau: các cây bên trái và bên phải sẽ được dịch chuyển tới các vị trí thích hợp, các trường `FFormulaCenter`, `FFormulaRect`, `FLabelRect` được cập nhật để phản ánh vị trí và kích thước của nút (có để ý tới việc xung quanh nút có cần vẽ dấu ngoặc hay không).
- `Draw`: thực hiện vẽ công thức cho bởi nút đang xét lên ngữ cảnh thiết bị (device context ) cho bởi tham số `DC`. Thủ tục này là thủ tục đệ qui, nó lần lượt vẽ từ các nút lá trái, nút lá phải rồi nút gốc dựa trên các thông tin trong các trường của nút. Đây là phương thức trung tâm của lớp này. Nó phải được gọi sau `UpdateRect`.

### 2.6.2. Cấu trúc ngăn xếp nút (*TNodeStack*)

Cấu trúc này dùng để lưu trữ danh sách nút trong quá trình biên dịch. Sau khi biên dịch xong một chương trình PL/0 thì mỗi biểu thức cần được ghi nhận sẽ tương ứng với một phần tử trong danh sách này. Mỗi phần tử trong danh sách là một đối tượng kiểu `TNode` đã trình bày ở trên (tức là một cây biểu diễn biểu thức số học).

Danh sách này làm việc giống như ngăn xếp trong tính toán biểu thức nhưng thay vì sinh ra kết quả số thì sinh ra cây văn phạm của biểu thức.

Đây là lớp kế thừa từ lớp `TList`, một lớp chuẩn rất hay được sử dụng trong VCL (Visual Component Library) của Borland. Lớp này về mặt ý nghĩa có thể coi là một mảng các con trỏ được đánh chỉ số từ 0 tới `Count-1`. Để thêm một phần tử vào mảng ta dùng thủ tục `Add`. Khác với các mảng được khai báo tĩnh, mảng này được tự động tăng thêm chiều dài khi cần thiết. Do vậy ta không sợ số phần tử lớn tràn ra khỏi mảng (có nghĩa là không cần phải lo lắng về số lượng các biểu thức cần phân tích).

Sau đây là mô tả một số phương thức quan trọng mà ta thêm vào lớp `TNodeStack`:

Các thủ tục làm việc với ngăn xếp:

- `Push(N: TNode)` : nạp nút `N` lên đỉnh ngăn xếp
- `Pop` : lấy phần tử cuối (TOS) ra khỏi ngăn xếp (loại bỏ phần tử này khỏi ngăn xếp)
- `TOS`: trả lại phần tử là đỉnh của ngăn xếp

Các thủ tục để thêm các phần tử vào ngăn xếp

- `AddIdent`: thêm một phần tử vào ngăn xếp. Phần tử này là một toán hạng đơn, Nó không có con cháu, tức đây sẽ là lá của cây văn phạm.



- `AddOperation`: thêm một toán tử vào cây văn phạm. Thủ tục này sẽ tạo nút mới, lấy hai phần tử trên đỉnh ra khỏi ngăn xếp làm các nút trái phải, và đưa nút mới này vào ngăn xếp. Các toán tử có thể là cộng, trừ, nhân, chia (hai ngôi)
- `AddFunc`: Thủ tục này sẽ tạo nút mới, lấy phần tử trên đỉnh ra khỏi ngăn xếp làm các nút trái và đưa nút mới này vào ngăn xếp trở lại. Hàm ở đây là các hàm toán học một biến hiện thời có các hàm `sin`, `cos`, `tan`, `arctan`, `ln`, `exp`, `sqrt`, `sqr`, `abs`.
- `AddNeg`: tương tự như `AddFunc`, chỉ khác là toán tử ở đây là toán tử đổi dấu.

Các phương thức này được gọi *song song* (đi ngay sau) các đoạn chương trình sinh mã tương ứng cho máy ảo PL/0 trong quá trình biên dịch:

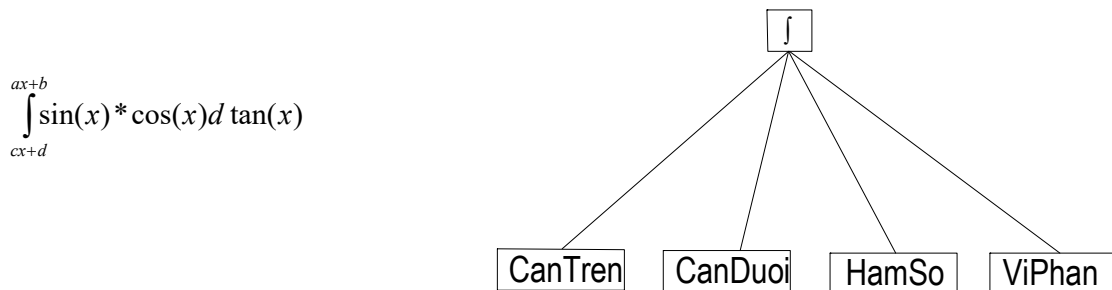
- ứng với việc sinh mã nạp một phần tử lên stack của máy ảo PL/0 ta gọi phương thức `AddItem` để thêm một phần tử lên `NodeStack`.
- ứng với việc sinh mã cho các toán tử hai ngôi như cộng, trừ, nhân chia ta gọi phương thức `AddOperation` cho toán tử tương ứng.
- ứng với việc sinh mã cho hàm dựng sẵn ta gọi phương thức `AddFunc`
- ứng với việc sinh mã đảo dấu ta gọi phương thức `AddNeg`.

#### Ghi chú:

Việc mở rộng để có thể biểu diễn nhiều loại biểu thức toán học có thể đòi hỏi

- cây có nhiều con hơn, còn ý tưởng chính vẫn giữ nguyên như cũ.

Ví dụ: Biểu diễn tích phân xác định



có thể cần dùng các cây con cho

- Cận trên
- Cận dưới
- Biểu thức trong dấu tích phân
- Biểu thức trong dấu vi phân ( $\tan(x)$ )
- Ngoài ra có thể còn phải nhớ tới kích thước chữ (vì biểu thức trên dấu mũ hay trên dấu tích phân thường được viết nhỏ hơn)
- Phải xử lý các trường hợp đặc biệt, ví dụ như với các hàm lượng giác ta thường viết
  - $\sin x$  thay vì  $\sin(x)$
  - $\sin^2 x$  thay vì  $(\sin x)^2$

- Phải xử lý các cây có nhiều nút.

Tuy nhiên đây là vấn đề có thể giải quyết được dù có thể phải tốn nhiều công sức lập trình. Ta không gặp các khó khăn mang tính nguyên tắc ở đây.

## 2.7. Phân tích trực tiếp biểu thức không nằm trong chương trình PL/0

---

### 2.7.1. Đặt vấn đề

Như trên đã nói, trong luận văn này ta thực hiện thực phân tích các biểu thức nằm trong chương trình PL/0. Tuy nhiên việc phân tích như vậy có vẻ khó hiểu và không trực quan lắm và việc sử dụng chương trình đòi hỏi người dùng phải có hiểu biết nhất định về ngôn ngữ lập trình PL/0. Để cho trực quan hơn ta sẽ thực hiện cài đặt sao cho chương trình có thể vẽ ngay biểu thức mà người dùng vừa đánh vào trên dòng soạn thảo mà không cần phải đánh biểu thức tường minh ở trong chương trình PL/0.

Công việc này được thực hiện như sau:

Khi người dùng đánh vào một biểu thức ta sẽ tự động thực hiện việc chèn biểu thức này vào trong chương trình PL/0 hợp lệ sao cho chương trình này có thể biên dịch được và như vậy ta có thể sử dụng toàn bộ các công cụ có sẵn để phân tích và vẽ biểu thức.

Ví dụ khi người dùng đánh vào biểu thức  $1 + 2/(x*x+1)$  thì ngay lập tức chương trình sau sẽ được sinh ra

```
var X, Y;  
begin  
  Y:= 1+ 2/(x*x+1);  
end.
```

Chương trình này ngay lập tức sẽ được tự động biên dịch và biểu thức sẽ được vẽ ra trên màn hình. Nếu biểu thức có sai sót thì thông báo cũng sẽ được đưa ra.

Khó khăn lớn nhất ở đây là việc thực hiện *tự động khai báo* các biến và hằng được dùng trong biểu thức cho chương trình PL/0 được sinh ra (ta nhớ là trình biên dịch PL/0 đòi hỏi các định danh phải được khai báo trước khi sử dụng).

Ta sẽ giả thiết như sau:

Khi người dùng đánh vào một định danh có tên là X, hoặc Y thì ta sẽ coi đó là biến, còn tất cả các định danh khác, nếu không phải là hàm dựng sẵn, ta sẽ coi là hằng.

Ví dụ với biểu thức  $1 + 2/(x*x+1) + m*HeSo$  thì chương trình PL/0 tương ứng sẽ là

```
const m=1, HeSo=1;
var X, Y;
begin
  Y:= 1+ 2/(x*x+1)+m*HeSo;
end.
```

### 2.7.2. Giải quyết vấn đề khai báo các định danh tự động

Biểu thức mới đánh vào  $(1 + 2/(x*x+1) + m * HeSo)$  sẽ được thực hiện *phân tích từ vựng*. Tất cả các định danh sẽ được ghi nhận vào trong một danh sách. Mỗi khi bộ phân tích từ vựng tìm ra từ tố `ident` trong biểu thức ta sẽ xem xem `id` của định danh này đã có trong danh sách hay chưa. Nếu chưa có thì ta thêm vào danh sách tên của định danh này.

Sau đó ta loại bỏ khỏi danh sách này các định danh có tên `X`, `Y` và các hàm dựng sẵn.

Nếu danh sách chưa rỗng (tức là có ít nhất một phần tử) thì ta sẽ sinh ra câu lệnh khai báo `const`.

Ví dụ: Giả sử trong danh sách còn lại hai định danh là `m` và `HeSo` như trong ví dụ trên ta sẽ sinh ra câu lệnh

```
const m=1, HeSo=1;
```

Sau đó ta tiếp tục sinh ra các câu lệnh

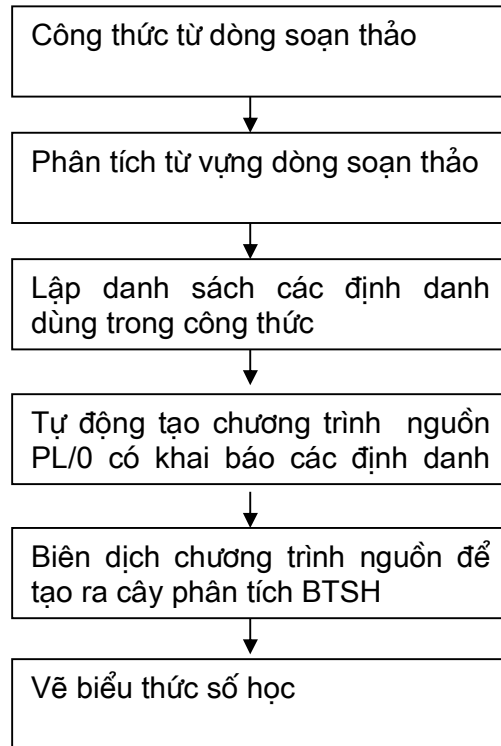
```
var X, Y;
begin
  Y := 1 + 2/(x*x+1)+m*HeSo;
end.
```

Chú ý tới dòng thứ 3 - ta thực hiện việc gán biểu thức vừa được nhập vào cho biến `Y`. Với dòng lệnh này cùng các khai báo tự động chương trình của chúng ta hoàn toàn thích hợp cho biên dịch và thỏa mãn các điều kiện cần thiết để vẽ đồ thị, tính toán giá trị mà ta đã qui định.

Chương trình trên sau đó được chuyển cho đối tượng kiểu `FCompiler` để thực hiện biên dịch. Nếu có lỗi thì ta sẽ đưa ra thông báo, ngược lại sau khi biên dịch thành công ta có thể vẽ ngay biểu thức.

Có một giới hạn ở đây là các hằng số được tự động khai báo có giá trị là 1. Nếu muốn biểu thức được tính toán đúng ta không được dùng các hằng như trên mà phải dùng số trực tiếp trong biểu thức. Như trong ví dụ trên để có thể thực hiện được tính

toán với  $m=2$ ,  $HeSo=3.5$  ta phải viết công thức  $1 + 2/(x*x+1)+2*3.5$  trên dòng soạn thảo.



### 2.7.3. Cài đặt

Để thực hiện hỗ trợ việc khai báo tự động ta dùng lớp `TIdentList`. Lớp này được khai báo như sau:

```

TIdentList = class(TStringList)
  procedure RecordIdent(exp: string);
  function HasIdent(id: string; var Index : Integer): Boolean;
  procedure AddIdent(id: string);
  function CreatePL0SourceList(Exp: string): TStringList;
  function CreatePL0Source(Exp: string): string;
  procedure RemoveKnownIds;
end;
  
```

- Thủ tục `RecordIdent(Exp: string);`

Thủ tục này nghi nhận tất cả các định danh *mới* trong biểu thức `Exp`. Nó khởi động đối tượng `aTuVung` có kiểu `TPhanTichTuVung` để thực hiện phân tích từ vựng của `Exp`. Mỗi khi phát hiện từ tố `ident` nó sẽ gọi `AddIdent` để thêm định danh chưa có vào danh sách.

- Thủ tục `AddIdent(id: string);`  
Thủ tục này kiểm tra xem `id` hiện đã có trong danh sách hay chưa (không phân biệt chữ hoa chữ thường). Nếu chưa `Id` sẽ được thêm vào danh sách.
- Thủ tục `RemoveKnownIds;`  
Thủ tục này loại bỏ khỏi danh sách những định danh đã biết trước. Đó là X, Y và các hàm dựng sẵn.
- Hàm `HasIdent`  
Hàm này kiểm tra xem `id` đã có trong danh sách hay chưa. Nếu có nó sẽ trả về vị trí (`Index`) của `id` trong danh sách. Dựa vào `Index` này ta có thể xóa `id` khỏi danh sách.
- `CreatePL0SourceList;`  
Hàm này trả lại chương trình PL/0 được tự động sinh ra cho biểu thức `Exp` trên dưới dạng một danh sách `TStringList`. Danh sách này thuận tiện để gán vào trình soạn thảo (Editor) chương trình nguồn.
- Hàm `CreatePL0Source;`  
Hàm này trả lại chương trình PL/0 được tự động sinh ra cho biểu thức `Exp` trên dưới dạng một string. Chương trình này có thể truyền cho `FCompiler` để biên dịch.

Trong Form chính của chương trình có phương thức xử lý sự kiện gắn liền với các thay đổi trong phần tử soạn thảo biểu thức. Mỗi khi có thay đổi văn bản (các ký tự được thêm vào hay xóa đi) thì phương thức này sẽ được gọi. Phương thức này sẽ thực hiện việc tạo chương trình PL/0 và biên dịch nó như đã nêu ở trên. Nếu quá trình biên dịch thành công thì biểu thức sẽ được vẽ ra ngay lập tức. Ngược lại thông báo lỗi được đưa ra.

# Chương 3

## Ngôn ngữ PL/0

### 3.1. Giới thiệu sơ lược về PL/0

---

Ngôn ngữ lập trình PL/0 do N. Wirth đưa ra trong tập sách nổi tiếng của ông : "Giải thuật + Cấu trúc dữ liệu = Chương trình". Wirth đã khéo léo chọn cho PL/0 để cho nó không tới nỗi tầm thường, vẫn giữ được những nét điển hình của ngôn ngữ lập trình cấp cao, với dáng dấp của Pascal, mà đồng thời nó cũng không quá phức tạp để vượt ra khỏi tầm với của người mới bước đầu làm quen với chương trình dịch.

Ta sẽ sửa đổi một chút PL/0 so với nguyên bản được trình bày trong cuốn "Thực hành kỹ thuật biên dịch - Nguyễn Văn Ba", viết trình biên dịch cho ngôn ngữ đã sửa đổi này và qua đó thể hiện các giải thuật giải quyết các vấn đề của luận văn. Các vấn đề trình bày dưới đây liên quan tới ngôn ngữ "mới" này và ta vẫn gọi là PL/0.

- Về kiểu dữ liệu: PL/0 chỉ chấp nhận kiểu duy nhất là kiểu số thực (double).
- PL/0 có một số hàm dựng sẵn - đó là các hàm sin , cos , tan , arctan , ln , exp , abs , sqrt , sqr (tương tự như trong Pascal có một số hàm dựng sẵn như cos, sin...). Chương trình dịch sẽ hiểu các hàm này và sinh mã tính toán cho chúng.
- Về câu lệnh : PL/0 có các câu lệnh sau:
  - câu lệnh gán với dấu :=
  - câu lệnh gộp với BEGIN END
  - câu lệnh IF
  - câu lệnh WHILE
  - câu lệnh gọi chương trình con CALL
  - không có lệnh vào/ra

PL/0 cũng có cấu trúc khối (*block*), thể hiện khá đầy đủ khái niệm chương trình con, lời gọi chương trình con, sự khai báo và phạm vi của các đối tượng cục bộ trong chương trình con. Nó cũng cho phép cả sự đệ qui, nghĩa là một chương trình con có thể gọi tới chính nó.

### 3.2. Biểu đồ cú pháp của PL/0

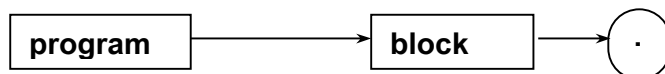
---

Sau đây là cú pháp của PL/0 diễn tả dưới dạng một hệ thống các biểu đồ cú pháp. Mỗi biểu đồ tương ứng với một ký hiệu không kết thúc của văn phạm (phi ngữ cảnh) của ngôn ngữ, hay còn gọi là một *đích triển khai* trong phân tích cú pháp của

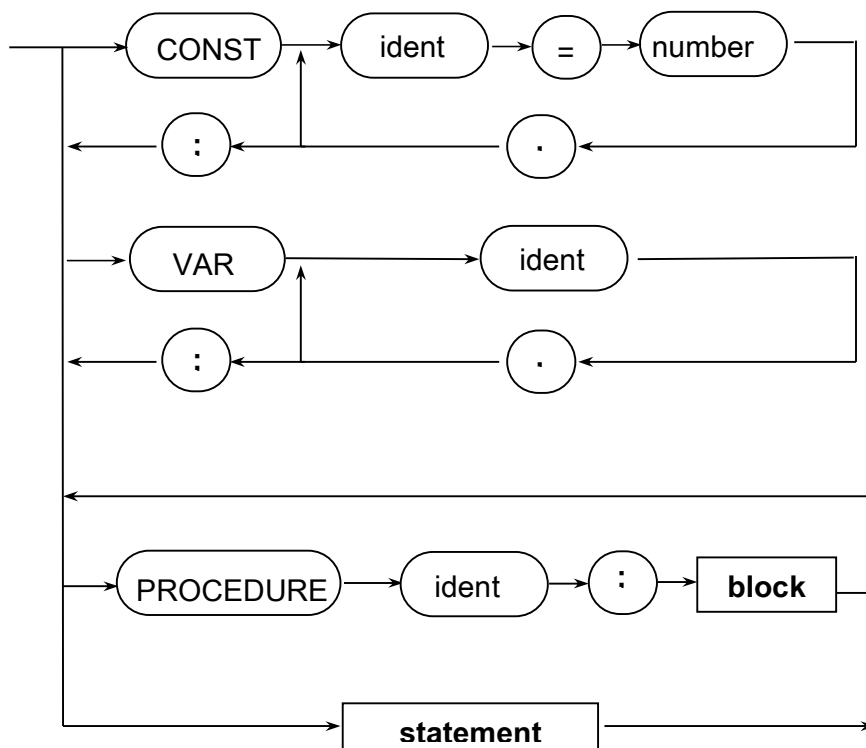
ngôn ngữ. Cách biểu diễn này rất phù hợp với phương pháp đệ qui trên xuống sử dụng trong chương trình dịch.

Trong các biểu đồ, những ký hiệu *không kết thúc* được viết trong ô chữ nhật vuông, những ký hiệu *kết thúc* được viết trong các dấu tròn hay hình chữ nhật tròn.

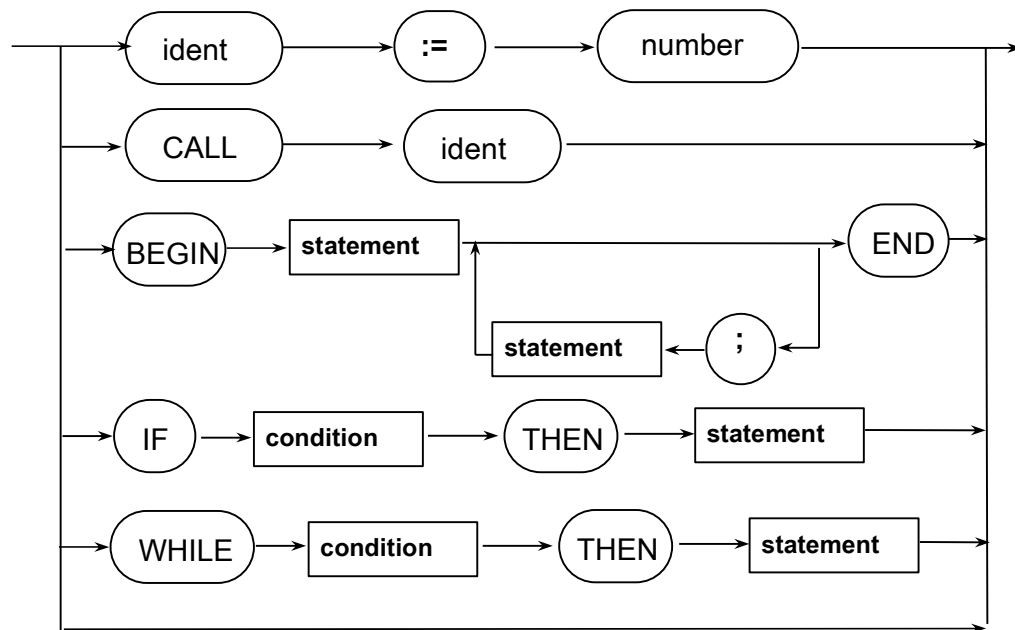
**program**



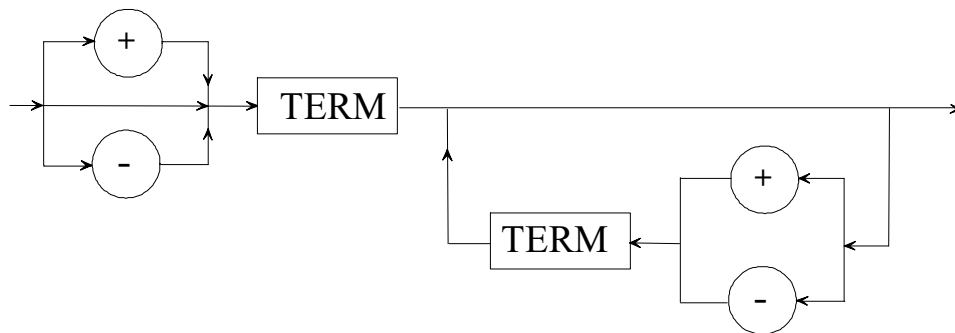
**Block**



**statement**

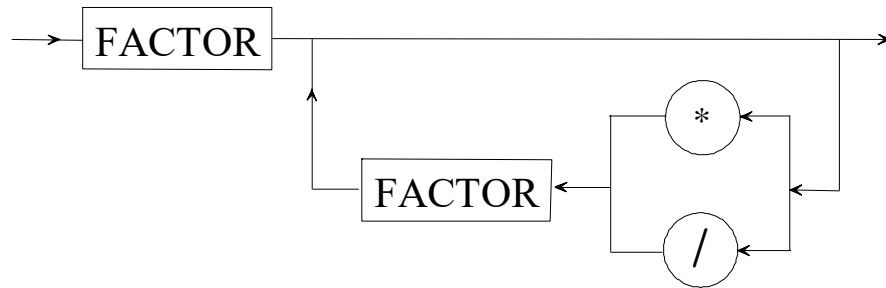


## Expression

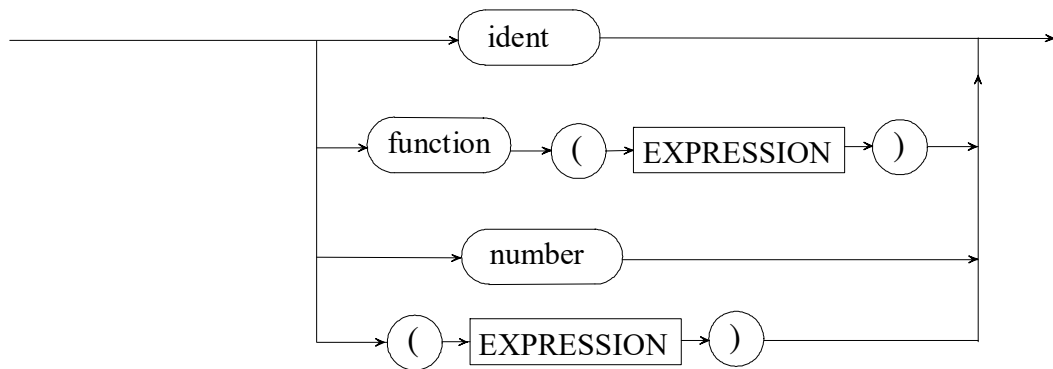




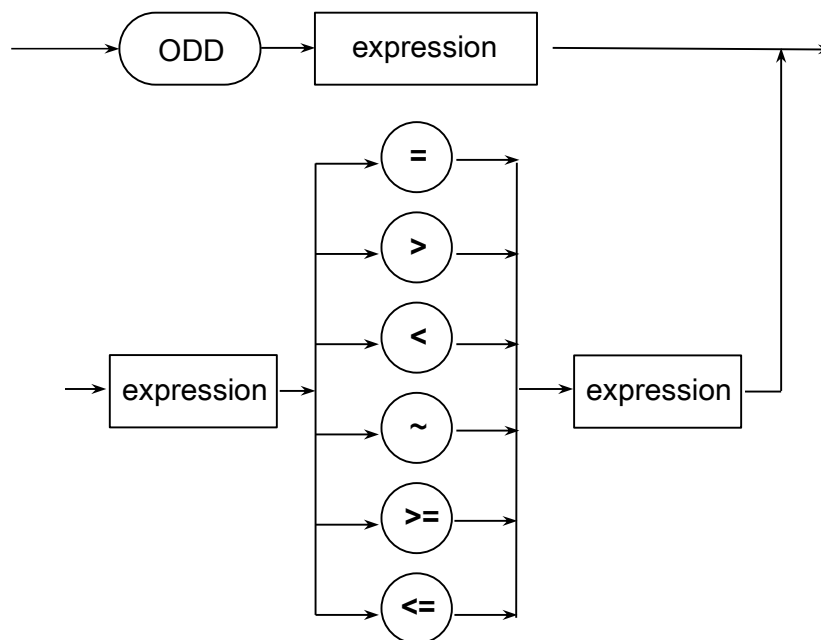
## Term



## Factor



## Condition



Chú ý tới biểu đồ **expression**. Biểu đồ này có một số thay đổi so với PL/0 nguyên bản để có thể chấp nhận một số hàm dựng sẵn. Biểu đồ **condition** cũng có thay đổi với các phép so sánh **>=**, **<=**.

# Các thành phần của trình biên dịch

Trình biên dịch PL/0 có thể chia làm 3 phần chính

- Bộ phân tích từ vựng
- Bộ phân tích cú pháp
- Máy ảo PL/0

## Chương 4

### Phân tích từ vựng

#### 4.1. Sơ đồ ô-tô-mat bộ phân tích từ vựng

Bộ phân tích từ vựng có nhiệm vụ phát hiện ra các *từ tố* (token) trong dòng dữ liệu ký tự cần phân tích (dòng ký tự này chính là văn bản chương trình nguồn).

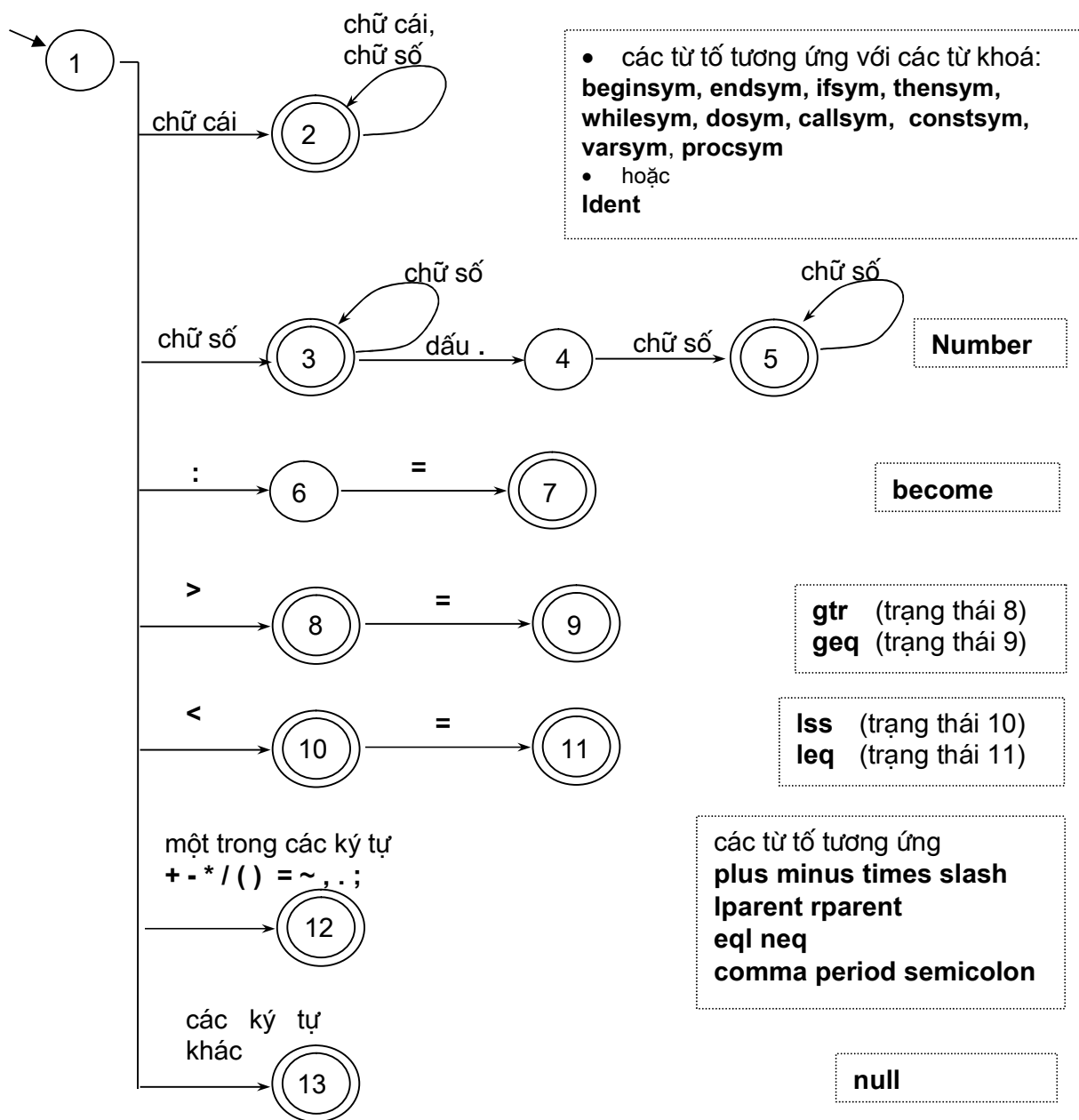
Ta coi văn bản chương trình nguồn chỉ là một chuỗi ký tự (*string*) duy nhất có chứa các ký tự phân cách hàng #13, #10 bên trong. Làm việc với một chuỗi ký tự sẽ đơn giản hơn và vị trí lỗi cũng dễ phản ánh hơn trên màn hình soạn thảo (sử dụng thành phần rich-edit làm bộ soạn thảo chương trình nguồn). Chú ý là ta sẽ dùng ngôn ngữ Object Pascal phiên bản 32bit, *string* có thể chứa tới 2GB ký tự, và do có các ký tự phân cách dòng bên trong chuỗi văn bản chương trình nguồn nên nó cũng được biểu diễn thành nhiều dòng trên màn hình soạn thảo. Như vậy độ dài thực tế của chương trình nguồn là đủ lớn.

Bộ phân tích từ vựng làm việc giống như một ô-tô-mát hữu hạn dùng để đoán nhận các từ tố. Sơ đồ ô-tô-mát như trong hình vẽ phía dưới (trang sau).

Có một số cách cài đặt khác nhau cho bộ phân tích từ vựng, hay nói chính xác hơn là cài đặt ô-tô-mát nói trên. Ví dụ phương pháp dựa trên phân tích bảng. Trong phương pháp này trước tiên ta xây dựng ma trận hàm dịch chuyển trạng thái của ô-tô-mát  $\delta$ .

Gọi  $Q = \{q_1, q_2, \dots, q_n\}$  là tập các trạng thái của ô-tô-mát,  $V = \{\text{tập các chữ cái được sử dụng trong PL/0}\} = \{'0', '1'.. '9', 'A', 'B'.. 'Z', '(', ')', '=', \dots\}$ . Ma trận hàm dịch chuyển  $\delta$  được biểu diễn dưới dạng mảng 2 chiều  $\delta(Q, V)$ , mỗi ô trong ma trận nhận giá trị trong tập  $Q$ . Như vậy ma trận này biểu diễn ánh xạ từ  $(Q, V)$  vào  $Q$ . Trong đó  $(Q, V)$  là tích Đề-các của hai tập  $Q, V$ .

Giả sử chuỗi cần phân tích là  $w$ ,  $\text{next}(w)$  trả ra ký tự kế tiếp trên xâu đầu vào. Thuật toán được biểu diễn như sau:



Sơ đồ ô-tô-mát hữu hạn mà phương thức `GetSym` thực hiện để trích ra các từ tố trong dòng các ký tự của văn bản chương trình nguồn.

Ta thấy `GetSym` có thể nhận biết được số thực dấu phẩy tính (ví dụ như 10.2340)

Phía bên phải, trong các hình vuông nét đứt là các từ tố được trả ra tương ứng vào các trạng thái kết thúc của ô-tô-mát.

```

begin
  q = q0
  c = next(w)
  while not EOL(w) do begin
    q =  $\delta(q, c)$ 
    c = next(w)
  end
  if q in F then
    DoanNhanDuoc
  else
    KhongDoanNhanDuoc
end.

```

Trong thuật toán trên

q: biến nhận giá trị trong tập trạng thái của ô-tô-mát  
 c: biến kiểu ký tự  
 q0: trạng thái đầu tiên của ô-tô-mát  
 EOL(w): hàm boolean, xác định xem đã đọc hết xâu ký tự w hay chưa

Thuật toán trên cho phép ta xác định chuỗi ký tự có được nhận diện bởi ô-tô-mát hay không, trong trường hợp của chúng ta có nghĩa là chuỗi ký tự có phải là một từ tố hay không. Như vậy trong cài đặt để chuyển chương trình nguồn thành chuỗi các từ tố ta còn cần phải cài đặt thêm một vòng lặp nữa để scan toàn bộ văn bản chương trình nguồn. Vòng lặp này sẽ nằm trong phần phân tích từ vựng. Ngoài ra sau khi nhận diện được một từ tố ta còn phải có thêm những xử lý thích hợp nhằm lưu giữ tên định danh, giá trị số mới thu được v.v...

Trong bài toán của chúng ta ô-tô-mát có 13 trạng thái, tuy nhiên ta cần thêm trạng thái không kết thúc là qnull để biểu diễn những chuyển đổi trạng thái không được thể hiện tường minh trên sơ đồ ô-tô-mat. Như vậy phải có ít nhất 14 trạng thái.

Ta thử liệt kê số lượng các ký tự:

Loại ký tự	Ký tự	Số lượng
Chữ số	0,1,...9	10
Chữ cái	A, B,...Z	26
Các ký hiệu khác	< > = + - * / ( ) . ; : , ~	14
	Blank	1
<b>Tổng số</b>		<b>51</b>

Đó là số lượng tối thiểu các ký tự cần dùng trong chương trình PL/0. Ta đã loại bỏ chữ cái thường cùng các ký tự khác. Ngoài ra trong văn bản chương trình nguồn còn có thể gặp các ký tự như &, # {}[] v.v... mà bộ phân tích từ vựng phải báo lỗi trong khi gặp chúng. Hàm chuyển  $\delta$  sẽ phải là mảng ít nhất cỡ 14x51.

Sử dụng hàm chuyển dịch có vẻ mang tính tổng quát, tuy nhiên lại ít trực quan hơn so với phương pháp diễn giải. Nó cũng khó debug hơn nếu có lỗi do chương trình không hoàn toàn bám sát sơ đồ ô-tô-mát hữu hạn mà thể hiện ô-tô-mát thông qua

hàm chuyển. Ta có thể thấy trong thuật toán trên các chương trình đoán nhận cho các ngôn ngữ khác nhau chỉ khác nhau ở hàm chuyển.

Trong luận văn này ta chọn phương pháp diễn giải. Phương pháp này thực chất là diễn giải từng nhánh trong sơ đồ ô-tô-mát hữu hạn. Với phương pháp này với mỗi ô-tô-mát ta cần viết thủ tục `GetSym` khác nhau, dù về cấu trúc các thủ tục này tương tự nhau. Phương pháp này có vẻ tự nhiên và dễ hiểu hơn phương pháp dùng bảng.

Phần 4.2. sẽ giải thích kỹ hơn về cách cài đặt bằng phương pháp này.

Bộ phân tích từ vựng được đặt trong đơn vị chương trình `TuVung.pas`.

## 4.2. Lớp phân tích từ vựng (TPhanTichTuVung)

---

Bộ phân tích từ vựng được viết lại là một đối tượng với phương thức trung tâm là `GetSym` - phương thức này trích ra từ tổ kế tiếp trong dòng ký tự cần phân tích.

Trong lúc làm việc, `GetSym` cũng xác định được định danh của biến, hằng, chương trình con, hàm dựng sẵn, giá trị của hằng số (là một số thực - dấu phẩy động).

Thủ tục `GetSym` trả lại 3 kết quả chính mà phân phân tích cú pháp sẽ quan tâm, đó là

- `Sym` : từ tổ mới đọc được
- `id` : định danh của từ tổ mới đọc được nếu từ tổ đó là thủ tục, hàm, biến hay hằng
- `Num` : nếu từ tổ mới đọc được là number thì giá trị số của tương ứng sẽ là `Num` (số thực).

Thủ tục `GetSym` gọi `GetCh` để đọc từng ký tự một trên dòng dữ liệu chương trình nguồn. Thủ tục `GetCh` đơn giản chỉ đọc ra 1 ký tự từ chuỗi đầu vào và dịch chuyển con trỏ ký tự hiện thời thêm một bước (tăng `cc` lên 1 đơn vị). Ngoài ra nó cũng sẽ lưu lại giá trị `cc` của lần gọi trước (tức vị trí cuối của từ tổ trước) để sau này định vị lỗi được chính xác hơn.

Để xác định được ranh giới bên phải của một từ tổ thì `GetSym` đọc lần lượt các ký tự cho đến khi gặp phải một ký tự mà nó xác định là không thể thuộc vào thành phần của từ tổ đó. Vì vậy khi phát hiện xong một từ tổ thì `GetSym` bao giờ cũng đọc dôi ra một ký tự, ký tự này có thể là dấu trắng hoặc là ký tự đầu tiên của từ tổ tiếp theo. Do đó mỗi lần được khởi động để tìm một từ tổ mới, `GetSym` làm việc ngay với ký tự đã được đọc sẵn từ trước đó mà không cần gọi `GetCh`. Đương nhiên lần đầu tiên, khi cần xác định từ tổ thứ nhất trong văn bản nguồn, thì chương trình chính phải cung cấp ký tự này trước khi gọi `GetSym`. Cụ thể hơn như sau:

Bắt đầu vào chương trình con `GetSym` sẽ bỏ qua các ký tự phân cách. Sau khi đọc được một ký tự đầu tiên khác ký tự phân cách `GetSym` sẽ phân tích

1. Nếu đó là một chữ cái thì đó sẽ là tên hay từ khóa. Ô-tô-mát lúc này ở trạng thái 2. và `GetSym` tiếp tục gọi `GetCh` để đọc cho tới khi gặp một ký tự không nằm trong tập `['A'..'Z', '0'..'9']`. (chú ý là `GetCh` luôn trả lại chữ hoa). Chuỗi ký tự thu được được kiểm tra xem có phải là từ khóa hay không.
  - Nếu là từ khóa thì giá trị của `Sym`-tức từ tố mới đọc được sẽ là từ tố tương ứng với từ khóa (như `beginsym`, `endsym`, `ifsym`, `thensym`, `whilesym`, `dosym`, `callsym`, `constsym`, `varsym`, `procsym`)
  - Nếu không đó sẽ là một tên và từ tố mới đọc được sẽ là `ident` và `id` sẽ là tên của biến, hằng hay thủ tục vừa đọc được.
2. Nếu là số nó sẽ gọi `GetCh` đọc liên tiếp các ký tự số cũng như dấu chấm và cố phân tích xem đây có phải là một số thực hay không. Nếu không chuyển đổi được chuỗi ký tự vừa đọc thành số thực thì sẽ báo lỗi. Nếu là số thực thì từ tố mới đọc được là `number` và giá trị của nó là `num`.
3. Nếu đó là ký tự `'.'` thì nó đọc tiếp một ký tự và xác định xem có phải là ký tự `'='` hay không. Nếu đúng thì đó là phép gán (từ tố `becomes`).
4. Nếu đó là ký tự `'>'` thì nó đọc tiếp một ký tự và xác định xem có phải là ký tự `'='` hay không. Nếu đúng thì đó là phép so sánh `>=` (từ tố `geq`), nếu không thì đó là từ tố `gtr`.
5. Nếu đó là ký tự `'<'` thì nó đọc tiếp một ký tự và xác định xem có phải là ký tự `'='` hay không. Nếu đúng thì đó là phép so sánh `<=` (từ tố `leq`), nếu không thì đó là từ tố `lss`.
6. Ngoài ra nếu là các ký tự `+`, `-`, `*`, `/`, `=`, `,`, `.`, `~`; thì `Sym` sẽ mang giá trị của từ tố tương ứng.
7. Trong các trường hợp khác: trả lại từ tố `null`.

Khi kết thúc xử lý mỗi nhánh bên trên chương trình luôn luôn gọi `GetCh`, tức là luôn có một ký tự được đọc dôi ra cho lần gọi `GetSym` kế tiếp.

Ngoài ra trong còn có các phương thức liên quan tới việc thông báo lỗi.

Sau đây là cài đặt cụ thể của phương thức `GetSym`

```
procedure TPhanTichTuVung.GetSym;  
  procedure GetCh;  
  begin  
    if cc = Length(Line) then begin  
      Ch := #255; // từ tố tương ứng sẽ là null  
    end else begin
```

```

        Inc(cc);
        Ch:= Line[cc];
        FOrgCh := FOrgLine[cc];
    end;
end;

var
    i, j, k: integer;
    s1, s2: string;
    Code: integer;

begin { GetSym }
    LastChar:= CC;
    while Ch in [' ', #0..#31] do GetCh; {skip white}
    if Ch in ['A'..'Z'] then begin { danh bieu hay reserved word}
        Fid:= Ch;
        FOrgID:= FOrgCh;
        GetCh;

        while Ch in ['A'..'Z', '0'..'9'] do begin
            Fid:= Fid + Ch;
            FOrgID:= FOrgID + FOrgCh;
            GetCh;
        end;
        i:= 1;
        j:= nrow;

        { binary search }

        repeat
            k:= (i+j) div 2;
            if id <= word[k] then j:= k-1;
            if id >= word[k] then i:= k+1;
        until i > j;

        if i - 1 > j then // đây là từ khoá
            Fsym := wsym[k]
        else
            Fsym := ident;

    end else if Ch in ['0'..'9'] then begin
        { số thực }
        k := 0;
        Fnum:= 0;
        Fsym:= number;
        S1:= '';
        repeat
            S1:= S1 + Ch;
            k:= k+1;
            GetCh;
        until not( Ch in ['0'..'9']);
        S2:='';

        if Ch='.' then begin
            S2 :='.';
            GetCh;
            repeat
                S2:= S2 + Ch;
                k:= k+1;
                GetCh;
            until not( Ch in ['0'..'9']);

```



```

    end;

    Val(S1+S2, FNum, Code);
    if Code <> 0 then begin
        Error(31);
    end;

    if k > nmax then Error(30);
end else if Ch = ':' then begin
    GetCh;
    if Ch = '=' then begin
        FSym := becomes;
        GetCh;
    end else
        FSym := null
end else if Ch = '>' then begin
    GetCh;
    if Ch = '=' then begin
        FSym := geq;
        GetCh;
    end else
        FSym := ssym['>'];
end else if Ch = '<' then begin
    GetCh;
    if Ch = '=' then begin
        FSym := leq;
        GetCh;
    end else
        FSym := ssym['<'];
end else begin
    FSym := ssym[Ch];
    GetCh;
end;
end; { GetSym }

```

## Chương 5

# Phân biên dịch (Phân tích cú pháp, ngữ nghĩa)

### 5.1. PL/0 là ngôn ngữ LL(1)

Chương trình biên dịch PL/0 dùng phương pháp đệ qui trên xuống để phân tích cú pháp. Để vận dụng được phương pháp này thì văn phạm của ngôn ngữ phải là một văn phạm LL(1). Văn phạm LL(1) là văn phạm thỏa mãn 2 điều kiện sau:

#### Điều kiện 1

Với mọi sản xuất có dạng

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

thì phải có

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \Phi \text{ với mọi } i, j$$

#### Điều kiện 2

Với mọi ký hiệu không kết thúc A mà  $A \Rightarrow^* \epsilon$  thì phải có

$$\text{FIRST}(A) \cap \text{FOLLOW}(A) = \Phi$$

Ta hiểu  $\text{FIRST}(\alpha)$  là tập các ký hiệu kết thúc đứng đầu một xâu nào đó suy dẫn từ  $\alpha$ , còn  $\text{FOLLOW}(A)$  là tập các ký hiệu kết thúc đứng liền sau A trong một dạng câu nào đó suy dẫn từ ký hiệu đầu S.

**Điều kiện 1** đòi hỏi mỗi lối rẽ trong biểu đồ cú pháp các nhánh phải bắt đầu bằng các ký hiệu (kết thúc) không giống nhau. Như vậy trong khi triển khai đích A, khi nhìn trước một ký hiệu ta có thể biết được phải rẽ nhánh theo hướng nào trong các hướng  $\alpha_1, \alpha_2, \dots, \alpha_n$ . Mọi biểu đồ của PL/0 đều thỏa mãn điều kiện này.

**Điều kiện 2:** áp dụng vào các biểu đồ cú pháp có chứa một đường rỗng. Trong PL/0 chỉ có một biểu đồ có đường rỗng, đó là biểu đồ *statement*. Điều kiện 2 đòi hỏi trong trường hợp đó thì mọi ký hiệu đứng liền sau các câu lệnh phải khác biệt với các ký hiệu đứng đầu các câu lệnh.

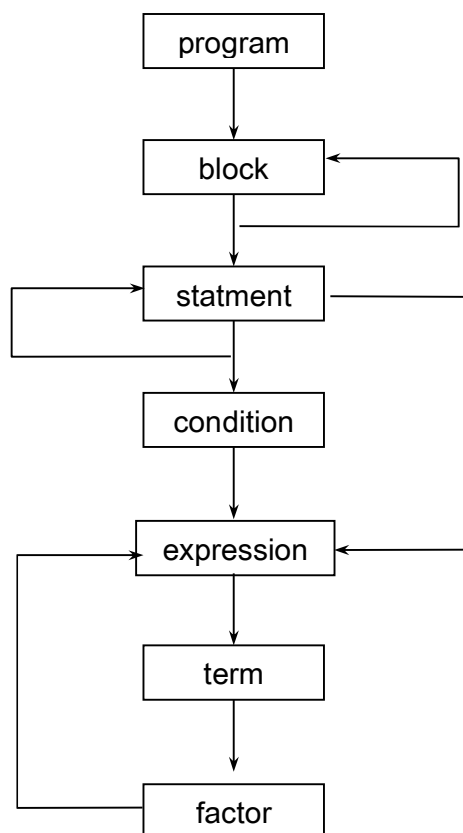
A	FIRST(A)	FOLLOW(A)
block	CONST VAR PROCEDURE ident IF WHILE CALL BEGIN	. ;
statement	ident IF WHILE CALL BEGIN	. ; END
condition	ODD + - ident number (	THEN DO

expression	+ - ( ident number	. ; R END THEN DO )
term	ident number (	. ; R END THEN DO ) + - )
factor	ident number (	. ; R END THEN DO ) + - ) * /

Chú thích R là tập ký hiệu { = ~ < > , >= , <= }

## 5.2. Bộ phân tích cú pháp

a) Bộ phân tích cú pháp là một tập các thủ tục. Mỗi thủ tục có nhiệm vụ suy dẫn một ký hiệu không kết thúc của văn phạm thành một đoạn trên văn bản chương trình nguồn. Ta gọi công việc của thủ tục là *triển khai một đích*. Có bao nhiêu ký hiệu không kết thúc (tức là bao nhiêu biểu đồ cú pháp) thì có bấy nhiêu thủ tục. Khi triển khai một đích (dựa vào biểu đồ cú pháp) lại có thể gặp một ký hiệu không kết thúc, đòi hỏi phải triển khai một đích mới (đích con) trong khi đích cũ chưa triển khai xong. Bởi vậy ta thấy các thủ tục phân tích cú pháp sẽ gọi lẫn nhau ( và gọi tới chính nó). Để biết thủ tục nào sẽ gọi thủ tục nào, ta hãy nhìn vào biểu đồ cú pháp xem trong đó có những nút (hình chữ nhật) chứa ký hiệu không kết thúc nào. Đồ thị sau diễn tả quan hệ gọi nhau giữa các thủ tục, trong đó mũi tên có nghĩa là "gọi tới".



b) Quá trình phân tích cú pháp theo lối đệ qui trên xuống diễn ra như sau:

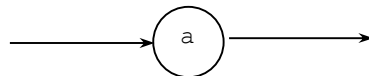
Tại mỗi thời điểm luôn có một đích đang được triển khai - tức là thủ tục tương ứng với đích đó đang được thực hiện, và việc kiểm tra cú pháp đã tiến đến một điểm nào đó trên văn bản chương trình nguồn. Bên trái điểm đó là phần văn bản nguồn đã được kiểm tra cú pháp, còn bên phải là phần chưa kiểm tra. Lúc khởi đầu thì đích được triển khai là *block* vì ta biết chương trình nguồn là một *block* với một chấm theo sau (biểu đồ cú pháp đầu tiên), còn điểm kiểm tra trên văn bản nguồn lúc đó là điểm khởi đầu văn bản.

Công việc của một thủ tục triển khai một đích là tiến hành kiểm tra một đoạn trên văn bản nguồn bằng cách đối chiếu nó với một đường trên biểu đồ cú pháp (từ lối vào đến lối ra của biểu đồ), xem đoạn văn bản đó có thể hiện đúng cú pháp diễn tả bởi các nút trên đường đó không.

Ở mỗi bước của quá trình đó, thì trước hết từ tố tiếp đến trên văn bản nguồn được đọc vào, rồi nó được đem đối chiếu với nút tiếp đến trên biểu đồ. Nếu nút đó là nút tròn thì nút đó phải chứa từ tố vừa đọc. Còn nếu nút đó là nút chữ nhật có chứa ký hiệu không kết thúc A, thì từ tố vừa đọc phải thuộc vào  $FIRST(A)$ , và bây giờ phải triển khai đích con A. Nếu không khớp được như vậy thì văn bản nguồn có lỗi cú pháp tại từ tố đang xét.

Để viết các thủ tục phân tích cú pháp thể hiện quá trình làm việc nói trên, ta dựa vào biểu đồ cú pháp tương ứng, rồi áp dụng các qui tắc

### 1. Biểu đồ



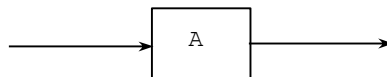
trong đó a là một ký hiệu kết thúc sẽ dùng đoạn chương trình như sau để kiểm tra

```

if Sym = a then
  GetSym
else
  Error('Thiếu a')
  
```

trong đó *Sym* là từ tố vừa đọc được trên văn bản chương trình nguồn, *GetSym* là thủ tục đọc từ tố kế tiếp trên văn bản chương trình nguồn.

### 2. Biểu đồ



trong đó A là một ký hiệu không kết thúc sẽ dùng đoạn chương trình như sau để kiểm tra

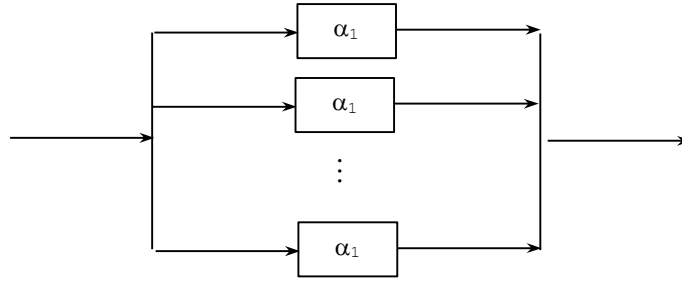
```

call A;
  
```

tức gọi thủ tục có tên A để kiểm tra.

### 3. Nếu có sản xuất $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ tức có biểu đồ tương ứng

và các  $\alpha_i$  đã được lập sơ đồ cú pháp dùng đoạn lệnh sau để kiểm tra



```

if Sym  $\in$  FIRST( $\alpha_1$ ) then T( $\alpha_1$ )
else if Sym  $\in$  FIRST( $\alpha_2$ ) then T( $\alpha_2$ )

...

else if Sym  $\in$  FIRST( $\alpha_n$ ) then T( $\alpha_n$ )
else Error('Sym này không thể có ở đây').
  
```

với giả thiết rằng  $T(x)$  là câu lệnh diễn dịch cho  $x$   
 nếu  $x = a$  thì đó là lệnh

```

if Sym = a then
  GetSym
else
  Error('Thiếu a')
  
```

còn nếu  $x = A$  thì đó là lệnh  
 Call A;

#### 4. Nếu có sản xuất

$A \rightarrow \alpha_1 \alpha_2 \dots \alpha_n$  thì có biểu đồ cú pháp

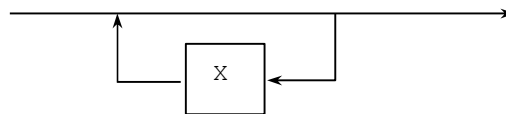


thì ta dùng các câu lệnh

```

T( $\alpha_1$ ); T( $\alpha_2$ ); T( $\alpha_n$ );
  
```

Nếu có  $A \rightarrow \{X\}$  tức  $A \rightarrow X, A \rightarrow XX, A \rightarrow X^*$   
 tức có sơ đồ cú pháp



thì ta dùng các câu lệnh sau

```

while Sym  $\in$  FIRST(X) do T(X);
  
```

- c) Chú ý rằng trong quá trình phân tích cú pháp nói trên, từ tố tiếp đến luôn luôn được đọc trước, rồi mới đối chiếu cú pháp sau. Chính việc đọc trước này cho phép chọn đúng đường đi trên biểu đồ khi gặp một chỗ rẽ nhánh (bởi tính chất của văn

phạm LL(1)). Cũng từ đó ta lại thấy khi ra khỏi một thủ tục phân tích cú pháp (tức triển khai xong một đích), thì một từ tố đã được đọc dôi ra. Từ tố này không thuộc cấu trúc A vừa triển khai, mà thuộc FOLLOW(A), và sẽ là từ tố đứng đầu của cấu trúc đứng tiếp sau A. Vì vậy các thủ tục phân tích cú pháp lúc vào đầu đều được khởi động làm việc ngay với một từ tố đã được đọc sẵn mà không phải gọi `GetSym`. Đương nhiên lúc xuất phát của quá trình phân tích cú pháp thì chương trình chính phải gọi `GetSym` để đọc từ tố đầu tiên của văn bản nguồn, rồi mới gọi phương thức triển khai đích là `Block`.

d) Để ghi nhận từ tố tiếp đến ( bởi thủ tục `GetSym`) thì ta dùng các kiểu dữ liệu sau (khai báo trong đơn vị chương trình `TuVung.pas`)

- Kiểu `symbol` cho mã của các từ tố. Trong đó
  - các danh biểu (tên) có mã chung là `ident`
  - các số có mã chung là `number`
  - các từ tố không hợp lệ có mã chung là `null`
  - phép gán có mã là `becomes` ( ký hiệu của phép gán là hai ký tự liên tiếp :=)
  - các từ khóa có các mã từ tố riêng. Ta dùng biến mảng `word` và `wsym` để chứa 11 từ khóa và từ tố tương ứng của chúng. `GetSym` sẽ tra trong bảng này để xác định xem một từ mà nó vừa đọc được, nếu bắt đầu bằng chữ cái có phải là từ khóa hay không. Nếu là từ khóa thì nó sẽ đặt giá trị của trường `FSym` là từ tố tương ứng, nếu không thì `FSym` sẽ có giá trị là `ident` và tên tương ứng được lưu trong `Fid`.
  - Mảng `ssym` có kiểu chỉ số là `char`. Các ký tự `+ - * / ( ) = ~ < > , . ;` có các từ tố tương ứng trong mảng `ssym` là `plus minus times slash lparent rparent eql neq lss gtr comma period semicolon`. Còn các phần tử khác trong mảng `ssym` chứa mã từ tố `null`.

### 5.3. Cài đặt trong lớp `TPhanTichTuVung`

Trong lớp phân tích từ vựng `TPhanTichTuVung` có các trường:

`FSym`: kiểu `Symbol` cho mã các từ tố.

`Fid`, kiểu `string` chứa danh biểu (tên) của từ tố vừa đọc được (chỉ có ý nghĩa khi từ tố vừa đọc được, tức `FSym` có giá trị là `ident`)

`FNum`: kiểu `double`. Nếu từ tố vừa đọc được, tức `FSym` là `number` thì `FNum` chứa giá trị số của nó.

Ngoài ra còn có các trường mang tính phụ trợ, một trong số đó là:

`FOrgId`: tên "nguyên gốc" của danh biểu. Trong khi `Fid` chứa danh biểu toàn chữ hoa thì `FOrgId` giữ tên danh biểu nguyên bản như nó được viết trong chương trình nguồn. Ví dụ trong chương trình nguồn ta viết `y:= Sin(GiaTriX)` thì các giá trị của `Fid` sẽ lần lượt là `Y`, `SIN`, `GIATRIX` còn giá trị của `FOrgId` sẽ là `y`, `Sin`, `GiaTriX`. Giá trị của

F`OrgId` được ghi nhận để vẽ công thức gắn với công thức được nhập vào hơn.

Vì các trường trên được khai báo là `private` nên chúng không thể truy cập được từ các đoạn mã bên ngoài tập tin chương trình nguồn này. Để truy nhập các trường này ta dùng các các property chỉ đọc (`read only`) tương ứng là `sym`, `id`, `Num`, `OrgId`... Đây là một thủ thuật rất hay dùng trong lập trình hướng đối tượng để bảo đảm tính đóng kín của đối tượng, cho phép viết chương trình "an toàn" hơn.

Trường `Table`, kiểu mảng, đóng vai trò bảng ký hiệu. Mỗi phần tử của nó có kiểu

```
TIdentInfo = record
  Name: alpha; { ten danh bieu }
  case kind: object_ of // kind: object_ ; { loai danh bieu }
    constant: (val: Double);
    variable, procedure_ : (level, adr: Integer; LValue, RValue: Boolean);
                          // LValue, RValue : chỉ dùng cho bien
end;
trong đó object_ được khai báo như sau:
object_ = (constant, variable, procedure_);
```

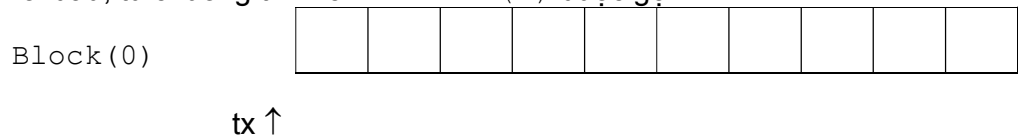
`Table` là nơi lưu giữ các danh biểu, tức các từ do người dùng đặt ra làm tên gọi cho các đối tượng trong chương trình như hằng, biến hay chương trình con. Trong quá trình phân tích cú pháp, mỗi khi gặp một danh biểu được *khai báo*, thì bộ phân tích cú pháp đưa nó vào trong bảng cùng với loại tương ứng. Còn khi gặp một đối tượng dùng trong một câu lệnh, thì bộ phân tích cú pháp lại phải lục tìm xem nó đã được khai báo và sử dụng phù hợp với khai báo đó không. Quá trình sử dụng bảng này liên quan tới phân tích ngữ nghĩa.

Các trường `LValue`, `RValue` trong bảng ghi `TIdentInfo` dùng để xem một biến xuất hiện ở bên trái, bên phải của một biểu thức nào đó hay không. Trong quá trình phân tích, nếu thấy biến nằm bên trái một phép gán nào đó thì `LValue` được cập nhật là `True`, nếu thấy xuất hiện trong biểu thức nào đó thì `RValue` được cập nhật `True`. Khi cả hai trường này đều là `False` thì biến được khai báo nhưng không được sử dụng.

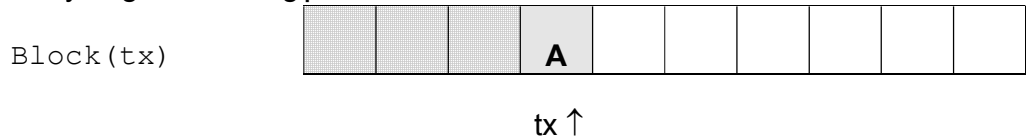
Bảng được tổ chức theo lối tuần tự. Các danh biểu được lần lượt đưa vào bảng theo các vị trí kế tiếp. Phần tử mới nhất trong bảng có chỉ số là `tx`. Chú ý rằng `tx` không phải là một biến, mà lại là một tham trị của phương thức `Block`. Khi phương thức `Block` được gọi để xử lý một khối (*block*) mới thì nó tiếp nhận giá trị cuối cùng của `tx`. Sau đó từng danh biểu cục bộ trong `Block` sẽ được lần lượt đưa vào bảng, làm tăng `tx` lên dần dần. Song tham trị cũng như biến cục bộ, chỉ tồn tại khi thủ tục đang hoạt động. Ra khỏi thủ tục, giá trị ghi nhận của chúng không còn nữa, và ta lại trở về với `tx` của thủ tục gọi. Chính vì vậy mà khi phân tích xong một khối thì các danh biểu cục bộ trong `Block` không còn có thể tìm lại trong bảng được nữa.

Có thể hình dung sự dịch chuyển của  $tx$  như sau:

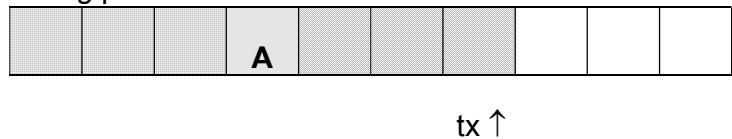
Lúc khởi đầu, từ chương trình chính  $Block(0)$  được gọi



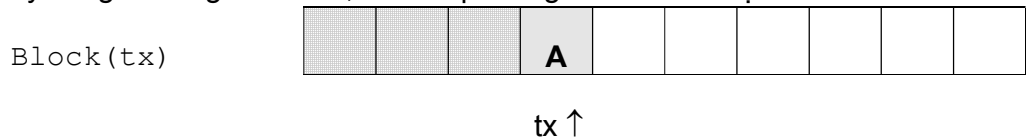
Đến một lúc  $Block(tx)$  được gọi để xử lý chương trình con A nào đó trong chương trình nguồn. Lúc đó bảng đã chứa sẵn một số danh biểu, và  $tx$  đang trỏ vào phần tử của bảng có chứa tên gọi A của chương trình con. Chính giá trị này của  $tx$  là tham số thực sự để chuyển giao cho lời gọi  $Block(tx)$



Tiếp đó một số danh biểu mới thuộc chương trình con đang được xử lý được ghi nhận lần lượt vào bảng, và  $tx$  dịch dần sang phải



Đến khi xử lý xong chương trình con, ra khỏi phương thức  $Block$  lại có



**Chú ý:** A là tên của một đối tượng thuộc khối mẹ chứ không thuộc khối vừa xử lý xong, do đó A vẫn còn sau khi đã loại bỏ mọi đối tượng cục bộ trong khối vừa xử lý.

Việc tìm kiếm một danh biểu trong bảng được thực hiện theo kỹ thuật lính canh (sentinel). Do đó có phần tử mang chỉ số 0 để đặt danh biểu cần tìm (làm lính canh) mỗi lần tiến hành tìm kiếm. Quá trình tìm kiếm được tiến hành từ phải qua trái, bắt đầu từ vị trí hiện thời của  $tx$ . Lúc đó trong bảng chỉ chứa toàn các danh biểu thuộc các khối bao nhau. Việc tìm kiếm từ phải qua trái trên bảng thể hiện phép duyệt các danh biểu đó từ khối trong ra khối ngoài, nghĩa là nó duyệt đúng những danh biểu được quyền truy cập theo đúng sự đòi hỏi của cấu trúc khối. Hơn nữa nếu có hai đối tượng trùng tên, thì cách tìm kiếm trên chỉ cho phép tìm ra đối tượng thuộc *khối trong* mà không đến được đối tượng thuộc khối ngoài. Điều đó thể hiện một nguyên tắc của cấu trúc khối là khả năng định nghĩa lại một danh biểu và sự che khuất của chúng.

#### 5.4. Các hàm dựng sẵn (đơn vị chương trình HamDungSan.pas)

Trình biên dịch có thể hiểu được một số hàm dựng sẵn và sẽ sinh mã cho các hàm này. Máy ảo PL/0 cũng sẽ hiểu những hàm này và thực thi nó trong khi chạy.



Trong khi xử lý một biểu thức, nếu gặp một danh biểu không có tên trong bảng và nằm trong biểu thức thì danh biểu đó có thể là tên một hàm dựng sẵn. Lúc này trình biên dịch sẽ kiểm tra xem danh biểu có phải là một hàm dựng sẵn hay không. Nếu là hàm dựng sẵn, nó sẽ tiếp tục kiểm tra xem cú pháp gọi hàm có đúng hay không, còn trong các trường hợp khác thì chương trình PL/0 bị viết sai cú pháp và thông báo lỗi sẽ được đưa ra.

Nếu một tên được khai báo trùng với tên hàm dựng sẵn thì khai báo này sẽ đè mất tên hàm. Điều này có nghĩa là trong một biểu thức thì hàm dựng sẵn được khai báo trước tất cả các biến, hằng khác (tương tự như trong ngôn ngữ lập trình Pascal).

Các hàm dựng sẵn hiện có là  
`sin, cos, tan, arctan, ln, exp, sqrt, sqr, abs`

Hiện thời hàm dựng sẵn là các hàm toán học chỉ có một tham số thực. Số lượng các hàm có thể tăng thêm theo nhu cầu. Việc thêm một hàm chỉ liên quan tới các thủ tục/hàm trong đơn vị chương trình `HamDungSan.pas`, ngoài ra không phải sửa đổi mã nguồn ở bất cứ chỗ nào khác. Điều này giúp cho việc bổ sung thêm các hàm dựng sẵn được dễ dàng. Tuy nhiên nếu hàm dựng sẵn có cách vẽ đặc biệt thì phải thêm một số dòng lệnh liên quan tới vẽ hàm (Xem hàm `abs`).

Hàm dựng sẵn cũng có ưu điểm là tốc độ tính toán cao do tính toán thực hiện không qua mã thông dịch của máy ảo mà thực hiện tính toán dựa vào thư viện thời gian chạy của Object Pascal.

Việc cài đặt các hàm dựng sẵn cho phép vẽ đồ thị, giải phương trình... với một lớp các hàm tương đối rộng, làm cho chương trình có một ý nghĩa thực tế nhất định chứ không chỉ đơn thuần là minh họa lý thuyết.

## 5.5. Ghi nhận các biểu thức

---

Ta biết trong PL/0 biểu thức có thể xuất hiện trong

- Vế trái câu lệnh gán
- Điều kiện trong câu lệnh `while`
- Điều kiện trong câu lệnh `if`

Trong phần xây dựng cây văn phạm của biểu thức ta đã biết cách ghi nhận cây của các biểu thức trong quá trình biên dịch.

Tuy nhiên những biểu thức trong câu lệnh `while` và `if` là những biểu thức logic, không phải là các biểu thức số học nên "Phân tích BTSH" chỉ giới hạn việc tìm các biểu thức trong các *câu lệnh gán* trong quá trình biên dịch. Hơn nữa các câu lệnh gán

trong các chương trình con cũng bị bỏ qua do việc tính toán giá trị của chúng có những phức tạp nhất định (việc vẽ không có cản trở gì).

Trình biên dịch được cài đặt thông qua đối tượng thuộc lớp `TCompiler` trong đơn vị chương trình `PL0Paser.pas`.

## 5.6. Lớp `TCompiler`

---

Trong `TCompiler` có khai báo các trường

```
Table: array[0..txmax] of TIdentInfo;  
TableCount: Integer;  
FVirtualCPU: TVirtualCPU;  
FPhanTichTuVung: TPhanTichTuVung;  
FNodeStack: TNodeStack;
```

sau khi gọi phương thức `ParseBuf(S: string)` quá trình biên dịch sẽ gọi `FPhanTichTuVung` để phân tích từ vựng đầu vào, sau đó sản sinh ra các thông tin:

- 1- Thông tin về các định danh dùng trong chương trình được lưu trong `Table`. `TableCount` là số phần tử trong bảng `Table`;
- 2- Mã chương trình được lưu trong `FVirtualCPU`;
- 3- Danh sách các cây phân tích của những biểu thức số học (ở mức tổng thể) được lưu trong `FNodeStack`.

## Chương 6

# Máy ảo PL0-VM và sinh mã

Khi biên dịch trình biên dịch sinh mã cho máy ảo PL0. Đây là một máy tính giả định có cấu trúc đơn giản do vậy khi sinh mã sẽ dễ dàng hơn.

Máy ảo PL/0 được cài đặt thông qua lớp `TVirtualCPU` trong đơn vị chương trình `Interpreter.pas`.

### 6.1. Các thanh ghi của máy ảo PL0

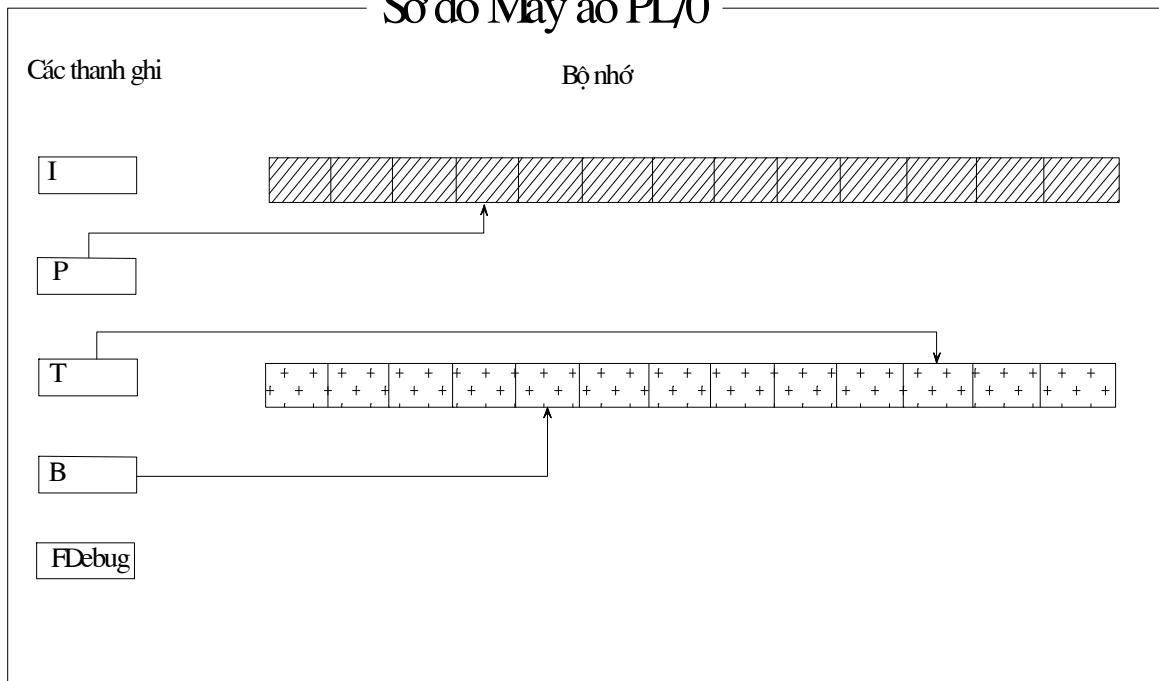
---

1. Thanh ghi lệnh `I`: dùng để chứa lệnh đang được thực hiện;
2. Con trỏ lệnh `P` (Program counter): Trỏ tới lệnh kế tiếp sẽ được thực hiện trong chương trình đích;
3. Thanh ghi địa chỉ `T`: Trỏ tới đỉnh hiện thời của ngăn xếp;
4. Thanh ghi địa chỉ `B` trỏ tới đáy của đoạn dữ liệu trên cùng ở trong ngăn xếp (tức là nơi bắt đầu của đoạn dữ liệu thuộc thủ tục đang được thực hiện);



Cả 4 thanh ghi trên được khai báo như những biến cục bộ trong phương thức `TVirtualCPU.interpret`

5. Thanh ghi trạng thái `FDebug`: được khai báo như là trường của lớp `TVirtualCPU`. Khi `FDebug` có giá trị `True` thì khi quá trình thực hiện từng lệnh sẽ được ghi ra tập tin ra chuẩn để xem (tab `CRT` trong chương trình). Điều này có ý nghĩa khi ta muốn xem các lệnh, kết quả tính toán trung gian xảy ra thế nào trong khi chạy chương trình trên máy ảo PL/0. Chạy trong chế độ `FDebug` là `true` sẽ chậm hơn đáng kể so với khi `FDebug` là `False`.

## Sơ đồ Máy ảo PL/0



### Chú thích

-  Bộ nhớ chương trình
-  Bộ nhớ dữ liệu

### Cấu trúc 1 "ô nhớ" trong bộ nhớ chương trình (cấu trúc lệnh)

f (mã lệnh)	l (chênh mức hay mã hàm số)	a (Integer) hoặc FloatValue (Double)
----------------	-----------------------------------	---

### Cấu trúc 1 "ô nhớ" trong bộ nhớ dữ liệu

IValue (Integer) hoặc FValue (Double)
--

Máy ảo PL/0 có hai bộ nhớ

## 6.2. Bộ nhớ chương trình

Bộ nhớ chương trình dùng để chứa các lệnh đích được trình biên dịch sản sinh ra. Nó được khai báo như một mảng `FCode` trong đối tượng `TVirtualCPU`. Bộ nhớ chương trình không thay đổi gì trong quá trình thực hiện các lệnh đích.

### 6.3. Dạng thức lệnh

Mỗi lệnh của máy ảo PL/0 được lưu trữ trong bộ nhớ chương trình như sau:

```
TRInstruction = packed record
  OpCode: fct;           { ma lenh }
  NestedLevel: Byte;     { Muc Cung co the chua ord(func) }
  case Integer of
    0: (a: 0..amax);     { dia chi hay do doi }
    1: (FloatValue: double);
  end;
```

OpCode (mã lệnh)	NestedLevel (chênh mức hay mã hàm số)	a (Integer) hoặc FloatValue (Double)
---------------------	---	---

### 6.4. Bộ nhớ dữ liệu

Bộ nhớ dữ liệu của máy ảo được tổ chức dưới dạng ngăn xếp. Đơn vị dữ liệu của ngăn xếp là TData được khai báo trong đơn vị chương trình `Interpreter.pas` như sau:

```
TData = record
  case Integer of
    0: (IValue: Integer);
    1: (FValue: Double)
  end;
```

Như vậy dữ liệu trên ngăn xếp vừa có thể là số thực, vừa có thể là số nguyên (IValue và FValue dùng chung vùng nhớ). Khi một lệnh cần làm việc với dữ liệu số nguyên nó sử dụng giá trị IValue, khi cần làm việc với số thực nó sử dụng FValue.

Mỗi lệnh sẽ hiểu dữ liệu theo cách của mình:

- Những lệnh làm việc với địa chỉ như lệnh trở về, lệnh gọi chương trình con sẽ sử dụng dữ liệu trên ngăn xếp như là số nguyên, các lệnh so sánh cũng lưu kết quả so sánh như là số nguyên (để thể hiện giá trị Boolean), lệnh nhảy có điều kiện cũng coi giá trị trên đỉnh ngăn xếp là số nguyên (đây là kết quả so sánh nên thực chất là giá trị Boolean);
- Các lệnh số học, gọi hàm coi các giá trị cần dùng trên ngăn xếp là các số thực, kết quả tính toán biểu thức được lưu lại trên ngăn xếp cũng là số thực. Khi so sánh hai giá trị trên ngăn xếp cũng được coi là các số thực;
- Các phép tính số học +, -, \*, / và các phép so sánh làm việc với hai phần tử nằm trên đỉnh ngăn xếp. Sau khi tính toán xong hai phần tử này bị loại bỏ và kết quả phép toán được đưa trở lại vào ngăn xếp;

- Phép gọi hàm dựng sẵn làm việc với đỉnh ngăn xếp. Sau khi tính toán xong phần tử này bị loại bỏ và kết quả tính toán của hàm được đưa trở lại vào ngăn xếp;
- Toán tử một ngôi - (đảo dấu) coi phần tử trên đỉnh ngăn xếp là số thực, nó thực hiện việc đảo dấu phần tử này.

## 6.5. Tổ chức trong bộ nhớ dữ liệu

Máy ảo chỉ có một bộ nhớ dữ liệu duy nhất, nhưng cùng một lúc có thể tồn tại nhiều đoạn dữ liệu (logic) trên đó. Mỗi đoạn dữ liệu ở trên ngăn xếp (stack) tương ứng với một thủ tục được gọi đến trong chương trình nguồn. Đoạn dữ liệu dùng làm chỗ nhớ cho các biến cục bộ của thủ tục đó (mỗi biến chiếm một phần tử). Chú ý rằng cứ mỗi lời gọi thủ tục thì lại cấp phát một đoạn dữ liệu mới. Cho nên một thủ tục chưa kết thúc mà lại gọi một thủ tục khác thì đoạn dữ liệu cho thủ tục đó vẫn tiếp tục tồn tại trên ngăn xếp, song phải đổi đoạn dữ liệu của thủ tục mới vừa được cấp phát trên đỉnh ngăn xếp. Cứ như vậy trên ngăn xếp tồn tại đồng thời nhiều đoạn dữ liệu tương ứng với các thủ tục đang chờ, trừ đoạn dữ liệu ở đỉnh ngăn xếp là tương ứng với thủ tục hiện hành. Nếu một thủ tục được gọi đệ qui, nghĩa là trước đó thủ tục đã được gọi và đang chờ, và nay lại được gọi lại, thì một đoạn dữ liệu mới (có cấu trúc giống cũ) vẫn phải được cấp phát. Điều đó có nghĩa là lần gọi sau của một thủ tục sẽ làm việc trên các biến cục bộ của nó tại những chỗ nhớ khác với lần gọi trước.

Các trình biên dịch ngôn ngữ cấp cao cho các máy x86 cũng thường sinh mã như sau cho chương trình con:

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, Locals
...
MOV     ESP, EBP
POP     EBP
RET     Params
```

Ta có thể thấy sự tương tự trong cách sử dụng ngăn xếp ở đây với việc tổ chức trong bộ nhớ dữ liệu của máy ảo PL/0.

Ngoài các biến cục bộ ra, thì ở đáy mỗi đoạn dữ liệu người ta dành ra 3 phần tử để lưu 3 thông tin cần thiết cho việc quản lý các block, đó là

- SL (static link): mối nối tĩnh
  - DL (dynamic link): mối nối động
  - RA (return address): địa chỉ trở về
- Mối nối tĩnh SL của một đoạn dữ liệu tương ứng với một thủ tục, được dùng để chứa địa chỉ (hay nói cách khác là trỏ đến) đáy của đoạn dữ liệu tương ứng với thủ tục mẹ của thủ tục nói trên (tức thủ tục chứa **khai báo** của thủ tục hiện hành). Như

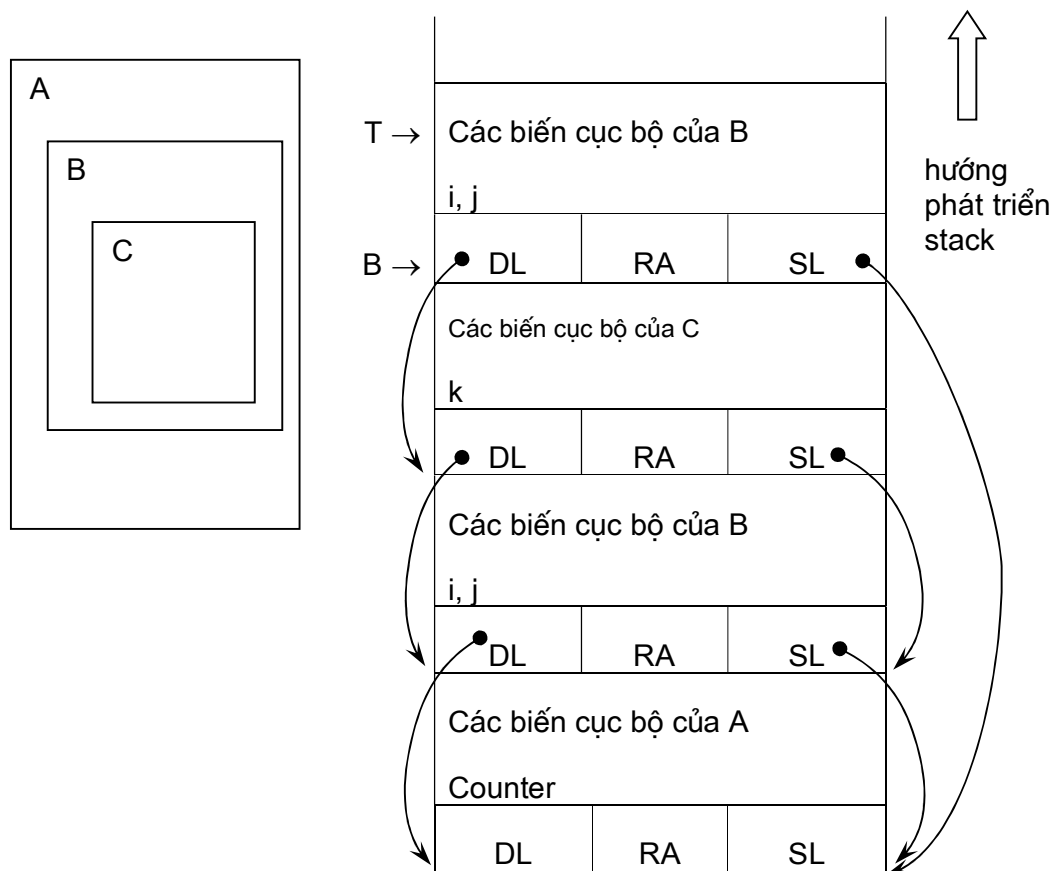
vây dây chuyền tĩnh (gồm các mối nối tĩnh liên tiếp) duy trì con *đường truy nhập* vào các đoạn dữ liệu tương ứng với các thủ tục bao nhau (lần lượt từ trong ra ngoài);

- Mối nối động DL trên một đoạn dữ liệu tương ứng với một thủ tục, được dùng để trở đến đáy của đoạn dữ liệu tương ứng với thủ tục đã **gọi** thủ tục nói trên. Đoạn dữ liệu được trở tới phải nằm ngay dưới đoạn dữ liệu của thủ tục nói trên. Như vậy dây chuyền động (gồm các mối nối động liên tiếp) duy trì *trật tự gọi* các thủ tục. Nó cho phép mỗi khi ra khỏi một thủ tục thì theo đó mà lùi về được với đoạn dữ liệu của thủ tục đã gọi thủ tục vừa hoàn thành;
- Địa chỉ trở về RA, trên một đoạn dữ liệu tương ứng với một thủ tục, được dùng để lưu *địa chỉ của lệnh kế tiếp* với lệnh gọi đã khởi động thủ tục này. Đây là nơi trong chương trình đích cần phải về để tiếp tục thực hiện chương trình khi ra khỏi thủ tục nói trên.

Để dễ hình dung hơn ta xét ví dụ:

Giả sử thủ tục A bao thủ tục B và thủ tục B bao thủ tục C. Trong quá trình chạy chương trình, giả sử A gọi B, rồi B lại gọi C, và sau đó C gọi B (một cách đệ qui). Lúc này ta có quang cảnh sau:

```
procedure A;  
  var Counter;  
  procedure B;  
    var j, i;  
    procedure C;  
      var k;  
      begin  
        if Counter < 5 then begin  
          call B ;  
          Counter := Counter + 1;  
        end  
      end;  
    begin  
      Call C;  
    end;  
  begin  
    Counter := 0;  
    call B;  
  end;
```



Như đã thấy ở trên, sự cấp phát chỗ nhớ cho các biến trong chương trình là cấp phát động, tức là sự cấp phát xảy ra trong lúc thực hiện chương trình chứ không phải trong giai đoạn dịch. Vì vậy mà trong giai đoạn dịch, trình biên dịch chưa biết trước được địa chỉ tuyệt đối (trên ngăn xếp S) của các biến. Tuy nhiên nó lại biết được vị trí tương đối của mỗi biến bên trong đoạn dữ liệu, bởi vì nó có khả năng tính được khoảng cách từ đáy đoạn dữ liệu tới mỗi biến (dựa trên các khai báo). Khoảng cách đó gọi là *độ dời* của biến. Sau này đến giai đoạn thực hiện, khi đoạn dữ liệu đã được định vị thì chính phương thức Interpret sẽ cộng địa chỉ tuyệt đối của đáy đoạn dữ liệu với độ dời của biến để xác định vị trí tuyệt đối của biến trên ngăn xếp.

Mặt khác, ta lại biết rằng các biến cục bộ của mình ngoài các biến cục bộ của mình thì một thủ tục còn có quyền truy nhập tới các biến của thủ tục bao nó nữa (thủ tục mẹ). Muốn thực hiện sự truy nhập này thì phải biết rõ đoạn dữ liệu chứa biến cần truy nhập đang nằm ở đâu trên ngăn xếp S. Điều này cũng không thể xác định được vào lúc dịch. Tuy nhiên vì một thủ tục đã khai báo biến này phải là một thủ tục bao thủ tục hiện hành, cho nên chắc chắn rằng đoạn dữ liệu tương ứng với nó phải ở một chặng nào đó trên day chuyển tính (bắt nguồn từ đoạn dữ liệu của thủ tục hiện hành). Vậy chỉ cần biết rõ mức của thủ tục chứa biến và mức của thủ tục hiện hành chênh



nhau bao nhiêu là ta có thể sử dụng dây chuyền tính để tìm đến đoạn dữ liệu tương ứng. Việc tính chênh lệch mức thì lại có thể làm được trong giai đoạn dịch, bởi vì trong giai đoạn dịch, mức của từng thủ tục đầu đã được ghi nhận vào bảng các ký hiệu Table, và có thể tìm lại ở đó để tính chênh lệch mức.

Tóm lại, trong giai đoạn dịch thì mỗi địa chỉ của biến đều được sản sinh dưới dạng một cặp: (chênh lệch mức, độ dời). Đến giai đoạn thực hiện thì interpreter sẽ truy nhập được tới biến đó ở trên ngăn xếp S bằng cách tụt theo dây chuyền tính một số nấc bằng chênh lệch mức, rồi dâng lên một đoạn bằng độ dời.

## 6.6. Tập lệnh của máy ảo PL/0

- (1) Lệnh `LIT` : đặt một số thực lên đỉnh stack
- (2) Lệnh `LOT` : đưa một biến lên đỉnh stack
- (3) Lệnh `STO` : cất một giá trị đang ở đỉnh stack vào một biến
- (4) Lệnh `CAL` : gọi một chương trình con
- (5) Lệnh `INT` : cấp phát chỗ nhớ trên đỉnh stack bằng cách tăng con trỏ T
- (6) Lệnh `JMP`: lệnh nhảy không điều kiện
- (7) Lệnh `JPC`: lệnh nhảy có điều kiện
- (8) Lệnh `OPR`: thể hiện các phép toán số học, gọi hàm dựng sẵn và quan hệ khác nhau.

Các lệnh này được đặt trong trường `OpCode` của bản ghi `TRInstruction` (xem trang 45).

- Nếu `Opcode` là `LIT` thì sử dụng trường `FloatValue` như là giá trị để nạp lên đỉnh ngăn xếp
- Nếu `Opcode` là `INT` thì sử dụng số nguyên `a` là số lượng đơn vị dữ liệu kiểu `TData` sẽ được cấp phát trên đỉnh ngăn xếp thông qua lệnh `INT`
- Nếu `Opcode` là `JMP`, `JPC` hay `CAL` thì sử dụng số nguyên `a` là địa chỉ trong chương trình (địa chỉ trong bộ nhớ chương trình)
- Nếu `Opcode` là `LOD` hay `STO` thì sử dụng số nguyên `a` là địa chỉ tương đối (độ dời) trên ngăn xếp (bộ nhớ dữ liệu)
- Nếu `Opcode` là `OPR` thì sử dụng số nguyên `a` là mã số của các phép toán, phép so sánh. `a` sẽ nhận một trong các giá trị sau

<code>oprTroVe</code>	quay về từ chương trình con. Chương trình con chấm dứt hoạt động và chương trình chính tiếp tục chạy
<code>oprDaoDau</code>	Đảo dấu phần tử trên đỉnh ngăn xếp
<code>oprCong</code> <code>oprTru</code> <code>oprNhan</code> <code>oprChia</code>	Cộng, trừ, nhân chia hai phần tử trên đỉnh ngăn xếp, loại bỏ hai phần tử này và đẩy kết quả lên đỉnh ngăn xếp.
<code>opr_eq</code> <code>opr_neq</code> <code>opr_lss</code> <code>opr_leq</code> <code>opr_geq</code>	thực hiện so sánh hai phần tử trên đỉnh ngăn xếp, loại bỏ hai phần tử này và đặt kết quả lên đỉnh ngăn xếp. Các phép so sánh: bằng nhau, khác nhau, nhỏ hơn, nhỏ hơn hoặc bằng, lớn hơn, lớn hơn hoặc bằng

opr_gtr	
oprODD	Số thực trên đỉnh ngăn xếp được làm tròn, sau đó kiểm tra xem có phải là số lẻ hay không. Phần tử trên đỉnh ngăn xếp sẽ được loại bỏ và kết quả được đầu vào đỉnh ngăn xếp.
opr_CallFunc	Thực hiện lời gọi hàm dựng sẵn. Mã hàm dựng sẵn được đặt trong trường <code>NestedLevel</code> , còn số thực trên đỉnh ngăn xếp sẽ là tham số của hàm

- còn `NestedLevel` là
  - chênh lệch mức được dùng trong các lệnh `LOD`, `STO`, `CAL` để tìm một đoạn dữ liệu trên ngăn xếp.
  - mã số hàm dựng sẵn trong trường hợp `Opcode` là `OPR` và `a` là `opr_CallFunc`

Để biết rõ nội dung từng lệnh, ta xem phương thức `Interpret` của lớp `TVirtualCPU`. Phương thức này sử dụng một hàm cục bộ `Base(1)` để tìm đáy của một đoạn dữ liệu trên ngăn xếp bằng cách tụt theo dây chuyền tính 1 mức. Phần cơ bản của thủ tục `Interpret` là một lệnh `case` cho ta cách *diễn giải* của từng lệnh tùy thuộc vào mã lệnh `Opcode`:

```
case OpCode of
  LIT: .....
  OPT: case a of .....
  .....
end;
```

## 6.7. Sản sinh lệnh

Như trên đã nói, khi sản sinh một lệnh đích, trình biên dịch phải tính được `a`, `NestedLevel`. Muốn thế nó phải tìm đến những thông tin lưu trong bảng ký hiệu `Table`.

Ta hãy xem lại cấu trúc phần tử của bảng `Table`:

```
TIdentInfo = record
  Name: alpha; { ten danh bieu }
  case kind: object_ of // kind: object_ ; { loại danh bieu }
    constant: (val: Double);
    variable, procedure_ : (level: Integer; LValue, RValue: Boolean);
                          // LValue, RValue : chỉ dùng cho biến
end;
```

Tất cả các danh biểu đều được lưu giữ tên. Ngoài ra

- Đối với hằng, giá trị số thực của nó được lưu trong `val`.
- Đối với một biến, giá trị cần lưu giữ là địa chỉ của nó bao gồm mức `level` (đây là mức của *block* đã khai báo nó), và độ dời `adr` (vị trí tương đối trên đoạn dữ liệu).

- Đối với một chương trình con giá trị cần lưu giữ là địa chỉ lối vào `adr` của nó trong chương trình đích và mức `level` của nó (ấy là mức của *block* đã định nghĩa chương trình con này).

Các thông tin trên được thu thập vào lúc chương trình dịch xử lý các phần khai báo và định nghĩa trong chương trình nguồn:

- Về giá trị một hằng (`val`) thì chính chương trình nguồn đã trực tiếp cung cấp nó trong khai báo hằng.
- Về độ dài của một biến hay địa chỉ một thủ tục (`adr`) thì trình biên dịch phải có biện pháp riêng để tính toán chúng:

Để xác định độ dài của một biến, chương trình sử dụng một biến cục bộ trong phương thức `Block` gọi là `dx` để ghi nhận số ô nhớ đã cấp phát trên đoạn dữ liệu cho tới một thời điểm nào đó. Lúc bắt đầu xử lý một chương trình con, thì `dx` được đặt giá trị 3 (điều đó có nghĩa là dành 3 ô đầu tiên trên đoạn dữ liệu lần lượt cho `SL`, `DL` và `RA`). Rồi sau đó, cứ mỗi lần một biến được khai báo, thì `dx` được tăng lên 1 (nhớ rằng trong `PL/0` thì mỗi một biến chỉ chiếm một ô trên ngăn xếp. Đây là ô nhớ logic, trong cài đặt cụ thể trong đồ án này nó là `TData`, chiếm 8 byte). Như vậy, các biến cục bộ trong một thủ tục được cấp phát chỗ nhớ lần lượt theo thứ tự khai báo của chúng trong chương trình nguồn. Và như thế, vào lúc xử lý khai báo của một biến mới thì `dx` giá trị của `dx` là bằng độ dài của biến đó trên đoạn dữ liệu. Bởi vậy, giá trị của `dx` được ghi nhận vào trường `adr` ứng với biến đó.

Cũng tương tự, để xác định địa chỉ lối vào của một thủ tục, chương trình biên dịch sử dụng một biến toàn cục gọi là `cx`. Trong quá trình sản sinh mã, `cx` luôn luôn trở tới vị trí trên mảng `FCode`, nơi sẽ ghi nhận lệnh được sản sinh tiếp theo. Chính phương thức `IGen` và `FGen` mỗi lần sản sinh một lệnh đích, lại tăng `cx` lên 1. Như vậy khi bước vào xử lý một *khối* trong chương trình nguồn, thì giá trị của `cx` vào lúc đó chính là địa chỉ lối vào của thủ tục (tương ứng với *khối* đó). Bởi vì vậy giá trị của `cx` được ghi vào trường `adr` trong bảng, tương ứng với tên thủ tục.

Về mức `Level` của tên biến hay tên thủ tục thì theo định nghĩa, đó chính là mức của *khối* đã khai báo tên đó. Để ghi nhận mức của *khối*, chương trình dịch sử dụng một tham số của phương thức `Block` gọi là `lev`. vào lúc chương trình dịch xử lý khai báo tên của một biến hay tên của thủ tục, thì tên đó được đưa vào bảng, đồng thời giá trị hiện thời của tham số `lev` được đưa vào trường `Level` tương ứng với tên đó trong bảng.

Cần chú ý rằng vừa rồi là nói về việc xử lý một khai báo biến hay khai báo thủ tục cùng với việc đưa tên biến hay thủ tục vào bảng. Còn khi xử lý các câu lệnh có dùng

đến các tên đó thì khác. Bây giờ, để sản sinh các lệnh LOD, STO hay CAL, lại cần tính *chênh lệch mức*  $L = lev - Level$  giữa mức cho bởi tham số *lev* của *khối* hiện thời (tức là chứa câu lệnh đang được xử lý) với mức cho bởi trường *level* ứng với tên biến hay tên thủ tục được đề cập đến trong lệnh.

Khi xử lý các câu lệnh *if* hay *while*, trình biên dịch gặp phải tình huống là phải sản sinh một lệnh rẽ (jump) vào lúc mà địa chỉ nhảy đến còn *chưa xác định* được. Quả vậy, lược đồ biên dịch của các câu lệnh *if* và *while* như sau:

<b>if C then S</b>	<b>while C do S</b>
Mã đích của C JPC N1 Mã đích của S N1:.....	N0 : Mã đích của C JPC N1 Mã đích của S JPC N0 N1: .....

Trong cả hai lược đồ trên thì khi sản sinh lệnh JPC N1, địa chỉ N1 chưa xác định được, do độ dài của khối các mã đích S là không biết trước được. Cách giải quyết cho tình huống này là: các địa chỉ chưa xác định được đó đành tạm thời xem là 0 trong mã được sản sinh, cho đến khi sản sinh xong khối các mã đích của S ( và lệnh JMP N0 trong *while*), các địa chỉ này đã xác định, thì mới quay lại địa chỉ trong lệnh nhảy JPC cho đúng.

Để sản sinh một lệnh trong ngôn ngữ đích với ba phần (Opcode, a, NestedLevel) trình biên dịch sử dụng hai thủ tục là IGen và FGen. IGen sản sinh các lệnh liên quan tới địa chỉ và số nguyên, FGen sinh lệnh tính toán dấu phẩy động. Các thủ tục này sau khi sinh một lệnh sẽ tăng con trỏ *cx* thêm 1. Nhớ rằng *cx* trỏ tới vị trí trên mảng FCode, nơi sẽ ghi nhận lệnh được sản sinh tiếp theo.

## 6.8. Ví dụ 1

```

procedure A;
  var x, y, Counter;

  procedure B;
    var j, i;
    procedure C;
      var k;
      begin
        if Counter <=2 then begin
          Counter := Counter + 1;
          call B ;
        end
      end;
    begin
      Call C;
    end;
  end;

```

```

    end;

begin
    Counter := 0;
    call B;
end;

begin
    call A;
end.

```

### mã sản sinh

Ma máy PL/0 sau khi biên dịch lần cuối cùng:

```

0  JMP 0 23
1  JMP 0 18
2  JMP 0 15
3  JMP 0 4
4  INT 0 4
5  LOD 2 5
6  LIT 0 2.0
7  OPR 0 13 (<=)
8  JPC 0 14
9  LOD 2 5
10 LIT 0 1.0
11 OPR 0 2 (+)
12 STO 2 5
13 CAL 2 2
14 OPR 0 0 (Tro ve)
15 INT 0 5
16 CAL 0 4
17 OPR 0 0 (Tro ve)
18 INT 0 6
19 LIT 0 0.0
20 STO 0 5
21 CAL 0 15
22 OPR 0 0 (Tro ve)
23 INT 0 3
24 CAL 0 18
25 OPR 0 0 (Tro ve)

```

**Chú ý:** sau câu lệnh cuối cùng thì thanh ghi p (program counter) của máy ảo sẽ bằng 0 và phương thức interpret ra khỏi vòng lặp repeat ... until p=0 và chấm dứt chương trình PL/0.

Khi chạy chương trình ta có kết quả sau trong tab "CRT". Cột đầu tiên là địa chỉ lệnh tương ứng với mã máy bên trên.

```

Start PL/0 interpreter
0 JMP to @23
23 INT 3 (tang con tro ngan xep)
24 Call sub @18
18 INT 6 (tang con tro ngan xep)
19 LID 0.00
20 STO @9 0.00
21 Call sub @15

```

```

15 INT 5 (tang con tro ngan xep)
16 Call sub @4
  4 INT 4 (tang con tro ngan xep)
  5 LOD @9 0.00
  6 LID 2.00
  7 <=: True (opr_leq)
  8 JPC to @14
  9 LOD @9 0.00
10 LID 1.00
11 + : 1.00=0.00 + 1.00
12 STO @9 1.00
13 Call sub @2
  2 JMP to @15
15 INT 5 (tang con tro ngan xep)
16 Call sub @4
  4 INT 4 (tang con tro ngan xep)
  5 LOD @9 1.00
  6 LID 2.00
  7 <=: True (opr_leq)
  8 JPC to @14
  9 LOD @9 1.00
10 LID 1.00
11 + : 2.00=1.00 + 1.00
12 STO @9 2.00
13 Call sub @2
  2 JMP to @15
15 INT 5 (tang con tro ngan xep)
16 Call sub @4
  4 INT 4 (tang con tro ngan xep)
  5 LOD @9 2.00
  6 LID 2.00
  7 <=: True (opr_leq)
  8 JPC to @14
  9 LOD @9 2.00
10 LID 1.00
11 + : 3.00=2.00 + 1.00
12 STO @9 3.00
13 Call sub @2
  2 JMP to @15
15 INT 5 (tang con tro ngan xep)
16 Call sub @4
  4 INT 4 (tang con tro ngan xep)
  5 LOD @9 3.00
  6 LID 2.00
  7 <=: False (opr_leq)
  8 JPC to @14
14 Return to @17
17 Return to @14
14 Return to @17
17 Return to @14
14 Return to @17
17 Return to @14
14 Return to @17
17 Return to @22
22 Return to @25
25 Return to @0
End PL/0

```

Trong đoạn cuối của chương trình ta thấy hàng loạt lệnh quay về từ các thủ tục B, C gọi đệ qui lẫn nhau.

## 6.9. Ví dụ 2

```
var x, y, sinx, cosx;  
procedure sincos;  
begin  
    sinx:= sin(x);  
    cosx:= cos(x)  
end;  
  
begin  
    x:= 1;  
    while x < 5 do begin  
        call sincos;  
        x:= x+1;  
    end;  
  
    y:= sin(x);  
end.
```

mã sản sinh

Ma máy PL/0 sau khi biên dịch lần cuối cùng:

```
0  JMP  0   10  
1  JMP  0   2  
2  INT  0   3  
3  LOD  1   3  
4  OPR  0  14  (Goi ham SIN)  
5  STO  1   5  
6  LOD  1   3  
7  OPR  1  14  (Goi ham COS)  
8  STO  1   6  
9  OPR  0   0  (Tro ve)  
10 INT  0   7  
11 LIT  0  1.0  
12 STO  0   3  
13 LOD  0   3  
14 LIT  0  5.0  
15 OPR  0  10  (<)  
16 JPC  0  23  
17 CAL  0   2  
18 LOD  0   3  
19 LIT  0  1.0  
20 OPR  0   2  (+)  
21 STO  0   3  
22 JMP  0  13  
23 LOD  0   3  
24 OPR  0  14  (Goi ham SIN)  
25 STO  0   4  
26 OPR  0   0  (Tro ve)
```

Khi cho chạy trong Tab "CRT" hiển thị các bước thực hiện chương trình như sau

```
Start PL/0 interpreter  
0  JMP to @10  
10 INT 7 (tang con tro ngan xep)  
11 LID 1.00  
12 STO @4 1.00  
13 LOD @4 1.00
```

```

14 LID 3.00
15 < : True (opr_lss)
16 JPC to @23
17 Call sub @2
  2 INT 3 (tang con tro ngan xep)
  3 LOD @4 1.00
  4 0.84=SIN(1.00)
  5 STO @6 0.84
  6 LOD @4 1.00
  7 0.54=COS(1.00)
  8 STO @7 0.54
  9 Return to @18
18 LOD @4 1.00
19 LID 1.00
20 + : 2.00=1.00 + 1.00
21 STO @4 2.00
22 JMP to @13
13 LOD @4 2.00
14 LID 3.00
15 < : True (opr_lss)
16 JPC to @23
17 Call sub @2
  2 INT 3 (tang con tro ngan xep)
  3 LOD @4 2.00
  4 0.91=SIN(2.00)
  5 STO @6 0.91
  6 LOD @4 2.00
  7 -0.42=COS(2.00)
  8 STO @7 -0.42
  9 Return to @18
18 LOD @4 2.00
19 LID 1.00
20 + : 3.00=2.00 + 1.00
21 STO @4 3.00
22 JMP to @13
13 LOD @4 3.00
14 LID 3.00
15 < : False (opr_lss)
16 JPC to @23
23 LOD @4 3.00
24 0.14=SIN(3.00)
25 STO @5 0.14
26 Return to @0
End PL/0

```

## 6.10. Một số vấn đề liên quan tới tính toán giá trị hàm

---

### 6.10.1. Khởi tạo bộ nhớ dữ liệu

Phương thức `Interpret` bây giờ không khởi tạo bộ nhớ dữ liệu. Bộ nhớ dữ liệu được khởi tạo bên ngoài phương thức này. Điều này cho phép ta viết vào bộ nhớ trước khi máy ảo chạy và đọc kết quả khi máy ảo chạy xong.

### 6.10.2. Tính chất "tĩnh" của biến tổng thể.

Trong máy ảo bộ nhớ dữ liệu được khai báo như một trường của đối tượng (chứ không phải là khai báo cục bộ như trong nguyên bản). Bộ nhớ dữ liệu được khởi tạo



bên ngoài phương thức `Interpret`. Điều này cho phép ta viết các thủ tục "can thiệp" vào vùng nhớ này, cụ thể là viết vào bộ nhớ của các biến trước khi máy ảo chạy và đọc kết quả khi máy ảo chạy xong (do thời gian tồn tại của vùng nhớ này trùng với thời gian tồn tại của đối tượng). Xét phương thức sau:

```
function TVirtualCPU.GiaTriHam(adrX, AdrY: Integer; X: TFloat): TFloat;
begin
    S[adrX+1].FValue := X;
    Interpret;
    Result:= S[adrY+1].FValue; //1: base của level 0
end;
```

Đầu tiên ta khởi tạo giá trị cho biến có địa chỉ `AdrX`, sau đó cho máy ảo chạy và lấy kết quả của biến `Y`.

Quá trình như trên chỉ thực hiện được khi `X` và `Y` được khai báo là tổng thể trong chương trình PL/0.

### 6.10.3. Mở rộng tính năng của máy ảo PL/0

Để thực hiện việc giải phương trình, vẽ đồ thị ta sẽ thêm vào một số tính năng mới cho máy ảo để thực hiện việc tính toán giá trị hàm tại một điểm.

Ta có một số giới hạn sau trong chương trình nguồn:

#### Một số giới hạn sau trong chương trình nguồn:

Để thực hiện các tính toán với các biểu thức thì chương trình phải có dạng

```
const
n=8, m=10, Coeff=200;
var x, y, z;
begin
    y:= Sin(x-0.5);
    y:= Sin(x*x-1);
    y:= Sin(x-sin(x));
end.
```

Tức là trong chương trình phải

- khai báo hai biến đầu tiên là X, Y.
- Trong các biểu thức phải có Y nằm bên phải và X nằm bên trái.
- Các phần tử nằm trong biểu thức (phía bên phải) phải là 1 trong những loại sau:
  - Số
  - Tên hàm dựng sẵn
  - Biến X
  - Các hằng khai báo trong phần khai báo hằng (**const**) ở phía trên

Trong trường hợp một biến khác (ví dụ như biến  $z$  ở trên xuất) hiện trong biểu thức phía phải thì lúc tính toán sẽ không xác định.

Tên biến không nhất thiết là  $x$ ,  $y$  nhưng thứ tự khai báo là quan trọng. Thực chất ở đây ta giả sử có phụ thuộc  $y = f(x)$

Số lượng các biểu thức không hạn chế về lý thuyết, tuy nhiên tổng số mã sinh ra trong chương trình không được vượt qua giá trị hằng `cmax` được khai báo trong đơn vị chương trình `Interpreter.pas`. Nói chung ta có thể biên dịch được hàng trăm biểu thức không phức tạp. Tất nhiên ta cũng có thể tăng `cmax` và biên dịch lại chương trình để số lượng này tăng lên.

#### *6.10.4. Chỉ thực hiện việc tính toán với các biểu thức ở mức tổng thể*

Các biểu thức trong chương trình con không được ghi nhận, chỉ có các biểu thức ở mức tổng thể được ghi nhận vào cây phân tích. Tuy nhiên trong tất cả mọi trường hợp thì chương trình PL/0 vẫn chạy bình thường vì việc ghi nhận biểu thức chỉ ảnh hưởng tới việc vẽ công thức và đồ thị.

Thực ra ta có thể quét qua hết tất cả các biểu thức trong chương trình nguồn nhưng các biểu thức mức trong nếu có truy nhập các biến cục bộ sẽ khó tính toán hơn do có địa chỉ tương đối. Do vậy việc sao chép mã thực thi sang máy ảo khác sẽ khó hơn.

#### *6.10.5. Tại sao ta phải cố định hai biến $X$ , $Y$ ?*

Sau khi biên dịch địa chỉ của các biến tổng thể là cố định. Trong trường hợp của ta hai địa chỉ này là 3 và 4. Điều này sẽ giúp cho ta có thể tính toán hàng loạt giá trị dễ dàng hơn.

#### *6.10.6. Các giá trị được tính toán thế nào.*

Trong khi biên dịch mỗi biểu thức sẽ được sinh ra thành một cây. Ngoài ra ta cũng đánh dấu luôn đoạn mã tương ứng sinh ra cho biểu thức đó. Sau này khi tính toán giá trị của biểu thức ta sẽ dùng đoạn mã trên để tính.

Quá trình thực hiện như sau:

##### Bước 1: Tạo máy ảo thực thi mã lệnh của biểu thức.

Ta sẽ sử dụng ý tưởng chính là: tạo ra 1 máy ảo mới có mã chỉ là mã tính toán biểu thức từ chương trình đã được biên dịch. Điều này tương đương với chương trình chỉ có duy nhất một dòng lệnh, ví dụ như với chương trình nêu ở trên đối với biểu thức thứ nhất mã sinh ra sẽ tương đương với

```

var x, y;
begin
  y:= Sin(x-0.5);
end.

```

1. Tạo một máy ảo mới
2. Khởi tạo 2 lệnh đầu tiên của máy ảo mới,

```

IGen(JMP, 0, 1);
IGen(INT, 0, 5);

```

Lệnh đầu tiên có trong tất cả các chương trình PL/0, lệnh thứ hai là lệnh dành ra phần bộ nhớ cần cho các biến tổng thể được khai báo trong chương trình. Trong ví dụ trên chương trình khai báo 3 biến tổng thể nên phải dành ra 6 ô nhớ: 3 ô cho SL, DL, RA (đối với mọi chương trình con và chương trình chính) , 2 ô cho các biến  $x, y$ . Biến  $z$  không được dành chỗ.

3. Chép đoạn lệnh của biểu thức vào máy mới
4. Thêm lệnh trở về bằng cách gọi

```

IGen(opr, 0, oprTroVe);

```

Các bước từ 2-4 trên được thực hiện thông qua thủ tục `PrepareExpressionCode`. Còn bước 1 chỉ đơn giản là thực hiện lệnh gọi phương thức tạo (constructor) của lớp `TVirtualCPU`.

Tóm lại hai câu lệnh sau trong chương trình sẽ khởi tạo máy ảo để tính toán biểu thức:

```

NewCPU:= TVirtualCPU.Create;
CPU.PrepareExpressionCode(FCXFrom, FCXTo, NewCPU);

```

Trong đó CPU là máy ảo do trình biên dịch sinh mã (chứa mã của tất cả các biểu thức cần tính). `CXFrom, CXTo` là địa chỉ bắt đầu và kết thúc trong đoạn mã lệnh dành cho việc tính toán giá trị biểu thức số học. Các giá trị này cũng được sinh ra trong quá trình biên dịch cùng với việc sinh ra cây văn phạm của biểu thức.

Sau khi tạo ra CPU thực thi mã của biểu thức ta có thể:

### Bước 2.1: Tính toán giá trị hàm:

Để tính toán giá trị hàm tại một điểm ta gọi phương thức

```

NewCPU.GiaTriHam(adrX, AdrY: Integer; X: TFloat);

```

- Trong đó `adrX, adrY` là địa chỉ của biến  $x, y$  do trình biên dịch sinh ra. Ta đã qui định  $x$  và  $y$  phải được khai báo đầu tiên nên giá trị của chúng là 3 và 4.

- $x$ : điểm cần tính giá trị hàm

Địa chỉ  $\text{adrX}$ ,  $\text{adrY}$  chính là địa chỉ biến trong bảng  $\text{FTable}$  được sinh ra khi biên dịch do vậy về nguyên tắc ta không cần phải qui định chặt chẽ về khai báo  $x$ ,  $y$  như trên. Tuy nhiên nếu bỏ qua điều này thì chương trình của chúng ta sẽ dài dòng thêm vì trước khi tính giá trị biểu thức nó phải tìm được địa chỉ của biến được dùng trong biểu thức.

## Bước 2.2: Giải phương trình $y = 0$ .

Ta cần nhập vào các giá trị khoảng tìm nghiệm  $[a, b]$  và sai số epsilon- nhỏ hơn nó thì giá trị hàm coi như bằng không.

Ta sử dụng phương pháp "chia đôi" ở đây. do vậy giá trị hàm tại  $a$  và  $b$  phải khác dấu, tức  $f(a) \cdot f(b)$  nhỏ hơn hay bằng 0.

Phương pháp này có ưu điểm là chắc chắn hội tụ khi có nghiệm nhưng nó cũng bỏ qua một số nghiệm trong trường hợp giá trị hàm tại  $a$ ,  $b$  cùng dấu.

Chú ý là phương pháp giải phương trình không quét hết được các nghiệm của hàm. Việc tìm tất cả các nghiệm của hàm trên một đoạn không phải là bài toán đơn giản, nhất là trong trường hợp tổng quát (tìm miền xác định của hàm số cũng tương tự).

### Bước 2.2.1 Giải phương trình bằng cách dò tìm nghiệm (phương thức Scan).

Phương pháp này chỉ đơn giản là chia đoạn cần tìm ra nhiều khoảng rồi dò nghiệm. Nói chung ta cần kết hợp *kiến thức toán học* và *đồ thị hàm số* để thực hiện việc tìm nghiệm này. Số đoạn do người dùng đưa vào. Phương thức Scan không thể tìm được tất cả các nghiệm, nó chỉ cho phép tìm ra một số nghiệm mà thôi. Tuy nhiên với các hàm kiểu đa thức hoặc các hàm có nghiệm cách xa nhau thì có hy vọng tìm ra được tất cả các nghiệm nếu chọn các giá trị tham số hợp lý. Với những hàm có vô số nghiệm trong một khoảng hữu hạn (như  $\sin(1/x)$ ) thì nó không thể tìm hết nghiệm được.

Trong quá trình tìm nghiệm cũng sử dụng phương pháp chia đôi.

## Bước 2.3 Tính toán hàng loạt giá trị $x, y$ để vẽ đồ thị

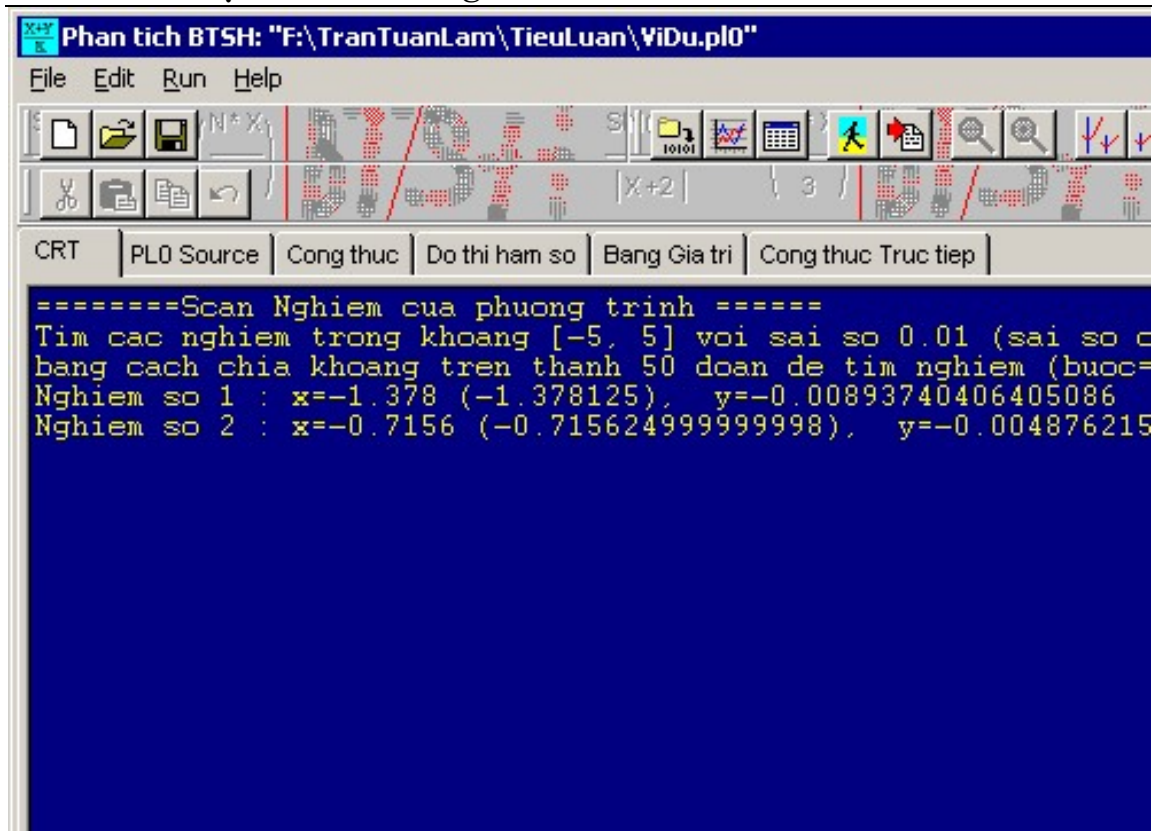
Việc vẽ đồ thị đòi hỏi một khối lượng tính toán tương đối lớn. Thay vì tính toán cho từng giá trị ta tính toán luôn cho một mảng thông qua thủ tục `GetYArray`.

Chú ý là ta không thể đơn thuần chỉ chạy chương trình PL/0 nguyên bản được vì nó không hợp với mục đích của chúng ta là tính toán các giá trị hàm. Do vậy bắt buộc phải chép mã sinh ra cho từng biểu thức sang máy ảo khác.

## Chương 7

# Các thành phần của "Phân tích BTSH" và cách sử dụng chương trình

### 7.1. Giao diện của chương trình



1. Tab **"CRT"** : cửa sổ thông báo. Khi biên dịch, khi chạy, vẽ, kết quả giải phương trình... các thông báo sẽ được đưa ra cửa sổ này.
2. Tab **"PL0 source"** : Đây là cửa sổ soạn thảo chương trình nguồn PL/0. Khi biên dịch có lỗi, con trỏ sẽ định vị tại nơi gây lỗi. Chú ý là phần văn bản phía sau từ khoá "end." trong chương trình sẽ không được biên dịch
3. Tab **"Cong thuc"**: chứa các phần giao diện để:
  - Vẽ công thức. Có menu cảm ngữ cảnh cho phép thay đổi font chữ dùng để vẽ công thức.

- Tính giá trị công thức tại một điểm
  - Giải phương trình
  - Giải phương trình bằng cách dò tìm nghiệm
4. Tab **"Do thi ham so"**: biểu diễn hàm số dưới dạng đồ thị
- Ta có thể Drag (ấn phím trái rồi dịch chuyển chuột) đồ thị
  - Có thể thay đổi font dùng để vẽ công thức (bằng menu cảm ngữ cảnh)
  - Có thể thay đổi khoảng [Xmin,Xmax] mà trong đó các giá trị biểu thức được tính toán
  - Có thể phóng to/thu nhỏ toàn đồ thị
  - Có thể phóng to/thu nhỏ trục Y
5. Tab **"Bang gia tri "**: Bảng các giá trị tính toán của từng biểu thức cho các điểm trên đồ thị.
6. Tab **"Cong thuc truc tiep"** : cho phép đánh công thức trong dòng soạn thảo và vẽ ngay công thức này dưới dạng toán học.

## 7.2. Hệ thống Menu và Thanh công cụ

---

Hệ thống menu và thanh công cụ chứa các lệnh cần thiết để mở tập tin, lưu tập tin, biên dịch chương trình...

Nhiều lệnh hay dùng có cả trên menu lẫn thanh công cụ.

Riêng phần Print chưa được xử lý hoàn chỉnh, mặc dù vẫn có thể in đồ thị ra để xem.

## 7.3. Khảo sát một công thức (hàm)

---

Giả sử ta cần khảo sát công thức

$$y = \sin(x) + \cos(2 \cdot x) + x \cdot x$$

ta thực hiện các bước sau:

1. Vào menu "File\BTSH moi (New)..." để tạo một chương trình PL/0 mới. Lúc này **"Phân tích BTSH"** sẽ tạo một chương trình mẫu trong Tab PL/0 Source.
2. Trong cửa sổ soạn thảo nhập công thức

$$y := \sin(x) + \cos(2 \cdot x) + x \cdot x;$$

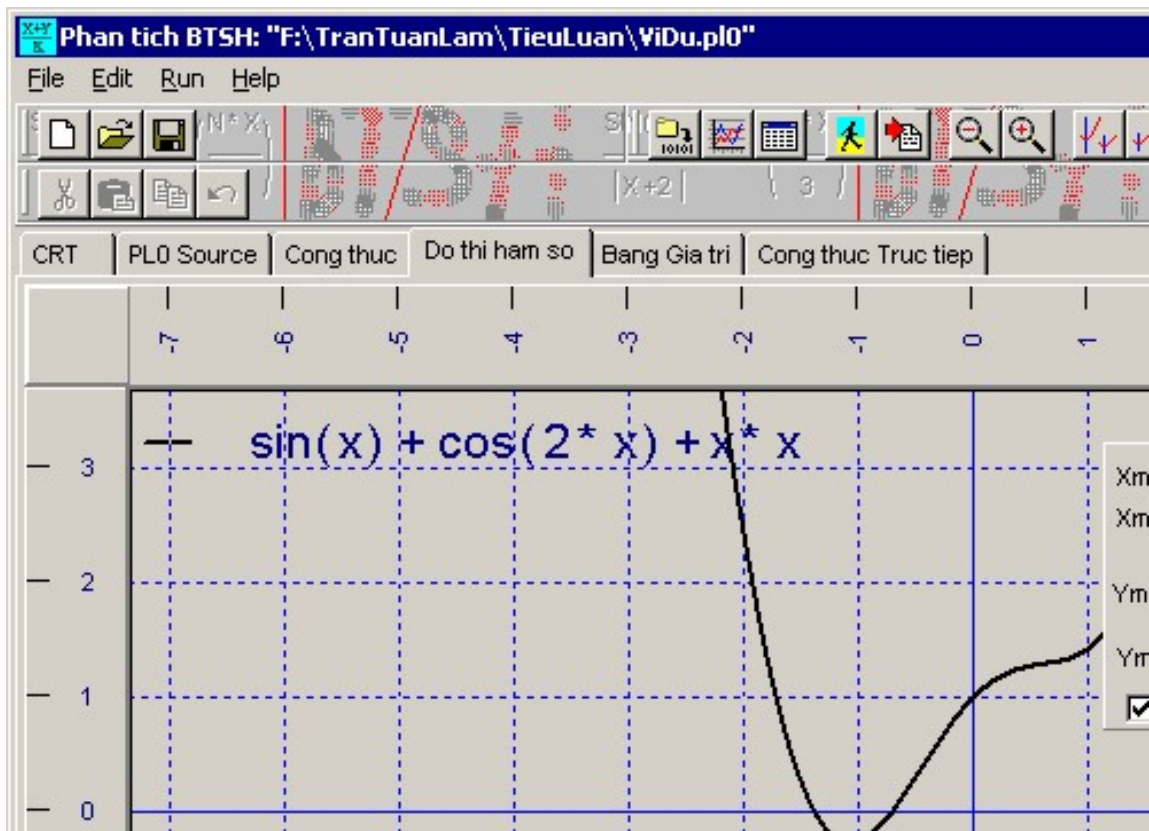
Xóa bỏ các công thức không cần thiết. Sau khi sửa trong cửa sổ soạn thảo ta có chương trình PL/0 như sau

```

const Heso1=1, Heso2=2;
var x, y;
begin
  y:=sin(x)+ cos(2*x) + x*x;
end.

```

3. Vào menu "Run\Biên dịch (Compile)" để dịch chương trình. Nếu có lỗi trong chương trình nguồn thì con trỏ trong cửa sổ soạn thảo sẽ định vị tới gần nơi gây lỗi và trên dòng Statusline ở phía dưới có thông báo lỗi. Nếu biên dịch thành công thì các lệnh có thể thực hiện tiếp theo như vẽ đồ thị, điền bảng kết quả,... được phép truy cập (Enabled). Mã chương trình PL/0 được ghi ra trong Tab **"CRT"**.
4. Vào menu "Run\Ve do thi (Graphic)" để "Phân tích BTSH" tính toán giá trị hàm số tại các điểm và vẽ đồ thị của hàm số. Lần sau chỉ cần bấm vào Tab **"Do thi ham so"** là có thể xem đồ thị, không cần thông qua menu để tính toán lại.



5. Vào menu "Run\Diện bảng (Fill Table)" để điền kết quả tính toán các điểm vào bảng. Cũng tương tự như trên lần sau xem không cần thông qua menu mà chỉ cần bấm vào tab "Bang Gia tri".

**Chú ý:** Muốn vẽ đồ thị nhiều hàm thì ta nhập nhiều công thức vào chương trình nguồn PL/0. Ví dụ sau sẽ đưa thêm hàm  $\sin(x)$  vào đồ thị



```

const Heso1=1, Heso2=2;
var x, y;
begin
  y:=sin(x)+ cos(2*x) + x*x;
  y:=sin(x);
end.

```

Phân tích BTSH: "F:\TranTuanLam\TieuLuan\ViDu.pl0"

File Edit Run Help

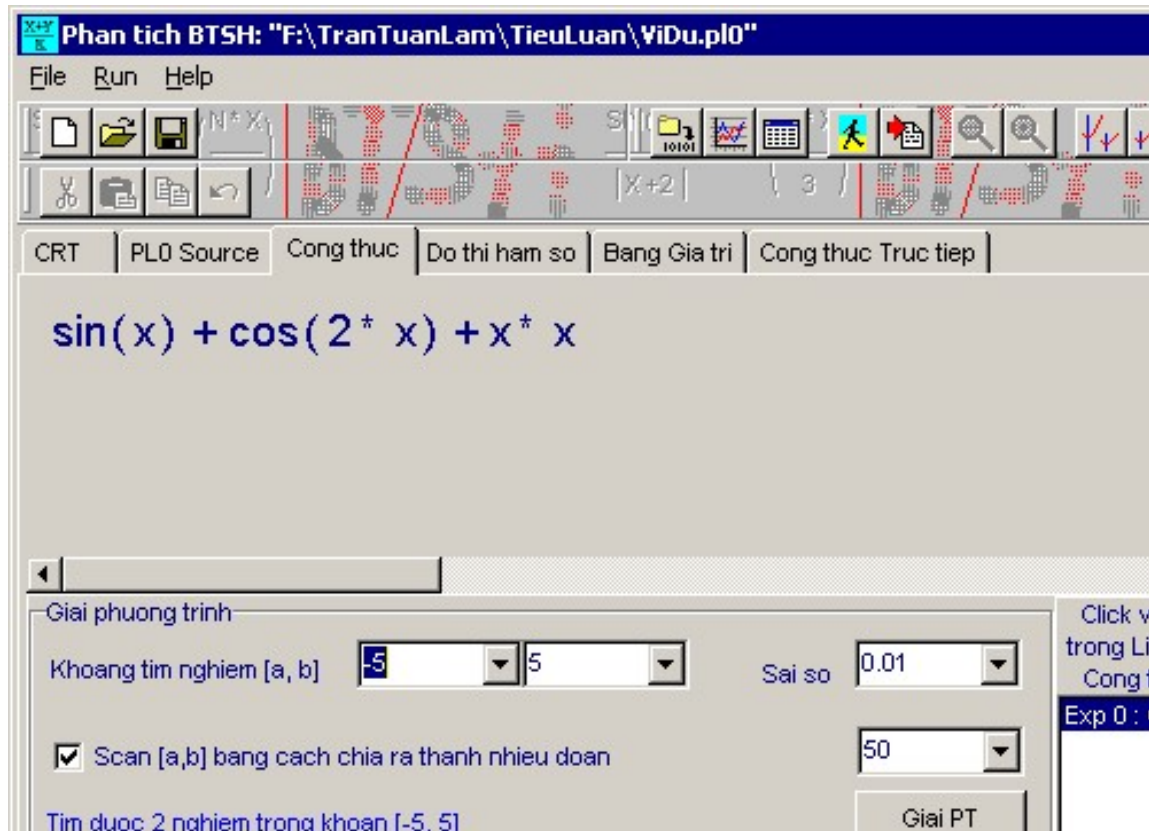
CRT | **PL0 Source** | Công thức | Đồ thị hàm số | Bảng Giá trị | Công thức Trục tiếp

NN	X	Exp 0
		$\sin(x) + \cos(2 * x) + x * x$
0	-10.000	100.952
1	-9.960	100.191
2	-9.920	99.430
3	-9.880	98.667
4	-9.840	97.904
5	-9.800	97.138
6	-9.760	96.370

## 7.4. Giải phương trình

- Thực hiện các bước 1-3 ở bên trên rồi bấm vào tab **"Công thức"**.
- Trong ListBox ở góc dưới bên trái bấm vào công thức cần khảo sát. Phía bên trên sẽ có hình vẽ biểu diễn toán học của công thức.
- Thực hiện giải phương trình:
  - Khi checkbox "Scan [a,b] bằng cách chia ra thành nhiều đoạn" **không được đánh dấu** thì phương trình sẽ được giải bằng phương pháp chia đôi. Lúc này giá trị hàm tại hai điểm đầu và cuối của khoảng tìm nghiệm phải khác nhau, nếu không "Phân tích BTSH" coi như phương trình không có nghiệm (mặc dù trong thực tế phương trình vẫn có nghiệm). Để chọn hai điểm đầu, cuối này ta nên xem trước đồ thị của hàm tương ứng.

Trong ví dụ trên nếu ta chọn khoảng tìm nghiệm là  $[-2, 0]$  rồi bấm nút Giải PT thì "Phân tích BTSH" sẽ thông báo là "Vo nghiệm", mặc dù phương trình có hai nghiệm trên đoạn này. Nếu ta chọn khoảng tìm nghiệm là  $[-1, 0]$  thì "Phân tích BTSH" sẽ báo là tìm được 1 nghiệm.



- b) Khi checkbox "Scan [a,b] bang cach chia ra thanh nhieu doan" được đánh dấu thì phương trình sẽ được giải bằng phương pháp dò tìm nghiệm. Khoảng tìm nghiệm sẽ được chia ra thành nhiều đoạn (ta có thể nhập vào số đoạn này trong ô bên cạnh). Trên mỗi đoạn cũng sẽ dùng phương pháp chia đôi để giải phương trình. Các nghiệm được ghi ra trong tab **"CRT"**.

Ta có thể nhập vào giá trị sai số trong ô "Sai so" - Khi giải phương trình, giá trị hàm nhỏ hơn giá trị này sẽ được coi là bằng không (zero).

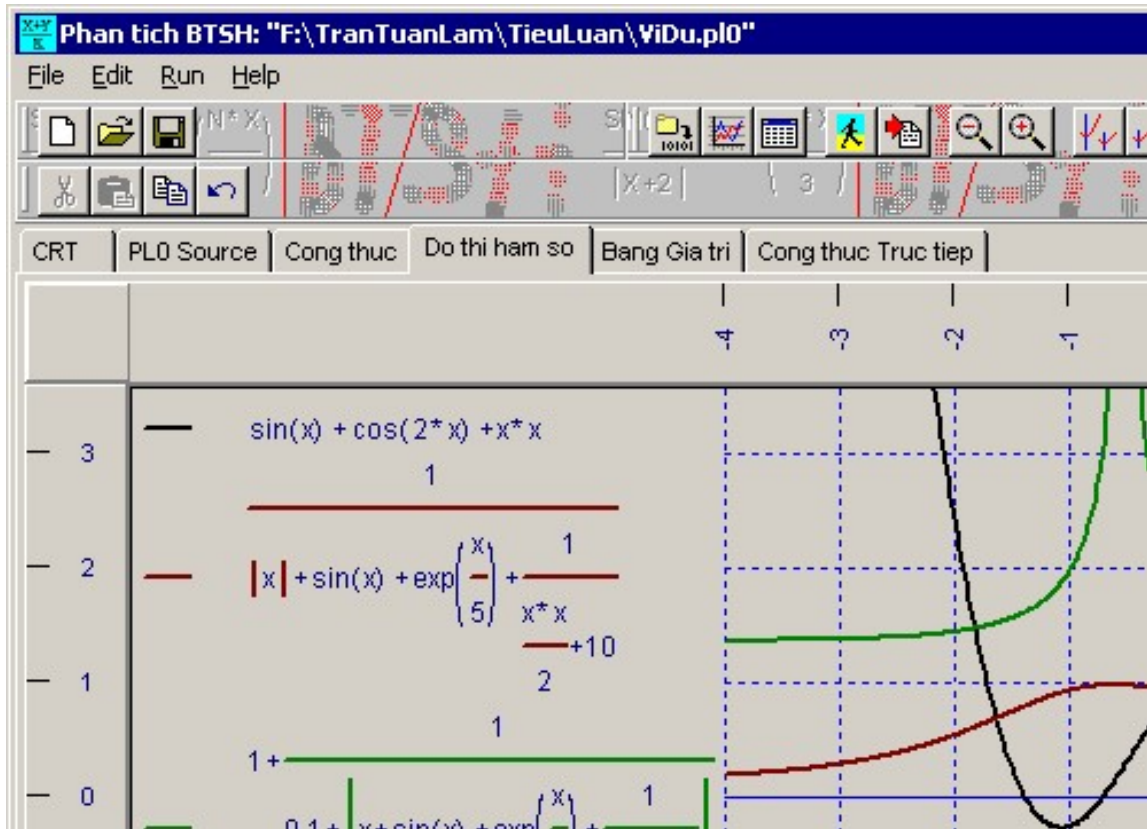
3. Ta cũng có thể tính giá trị hàm tại điểm quan tâm: nhập vào ô "Gia tri ham tai diem" giá trị x rồi bấm nút "Gia tri ham", kết quả tính toán sẽ hiện ra.

## 7.5. Vẽ một biểu thức số học

Trong phần vẽ đồ thị hàm số, trong phần tính toán giá trị hàm và giải phương trình, trong bảng giá trị hàm các công thức đều được thể hiện hiện dưới dạng toán học.

Sau đây là một số ví dụ khác

```
var x, y ;
begin
  y :=sin(x)+ cos(2*x) + x*x;
  y:= 1/(abs(x)+sin(x)+exp(x/5)+ 1/(x*x/2+10));
  y:= 1+1/(0.1+abs(x+sin(x)+exp(x/5)+ 1/(x*x/2+10)));
end.
```



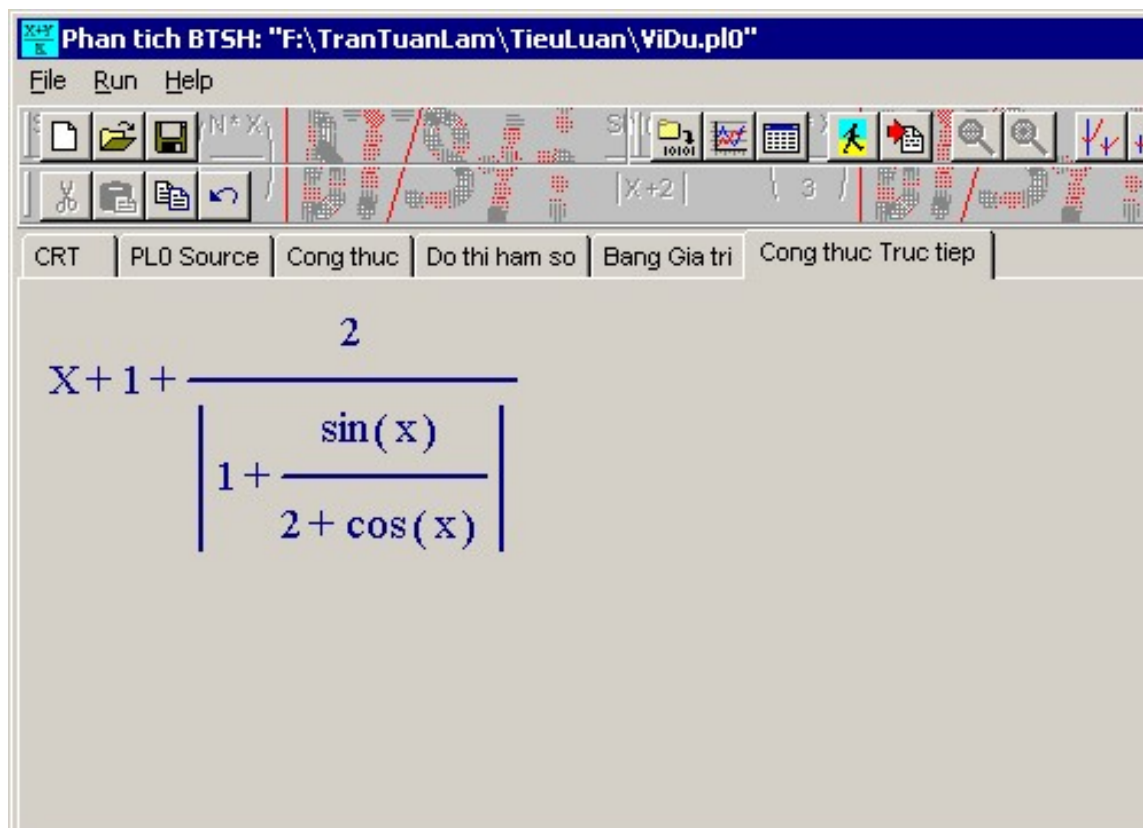
## 7.6. Vẽ một trực tiếp biểu thức số học

Bật tab "**Cong thucTruc tiep**".

Trên dòng soạn thảo "Cong thuc" đánh công thức cần nhập.

Trong khi nhập công thức nếu công thức đúng thì sẽ được vẽ ra ngay trên màn hình. Nếu có sai sót thì sẽ báo lỗi "**Error : Cong thuc chua dung**" màu đỏ ngay ở phía dưới. Trên dòng status line có thông báo lỗi của trình biên dịch.

Ta có thể chuyển công thức này cùng chương trình nguồn PL/0 được tự động sinh ra sang của sổ soạn thảo bằng cách bấm vào nút "To Source". Khi chuyển vào của sổ soạn thảo ta có thể thực hiện tất cả những tính năng đã mô tả bên trên.



## Thay lời kết

*Trong những năm tháng tự học lập trình, một câu hỏi luôn nảy sinh trong tôi là làm thế nào mà các trình biên dịch lại hiểu được chương trình nguồn mà ta viết. Tôi cũng có để ý tìm hiểu và suy nghĩ nhưng những năm trước đây tài liệu còn khan hiếm, thỉnh thoảng đọc được vài ba thông tin rời rạc nên câu hỏi vẫn không có câu trả lời. Và ngay khi có tài liệu cũng rất khó hiểu được lĩnh vực này, thời gian đầu tư để đọc và hiểu được những cuốn sách như "Compilers, principles, techniques and tools - Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman." là rất lớn, chưa nói tới việc cài đặt một trình biên dịch.*

*Rất may là tại khóa đào tạo kỹ sư II tin học, trong một thời gian giảng bài trên lớp tương đối ngắn, các thầy Nguyễn Văn Ba và Phạm Đăng Hải đã truyền đạt được nền tảng lý thuyết, những ý tưởng và phương pháp cơ bản trong việc cài đặt một trình biên dịch. Khi đọc cuốn "Thực hành kỹ thuật biên dịch - Nguyễn Văn Ba" tôi thật sự ngạc nhiên vì không ngờ một trình biên dịch lại có thể được cài đặt trong một số ít dòng lệnh như vậy. Và chương trình được viết rất hay.*

*Cái hay đó đã thúc đẩy tôi nhận đề tài tốt nghiệp này. Một đề tài không "truyền thống". Khi nhận đề tài tôi vẫn chưa hiểu mình có thực hiện được hay không. Tuy nhiên khi suy nghĩ kỹ về cây phân tích (đối tượng TNode) thì tôi tin rằng có thể giải được bài toán này, vì lý thuyết đã rõ ràng, tất cả chỉ còn chờ kỹ thuật cài đặt.*

*Khi sử dụng chương trình BTSH có thể có người sẽ đánh giá cao phần vẽ đồ thị các hàm số hơn là vẽ công thức. Tuy nhiên trong cài đặt thì việc vẽ công thức đòi hỏi lý thuyết, kỹ thuật, "chất xám", sáng kiến lớn hơn rất nhiều.*

*Một lần nữa xin cảm ơn các thầy cô và kiến thức mà các thầy cô đã truyền đạt!*

Thực hiện tại Vũng tàu 12/2001

Trần Tuấn Lâm,

Tel : 064-837087

EMail: trantuanlam@hotmail.com

# Tài liệu tham khảo

1. Thực hành kỹ thuật biên dịch - Nguyễn Văn Ba.
2. Ngôn ngữ hình thức - Nguyễn Văn Ba.
3. Compilers, principles, techniques and tools - Alfred V.Aho, Ravi Sethi, Jeffrey D. Ullman.

# Phụ lục

## Về chương trình nguồn (Source)

Chương trình viết trên Delphi 5 và bao gồm trên 4400 dòng lệnh Pascal.

Chương trình gồm các tập tin nguồn sau:

STT	Tập tin	Tóm tắt
1	BTSH.dpr	Chương trình chính
2	BTSH.res	Tài nguyên của chương trình
3	GlobalTypes.pas	Một số kiểu tổng thể
4	TuVung.pas	Phân tích từ vựng
5	HamDungSan.pas	Các thủ tục với Hàm dựng sẵn
6	Interpreter.pas	Máy ảo PL/0
7	ParseTree.pas	Cây phân tích biểu thức số học
8	PL0Paser.pas	Phần lõi của chương trình dịch
9	DrawFunc.pas	Một số thành phần dùng để vẽ đồ thị
10	FormPL0Crt.pas	Chương trình nguồn của Form chính
11	FormPL0Crt.dfm	Form chính
12	BTSHAbout.pas	Chương trình nguồn của Form About
13	BTSHAbout.dfm	Form About
14	MemoProcs.pas	Một số thủ tục làm việc với Editor

Ngoài ra Delphi có sinh ra một số tập tin phụ trợ khác. Tuy nhiên thiếu các tập tin này chương trình vẫn biên dịch được.

# Mục lục

## PHÂN TÍCH BIỂU THỨC SỐ HỌC

LỜI NÓI ĐẦU .....	2
Đề tài .....	3
Chương 1 .....	4
Mở đầu .....	4
1.1. Biên dịch biểu thức số học để làm gì? .....	4
1.2. Nội dung của luận văn.....	5
1.2.1. Phần lý thuyết:.....	5
1.2.2 Phần cài đặt:.....	5
1.3. Các điểm chính về cài đặt chương trình "Phân tích BTSH".....	5
1.4. Tại sao ta lại không làm việc biên dịch biểu thức riêng mà lại lồng vào trong trình biên dịch PL/0?.....	6
1.5. Làm thế nào mở rộng "Phân tích BTSH" để tránh việc phải viết biểu thức trong ngôn ngữ PL/0. ....	6
Chương 2 .....	7
Phân tích biểu thức số học.....	7
2.1. Sơ đồ tổng quát của chương trình.....	7
2.2. Văn phạm của biểu thức số học .....	9
2.3. Phân tích biểu thức số học trong quá trình biên dịch .....	9
2.4. Về sinh mã tính toán cho các biểu thức số học. ....	9
2.5. Xây dựng cây văn phạm của biểu thức số học. ....	11
2.6. Làm thế nào để vẽ được biểu thức toán học.....	12
2.6.1. Cấu trúc dữ liệu thể hiện nút .....	14
Các trường của nút (TNode):.....	14
2.6.2. Cấu trúc ngăn xếp nút (TNodeStack).....	16
2.7. Phân tích trực tiếp biểu thức không nằm trong chương trình PL/0 .....	18
2.7.1. Đặt vấn đề .....	18
2.7.2. Giải quyết vấn đề khai báo các định danh tự động .....	19
2.7.3. Cài đặt .....	20
Chương 3 .....	22
Ngôn ngữ PL/0 .....	22
3.1. Giới thiệu sơ lược về PL/0.....	22
3.2. Biểu đồ cú pháp của PL/0 .....	22
Các thành phần của trình biên dịch.....	27
Chương 4 .....	27
Phân tích từ vựng .....	27
4.1. Sơ đồ ô-tô-mat bộ phân tích từ vựng .....	27
4.2. Lớp phân tích từ vựng (TPhanTichTuVung) .....	30
Chương 5 .....	34
Phân biên dịch (Phân tích cú pháp, ngữ nghĩa).....	34
5.1. PL/0 là ngôn ngữ LL(1).....	34
5.2. Bộ phân tích cú pháp.....	35
5.3. Cài đặt trong lớp TPhanTichTuVung .....	38
5.4. Các hàm dựng sẵn (đơn vị chương trình HamDungSan.pas) .....	40
5.5. Ghi nhận các biểu thức.....	41
	72



5.6. Lớp TCompiler.....	42
Chương 6 .....	43
Máy ảo PL0-VM và sinh mã.....	43
6.1. Các thanh ghi của máy ảo PL0 .....	43
6.2. Bộ nhớ chương trình .....	44
6.3. Dạng thức lệnh .....	45
6.4. Bộ nhớ dữ liệu .....	45
6.5. Tổ chức trong bộ nhớ dữ liệu .....	46
6.6. Tập lệnh của máy ảo PL/0.....	49
6.7. Sản sinh lệnh.....	50
6.8. Ví dụ 1 .....	52
6.9. Ví dụ 2 .....	55
6.10. Một số vấn đề liên quan tới tính toán giá trị hàm .....	56
6.10.1. Khởi tạo bộ nhớ dữ liệu .....	56
6.10.2. Tính chất "tĩnh" của biến tổng thể. ....	56
6.10.3. Mở rộng tính năng của máy ảo PL/0 .....	57
Một số giới hạn sau trong chương trình nguồn: .....	57
6.10.4. Chỉ thực hiện việc tính toán với các biểu thức ở mức tổng thể.....	58
6.10.5. Tại sao ta phải cố định hai biến X, Y?.....	58
6.10.6. Các giá trị được tính toán thế nào. ....	58
Bước 1: Tạo máy ảo thực thi mã lệnh của biểu thức.....	58
Bước 2.1: Tính toán giá trị hàm: .....	59
Bước 2.2: Giải phương trình $y = 0$ . ....	60
Bước 2.2.1 Giải phương trình bằng cách dò tìm nghiệm (phương thức Scan ).	60
Bước 2.3 Tính toán hàng loạt giá trị x,y để vẽ đồ thị.....	60
Chương 7 .....	62
Các thành phần của "Phân tích BTSH" và cách sử dụng chương trình .....	62
7.1. Giao diện của chương trình.....	62
7.2. Hệ thống Menu và Thanh công cụ .....	63
7.3. Khảo sát một công thức (hàm) .....	63
7.4. Giải phương trình .....	65
7.5. Vẽ một biểu thức số học.....	66
7.6. Vẽ một trực tiếp biểu thức số học .....	67
Thay lời kết.....	69
Tài liệu tham khảo.....	70
Phụ lục.....	71
Về Chương trình nguồn .....	71
Mục lục .....	72

"Phân tích BTSH"