



TRƯỜNG ĐẠI HỌC CÔNG NGHỆ - ĐHQGHN

PHẠM HỒNG NGUYÊN

GIÁO TRÌNH

CHƯƠNG TRÌNH DỊCH

THU VIEN DH NHA TRANG

* 3 0 0 0 0 1 8 1 4 9 *

3000018149

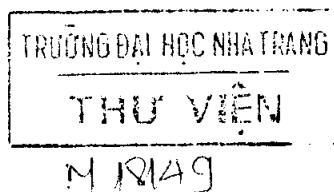


NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

PHẠM HỒNG NGUYÊN

GIÁO TRÌNH
CHƯƠNG TRÌNH DỊCH

(In lần thứ hai)



NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

MỤC LỤC

	Trang
Lời nói đầu.....	vii
Phần I. LÝ THUYẾT	1
Chương 1. MỞ ĐẦU.....	3
I. Giới thiệu môn học chương trình dịch.....	3
II. Chương trình dịch (compiler)	5
1. Định nghĩa chương trình dịch	5
2. Phân loại.....	6
3. Cấu trúc của chương trình dịch.....	7
4. Vị trí của chương trình dịch trong một hệ thống dịch thực sự	12
III. Phát triển dự án chương trình dịch.....	12
1. Mục đích	12
2. Các bước tiến hành	12
IV. Vì sao chúng ta cần học môn chương trình dịch	17
Chương 2. SƠ LUỢC VỀ NGÔN NGỮ HÌNH THÚC	20
A. Khái niệm chung về ngôn ngữ	20
1. Ký hiệu, bộ chữ cái, xâu, ngôn ngữ	21
2. Biểu diễn ngôn ngữ	21
3. Văn phạm	22
4. Phân loại Chomsky	26
B. Văn phạm chính quy và ôtômát hữu hạn.....	27
I. Văn phạm chính quy	27
II. Ôtômát hữu hạn	28
C. Văn phạm phi ngữ cảnh và ôtômát đẩy xuống.....	30
I. Văn phạm phi ngữ cảnh	30

<i>II. Bài toán phân tích ngôn ngữ đối với lớp vppnc</i>	34
<i>III. Ôtômát đẩy xuống</i>	35
Chương 3. PHÂN TÍCH TỪ VỰNG	41
<i>I. Mục đích và nhiệm vụ</i>	41
<i>II. Xác định từ tố</i>	45
1. Biểu diễn từ tố	45
2. Viết chương trình cho đồ thị chuyển đều	47
<i>III. Xác định lỗi trong phần phân tích từ vựng</i>	53
<i>IV. Các bước để xây dựng một bộ phân tích từ vựng</i>	54
Chương 4. PHÂN TÍCH CÚ PHÁP VÀ CÁC PHƯƠNG PHÁP PHÂN TÍCH CƠ BẢN	57
<i>I. Mục đích</i>	57
<i>II. Hoạt động của bộ phân tích</i>	57
1. Văn phạm	57
2. Mô tả văn phạm	58
3. Các phương pháp phân tích	58
4. Phát hiện lỗi	64
Chương 5. CÁC PHƯƠNG PHÁP PHÂN TÍCH HIỆU QUẢ	71
<i>I. Phân tích LL</i>	72
1. Mô tả	72
2. FIRST và FOLLOW	76
3. Lập bảng phân tích	79
4. Văn phạm LL (k) và LL (1)	80
5. Khôi phục lỗi trong phân tích tất định	80
<i>II. Phân tích LR</i>	83
1. Giới thiệu	83
2. Thuật toán phân tích LR	84
3. Văn phạm LR	88
4. Xây dựng bảng phân tích SLR	90
5. Xây dựng bảng phân tích LR chuẩn	98
6. Xây dựng bảng phân tích LALR	103
7. Khôi phục lỗi trong phân tích LR	108

Chương 6. ĐIỀU KHIỂN DỰA CÚ PHÁP	113
<i>I. Mục đích</i>	113
<i>II. Định nghĩa điều khiển dựa cú pháp</i>	114
1. Dạng của định nghĩa điều khiển dựa cú pháp	115
2. Đồ thị phụ thuộc.....	118
3. Thứ tự đánh giá thuộc tính.....	118
3.1 Các bộ đánh giá thuộc tính dựa theo cây phân tích	119
3.2 Các bộ đánh giá thuộc tính quên lãng	120
<i>III. Lược đồ chuyển đổi</i>	122
<i>IV. Dụng cây cú pháp</i>	123
Chương 7. PHÂN TÍCH NGỮ NGHĨA	128
<i>I. Nhiệm vụ</i>	128
<i>II. Các hệ thống kiểu</i>	129
1. Các hệ thống kiểu.....	129
2. Ví dụ về một bộ kiểm tra kiểu đơn giản	133
<i>III. Một số vấn đề khác của kiểm tra kiểm</i>	136
1. Sự tương đương của kiểu biểu thức	136
2. Đổi kiểu.....	136
3. Định nghĩa chồng (overloading) của hàm và các phép toán	138
Chương 8. BẢNG KÝ HIỆU	140
<i>I. Mục đích, nhiệm vụ</i>	140
<i>II. Các yêu cầu đối với bảng ký hiệu</i>	140
<i>III. Cấu trúc dữ liệu của bảng ký hiệu</i>	143
Chương 9. SINH MÃ TRUNG GIAN	147
<i>I. Mục đích, nhiệm vụ</i>	147
<i>II. Các ngôn ngữ trung gian</i>	148
1. Ký pháp hậu tố	148
2. Đồ thị	149
3. Mã ba địa chỉ (three-address code)	149
Chương 10. SINH MÃ	155
<i>I. Mục đích, nhiệm vụ</i>	155
<i>II. Các dạng mã đối tượng (object code).....</i>	156

1. Mã máy định vị tuyệt đối	156
2. Mã đổi tượng có thể định vị lại được	156
3. Mã đổi tượng thông dịch.....	157
III. Các vấn đề thiết kế của bộ sinh mã	158
1. Đầu vào	158
2. Đầu ra	158
3. Quản lý bộ nhớ.....	159
4. Chọn chỉ thị lệnh.....	159
5. Sử dụng thanh ghi	160
6. Thứ tự làm việc	161
IV. Máy dịch	161
V. Một bộ sinh mã đơn giản	161
 Phần II. THỰC HÀNH	165
 Chương 1. ĐẶT VÂN ĐÈ	167
 Chương 2. NGÔN NGỮ NGUỒN VÀ MÁY TÍNH ẢO	171
A. Ngôn ngữ nguồn SLANG (bước 1)	171
I. Mô tả ngôn ngữ SLANG	172
II. Định nghĩa ngôn ngữ SLANG	173
B. Máy tính ảo (Virtual maniche-VIM)	181
I. Version đơn giản đầu tiên của máy tính ảo VIM.....	181
1. Cấu trúc của máy tính ảo VIM.....	182
2. Trình thông dịch cho máy tính ảo VIM	186
3. Chuyển đổi từ SLANG sang VIM	191
II. Cài tiến máy tính ảo VIM - version thực sự	195
1. Cài tiến nhằm thêm thủ tục	198
2. Cài tiến làm dễ cho việc sinh các chỉ thị nhảy và phân đoạn	201
3. Bộ thông dịch VIM - version thực sự	203
 Chương 3. PHÂN TÍCH TỪ VỰNG	211
I. Mục đích, nhiệm vụ	211
II. Đồ thị dịch chuyển	211
III. Thực hiện	212
IV. Hoàn thiện chương trình phân tích từ vựng	218
V. Kiểm tra, thử nghiệm chương trình	219

Chương 4. PHÂN TÍCH CÚ PHÁP	223
<i>I. Mục đích, nhiệm vụ</i>	223
<i>II. Phương pháp phân tích để quy đổi xuống</i>	223
<i>III. Bộ phân tích cú pháp cho SLANG</i>	230
<i>IV. Thủ nghiệm chương trình</i>	236
Chương 5. BẢNG KÝ HIỆU VÀ PHÂN TÍCH NGỮ NGHĨA.....	242
<i>I. Mục đích, nhiệm vụ</i>	242
<i>II. Thiết kế bảng ký hiệu và phân tích ngữ nghĩa</i>	244
<i>III. Hoàn thiện và thử nghiệm chương trình</i>	257
1. Hoàn thiện.....	257
2. Thủ nghiệm chương trình	261
Chương 6. SINH MÃ	264
<i>I. Mục đích, nhiệm vụ</i>	264
<i>II. Sinh mã cho VIM</i>	265
<i>III. Hoàn chỉnh và thử nghiệm</i>	275
1. Hoàn chỉnh.....	275
2. Kiểm thử chương trình.....	279
Chương 7. XỬ LÝ LỖI.....	283
<i>I. Mục đích, nhiệm vụ</i>	283
<i>II. Xử lý lỗi trong phần phân tích từ vựng</i>	284
<i>III. Xử lý lỗi trong phần phân tích cú pháp</i>	286
1. Phát hiện lỗi	286
2. Khôi phục lỗi.....	287
<i>IV. Xử lý lỗi trong phần phân tích ngữ nghĩa</i>	295
Chương 8. MỞ RỘNG NGÔN NGỮ NGUỒN VÀ MÁY TÍNH ẢO	296
<i>I. Mục đích</i>	296
<i>II. Mở rộng ngôn ngữ nguồn SLANG (bước 2)</i>	297
1. Mở rộng từ tổ	297
2. Mở rộng các luật cú pháp.....	297
<i>III. Mở rộng VIM cho mảng</i>	302
1. Kiến trúc.....	302
2. Tập chi thị	305
3. Trình thông dịch.....	308

4. Chuyển đổi sang mã VIM cho mảng	312
IV. Mở rộng VIM cho tham số	314
1. Kiến trúc.....	314
2. Tập chỉ thị	316
3. Trình thông dịch.....	318
4. Chuyển đổi sang mã VIM cho tham số.....	322
V. Hướng dẫn mở rộng chương trình dịch.....	323
1. Mở rộng bảng ký hiệu	323
2. Các thủ tục và hàm.....	325
Các bài tập tổng hợp	333
Phụ lục A. Mô tả một số ngôn ngữ lập trình	341
Phụ lục B. Định dạng chương trình nguồn trong MS Word	358
Phụ lục C. Các chương trình nguồn Pascal	363
Tài liệu tham khảo	403

Lời nói đầu

Chương trình dịch là một trong các công cụ chính của người lập trình cho máy tính. Một khi bạn đã đi vào việc lập trình thì cần tìm hiểu sâu về nguyên lý của các công cụ này. Các kiến thức về chương trình dịch không những giúp cho người lập trình sử dụng chương trình dịch tốt hơn, hiểu biết sâu hơn về các ngôn ngữ lập trình, giúp ra các quyết định lựa chọn đúng đắn, không những thế các kiến thức này còn giúp ích cho chúng ta trong việc xử lý nhiều việc khác, như xây dựng các chương trình dựa trên các thành phần của chương trình dịch, ôtômát, xử lý ngôn ngữ tự nhiên.

Để đọc được giáo trình này bạn cần một số kiến thức nhất định về lập trình.

Cuốn sách này dựa theo giáo trình chuyên ngành đã được giảng dạy tại khoa Công nghệ, trường Đại học Công nghệ (trước đây là khoa Công nghệ thông tin, Đại học Khoa học Tự nhiên), thuộc Đại học Quốc gia Hà Nội từ năm 1998 với thời lượng khoảng 60 tiết (bao gồm cả thực hành).

Chương trình dịch là một trong các môn chuyên ngành sâu và cũng không quá khó đối với những người lập trình.

Giáo trình gồm hai phần: Phần Lý thuyết có 10 chương, đề cập đến các nguyên lý của chương trình dịch. Các chương ứng với các khối xử lý chính. Phần Thực hành cũng có những chương, ứng với các chương của phần lý thuyết. Phần thực hành của giáo trình là một chương trình dịch hoàn chỉnh, có thể dịch được. Tuy nhiên để tránh sử dụng chương trình có sẵn mà không thật hiểu các lệnh lập trình, chương trình mẫu đã được "tháo rời" ra theo các chương. Bạn cần phải đọc một chút thì mới có thể lắp chúng lại thành một chương trình dịch hoàn chỉnh.

Tôi rất hoan nghênh mọi phê bình, góp ý cho sách cũng như các chương trình cờ làm cho chúng ngày càng hoàn thiện hơn (các góp ý của các bạn sẽ được tập hợp biên soạn và phổ biến đến các bạn khác). Nếu các bạn cần liên hệ, góp ý cũng như cần giúp đỡ, xin hãy viết thư cho tôi theo địa chỉ sau:

Email: phhnguyen@yahoo.com

Nhân đây, tôi xin bày tỏ lòng biết ơn sâu sắc đối với gia đình đã tạo mọi điều kiện cho tôi trong mọi việc cũng như viết cuốn sách này. Tôi cũng xin cảm ơn các đồng nghiệp trong khoa Công nghệ Thông tin đã giúp đỡ nhiều trong quá trình viết sách. Cảm ơn các em sinh viên đã đọc và giúp tôi hiệu chỉnh lại toàn bộ bản thảo và các chương trình mẫu.

TS. Phạm Hồng Nguyên

Phân I

LÝ THUYẾT

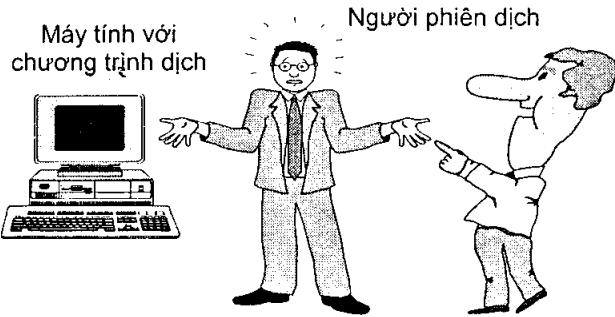
Chương 1

MỞ ĐẦU

I. GIỚI THIỆU MÔN HỌC CHƯƠNG TRÌNH DỊCH

Trước đây, con người thường ra lệnh cho các máy móc của họ bằng các nút bấm, công tắc và cần gạt. Ngày nay, việc cung cấp các dữ liệu và điều khiển máy tính đa dạng và phức tạp đến nỗi con người buộc phải trao đổi, “nói chuyện” với máy bằng một ngôn ngữ nào đấy mới đủ để ra lệnh và chuyển tải hết thông tin.

Ngôn ngữ của bản thân máy tính thường quá phức tạp, khó hiểu (chi toàn là 0,1) đối với con người. Nhưng ngôn ngữ của con người (gọi là ngôn ngữ tự nhiên) cũng dài dòng, đầy những chi tiết mập mờ, lại quá sức đối với khả năng máy tính hiện nay. Trong tương lai xa, người và máy có khả năng đối thoại với nhau bằng chính ngôn ngữ của con người. Còn hiện nay, con người phải sáng tác ra các ngôn ngữ trung gian, đơn giản và chặt chẽ, rõ ràng hơn nhiều so với ngôn ngữ tự nhiên và dùng để đối thoại với máy tính - đó là các ngôn ngữ lập trình (còn gọi là ngôn ngữ nhân tạo). Các ngôn ngữ này tương đối dễ học vì chúng phỏng theo các ngôn ngữ tự nhiên (thường phỏng theo tiếng Anh). Nhưng điều quan trọng nhất là có thể *chuyển tự động* sang ngôn ngữ máy, nhờ một “người phiên dịch”. Người phiên dịch thường là một chương trình của máy tính và được gọi là chương trình dịch (compiler).



Hình 1.1. Người phiên dịch này không “nói” được ngôn ngữ chỉ gồm 0 và 1 của máy tính. Bạn phải dùng người phiên dịch tự động là chương trình dịch, và học một ngôn ngữ (ngôn ngữ lập trình) mà nó hiểu để “nói chuyện” với máy thông qua nó.

Môn học chương trình dịch nghiên cứu hai vấn đề chính:

- Lý thuyết thiết kế các ngôn ngữ lập trình. Nói cách khác, nó nghiên cứu các cách tạo ra một ngôn ngữ nhân tạo giúp người lập trình có thể đối thoại với máy và có thể dịch tự động được.
- Cách viết chương trình chuyển đổi từ một ngôn ngữ lập trình này sang một ngôn ngữ lập trình khác.

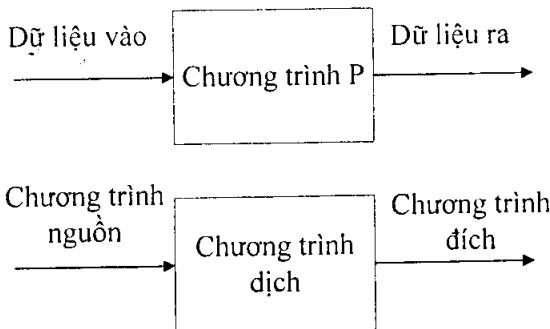
Kiến thức về chương trình dịch gắn chặt với kiến thức về ngôn ngữ và phân tích ngôn ngữ cũng như các kiến thức tổng hợp khác về máy tính.

Ngày nay các ứng dụng của chương trình dịch thật đa dạng và phong phú. Ta có thể bắt gặp chúng ở khắp các nơi, từ các chương trình dịch đơn thuần như Turbo Pascal, MS C, Visual C++, gcc, javac, Visual Basic... cho đến các bộ chuyển đổi văn bản (như bộ chuyển đổi dạng file DOC sang RTF và ngược lại của MS Word) hoặc là một thành phần quan trọng của các phần mềm hiện đại như các chương trình xử lý văn bản và duyệt WEB. Các ngôn ngữ lập trình mới, chương trình dịch và phần mềm liên quan ngày càng xuất hiện nhiều và mau, đòi hỏi chúng ta phải nghiên cứu kỹ lưỡng về lý thuyết lẫn thực hành chương trình dịch.

Hiện nay, các kiến thức về chương trình dịch đã được nghiên cứu tương đối tường tận. Để thiết kế một ngôn ngữ lập trình mới không quá phức tạp và xây dựng một chương trình dịch hoàn chỉnh cho ngôn ngữ này, một sinh viên có thể chỉ cần bỏ ra một vài tháng.

II. CHƯƠNG TRÌNH DỊCH (COMPILER)

Trừ các trường hợp đặc biệt, một chương trình có thể xem là một bản mô tả một quá trình hay một thuật toán dùng để biến đổi các dữ liệu ở đầu vào thành dữ liệu đã được xử lý ở đầu ra. Hình 1.2 cho thấy quá trình đó:



Hình 1.2. So sánh chương trình dịch và chương trình máy tính thông thường

Nếu dữ liệu vào lại ở dạng là một chương trình viết trong một ngôn ngữ nào đó và chương trình **P** sẽ chuyển đổi chúng thành một chương trình trong một ngôn ngữ khác, thì **P** được gọi là *chương trình dịch*. Dữ liệu vào lúc này gọi là *chương trình nguồn* (source program) và dữ liệu ra gọi là *chương trình đích* (target program). Chương trình nguồn thường ở dạng một ngôn ngữ bậc cao, như ngôn ngữ tựa tiếng Anh, khác xa với ngôn ngữ máy, chương trình đích lại thường ở ngôn ngữ bậc thấp, thường là ngôn ngữ máy hoặc gần với ngôn ngữ máy hơn.

Thuật ngữ *Compiler* bắt đầu xuất hiện vào những năm 1950, do Grace Murray Hopper đưa ra. Chương trình dịch thật sự và hoàn chỉnh đầu tiên là FORTRAN cũng hoàn thành vào cuối những năm 1950. Để xây dựng được chương trình này, người ta đã phải bỏ ra 18 năm nhân công.

Đối với đa số người lập trình, một chương trình dịch có thể coi là một “hộp đen” mà họ chỉ quan tâm đến đầu vào và đầu ra của nó chứ không cần quan tâm đến bên trong nó. Còn nhiệm vụ của chúng ta là đi sâu vào cấu trúc của “hộp đen” này.

1. Định nghĩa chương trình dịch

Chương trình dịch là một chương trình dùng để chuyển một chương trình từ một ngôn ngữ (gọi là ngôn ngữ nguồn) thành một chương trình tương đương trong một ngôn ngữ khác (gọi là ngôn ngữ đích).

Tương đương ở đây hiểu theo nghĩa là chương trình đích sẽ thực hiện được chính xác các công việc mà người lập trình *đã thể hiện* thông qua chương trình nguồn.

2. Phân loại

Các chương trình dịch có thể phân thành nhiều loại tùy theo các tiêu chí đưa ra để phân loại:

- Theo số lần duyệt: duyệt đơn, duyệt nhiều lần (ta sẽ đề cập thêm ở phần sau).

- Theo mục đích: tải và chạy, gỡ rối, tối ưu, chuyển đổi ngôn ngữ, chuyển đổi định dạng...

- Theo độ phức tạp của chương trình nguồn và chương trình đích:

- + Assembler (chương trình hợp dịch): dịch từ ngôn ngữ Assembly ra ngôn ngữ máy. Assembly là một ngôn ngữ cấp thấp, rất gần với ngôn ngữ máy.

- + Preprocessor (tiền xử lý): dịch từ ngôn ngữ cấp cao ra ngôn ngữ cấp cao khác. Thực chất chỉ là dịch một số cấu trúc mới sang cấu trúc cũ.

- + Compiler (biên dịch): dịch từ ngôn ngữ cấp cao sang ngôn ngữ cấp thấp.

- Theo phương pháp dịch -chạy:

- + Thông dịch, còn gọi là diễn giải (interpreter): hành động do câu lệnh của ngôn ngữ quy định được thực hiện trực tiếp. Thông thường với mỗi hành động đều có tương ứng một chương trình con để thực hiện nó.

Ví dụ: bộ lệnh của DOS, FoxPro có thể chạy theo chế độ thông dịch.

- + Biên dịch: chương trình nguồn được dịch toàn bộ thành chương trình đích rồi mới chạy.

- Theo lớp văn phạm:

- + LL(1)

- + LR(1).

Tuy có nhiều cách phân loại, các chương trình dịch là giống nhau về nguyên lý. Chúng ta có thể tạo ra nhiều loại chương trình dịch cho các ngôn ngữ nguồn khác nhau, chương trình đích chạy trên các loại máy tính khác nhau mà vẫn sử dụng cùng một kỹ thuật cơ bản.

3. Cấu trúc của chương trình dịch

a. Cấu trúc tinh hay cấu trúc logic

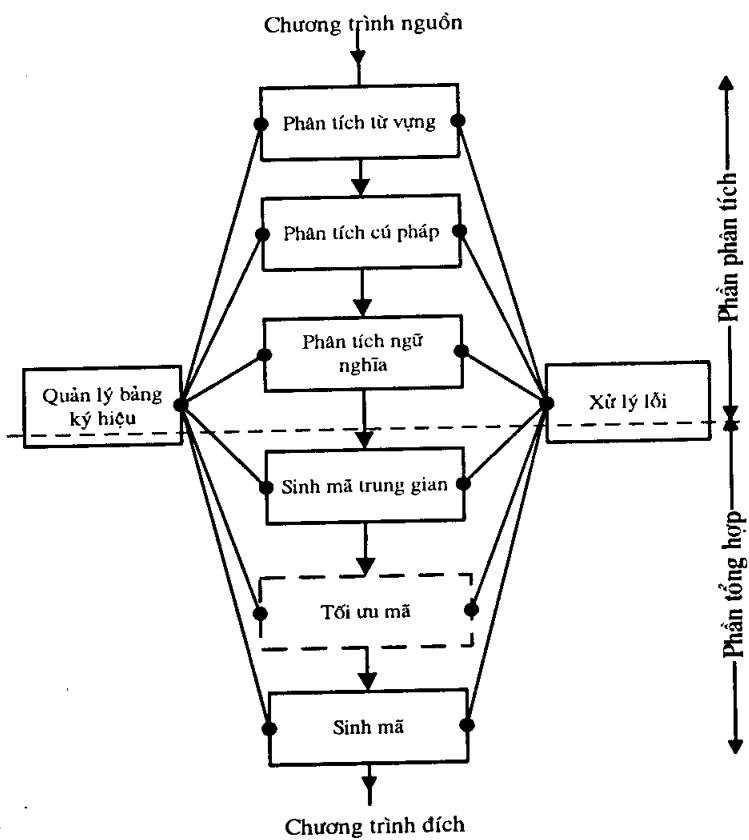
Giống như khi dịch một câu trong ngôn ngữ tự nhiên Q (ví dụ như tiếng Anh) sang một ngôn ngữ tự nhiên khác Z (như tiếng Việt), việc chuyển đổi một chương trình nguồn thành một chương trình đích có thể chia thành hai giai đoạn.

Đầu tiên câu cần dịch trong ngôn ngữ *Q* phải là "*hiểu được*". "*Hiểu được*" có nghĩa là xác định được các từ là đúng và các thành phần của câu tuân theo các luật ngữ pháp (câu đó là đúng ngữ pháp). Để "*hiểu*" ta phải có sự "*phân tích*".

Sau đó là việc *tổng hợp* các thông tin thu được thành câu mới trong ngôn ngữ Z sao cho nó có cùng ý nghĩa với câu ban đầu.

Tương tự, việc dịch cũng phải có hai giai đoạn *phân tích* và *tổng hợp*.

Giai đoạn phân tích: chương trình nguồn phải trải qua các bước sau:



Hình 1.3. Sơ đồ khái của một chương trình dịch diễn hình.

1. *Phân tích từ vựng* (lexical analyzer): đọc luồng ký tự tạo thành chương trình nguồn từ trái sang phải, nhóm thành các ký hiệu mà ta gọi là từ tố như là tên, số hay các phép toán.

2. *Phân tích cú pháp* (syntax analyzer): phân tích cấu trúc ngữ pháp của chương trình. Các từ tố sẽ được nhóm lại theo các cấu trúc phân cấp. Đôi khi ta gọi đây là phần *phân tích phân cấp*.

3. *Phân tích ngữ nghĩa* (semantic analyzer): phân tích tất cả các đặc tính khác của chương trình mà không thuộc đặc tính cú pháp. Nó kiểm tra chương trình nguồn để tìm những lỗi ngữ nghĩa và sự hợp kiều (ví dụ như không được gán bản ghi cho biến số nguyên).

Hai giai đoạn *phân tích cú pháp* và *phân tích ngữ nghĩa* có thể hoạt động như hai chức năng tách rời hoặc kết hợp làm một.

Giai đoạn tổng hợp: Chương trình đích được sinh ra từ các ngôn ngữ trung gian theo các bước sau:

1. *Sinh mã trung gian* (intermediate code generator): Sinh chương trình trong ngôn ngữ trung gian nhằm hai mục đích: dễ sinh và tối ưu hơn mã máy và dễ chuyển đổi về mã máy hơn.

2. *Tối ưu mã* (code optimizer): Sửa đổi chương trình trong ngôn ngữ trung gian nhằm cải tiến chương trình đích về hiệu năng. Do tính phức tạp quá cao nên giáo trình sẽ không đề cập đến khái niệm này.

3. *Sinh mã* (code generator): Tạo ra chương trình đích từ chương trình trong ngôn ngữ trung gian đã tối ưu.

Ngoài hai giai đoạn chính như trên, chương trình dịch còn phải thực hiện các nhiệm vụ sau:

Quản lý bảng ký hiệu (symbol-table manager): Đây là một chức năng rất cơ bản của mọi chương trình dịch, nhằm ghi lại các ký hiệu, tên... đã sử dụng trong chương trình nguồn cùng các thuộc tính kèm theo như kiều, phạm vi, giá trị... để dùng cho các bước cần đến.

Xử lý lỗi (error handler): Mọi giai đoạn trong quá trình dịch đều có thể gặp lỗi. Khi phát hiện ra lỗi, chức năng này phải ghi lại vị trí có lỗi, loại lỗi, những lỗi khác có liên quan đến lỗi này để thông báo cho người lập trình. Điều đó giúp cho chương trình dịch bỏ qua được các lỗi không quan trọng, hay những lỗi dây chuyền để có thể dịch tiếp hay buộc phải dừng lại.

Như vậy, giai đoạn phân tích có đầu vào là ngôn ngữ nguồn và đầu ra là ngôn ngữ trung gian; phần tổng hợp có đầu vào là ngôn ngữ trung gian và đầu ra là ngôn ngữ đích. Giai đoạn phân tích được coi như là *mặt trước* (front-end); giai đoạn tổng hợp được coi như là *mặt sau* (back-end) của chương trình dịch. Mặt trước độc lập với ngôn ngữ đích, mặt sau độc lập với ngôn ngữ nguồn.

b. Cấu trúc động

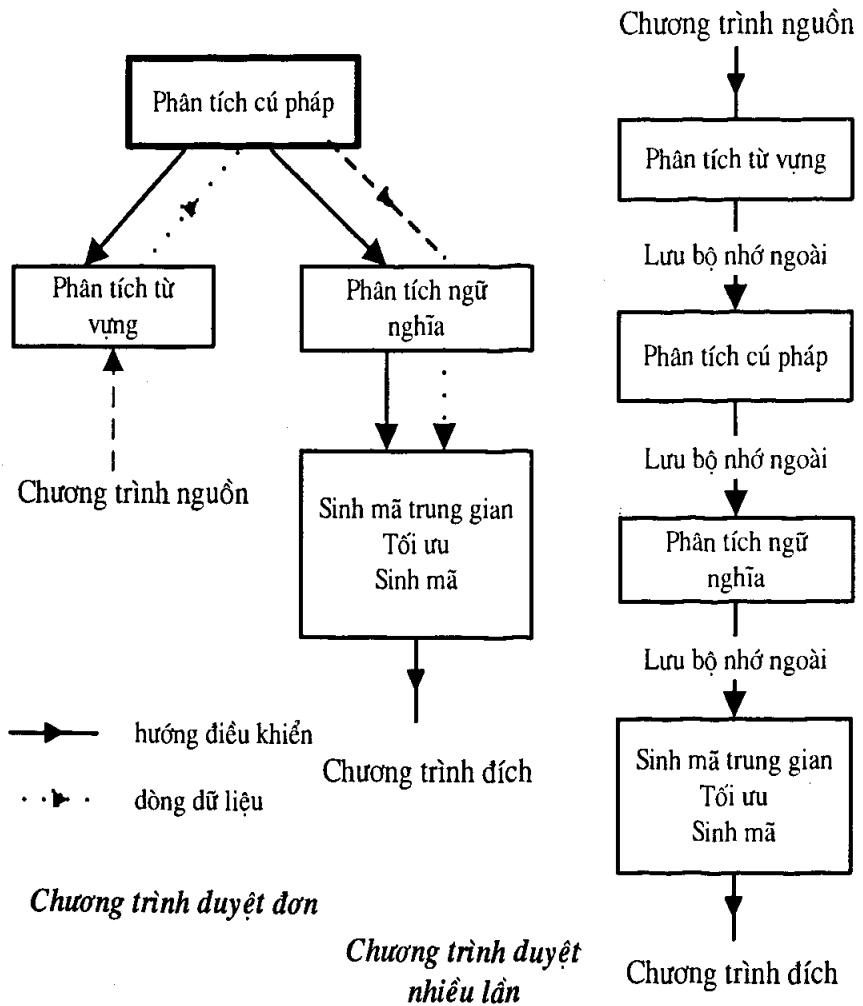
Cấu trúc động của chương trình dịch (hay cấu trúc theo thời gian) cho biết quan hệ giữa các phần của nó khi hoạt động.

Các thành phần độc lập của một chương trình dịch (phân tích từ vụng, phân tích cú pháp, phân tích ngữ nghĩa, tối ưu, sinh mã) có thể hoạt động theo hai cách: *lần lượt* hay *đồng thời*. Mỗi khi một phần nào đó của chương trình dịch đọc xong toàn bộ chương trình nguồn hoặc chương trình trung gian thì ta gọi đó là một lần *duyệt* (pass). Do vậy các chương trình dịch được chia thành *duyệt đơn* (single pass) và *duyệt nhiều lần* (multi-pass).

Đối với chương trình dịch duyệt đơn, bộ phân tích cú pháp đóng vai trò trung tâm, điều khiển cả chương trình. Nó sẽ gọi bộ phân tích từ vụng khi nó cần một từ tố tiếp theo trong chương trình nguồn, và nó gọi bộ phân tích ngữ nghĩa khi nó muốn chuyển cho một cấu trúc cú pháp đã được phân tích. Bộ phân tích ngữ nghĩa lại đưa cấu trúc này sang phần sinh mã trung gian để sinh ra các mã trong ngôn ngữ trung gian rồi đưa vào bộ tối ưu và sinh mã.

Đối với các chương trình dịch duyệt nhiều lần, mỗi một phần của nó hoạt động độc lập với nhau. Qua mỗi một phần, một chương trình trung gian sẽ được sinh ra và ghi vào các thiết bị lưu trữ ngoài để lại được đọc vào trong bước tiếp theo.

Các chương trình dịch duyệt đơn thường nhanh hơn duyệt nhiều lần vì chúng tránh được việc truy xuất các thiết bị lưu trữ ngoài để đọc các chương trình trung gian. Trái lại, các chương trình dịch duyệt nhiều lần cần ít khoảng lưu trữ hơn đối với mỗi khối. Mặt khác logic của chương trình đơn giản hơn vì các phần khác nhau rất độc lập với nhau. Một vài ngôn ngữ nguồn không cho phép dịch bằng các chương trình dịch duyệt đơn vì chúng có chứa các cấu trúc cú pháp cần đến các thông tin chỉ có được từ các phần sau. Ví dụ, các ngôn ngữ cho phép dùng biến không cần phải khai báo (như các ngôn ngữ PL /1, Algol 68, FoxPro, PHP).

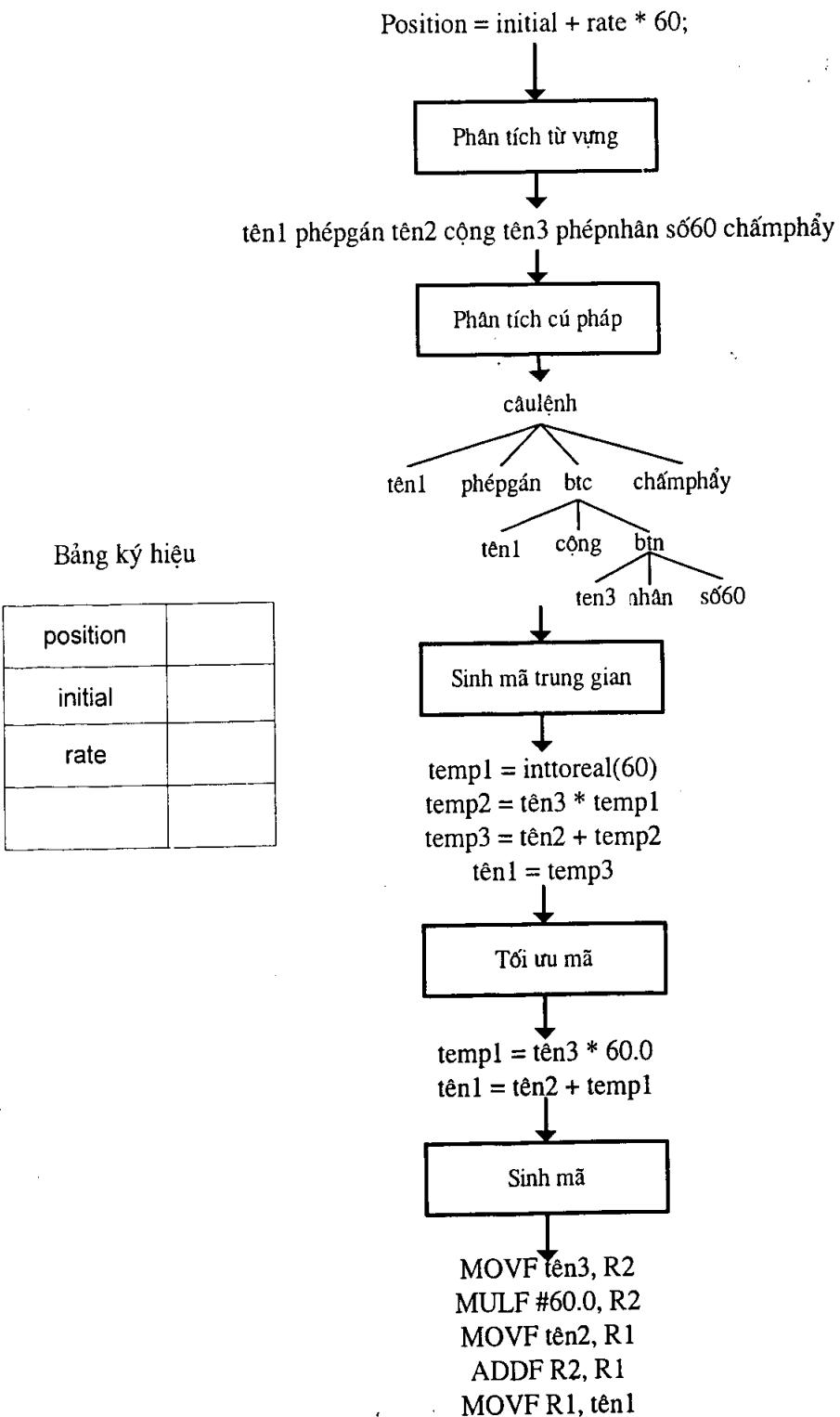


Hình 1.4. Các cấu trúc động của chương trình dịch.

Bảng dưới đây tổng kết các ưu và nhược điểm của từng phương pháp:

So sánh	Duyệt đơn	Duyệt nhiều lần
Tốc độ	Tốt	Kém
Bộ nhớ	Kém	Tốt
Độ phức tạp	Kém	Tốt
Các ứng dụng lớn	Kém	Tốt

Hình 1.5 là một ví dụ về quá trình dịch một biểu thức trong ngôn ngữ nguồn thành mã máy trong ngôn ngữ đích nào đó:



Hình 1.5. Ví dụ một quá trình dịch một biểu thức

4. Vị trí của chương trình dịch trong một hệ thống dịch thực sự

Chúng ta chỉ nghiên cứu về chương trình dịch đơn thuần. Trong thực tế, chương trình dịch thường được dùng trong một hệ thống liên hoàn nhiều chức năng, tạo ra một môi trường chương trình dịch hoàn chỉnh (Hình 1.6).

III. PHÁT TRIỂN DỰ ÁN CHƯƠNG TRÌNH DỊCH

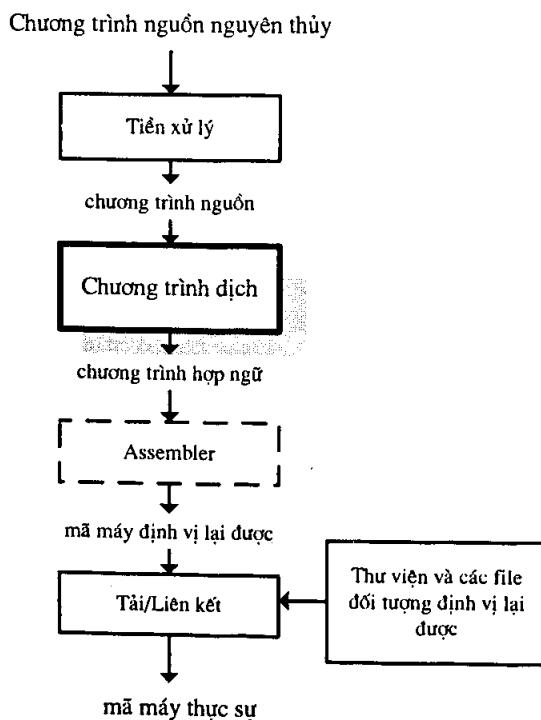
1. Mục đích

Phát triển dự án chương trình dịch nhằm tạo ra một ngôn ngữ lập trình mới (nếu cần thiết) và viết chương trình dịch hoàn chỉnh cho ngôn ngữ này.

2. Các bước tiến hành

a. Tìm hiểu hoặc thiết kế ngôn ngữ nguồn và đích

Thông thường trọng tâm tìm hiểu hoặc thiết kế chương trình dịch là cấu trúc từ vựng, ngữ pháp, ngữ nghĩa của chúng. Nếu ngôn ngữ nguồn và đích đã có sẵn thì nhiệm vụ phần này cũng đơn giản. Trong các chương sau, ta sẽ nghiên cứu từng khía cạnh khác nhau của ngôn ngữ nguồn và đích để xây dựng được các chức năng xử lý tương ứng.



Hình 1.6. Một hệ thống dịch đầy đủ

Thiết kế một ngôn ngữ lập trình mới (làm ngôn ngữ nguồn hoặc dịch) là một công việc tương đối khó khăn và ít xảy ra hơn so với việc viết các chương trình dịch. Điều này cũng giống như các ngôn ngữ của con người tương đối ít biến đổi, rất lâu mới có ngôn ngữ mới, trong khi đó, nhu cầu cần thêm người phiên dịch giữa các ngôn ngữ lại rất nhiều. Tuy nhiên, việc học thiết kế một ngôn ngữ lập trình mới sẽ giúp chúng ta nắm vững được chúng và hiểu được sâu hơn chương trình dịch sẽ hoạt động như thế nào.

Muốn thiết kế một ngôn ngữ lập trình mới, ta phải xem xét và tự trả lời những câu hỏi sau:

- *Có nhu cầu thật sự về ngôn ngữ lập trình mới hay không:* Thông thường có rất ít nhu cầu phải tạo ra một ngôn ngữ lập trình mới. Điều này thường xảy ra trong những tình huống như khi có một vi mạch được sáng chế mà vi mạch này có bộ lệnh và thanh ghi khác những cái đã có nên phải cần một ngôn ngữ lập trình riêng; bạn cần giải một lớp bài toán mới và có ý tưởng về ngôn ngữ lập trình cho riêng lớp bài toán này.
- *Các ngôn ngữ lập trình đang tồn tại không đáp ứng được nhu cầu mới này.* Trước khi quyết định viết một ngôn ngữ lập trình mới, bạn phải thận trọng tìm và xem xét lại tất cả các ngôn ngữ lập trình đang có sẵn, phân tích cái mạnh, cái dở của nó đối với nhu cầu của bạn. Trong rất nhiều trường hợp, bạn chỉ cần tìm ra hoặc cải tiến thêm một chút, hoặc đơn giản cắt xén các ngôn ngữ lập trình đã có là có thể có được thứ mình cần.

Giáo sư Niklaus Wirth ở Trường Đại học ETH Zuerich đã kể lại cẩn nguyên xây dựng một hệ thống dịch PASCAL -S. Ông và sinh viên phải thường xuyên dùng Pascal chạy trên những máy tính lớn vào những năm 1970. Giá chạy máy rất đắt, thời gian lại giới hạn nghèo, hệ thống trình dịch Pascal lớn nên nạp rất lâu lại vừa chạy chậm, vừa tốn bộ nhớ nên không cho phép chạy một số bài toán. Do đó ông đã quyết định sáng tác ra một ngôn ngữ lập trình mới, gọi là Pascal -S (Subset Pascal) thực chất là một phần của ngôn ngữ Pascal, chỉ bao gồm các lệnh hay chạy nhất. Nhờ đó, hệ chương trình dịch cho PASCAL -S của ông viết ra đã nhanh và nhỏ gọn đi rất nhiều, giải quyết được phần lớn các bài toán của ông và sinh viên, giúp tiết kiệm được nhiều tiền bạc lẫn thời gian chạy máy.

Ở Việt Nam đã từng có các thử nghiệm của một số người tìm cách sáng tạo ra những ngôn ngữ lập trình mới là bản Việt hóa các ngôn ngữ lập trình đã có như Việt-Pascal, Việt-Foxpro.

Nếu buộc phải bắt tay vào thiết kế một ngôn ngữ lập trình mới, bạn phải đảm bảo được những điều kiện sau:

- *Có thiết kế tốt, đáp ứng được nhu cầu mới:* hiển nhiên, thiết kế một ngôn ngữ lập trình mới này phải đáp ứng được các nhu cầu mới do bạn đề ra.
- *Ngôn ngữ lập trình này phải tuân theo các tiêu chuẩn chung về ngôn ngữ lập trình và về công nghệ lập trình* như tính đối xứng, tính thống nhất, tính an toàn... Đã có người thiết kế ngôn ngữ lập trình cho phép người dùng viết:

begin	hoặc	bắt đầu
end;		end;

đều được. Đây là một sự vi phạm nghiêm trọng tính nhất quán. Trong quá khứ, người ta từng cho rằng, một vụ phóng tàu vũ trụ của Mỹ bị hỏng là do lỗi lập trình FORTRAN nhầm có một dấu phẩy thay cho dấu chấm trong câu lệnh - và đã quy kết trách nhiệm cao hơn là do lỗi của người thiết kế ngôn ngữ lập trình FORTRAN đã tạo ra những kẽ hở cho phép viết những lỗi như vậy mà không bị phát hiện.

Ngoài ra, việc tuân theo các chuẩn sẽ giúp cho việc học và chuyển đổi ngôn ngữ dễ dàng hơn nhiều.

- *Viết được chương trình dịch cho ngôn ngữ này bằng những kỹ thuật dịch hiện có:* Tất nhiên bạn có thể tự mình phát triển các kỹ thuật dịch riêng đáp ứng cho thiết kế ngôn ngữ lập trình đặc biệt của bạn, nhưng điều đó sẽ làm cho thời gian phát triển hệ thống dài hơn nhiều, và bạn không đảm bảo được thành công.

Thông thường, một ngôn ngữ lập trình mới sẽ ảnh hưởng đến các khối thuộc phần phân tích của chương trình dịch: khối phân tích từ vựng, khối phân tích cú pháp và khối phân tích ngữ nghĩa (các khối thuộc phần tổng hợp phía sau sẽ không bị ảnh hưởng). Để dịch được đòi hỏi chương trình viết trong ngôn ngữ lập trình mới phải chạy trôi chảy qua các khối này. Trong các chương sau, ta sẽ đi lần lượt từng bước để xem các yêu cầu về ngôn ngữ lập trình để có thể đi qua các khối này như thế nào.

b. Thiết kế và viết chương trình dịch

Có vài phương án khác nhau để phát triển một chương trình dịch. Cách đơn giản nhất là *tìm và sửa lại* từ một hoặc nhiều chương trình dịch đã có

sẵn sao cho đáp ứng được các mục đích yêu cầu mới của chúng ta. Nếu không có sẵn các chương trình dịch thích hợp thì ta buộc phải *xây dựng lại toàn bộ*. Để xây dựng toàn bộ, lại có hai phương pháp: tự xây dựng lấy các thành phần rồi lắp ráp lại với nhau hoặc dùng các chương trình sinh chương trình dịch tạo ra một số thành phần, viết thêm các thành phần còn thiếu rồi ghép lại với nhau. Nội dung phần thực hành của giáo trình này là hướng dẫn học và viết về từng thành phần và lắp ghép chúng lại thành một chương trình dịch hoàn chỉnh.

Tìm hiểu lý thuyết và cấu trúc từng bộ phận của một chương trình dịch vừa giúp ta có khả năng tự xây dựng được một chương trình dịch hoàn chỉnh từ các thành phần của nó, đồng thời cho phép ta tìm hiểu các cách thiết kế một ngôn ngữ lập trình như thế nào để có thể xây dựng được chương trình dịch cho nó.

Ngoài ra, nhờ việc nắm được cách thiết kế một ngôn ngữ và hiểu được cẩn kẽ các thành phần của một chương trình dịch, ta có thể viết được các mô tả về một ngôn ngữ lập trình cho chương trình sinh chương trình dịch và giám sát để nó tạo cho ta các chương trình dịch thích hợp.

Các bộ phận cần tìm hiểu chính là các bộ phận trong sơ đồ cấu trúc tĩnh. Trong giáo trình này chúng ta không nghiên cứu khói chức năng *tối ưu mã* do tính phức tạp và nhiều khi không cần thiết của nó. Cụ thể ta sẽ nghiên cứu các khói sau:

Phân tích từ vựng

Đây là chức năng đầu tiên, có nhiệm vụ đọc lần lượt các ký tự từ văn bản của chương trình nguồn vào và phân tích, đưa ra các từ tố. Việc này có thể hình dung gần giống như ta tìm hiểu một câu trong tiếng Việt xem các từ có viết đúng không và đâu là danh từ, đâu là tính từ...

Kiến thức chủ yếu cần nắm vững trong phần này là cách kiểm tra và xác định từ tố. Các từ trong một ngôn ngữ thường phải tuân theo một số quy tắc nào đó, do đó phải học cách biểu diễn từ qua một ngôn ngữ riêng thích hợp: ngôn ngữ chính quy và bộ phân tích ngôn ngữ chính quy - ôtômát hữu hạn.

Phân tích cú pháp

Đầu vào của khói này là dòng các từ tố. Ta có thể hình dung giống như thông báo câu vào bao gồm danh từ, tiếp sau là một tính từ, rồi động từ...

Khối này phải phân tích xem dòng từ tố có theo đúng văn phạm (ngữ pháp) hay không và dựng cây phân tích cho biết quan hệ giữa các thành phần đó.

Phần này ta phải làm quen với cách biểu diễn một văn phạm của một ngôn ngữ nói chung (thường là văn phạm phi ngữ cảnh) và các bộ phân tích văn phạm khác nhau để có thể tự chọn một văn phạm cho ngôn ngữ lập trình và xây dựng được bộ phân tích thích hợp.

Phân tích ngữ nghĩa

Dựa vào các luật cần thiết để xây dựng câu nhận được qua bộ phân tích cú pháp, phần này sẽ tìm các luật tương ứng với các luật đó để kiểm tra tính hợp lý. Trước phần này ta cũng học cách chuyển đổi từ một cây phân tích thành một cây cú pháp và thêm các thuộc tính để tiện cho phần này và phần sau.

Sinh mã trung gian

Ở đây ta sẽ tìm hiểu về cách xây dựng một bộ mã trung gian và cách chuyển đổi từ cây phân tích hoặc cây cú pháp sang mã trung gian này.

Sinh mã

Đây là chức năng cuối của một chương trình dịch. Ở phần này ta sẽ xem cách biến đổi chương trình ở dạng mã trung gian sang mã đối tượng cho một máy tính cụ thể.

Quản lý bảng ký hiệu

Ta sẽ nghiên cứu các biện pháp cụ thể để tổ chức bảng và một chương trình quản lý hiệu quả.

Xử lý lỗi

Trong tất cả các chức năng trên, ta đều học các cách phát hiện, thông báo và xử lý nếu gặp lỗi trong chương trình nguồn.

Xây dựng môi trường phát triển của chương trình dịch

Trong thực tế, một chương trình dịch là một chương trình trong hệ thống liên hoàn giúp cho người lập trình có được một môi trường làm việc hoàn chỉnh để phát triển nhanh các ứng dụng của họ. Ví dụ như hệ thống soạn thảo chương trình với các tính năng như cho phép soạn thảo nhiều chương

trình một lúc, hiện các từ với màu khác nhau; hệ thống cho phép dùng macro; hệ thống debug để tìm lỗi hay dõi theo hoạt động của chương trình; hệ thống hướng dẫn trực quan; hệ thống quản lý các file lập trình...

Kiểm tra và bảo trì chương trình dịch

Một chương trình dịch phải tạo chương trình đích có mã đúng. Lý tưởng, chúng ta mong muốn thiết kế được chương trình đích hoạt động một cách máy móc nhưng trung thực. Các chương trình đích thường có các chức năng phức tạp, do đó cũng thường có vấn đề trong việc kiểm tra xem nó có hoạt động đúng như thiết kế hay không.

Thông thường, người ta hay kiểm tra một chương trình đích bằng phương pháp “hồi quy”. Đầu tiên người ta xây dựng một bộ chương trình trong ngôn ngữ nguồn của chương trình đích đó. Sau đó, mỗi khi có một sự thay đổi nào đó trong chương trình đích thì bộ chương trình này được đem ra dịch lại và được so sánh với mã của chính nó khi được dịch bằng bản cũ. Toàn bộ sự giống nhau và khác nhau sẽ được khảo sát kĩ lưỡng để lấy làm căn cứ sửa đổi lại chương trình đích. Cũng có thể thay cho việc so sánh với mã đích lần trước, người ta so sánh với chương trình tương đương đích bằng chương trình đích khác.

Một số phép kiểm tra khác cũng thường được thực hiện. Đó là việc so sánh kích thước chương trình đích, kích thước mã chương trình do nó tạo ra, thời gian chạy... giữa các bản khác nhau và với các chương trình khác.

Việc bảo trì chương trình đích cũng là một việc quan trọng, đòi hỏi chương trình đích phải có đủ tài liệu, thiết kế, viết phải dễ đọc, dễ sửa đổi.

IV. VÌ SAO CHÚNG TA CẦN HỌC MÔN CHƯƠNG TRÌNH DỊCH

Việc học môn chương trình đích sẽ giúp chúng ta:

a) *Nắm vững các nguyên lý ngôn ngữ lập trình và công cụ quan trọng của các nhà tin học - chương trình đích:*

+ Hiểu sâu từng ngôn ngữ lập trình. Nắm được các điểm mạnh, điểm kém của từng ngôn ngữ (nhất là về phương diện chương trình đích). Từ đó biết chọn các ngôn ngữ thích hợp cho các dự án của mình.

+ Biết lựa chọn chương trình đích thích hợp. Ví dụ: Turbo Pascal của hãng Borland dường như là hệ dịch vô địch đối với ngôn ngữ lập trình Pascal dưới DOS. Nhưng với ngôn ngữ C (cùng cho DOS) thì bạn phải cân

nhắc lựa chọn dùng Turbo C, Borland C (cũng của Borland), MSC, Visual C (của Microsoft) hay Watcom C. Các sản phẩm của hãng Borland nổi tiếng về sự tiện lợi và dễ dùng, của Microsoft thì sinh mã thường tốt hơn (gọn và nhanh) và không phải lo về vấn đề tương thích với hệ điều hành nhưng lại khó dùng hơn. Sản phẩm gcc nổi tiếng vì miễn phí và dùng được với đa số hệ điều hành phổ biến.

+ Hiểu kĩ hơn về các lựa chọn (option) trong các chương trình dịch. Các chương trình dịch hiện đại thường có rất nhiều lựa chọn. Việc hiểu cẩn kẽ bản chất các lựa chọn sẽ giúp bạn làm chủ tốt hơn các công cụ này.

+ Phân biệt rạch ròi các công việc do chương trình dịch thực hiện với công việc của các chương trình ứng dụng thực hiện. Do đó có thể tối ưu chương trình.

Ví dụ: Các lệnh sau:

```
Pascal: const SIZESTACK = 1024*9+1;  
          i := SIZESTACK;  
C:         i = sizeof(word);
```

có các phép tính **1024*9+1** hay **sizeof(word)** là do chương trình dịch thực hiện nên không ảnh hưởng đến kích thước mã hay tốc độ chương trình đích. Cách viết này cho phép chương trình nguồn dễ hiểu hơn. Cũng do hiểu hơn khả năng của chương trình đích, bạn sẽ tránh được các "ảo tưởng" về khả năng của máy tính và biết mình phải làm gì.

+ Nâng cao trình độ hiểu biết và tay nghề: Bạn sẽ nhanh chóng cải thiện hiểu biết và kỹ năng của mình do việc thiết kế và viết các chương trình dịch đòi hỏi nhiều kiến thức tổng hợp và tay nghề lập trình cao.

b) *Vận dụng:*

+ Thực hiện các dự án xây dựng chương trình dịch. Nhu cầu viết chương trình dịch thường có rất nhiều do các ngôn ngữ lập trình mới xuất hiện rất thường xuyên nhằm đáp ứng một mục đích nào đó.

+ Dùng các kiến thức của môn chương trình dịch áp dụng vào các ứng dụng khác như các bộ đọc, phân tích hay chuyển đổi các văn bản được viết dưới dạng chương trình như dạng TEX, RTF, SGML, HTML (của WWW).

+ Áp dụng các kiến thức thu được vào các ngành khác như xử lý ngôn ngữ tự nhiên.

Bài tập

1. Thế nào là một chương trình dịch?
2. Phân tích cấu trúc tĩnh và động của chương trình dịch.
3. So sánh các ưu, nhược điểm của chương trình dịch duyệt một lần và nhiều lần.
4. Bạn hãy thảo luận về các vấn đề sau:
 - + Sự giống, khác nhau giữa ngôn ngữ lập trình và ngôn ngữ con người (ngôn ngữ tự nhiên).
 - + Sự giống, khác nhau giữa một chương trình dịch và một người biên dịch.
5. Bạn hãy suy nghĩ tìm hiểu và cho biết:
 - + Bạn đã biết các ngôn ngữ lập trình nào, mức độ (hiểu biết sơ, nắm vững, thành thạo). Bạn thích nhất ngôn ngữ lập trình nào, tại sao?
 - + Bạn thường làm việc với các chương trình dịch nào (ngôn ngữ lập trình của nó, tên chương trình dịch, version, tên hãng viết chương trình đó). Bạn thích nhất chương trình dịch nào, tại sao?
 - + Trong MS Word (nếu bạn biết rõ về nó) có những chức năng nào là một phần hoặc là toàn bộ chương trình dịch hoàn chỉnh?
 - + Hãy kể tên các chương trình không phải là chương trình dịch nhưng theo bạn lại có các chức năng tựa chương trình dịch.
6. Trước khi học bạn nghĩ thế nào về môn học này:
 - + Về mức độ khó hay dễ.
 - + Có khái niệm về nguyên lý hoạt động của nó.

Bạn hãy ghi các suy nghĩ này ra giấy để tiện kiểm tra tính đúng đắn của chúng.

Bài tập thực hành

1. Tìm hiểu về cấu trúc các file văn bản có cấu trúc loại: .RTF (có thể tạo ra bằng MS Word), .HTML (có sẵn trên Internet và có thể tạo ra bằng các trình soạn thảo Web hoặc MS Word97 trở lên), .TEX (tạo bằng LaTeX, PCTeX).
2. Bạn hãy thử xóa hoặc biến đổi một vài thành phần của các file loại này và tìm hiểu xem các chương trình sử dụng chúng phản ứng như thế nào đối với các biến đổi đó.
3. Chỗ giống và khác nhau giữa các cấu trúc văn bản trên với các ngôn ngữ lập trình như Pascal, C, Java?

Chương 2¹

SƠ LƯỢC VỀ NGÔN NGỮ HÌNH THỨC

Một chương trình được nói là viết đúng khi nó được xác định là đúng về từ vựng, về ngữ pháp và ngữ nghĩa. Đồng thời lúc biết được đúng sai thì ta cũng xác định luôn được vai trò và quan hệ giữa các thành phần, từ đó có thể tiến hành chuyển đổi ngôn ngữ. Việc nghiên cứu đúng sai và quan hệ các thành phần này là một bộ phận trong một môn học: *ngôn ngữ hình thức*.

Trong chương này, chúng ta sẽ nghiên cứu sơ lược về môn học đó - môn học nghiên cứu về bản chất của ngôn ngữ nói chung mà không phải là một ngôn ngữ cụ thể nào. Nói cách khác, ngôn ngữ hình thức chỉ tập trung vào việc nghiên cứu một thứ gọi là siêu ngôn ngữ - ngôn ngữ của mọi ngôn ngữ. Nó đã lược bỏ đi các chi tiết rườm rà, hướng tới việc trừu tượng hoá, chính xác hoá các khái niệm. Trong quá trình nghiên cứu ngôn ngữ hình thức nếu ta luôn luôn liên hệ với các kiến thức về ngôn ngữ đã biết (như các ngôn ngữ Việt, Anh, C, Java, Pascal) thì việc nghiên cứu sẽ trở nên được định hướng, đơn giản và dễ dàng hơn nhiều.

A. KHÁI NIỆM CHUNG VỀ NGÔN NGỮ

Ngôn ngữ là gì? Cái gì tạo nên ngôn ngữ? Tại sao con người lại hiểu một ngôn ngữ nào đó (như hiểu tiếng Việt)? Giữa các ngôn ngữ tiếng Việt, tiếng Anh, C, Pascal có gì giống nhau, khác nhau? Phần sau đây là một số khái niệm và định nghĩa về ngôn ngữ.

¹ Nếu bạn đã học môn Ngôn ngữ hình thức thì có thể bỏ qua hoặc xem lướt phần này, chỉ cần chú ý xem các quy ước và các ký hiệu sẽ dùng trong sách.

1. Ký hiệu, bộ chữ cái, xâu, ngôn ngữ

Ký hiệu (symbol)

Ký hiệu là một khái niệm trừu tượng mà ta không định nghĩa hình thức được, giống như các khái niệm về điểm và đường thẳng trong hình học. Các chữ cái và các chữ số có thể coi là các ví dụ của ký hiệu.

Bộ chữ cái (alphabet)

Một bộ chữ cái là một tập hữu hạn các ký hiệu.

Ví dụ 2.1: Bộ chữ 26 chữ cái La Mã; bộ chữ $\{0,1\}$.

Xâu (string)

Một xâu là một dãy hữu hạn các ký hiệu xếp kề liền nhau. Chẳng hạn 010001 là một xâu trên tập ký hiệu $\{0,1\}$, còn "*tôi đi học*" là một xâu trên bộ chữ là tập hợp các chữ cái tiếng Việt.

Ngôn ngữ (language)

Ngôn ngữ là một tập hợp các xâu trên một bộ chữ nào đó.

Lưu ý đây là định nghĩa tổng quát nhất của ngôn ngữ. Trong thực tế, các ngôn ngữ tự nhiên hay lập trình đều có thể xem như một tập hợp các câu có một cấu trúc quy định nào đó. Điều đó nói lên chúng có một số ràng buộc, và các ngôn ngữ đó chỉ là một tập con của ngôn ngữ tổng quát.

Ví dụ 2.2: Tập rỗng \emptyset và tập chứa xâu rỗng $\{\epsilon\}$ là các ngôn ngữ trên mọi bộ chữ bất kỳ. Lưu ý rằng \emptyset và $\{\epsilon\}$ là khác nhau.

Tập tất cả các xâu trên một bộ chữ Σ , ký hiệu Σ^* , cũng là một ngôn ngữ. Mỗi ngôn ngữ trên bộ chữ Σ là một tập con của Σ^* . Tập Σ^* là vô hạn đếm được, vì có thể kể lần lượt mọi xâu của nó theo thứ tự độ dài tăng dần, và khi có cùng độ dài thì theo thứ tự từ điển. Chẳng hạn với $\Sigma = \{0,1\}$ thì

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$$

Tập tất cả các xâu trên bộ chữ Σ , loại trừ xâu rỗng ϵ , được ký hiệu là Σ^+ .

2. Biểu diễn ngôn ngữ

Như đã định nghĩa ở trên, một ngôn ngữ L trên một bộ chữ Σ là một tập con của tập Σ^* . Vậy vấn đề đặt ra đối với một ngôn ngữ L là:

- Cho một xâu w , xâu đó có thuộc ngôn ngữ L hay không?
- Làm thế nào để sinh ra một xâu w trong ngôn ngữ L ?

Đây là vấn đề biểu diễn ngôn ngữ và cũng là vấn đề cơ bản trong lý thuyết ngôn ngữ hình thức. Người ta cũng thường gọi đó là bài toán đoán nhận và tổng hợp ngôn ngữ. Hai vấn đề trên là tương đương với nhau.

Có một vài phương pháp để biểu diễn ngôn ngữ. Đối với các ngôn ngữ hữu hạn thì để biểu diễn chỉ cần liệt kê tất cả các xâu.

Ví dụ 2.3:

$$L_1 = \{a, ba, aaba, bbbb\}$$

Nhưng đối với các ngôn ngữ vô hạn (như các ngôn ngữ tự nhiên, ngôn ngữ lập trình), thì ta không thể liệt kê hết các xâu của chúng được. Vậy vấn đề là *phải tìm một cách biểu diễn hữu hạn cho một ngôn ngữ vô hạn*.

Trong những trường hợp không phức tạp lắm, ta có thể xác định các xâu của ngôn ngữ bằng cách chỉ rõ một đặc điểm cốt yếu của mỗi xâu đó. Đặc điểm của xâu có thể được mô tả bằng một câu tiếng Việt hay nói chung là một tân từ.

Ví dụ 2.4:

$$L_2 = \{ a^i \mid i \text{ là số nguyên tố} \}$$

$$L_3 = \{ w \in \{a,b\}^* \mid \text{số } a \text{ trong } w = \text{số } b \text{ trong } w \}$$

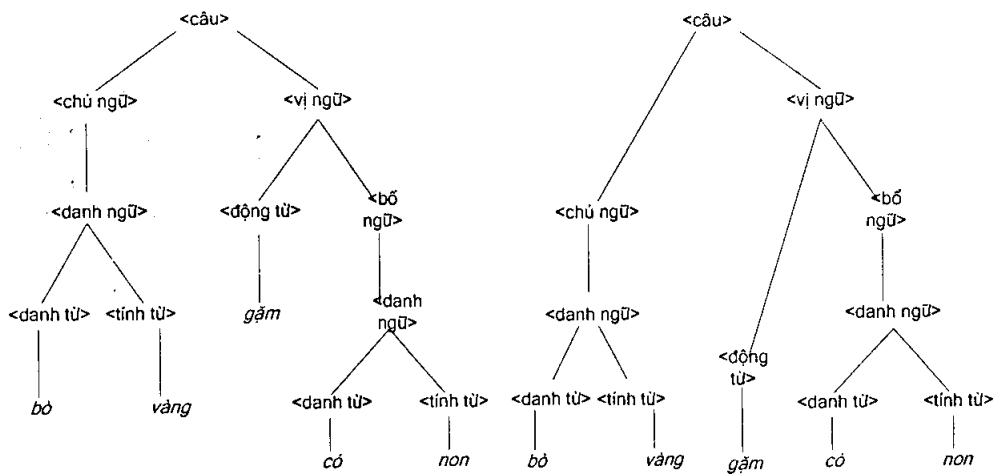
Tuy nhiên, người ta thường biểu diễn ngôn ngữ nhờ *một văn phạm* vì:

- Văn phạm là một công cụ mô tả hữu hạn ngôn ngữ rất hiệu quả.
- Là công cụ có định nghĩa toán học chặt chẽ, đã được nghiên cứu kỹ và là một thành phần chủ yếu của lý thuyết ngôn ngữ.

Lưu ý: Trong giáo trình này những từ *văn phạm*, *ngữ pháp*, *cú pháp* được dùng hoàn toàn tương đương với nhau.

3. Văn phạm (grammar)

Ta xét một ví dụ một cây diễn tả quá trình phân tích ra một câu trong tiếng Việt là: "*bò vàng gặm cỏ non*" như hình 2.1. Việc phân tích (hoặc ngược lại thành tổng hợp) bắt nguồn từ định *<định>* đi xuống mãi cho đến khi tạo thành cả câu.



Hình 2.1. Cây phân tích của một câu tiếng Việt. Hai cây trên chỉ khác nhau ở cách trình bày.

Chú ý rằng các lá của cây, như "bò", "vàng",... là những từ tạo thành câu được sản sinh. Ta gọi đó là các *ký hiệu kết thúc* (terminal), bởi vì với sự xuất hiện của chúng thì nhánh đó của cây cũng không được phát triển nữa (kết thúc). Trái lại các nút trong cây như là , , ,..., rốt cuộc thì sẽ không có mặt trong câu được sản sinh. Chúng chỉ có vai trò trung gian trong quá trình sản sinh, để diễn tả các cấu trúc bộ phận trong câu. Ta gọi đó là các *phạm trù cú pháp*, hay là các ký hiệu *không kết thúc* (non-terminal) hoặc *biến* do chúng còn được pháp triển, biến đổi tiếp.

Quá trình phân tích câu như trên chính là phương thức hoạt động của một văn phạm, mà định nghĩa hình thức cho như sau:

Định nghĩa 2.1: Một văn phạm là một hệ thống $G = (\Sigma, \Delta, P, S)$ trong đó:

1. Σ là tập hữu hạn các ký hiệu, gọi là ký hiệu kết thúc (terminal - còn gọi là ký hiệu cuối),
2. Δ là tập hữu hạn các ký hiệu, gọi là ký hiệu không kết thúc (nonterminal - còn gọi là ký hiệu trung gian hay biến), $\Sigma \cap \Delta = \emptyset$,
3. $S \in \Delta$ gọi là ký hiệu khởi đầu (initial),
4. P là tập hữu hạn các cặp xâu (α, β) và được gọi là *sản xuất* (production) hay *luật cú pháp* (rule) và thường được viết ở dạng là $\alpha \rightarrow \beta$. Các xâu này có thể bao gồm các ký hiệu kết thúc hoặc không kết thúc, xâu α phải có ít nhất một ký hiệu không kết thúc (có như thế nó mới phát triển tiếp tạo thành xâu β được).

Ví dụ 2.5: Từ hình 2.1, ta có:

$$\Sigma = \{ “bò”, “vàng”, “găm”, “cỏ”, “non” \}$$

$$\Delta = \{ <\text{câu}>, <\text{chủ ngữ}>, <\text{vị ngữ}>, <\text{danh ngữ}>, <\text{động từ}>, <\text{bổ ngữ}>, \\ <\text{danh từ}>, <\text{tính từ}> \}$$

$$S = <\text{câu}>$$

$$P = \{ <\text{câu}> \rightarrow <\text{chủ ngữ}> <\text{vị ngữ}>; <\text{chủ ngữ}> \rightarrow <\text{danh ngữ}>; \\ <\text{danh ngữ}> \rightarrow <\text{danh từ}> <\text{tính từ}>; <\text{vị ngữ}> \rightarrow <\text{động từ}> <\text{bổ ngữ}>; \\ <\text{bổ ngữ}> \rightarrow <\text{danh ngữ}>; <\text{danh từ}> \rightarrow “bò” | “cỏ”; <\text{tính từ}> \rightarrow \\ “vàng” | “non”; <\text{động từ}> \rightarrow “găm”; \}$$

Trong quá trình phát triển cây, ta có thể có xâu α :

Đầu tiên chỉ gồm một phần tử:

$$\{ <\text{câu}> \}$$

và có các xâu α, β như sau:

$$\{ <\text{chủ ngữ}> <\text{vị ngữ}> \}$$

$$\{ <\text{danh ngữ}> <\text{vị ngữ}> \}$$

$$\{ “bò” “vàng” <\text{vị ngữ}> \}$$

...

kết quả cuối cùng là xâu β như sau:

$$“bò” “vàng” “găm” “cỏ” “non”$$

Quy ước 2.1: Để tiện việc theo dõi, ta quy ước về cách viết thống nhất xuyên suốt giáo trình này như dưới đây. Một số quy ước khác chỉ làm chi tiết thêm hoặc là một phần của quy ước này.

1) Dùng các chữ in hoa $A, B, C, D, E\dots$ hoặc cụm từ trong cặp ngoặc nhọn (như $<\text{chủ ngữ}>$) để trả các ký hiệu không kết thúc;

2) Dùng các chữ thường $a, b, c, d, e\dots$ và các con số, các phép toán $+, -, *, /$, cặp ngoặc đơn để trả các ký hiệu kết thúc. Trong một số trường hợp dùng quy ước là một từ được in đậm (như **số** và **chữ**) hoặc cụm từ trong cặp ngoặc kép (như “**bò**”);

3) Dùng các chữ in hoa X, Y, Z để trả các ký hiệu có thể là kết thúc hoặc không kết thúc;

4) Dùng các chữ thường u, v, w, x, y, z để trả các xâu ký hiệu cuối;

5) Dùng các chữ thường Hy Lạp α, β, γ để trả các xâu gồm các biến và ký hiệu cuối;

6) Nếu có các sản xuất cùng về trái $A \rightarrow \alpha$ và $A \rightarrow \beta$ thì ta viết gộp lại cho gọn thành $A \rightarrow \alpha \mid \beta$. Các sản xuất có cùng một ký hiệu không kết thúc về trái có thể gọi chung bằng tên ký hiệu về trái. Ví dụ, sản xuất $-A$.

Ngoài ra, ta cũng có một số quy ước phụ khác như sau:

7) $V = (\Sigma \cup \Delta)^*$ là một xâu (có thể rỗng) bao gồm cả ký hiệu không kết thúc và ký hiệu kết thúc;

8) V^* là tập tất cả các xâu V có thể có;

9) V^+ cũng như vậy trừ xâu rỗng;

10) $||$ là ký hiệu độ dài xâu (ví dụ $|\alpha|$ là độ dài của xâu α);

11) Ký hiệu ε là một ký hiệu đặc biệt, chỉ xâu rỗng hoặc ký hiệu rỗng.

Sau khi định nghĩa văn phạm, ta sẽ tiếp tục định nghĩa tiếp một số khái niệm cơ bản.

Định nghĩa 2.2: Suy dẫn (derivation). Cho văn phạm $G = (\Sigma, \Delta, P, S)$ như trên, ta gọi *suy dẫn trực tiếp* là một quan hệ hai ngôi ký hiệu \Rightarrow trên tập V^* nếu $\alpha\beta\gamma$ là một xâu thuộc V^* và $\beta \Rightarrow \delta$ là một sản xuất trong P , thì $\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$.

Ta gọi *suy dẫn k bước* ký hiệu là $\stackrel{k}{\Rightarrow}$ hay $\stackrel{k}{\Rightarrow} \beta$ nếu tồn tại dãy $\alpha_0, \alpha_1, \dots, \alpha_k$ của $k+1$ xâu sao cho:

$$\alpha = \alpha_0, \dots, \alpha_{i-1} \Rightarrow \alpha_i \text{ với } 1 \leq i \leq k \text{ và } \alpha_k = \beta$$

Ta nói xâu α *suy dẫn* xâu β và viết là $\alpha \stackrel{*}{\Rightarrow} \beta$ khi và chỉ khi $\alpha \stackrel{i}{\Rightarrow} \beta$ với một i nào đó, $i \geq 0$.

Ta nói xâu α *suy dẫn không tầm thường* xâu β (derives in a nontrivial) và viết là $\alpha \stackrel{+}{\Rightarrow} \beta$ khi và chỉ khi $\alpha \stackrel{i}{\Rightarrow} \beta$ với một i nào đó, $i \geq 1$.

Định nghĩa 2.3: Ngôn ngữ của văn phạm G là tập hợp các xâu ký hiệu kết thúc, ta ký hiệu là $L(G)$, được suy dẫn từ S :

$$L(G) = \{ w \mid w \in \Sigma^* \text{ và } S \stackrel{*}{\Rightarrow} w \}$$

hoặc:

$$L(G) = \{ w \mid w \in \Sigma^* \text{ và } w \stackrel{*}{\Rightarrow} S \}$$

Định nghĩa 2.4: Hai văn phạm G_1 và G_2 (sản sinh hoặc đoán nhận) là *tương đương* khi và chỉ khi $L(G_1) = L(G_2)$.

Định nghĩa đơn giản này cho ta khả năng để biến đổi văn phạm, mà sự nhận biết ngôn ngữ sinh ra bởi văn phạm đó phức tạp, thành văn phạm sinh ra cùng ngôn ngữ mà sự nhận biết dễ dàng và có hiệu quả hơn (xét về thời gian và bộ nhớ).

4. Phân loại Chomsky (Chomsky hierarchy)

Trên cơ sở định nghĩa về văn phạm nói chung, Chomsky (1950) đã chia bốn lớp văn phạm như sau:

Cho văn phạm $G = (\Sigma, \Delta, P, S)$, ta gọi nó là văn phạm:

1. Lớp 0, văn phạm ngữ cầu (phrase - structure) nếu sản xuất có dạng:

$$\alpha \rightarrow \beta \text{ trong đó } \alpha \in V^+, \beta \in V^*$$

hoặc cũng có thể nói, lớp văn phạm này không bị ràng buộc gì.

2. Lớp 1, văn phạm cảm ngữ cảnh (context - sensitive) nếu sản xuất có dạng:

$$\alpha \rightarrow \beta \text{ thỏa mãn điều kiện } |\alpha| \leq |\beta|$$

3. Lớp 2, văn phạm phi ngữ cảnh (context free - viết tắt là VPPNC) nếu sản xuất có dạng:

$$A \rightarrow \alpha \text{ trong đó } A \in \Delta, \alpha \in V^*$$

4. Lớp 3, văn phạm chính quy (regular - viết tắt là VPCQ) nếu sản xuất có dạng:

$$A \rightarrow a, A \rightarrow Ba \quad \text{trong đó} \quad A, B \in \Delta, a \in \Sigma$$

hoặc tương tự

$$A \rightarrow a, A \rightarrow aB \quad \text{với} \quad A, B \in \Delta, a \in \Sigma$$

Các ngôn ngữ đoán nhận bởi các văn phạm trên cũng được xếp loại theo tên của văn phạm. Ta gọi đó là *sự phân cấp Chomsky* về ngôn ngữ.

Để đoán nhận các loại ngôn ngữ trên, ta dùng các công cụ sau:

- Ngôn ngữ loại 0 được đoán nhận bởi một máy Turing;



Hình 2.2. Noam Chomsky (sinh năm 1928). Nhà ngôn ngữ người Mỹ đã tạo ra một cuộc cách mạng trong nghiên cứu ngôn ngữ.

- Ngôn ngữ loại 1 (cảm ngữ cảnh) được đoán nhận bởi một ôtômát tuyến tính giới nội (sai khác xâu rỗng);
- Ngôn ngữ loại 2 (phi ngữ cảnh - viết tắt là NNPNC) đoán nhận bởi một ôtômát đẩy xuống (không đơn định);
- Ngôn ngữ loại 3 (chính quy - viết tắt là NNCQ) được đoán nhận bởi một ôtômát hữu hạn (sai khác xâu rỗng).

Theo thứ tự, các lớp ngôn ngữ giảm dần về độ phức tạp cũng như phạm vi.

Ví dụ 2.6

Cho VPPNC $G = (\Sigma, \Delta, P, S)$ với $\Sigma = \{a,b\}$, $\Delta = \{S,A,B\}$ và P có các sản xuất như sau:

$$S \rightarrow aB \mid bA, A \rightarrow aS \mid bAA \mid a, B \rightarrow bS \mid aBB \mid b$$

Trong bốn lớp văn phạm theo phân loại Chomsky, lớp VPPNC và lớp con của nó VPCQ là những lớp quan trọng nhất đối với ngôn ngữ lập trình và chương trình dịch. VPPNC được dùng trong đa số các cấu trúc cú pháp của ngôn ngữ lập trình. VPCQ dùng trong đa số các cấu trúc từ vựng.

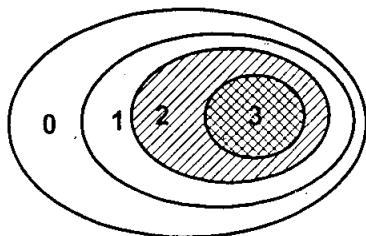
Một văn phạm chỉ ra thứ tự mà các ký hiệu trong ngôn ngữ có thể kết hợp với nhau để tạo ra các câu đúng trong một ngôn ngữ. Con người thường dùng một ngôn ngữ có văn phạm tương đối thoải mái. Các ngôn ngữ lập trình cố gắng mô phỏng ngôn ngữ con người, nhưng do hạn chế của máy, chúng phải dùng các văn phạm đơn giản hơn, ít khuôn dạng và cứng nhắc hơn - hay nói cách khác, ngôn ngữ của chúng bị giới hạn chặt chẽ (và nhỏ) hơn ngôn ngữ con người.

B. VĂN PHẠM CHÍNH QUY VÀ ÔTÔMÁT HỮU HẠN

I. VĂN PHẠM CHÍNH QUY

Theo phân loại của Chomsky ở phần A thì văn phạm chính quy là văn phạm mà tập hữu hạn các sản xuất P đều có dạng:

$$A \rightarrow a \text{ và } A \rightarrow Ba \quad \text{hoặc} \quad A \rightarrow a \text{ và } A \rightarrow aB \\ A, B \in \Delta, a \in \Sigma$$



Hình 2.3. Các lớp ngôn ngữ theo phân loại Chomsky và hai lớp trọng tâm đối với giáo trình này.

Trong chương trình dịch, văn phạm chính quy dùng rất nhiều cho việc biểu diễn và phân tích từ vựng.

VPCQ có một cách biểu diễn tương đương là biểu thức chính quy.

Biểu thức chính quy (Regular Expression)

Biểu thức chính quy dùng bộ ký hiệu quy ước sau:

| nghĩa là hoặc (or)

() nhóm các thành phần

* lặp lại không hoặc nhiều lần

Ví dụ 2.7: Để biểu diễn luật tạo một tên mới trong ngôn ngữ lập trình C hoặc Pascal bạn có thể biểu diễn bằng những cách như sau:

Bằng lời: Một tên là một xâu bắt đầu bằng chữ cái, tiếp sau có thể là chữ cái hoặc số.

Bằng văn phạm chính quy:

$\langle \text{tên} \rangle \rightarrow \text{chữ} \langle \text{phần tiếp theo của tên} \rangle$

$\langle \text{phần tiếp theo của tên} \rangle \rightarrow \epsilon$

$\langle \text{phần tiếp theo của tên} \rangle \rightarrow \text{chữ} \langle \text{phần tiếp theo của tên} \rangle$

$\langle \text{phần tiếp theo của tên} \rangle \rightarrow \text{số} \langle \text{phần tiếp theo của tên} \rangle$

Ta có thể biểu diễn gọn gàng bằng biểu thức chính quy:

$\langle \text{tên} \rangle \rightarrow \text{chữ} (\text{chữ} | \text{số})^*$

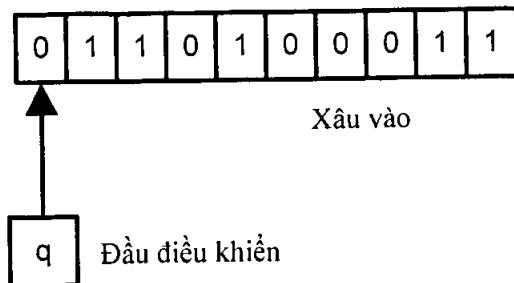
Ngoài ra, do VPCQ là một lớp con của VPPNC, nên nó có thể dùng các cách biểu diễn dành cho VPPNC. Một cách thường dùng là *đồ thị chuyển*. Trong một ví dụ ở phần sau ta sẽ thấy cách dùng đồ thị chuyển để biểu diễn khái niệm *tên* của nhiều ngôn ngữ lập trình.

II. ÔTÔMAT HỮU HẠN (finite automata – FA)

Ôtômat đầy xuống là một cái *máy đoán nhận* ngôn ngữ đối với lớp NNPNC. Nó là một cái máy trừu tượng, có cơ cấu và nguyên lý hoạt động hết sức đơn giản và được mô tả như dưới đây.

Hình 2.4 là sơ đồ của "máy". Máy đang ở một trạng thái q nào đó thuộc tập Q , đọc một ký hiệu a thuộc Σ nằm ở trên băng, và tùy thuộc vào q và a

nó sẽ đổi sang trạng thái mới $\delta(q, a)$ và chuyển đầu đọc sang ký hiệu tiếp theo trên băng ở bên phải, thực hiện một bước chuyển.



Hình 2.4. Mô hình ôtômát hữu hạn

Nếu $\delta(q, a)$ là một trong các trạng thái cho phép, thì FA thừa nhận xâu đang xét đoạn từ đầu xâu đến đầu đọc. Nếu đầu đọc lại ở cuối xâu (bên phải của băng) thì toàn bộ xâu là được thừa nhận. Một xâu vào w được gọi là *thừa nhận (accepted)* bởi một FA nếu sau khi đọc hết xâu vào và ôtômát chuyển sang được một trong các trạng thái mong muốn.

Chú ý rằng tập Q thể hiện các trạng thái ghi nhớ của ôtômát trong quá trình đoán nhận và như vậy khả năng ghi nhớ của ôtômát là hữu hạn. Vì vậy, ôtômát mô tả như trên được gọi là ôtômát hữu hạn.

Có nhiều cách biểu diễn hàm chuyển trong máy tính. Phương pháp đơn giản nhất là dùng một *bảng chuyển*. Trong bảng này, mỗi trạng thái có một dòng, mỗi một ký hiệu vào có một cột tương ứng. Vị trí dòng i , cột a trong bảng là một trạng thái hoặc một tập trạng thái có thể đạt tới bằng một dịch chuyển từ trạng thái i và ký hiệu vào a .

Trong mô tả trên, nếu ôtômát đang ở một trạng thái nào đó, đọc một ký hiệu vào và chỉ có duy nhất một chuyển đổi thì ôtômát này gọi là *ôtômát hữu hạn đơn định (Deterministic Finite Automata - DFA)*. Hàm chuyển này là hàm toàn phần và đơn trị, cho nên bước chuyển của ôtômát luôn luôn được xác định một cách duy nhất. Ngược lại, nếu từ một trạng thái và một ký hiệu vào, có thể có một hoặc nhiều khả năng đổi sang các trạng thái khác thì ôtômát này gọi là *ôtômát hữu hạn không đơn định (Nondeterministic Finite Automata - NFA)*. Như vậy DFA có thể coi là một trường hợp đặc biệt của NFA, khi với mỗi trạng thái, chỉ có duy nhất một chuyển đổi ứng với một ký hiệu vào. Cả hai ôtômát này (DFA và NFA) *tương đương về khả năng đoán nhận ngôn ngữ*.

C. VĂN PHẠM PHI NGỮ CẢNH VÀ ÔTÔMÁT ĐẦY XUỐNG

I. VĂN PHẠM PHI NGỮ CẢNH (context free grammar)

Theo phân loại của Chomsky như trên thì văn phạm phi ngữ cảnh là văn phạm mà tập hữu hạn các sản xuất P đều có dạng:

$$A \rightarrow \alpha \text{ với } A \in \Delta \text{ và } \alpha \in (\Sigma \cup \Delta)^*$$

Trong chương trình dịch, văn phạm phi ngữ cảnh dùng rất nhiều cho việc biểu diễn và phân tích cú pháp.

Dạng BNF

Dạng BNF (Backus - Naur Form) thực chất chỉ là một cách biểu diễn khác của VPPNC và có quy ước như sau:

- Các ký tự viết hoa biểu diễn các ký hiệu không kết thúc (nonterminal), cũng có thể thay bằng một xâu đặt trong cặp dấu $< >$ hoặc một từ in nghiêng;
- Các ký tự viết chữ nhỏ và dấu toán học biểu diễn các ký hiệu kết thúc (terminal), cũng có thể thay bằng một xâu đặt trong cặp dấu nháy kép “ ” hoặc một từ in đậm;
- Ký hiệu \rightarrow hoặc $=$ là ký hiệu chỉ phạm trù cú pháp ở về trái được giải thích bởi về phải;

Ký hiệu | chỉ sự lựa chọn.

Ví dụ 2.8: Định nghĩa toán hạng có thể là một biến (tên), số hoặc một biểu thức ở trong cặp dấu ngoặc đơn, viết dưới dạng BNF như sau:

$$<\text{Toán hạng}> = <\text{Tên}> | <\text{số}> | "(" <\text{Biểu thức}> ")"$$

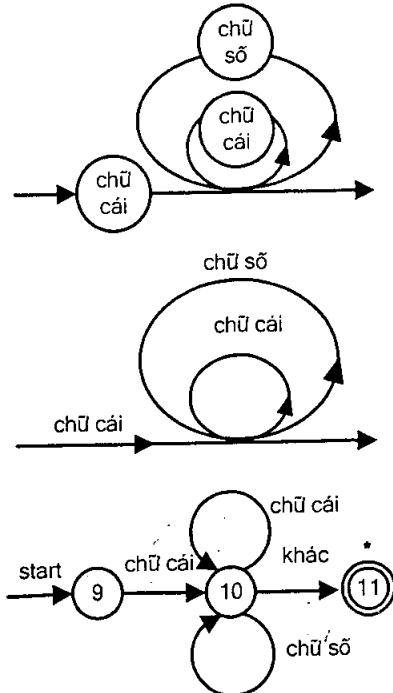
hoặc $\text{ToánHạng} \rightarrow \text{Tên} | \text{số} | (\text{BiểuThức})$

Đồ thị chuyển (transition diagram)

Một cách biểu diễn trực quan của dạng BNF là đồ thị chuyển. Có nhiều cách thể hiện đồ thị chuyển khác nhau như ví dụ ở hình 2.5 các đồ thị cùng biểu diễn khái niệm *tên* trong ngôn ngữ lập trình C và Pascal.

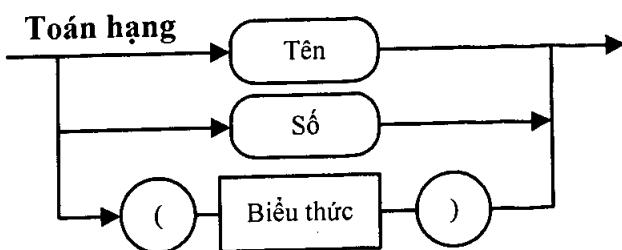
Người ta thường dùng các quy ước như sau: các vị trí trong đồ thị chuyển là các hình tròn và còn được gọi là các *trạng thái*. Các trạng thái lại được nối với nhau bởi các mũi tên, gọi là các *cạnh*. Các cạnh từ một trạng thái s đều có tên cho biết ký hiệu vào tiếp theo có thể xuất hiện sau khi đồ

thì đã đi tới s . Nhãn *khác* (còn lại) chỉ mọi ký hiệu khác các ký hiệu có trên các cạnh của s . Nhãn *start* (cũng có thể bỏ nhãn này) là trạng thái bắt đầu. Trạng thái vẽ bằng một vòng tròn kép chỉ ra rằng đó là một trạng thái cho phép, khi nó nhận ra dãy ký hiệu vào là một chuỗi xác định nào đó. Ví dụ như ở hình trên thì khi ở vòng tròn kép có nghĩa là đồ thị chuyển này đã chấp nhận xâu vào là một cái *tên* (theo các quy định của Pascal).



Hình 2.5. Đồ thị chuyển

Hình 2.6 là một đồ thị chuyển khác đã được biến đổi một chút để biểu diễn toán hạng trong các biểu thức số học.



Hình 2.6. Định nghĩa toán hạng

Ta cũng có thể kết hợp các đồ thị chuyển lại với nhau thành một đồ thị chuyển lớn hơn.

Cây suy dẫn (cây phân tích)

Để hình dung sự sản sinh ra một xâu trong văn phạm phi ngữ cảnh ta thường diễn tả một suy dẫn bằng một cây. Hình 2.1 là một ví dụ về cây suy dẫn. Cây suy dẫn là một đồ thị được định nghĩa một cách hình thức như dưới đây.

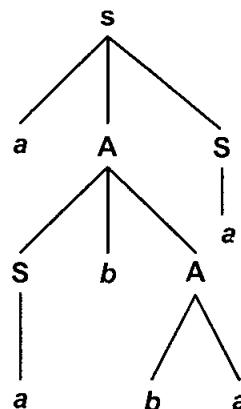
Định nghĩa 2.5: Cây suy dẫn trong một VPPNC $G = (\Sigma, \Delta, P, S)$ là một cây, trong đó:

1. Mọi nút có một nhãn, là một ký hiệu trong $(\Sigma \cup \Delta \cup \{ \epsilon \})$;

2. Nhãn của gốc là S ;

3. Nếu một nút có nhãn X là một nút trong thì $X \in \Delta$;

4. Nếu nút n có nhãn là X và các nút n_1, n_2, \dots, n_k là các con của nút n , theo thứ tự từ trái sang phải, và lần lượt mang các nhãn X_1, X_2, \dots, X_k thì $X \rightarrow X_1 X_2 \dots X_k$ phải là một sản xuất trong P ;



Hình 2.7. Cây suy dẫn.

Nếu nút n có nhãn là ϵ , thì n phải là một lá, và là con duy nhất của cha nó.

Ví dụ 2.9: Cho VPPNC $G = (\{a, b\}, \{S, A\}, P, S)$, với P gồm: $S \rightarrow aAS \mid a, A \rightarrow SbA \mid SS \mid ba$

Ta có một cây suy dẫn như hình 2.7.

Nếu ta đọc các nhãn của các lá, theo thứ tự từ "trái qua phải" ta có một dạng câu và gọi là kết quả của cây suy dẫn. Chẳng hạn $aabbba$ là kết quả của cây suy dẫn ở hình trên.

Ta gọi cây con của một cây suy dẫn là một nút nào đó cùng với các nút bè dưới của nó, các nhánh nối chúng và các nhãn kèm theo. Cây con cũng giống cây suy dẫn, chỉ khác là nhãn của gốc không nhất thiết là ký hiệu đầu S . Nếu nhãn của gốc của cây con là A , thì gọi đó là một A -cây.

Trong một số chương sau, ta sẽ làm việc với cây con của cây suy dẫn nên ta sẽ chỉ ra cụ thể định của nó là ký hiệu nào.

Đệ quy (recursive)

Định nghĩa 2.6: Ký hiệu không kết thúc A của văn phạm $G = (\Sigma, \Delta, P, S)$ gọi là đệ quy nếu tồn tại

$$A \stackrel{+}{\Rightarrow} \alpha A \beta \quad \text{với } \alpha, \beta \in V^*$$

Nếu $\alpha = \varepsilon$, khi đó A gọi là *dễ quy trái*.

Nếu $\beta = \varepsilon$, khi đó A gọi là *dễ quy phải*.

Nếu $\alpha \neq \varepsilon$ và $\beta \neq \varepsilon$, khi đó A gọi là *dễ quy trong*.

Các suy dẫn bên trái nhất và bên phải nhất

Định nghĩa 2.7: Ta gọi *suy dẫn bên trái nhất* (nói gọn là *suy dẫn trái*), nếu ở mỗi bước suy dẫn, biến được thay thế là biến nằm bên trái nhất trong dạng câu. Tương tự ta gọi *suy dẫn bên phải nhất* (nói gọn là *suy dẫn phải*), nếu ở mỗi bước suy dẫn, biến được thay thế là biến nằm bên phải nhất trong dạng câu.

Mỗi cây suy dẫn với kết quả α tương ứng với nhiều suy dẫn $S \Rightarrow \alpha$. Các suy dẫn này có cùng độ dài, vì áp dụng cùng một số các sản xuất như nhau (là các sản xuất tương ứng với các nút trong của cây). Chúng chỉ khác nhau ở thứ tự áp dụng các sản xuất đó. Trong số các suy dẫn này chỉ có một suy dẫn bên trái nhất và một suy dẫn bên phải nhất. Đôi khi để ký hiệu đó là suy dẫn bên trái nhất, người ta thêm chữ lm (leftmost) vào dưới dấu suy dẫn thành \Rightarrow_{lm} . Tương tự ta có \Rightarrow_{rm} chỉ suy dẫn phải nhất.

Tuy nhiên với một xâu $\alpha \in L(G)$, rất có thể có nhiều cây suy dẫn với kết quả chung α . Điều đó có nghĩa là xâu α có thể phân tích cú pháp theo nhiều cách khác nhau.

Nhập nhằng (ambiguity)

Định nghĩa 2.8:

- Một VPPNC G gọi là *văn phạm nhập nhằng* nếu có một xâu α là kết quả của hai cây suy dẫn khác nhau trong G . Ngôn ngữ do văn phạm này sinh ra gọi là *ngôn ngữ nhập nhằng*.
- Một NNPNC L được gọi là *ngôn ngữ nhập nhằng có hữu* nếu mọi VPPNC sản sinh ra L đều nhập nhằng. Người ta chứng minh rằng tồn tại NNPNC nhập nhằng có hữu.

Ví dụ 2.10 : Xét VPPNC G_0 cho bởi các sản xuất sau:

$$S \rightarrow S + S \mid S * S \mid (S) \mid a$$

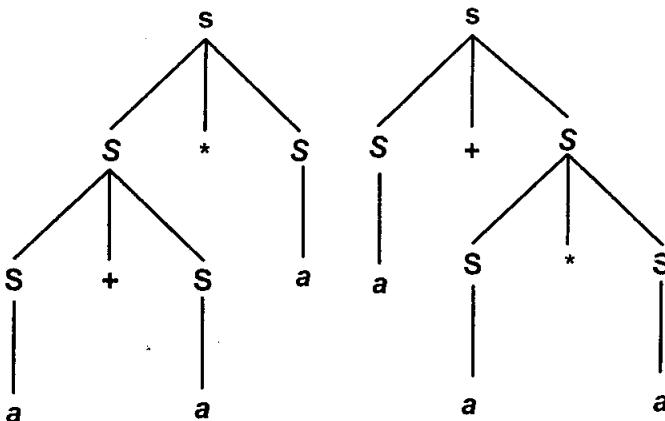
Văn phạm này cho ta viết các biểu thức số học với các phép toán + và *. Cây suy dẫn cho ở hình trên có kết quả là $a + a * a$. Suy dẫn bên trái nhất ứng với cây suy dẫn đó là:

$$S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * a$$

Suy dẫn bên phải nhất ứng với cây đó là:

$$S \Rightarrow S * S \Rightarrow S * a \Rightarrow S + S * a \Rightarrow S + a * a \Rightarrow a + a * a$$

Như vậy có hai cây suy dẫn khác nhau với cùng kết quả là $a + a * a$. Điều đó có nghĩa là biểu thức $a + a * a$ có thể được hiểu theo hai cách khác nhau: thực hiện phép cộng trước hay thực hiện phép nhân trước.



Hình 2.8. Nhập nhằng

II. BÀI TOÁN PHÂN TÍCH NGÔN NGỮ ĐÓI VỚI LỚP VPPNC

Bài toán thành viên với ngôn ngữ phi ngữ cảnh

Định lý: Tồn tại giải thuật để xác định, với mỗi văn phạm phi ngữ cảnh bất kỳ $G = (\Sigma, \Delta, P, S)$ và một xâu bất kỳ $w \in \Sigma^*$, xác định được $w \in L(G)$ hay không.

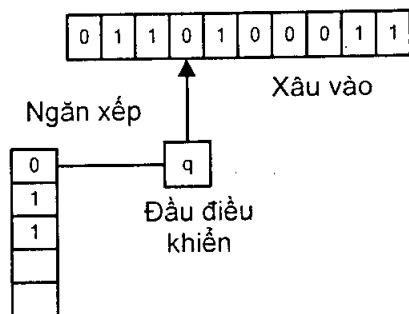
Người ta đã chứng minh được định lý này bằng cách đưa ra các giải thuật cài đặt trên thực tế, ví dụ như:

- Thuật toán phân tích Top - Down.
- Thuật toán phân tích Bottom - Up.
- Thuật toán phân tích CYK (Coke-Younger-Kasami).
- Thuật toán phân tích Earley.

Một số thuật toán phân tích trên ta sẽ đề cập đến trong các chương sau.

III. ÔTÔMÁT ĐẦY XUỐNG (pushdown automata – PDA)

Lớp các NNPNC vốn được sinh ra từ các VPPNC, có thể được đoán nhận bởi một cái "máy" sẽ trả lời *đúng* nếu xâu vào là một câu trong một ngôn ngữ nào đó mà nó biết, và *sai* trong trường hợp khác. Cái máy này được gọi là các ôtômát đầy xuồng không đơn định (Pushdown automata-PDA). Do lớp VPPNC phức tạp hơn VPCQ nên cái máy này cũng phải phức tạp hơn.



Hình 2.9. Mô hình ôtômát đầy xuồng

Ôtômát đầy xuồng bao gồm một bộ điều khiển hữu hạn trạng thái, một đầu đọc cho phép đọc lần lượt từ trái sang phải các ký hiệu của xâu vào ghi trên một băng vào, và một ngăn xếp được tổ chức theo nguyên tắc "vào sau ra trước" (last in first out, hay LIFO). Khi đưa một ký hiệu vào ngăn xếp, thì ký hiệu đó được đặt lên đầu ngăn, và đẩy các ký hiệu cũ xuồng dưới nó. Còn khi đọc thì chỉ có một ký hiệu được đọc, là ký hiệu trên cùng. Khi đọc xong thì ký hiệu được đọc bị loại khỏi ngăn xếp và ký hiệu kè dưới nó sẽ trồi lên vị trí đầu ngăn. Đôi khi người ta còn gọi ngăn xếp này bằng tên danh sách đầy xuồng. Như vậy so với ôtômát hữu hạn, ôtômát này có thêm một ngăn xếp.

Một bước chuyển của ôtômát là như sau: căn cứ vào trạng thái hiện tại của bộ điều khiển, ký hiệu vào mà đầu đọc đang quan sát vào lúc đó và ký hiệu ở đầu ngăn xếp, ôtômát sẽ chuyển sang một trạng thái mới nào đó, ghi một xâu ký hiệu nào đó vào đầu ngăn xếp và dịch chuyển đầu đọc sang phải một ô. Cũng có khi ký hiệu vào không ảnh hưởng tới bước chuyển. Ta gọi đó là một bước chuyển "*nhấp mắt*", và trong bước chuyển đó đầu đọc vẫn đứng yên tại chỗ cũ. Thực chất của bước chuyển đặc biệt đó là một sự tạm ngừng quan sát băng vào để chấn chỉnh lại ngăn xếp.

Có hai cách khác nhau để thửa nhận xâu vào:

- Xâu vào được đọc xong và ôtômát đến được một trong các trạng thái cuối mong muốn.
- Xâu vào được đọc xong và ngăn xếp trở thành rỗng.

Hai cách thửa nhận xâu vào như trên là tương đương với nhau.

Ta sẽ đề cập lại và sử dụng ôtômát này trong chương 5 "*Các phương pháp phân tích hiệu quả*".

Bài đọc

Các thể hệ ngôn ngữ lập trình

Các chương trình dịch đầu tiên xuất hiện vào những năm đầu thập kỷ 50. Kho có thể chỉ ra thời điểm chính xác của sự kiện đó vì vào thời điểm đó đã có vài nhóm độc lập với nhau cùng nghiên cứu và thực hiện công việc này.

Các thể hệ đầu tiên

Trước khi một máy tính có thể thực hiện được một nhiệm vụ, nó cần phải được lập chương trình để hoạt động bằng cách đặt các thuật toán, biểu thức trong ngôn ngữ máy vào bộ nhớ chính ở dạng các con số (ngôn ngữ máy). Nguồn gốc của việc xuất hiện quá trình lập trình này là do các nhu cầu tính toán rất đa dạng và người lập trình mong muốn diễn đạt được tất cả các thuật toán trong ngôn ngữ máy. Sau đó lại xuất hiện nhu cầu cải tiến phương pháp để ngoài nhiệm vụ thiết kế thuật toán còn giúp người lập trình tránh hay phát hiện, tìm được lỗi và giúp chữa chúng (một quá trình gọi là debug) trước khi công việc được hoàn thành.

Bước đầu tiên nhằm loại bỏ các khó khăn có từ quá trình lập trình như trên là vứt bỏ việc dùng các con số nhảm chán khô khan và dễ gây lỗi (dùng để biểu diễn các mã phép toán và các toán tử có trong ngôn ngữ máy). Chỉ đơn giản bằng cách chọn các tên mô tả cho các ô bộ nhớ và các thanh ghi để biểu diễn các mã phép toán, người lập trình có thể tăng được rất nhiều khả năng đọc hiểu được của một chuỗi các lệnh. Ví dụ, chúng ta hãy xem một thủ tục viết bằng mã máy có nhiệm vụ cộng nội dung của các ô nhớ 6C và 6D lại với nhau và đặt kết quả vào ô 6E. Trước đây, các lệnh thực hiện công việc này viết trong mã 16 như sau:

156C

166D

5056

306E

C000

Nếu bây giờ chúng ta gán một cái tên PRICE (giá tiền) cho vị trí 6C, TAX (thuế) cho 6D và TOTAL (tổng số) cho 6E, và chúng ta có thể biểu diễn cùng thủ tục này như dưới đây sử dụng kỹ thuật gọi là *đặt ký hiệu gợi nhớ*:

LD	R5, PRICE
LD	R6, TAX
ADDI	R0, R5 R6
ST	R0, TOTAL
HLT	

Đa số chúng ta sẽ công nhận là dạng thứ hai dù vẫn còn khó, thì việc thực hiện công việc biểu diễn mục đích và ý nghĩa của thủ tục tốt hơn nhiều dạng đầu.

Khi kỹ thuật này lần đầu tiên được công bố, người lập trình dùng bộ ký hiệu này để thiết kế chương trình gốc trên giấy và sau đó sẽ dịch nó ra dạng mã máy. Việc này cũng không mất nhiều thời gian lắm do việc chuyển đổi thực hiện tương đối máy móc. Hệ quả là việc dùng các ký hiệu này dẫn đến việc hình thức hóa nó thành một ngôn ngữ lập trình gọi là ngôn ngữ Assembly, và một chương trình gọi là Assembler dùng để dịch tự động các chương trình khác viết trong ngôn ngữ Assembly thành ngôn ngữ máy tương ứng. Chương trình này được gọi là Assembler (trình dịch hợp ngữ) do nhiệm vụ của nó là tổng hợp thành các lệnh máy bằng cách dịch các ký hiệu gợi nhớ và các tên.

Ngày nay, Assembler trở thành một chương trình tiện ích thông thường trong hầu hết các máy tính. Với những hệ thống như vậy, người lập trình có thể nhập một chương trình ở dạng ký hiệu gợi nhớ nhờ các bộ soạn thảo của hệ thống, rồi dùng Assembler để dịch chương trình đó và lưu thành tệp mà sau đó có thể dùng để chạy được.

Như vậy, các ngôn ngữ Assembly đã được phát triển đầu tiên, chúng xuất hiện như là một bước tiến khổng lồ trên con đường tìm kiếm các môi trường lập trình tốt hơn. Trong thực tế, có nhiều nghiên cứu về chúng để biểu diễn một ngôn ngữ lập trình mới và toàn diện. Do đó, *các ngôn ngữ Assembly được coi là các ngôn ngữ thế hệ thứ hai*, còn *ngôn ngữ thế hệ đầu tiên chính là bản thân các ngôn ngữ máy*.

Mặc dù ngôn ngữ thế hệ thứ hai này có rất nhiều ưu điểm so với ngôn ngữ máy, chúng vẫn còn quá vắn tắt. Ngôn ngữ Assembly về nguyên tắc cũng giống như ngôn ngữ máy tương ứng. Chúng dẫn đến việc ngôn ngữ Assembly phụ thuộc vào từng loại máy cụ thể. Các lệnh dùng trong chương trình chỉ là biểu diễn các thuộc tính của máy. Mặt khác, một chương trình viết trong ngôn ngữ Assembly không dễ chuyển sang một loại máy khác và thường phải viết lại cho thích ứng với các thanh ghi và tập lệnh của máy mới.

Một nhược điểm nữa của ngôn ngữ Assembly là một người lập trình, mặc dù không buộc phải mã các lệnh ở dạng từng bit, thì thường bị bắt phải nghĩ đến các chi tiết, các thành phần nhỏ này, không được tập trung vào việc tìm các giải pháp tốt hơn. Tình trạng này cũng giống như thiết kế một ngôi nhà mà người thiết kế phải chú ý đến xi măng, vôi vữa, gạch, ngói, đinh, đá... Tuy quá trình thiết kế một ngôi nhà từ những thành phần nhỏ như thế cũng thực hiện được, nhưng việc thiết kế sẽ đơn giản đi rất nhiều nếu chúng ta suy nghĩ bắt đầu từ các phòng, nền, cửa sổ, mái nhà... và kết quả tổng thể cũng tốt hơn.

Như vậy, nguyên lý thiết kế dựa trên các phần tử nhỏ đi lên không phải là nguyên lý thích hợp trong việc thiết kế. Quá trình thiết kế tốt hơn là dùng các nguyên lý ở mức cao hơn, mỗi nguyên lý này biểu diễn một khái niệm liên quan với các thuộc tính chính của sản phẩm. Mỗi khi một thiết kế được hoàn tất, các thiết kế gốc có thể được dịch sang các khái niệm ở mức thấp hơn, liên quan đến việc thực hiện, giống một nhà xây dựng cuối cùng sẽ chuyển thiết kế trên giấy sang chi tiết các vật liệu xây dựng.

Theo triết lý này, các nhà khoa học máy tính bắt đầu phát triển các ngôn ngữ lập trình tốt hơn cho việc viết phần mềm so với các ngôn ngữ lập trình bậc thấp Assembly. Kết quả là *các ngôn ngữ thế hệ thứ ba đã ra đời, khác với các thế hệ trước ở chỗ chúng vừa là ngôn ngữ ở mức cao, lại vừa độc lập với máy.*

Nói chung, phương pháp của các ngôn ngữ lập trình thế hệ thứ ba này là nhận biết bộ các nguyên lý bậc cao cho việc phát triển phần mềm. Mỗi một nguyên lý này được thiết kế sao cho nó có thể thực hiện như là một chuỗi các nguyên lý mức thấp có trong ngôn ngữ máy. Ví dụ, câu lệnh:

assign Total the value Price plus Tax

hoặc Total = Price + Tax

cho thấy một hành động (câu lệnh) ở mức cao không cần quan tâm đến việc một máy tính cụ thể phải thực hiện nó như thế nào.

Các phát triển gần đây

Nói chung, thuật ngữ *ngôn ngữ thế hệ thứ tư* được dùng trong một số sản phẩm phần mềm mà có *cho phép người dùng tự biến đổi (tùy biến) phần mềm của họ mà không cần có chuyên môn*. Người lập trình trong các ngôn ngữ như vậy thường được yêu cầu chọn từ những gì hiện trên màn hình ở dạng câu hoặc biểu tượng. Những sản phẩm phần mềm bao gồm cả bảng tính giúp duy trì các bảng dữ liệu ở dạng các bản ghi kế toán; hệ CSDL giúp duy trì và lấy lại thông tin; các phần mềm đồ họa giúp phát triển các đồ họa, đồ thị, ảnh...; các bộ xử lý văn bản mạnh cho phép các tài liệu có thể kết hợp, sắp xếp lại, và đổi dạng. Thêm nữa, các phần mềm này nhiều khi lại được bổ lại tạo nên các hệ thống tổng thể. Với những hệ thống như vậy, một nhà kinh tế có thể kiến trúc nền và biến đổi các mô hình kinh tế, phân tích những thay đổi ảnh hưởng khác nhau có thể có trong nền kinh tế nói chung hoặc trong một lĩnh vực kinh doanh cụ thể nào đó, và đưa ra kết quả ở dạng một tài liệu viết với các hình đồ họa, lược đồ làm các phương tiện trợ giúp trực quan. Một người quản lý doanh nghiệp nhỏ có thể tùy biến cùng sản phẩm này để phát triển một hệ thống cho việc duy trì kho và tìm ra các ảnh hưởng của các mục lưu chuyển thấp nào đó... Rõ ràng là nếu ta phải viết các chương trình làm tất cả các tùy biến này thì đó đều là những chương trình lớn.

Những hệ thống này được coi là *thay thế cho các ngôn ngữ lập trình* do môi trường lập trình của chúng gần gũi với ứng dụng hơn các môi trường mà các ngôn ngữ thế hệ thứ ba cung cấp. Ví dụ, thay cho việc mô tả các thông tin được biểu diễn trong máy như thế nào, một bảng dữ liệu có thể hiển thị trên màn hình máy tính ra làm sao, hoặc cả hệ thống được cập nhật thông tin như thế nào, thì người lập trình dùng các phần mềm thế hệ thứ tư chỉ cần mô tả các mục dữ liệu sẽ xuất hiện trên bảng tính và chúng quan hệ với nhau như thế nào. Do vậy, người dùng có thể tùy biến và dùng bảng tính mà không cần quan tâm (hoặc hiểu) về các chi tiết liên quan đến các kỹ thuật đang được sử dụng.

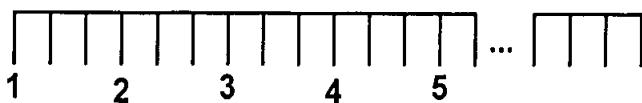
Thuật ngữ *ngôn ngữ thế hệ thứ năm* được dùng cho khái niệm lập trình mô tả, với việc nhấn mạnh phương pháp đặc biệt được biết như lập trình logic. Tư tưởng này thông minh hơn các tư tưởng trước đây. Đến giờ chúng ta sẽ chú ý rằng tư

tưởng lập trình khai báo cho phép người dùng máy tính giải các bài toán mà chỉ cần quan tâm đến đó là bài toán gì chứ không phải là nó được giải như thế nào. Quan điểm này có thể là thái quá đối với bạn. Tại sao chúng ta lại hy vọng giải được một bài toán mà không hề phải quan tâm đến cách giải nó như thế nào. Câu trả lời là chúng ta không giải bài toán này mà để máy tính giải hộ. Với phương pháp này, nhiệm vụ của chúng ta đơn thuần chỉ là khai báo về bài toán, trong khi đó máy tính phải thực hiện nhiệm vụ tìm lời giải cho nó.

Từ tư tưởng của thế hệ ngôn ngữ lập trình thứ năm, ta thấy nếu vẫn cứ được phát triển lên như vậy, thì cho đến một lúc nào đó máy tính sẽ hiểu được trực tiếp ngôn ngữ tự nhiên của con người.

Truyền thông giữa người và máy bằng
ngôn ngữ máy, trong đó con người
phải mô tả chi tiết mỗi bước lời giải.

Truyền thông giữa người và máy bằng
ngôn ngữ tự nhiên của con người, trong
đó máy sẽ tự sinh ra toàn bộ lời giải.



Các thế hệ ngôn ngữ lập trình.

Chuyện vui nhập nhằng ngôn ngữ

Một nhà ngôn ngữ học đi chơi cùng con. Đứa con chỉ vào tẩm biển của một cửa hiệu “GIẬT LÀ TÂY HẤP” và nhờ bố giải thích ý nghĩa.

Người bố giải thích: Chữ LÀ ở đây không có nghĩa LÀ LÀ LÀ mà nó có nghĩa LÀ LÀ LÀ.

Con: ????

Bài tập

1. Các vấn đề quan trọng nhất (các bài toán cơ bản) của biểu diễn ngôn ngữ là gì?
2. Người ta thường dùng những cách nào để biểu diễn ngôn ngữ?
3. Văn phạm là gì?
4. Có bao nhiêu lớp ngôn ngữ theo phân loại Chomsky?
5. Những lớp nào có thể phân tích được? Do đâu khẳng định?
6. Những lớp ngôn ngữ nào là quan trọng đối với các ngôn ngữ lập trình và chương trình dịch (thông thường)? Tại sao?
7. Mô tả cấu trúc của các PDA, NFA và DFA.

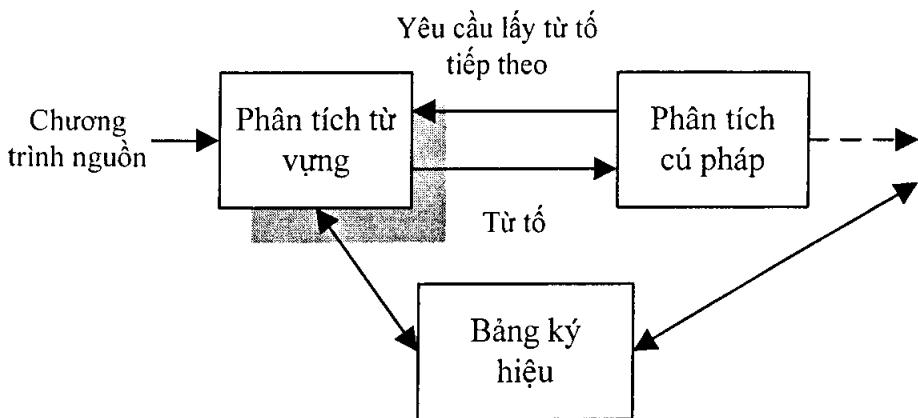
8. PDA và FA trả lời trực tiếp cho bài toán biểu diễn ngôn ngữ nào?
 9. PDA và FA dùng để đoán nhận các lớp văn phạm nào?
 10. PDA và FA dùng để đoán nhận ngôn ngữ. Ví dụ PDA có thể dùng để trả lời xem một xâu vào có đúng là xâu tiếng Việt không, vậy nó căn cứ vào đâu để xét? Nói cách khác, thông tin về tiếng Việt đặt ở đâu? Nếu muốn sửa ôtômát này để đoán nhận câu vào có phải là Pascal không có được không? Nếu được thì phải sửa ở chỗ nào? Sửa như vậy là sửa dữ liệu hay chương trình (hay cả hai)?
- Bài tập thực hành**
1. Bạn hãy xem xét một ngôn ngữ lập trình nào đó mà bạn thành thạo nhất (ví dụ như C hoặc Pascal). Thử tìm tất cả các luật ngữ pháp (văn phạm) của nó. Kiểm tra xem chúng thuộc lớp văn phạm nào.
 2. Bạn hãy sưu tầm tất cả luật ngữ pháp (văn phạm) tiếng Việt có thể được. Kiểm tra xem chúng thuộc lớp văn phạm nào.
 3. Viết chương trình mô phỏng PDA, NFA và DFA. Thử dùng các chương trình này kiểm tra một xâu vào có phải là C (hoặc Pascal) không.

Chương 3

PHÂN TÍCH TỪ VỰNG

I. MỤC ĐÍCH VÀ NHIỆM VỤ

Nhiệm vụ chính của phần phân tích từ vựng (lexical analysis) là đọc các ký tự vào từ văn bản chương trình nguồn và đưa ra lần lượt các từ tố (token) cùng một số thông tin thuộc tính (attribute). Phân tích từ vựng còn được gọi là *phân tích tuyến tính* hoặc đơn giản hơn: *phân quét*.



Hình 3.1. Vị trí khối Phân tích từ vựng.

Hình 3.1 là vị trí của bộ phân tích từ vựng trong một chương trình dịch. Mỗi khi nhận được yêu cầu lấy một từ tố tiếp theo từ bộ phân tích cú pháp, bộ phân tích từ vựng sẽ đọc các ký tự vào cho đến khi đưa ra được một từ tố.

Phần này có thể coi là phần tiền xử lý văn bản chương trình nguồn, làm cho nhiệm vụ của các giai đoạn sau đơn giản hơn. Quá trình này bao gồm các công việc:

1. *Xoá bỏ các ký tự không có nghĩa*. Các chú thích, dòng trống, các ký tự xuống dòng, dấu tab, các khoảng trắng không cần thiết đều bị xoá bỏ.

2. *Nhận dạng các ký hiệu*. Nhận dạng các ký tự liên nhau tạo thành một ký hiệu. Các dạng ký hiệu gọi là các từ tố. Các từ tố có thể là:

- a. *Từ khoá* như IF, WHILE, END,...;
- b. *Tên* của hằng số, kiểu, biến, hàm,...;
- c. *Các chữ số* như 3.14,...;
- d. *Xâu*, thường là nằm trong cặp dấu nháy như 'Đây là một xâu';
- e. *Các ký tự kết hợp*, như ':=' , '<=' , ...;
- f. *Các ký tự đứng riêng biệt*, như '(', '+';

3. *Số hoá ký hiệu*. Do các con số được xử lý dễ dàng hơn là các xâu, từ khoá, tên nên các xâu sẽ được thay bằng số, các chữ số sẽ được đổi thành số thật sự biểu diễn trong máy. Quá trình này được gọi là số hoá. Các tên sẽ được cắt trong danh sách tên, các xâu sẽ được cắt trong danh sách xâu, còn các chuỗi số trong danh sách hằng số.

Ví dụ 3.1: Một biểu thức (viết bằng ngôn ngữ lập trình Pascal) được đưa vào khối phân tích từ vựng có dạng là một đoạn văn bản:

*position := initial + rate * 60;*

Đoạn văn bản này có chứa các tên (biến) *position*, *initial*, *rate* ; một con số có giá trị là 60; ký tự kết hợp ':=' và các ký tự độc lập là '+', '*' và ','.

Kết quả ra sau khi phân tích từ vựng, đầu ra nhận được:

tên phép_gán tên toán_tử_cộng tên toán_tử_nhân số chấm_phẩy

Một số ký hiệu trong chuỗi này được xác định là duy nhất (ví dụ ký hiệu gán); một số khác như các tên và con số phải căn cứ vào lớp ký hiệu và phải phân biệt nhau bằng một giá trị (ví dụ dùng chỉ số của danh sách thay cho tên).

Ngoài ra, bộ phân tích từ vựng còn làm một nhiệm vụ phụ: đóng vai trò *giao diện với người sử dụng*. Nó xoá bỏ các ký tự thừa như các khoảng trắng, các chú thích, các ký tự hết dòng... làm cho chương trình không phụ thuộc vào chúng. Như vậy cũng có nghĩa nó cho phép người lập trình trình bày chương trình nguồn của mình tùy ý và dễ đọc hơn. Mặt khác, đây là phần duy nhất lưu các thông tin phụ về từ tố như số dòng, số cột của nó...

cho phép bộ phận báo lỗi chỉ chính xác nơi xảy ra lỗi trong chương trình nguồn. Nếu ngôn ngữ nguồn cho phép dùng các hàm macro, thì việc xử lý các macro cũng thường được đặt trong phần này.

Đôi khi người ta lại chia nhỏ bộ phân tích từ vựng thành hai giai đoạn nhỏ, chồng lên nhau. Giai đoạn thứ nhất gọi là "*quét*" (scanner), thường chỉ thực hiện các công việc đơn giản như bỏ qua các ký tự không có ý nghĩa và đọc vào một ký tự có nghĩa. Giai đoạn thứ hai gọi là "*phân tích từ vựng*", thường làm các công việc còn lại.

Sự cần thiết phải tách rời phân tích từ vựng với phân tích cú pháp

Như trên đã trình bày, phần phân tích của chương trình dịch bao gồm phần phân tích từ vựng, phần phân tích cú pháp và phân tích ngữ nghĩa. Trước đây phần phân tích từ vựng và phần phân tích cú pháp có thể viết làm một khối. Nay giờ có một số lý do để ta nên chia hai phần phân tích này tách rời nhau:

1. Thiết kế từng phần đơn giản hơn. Đây là lý do quan trọng nhất, phù hợp với các quy tắc thiết kế và bảo dưỡng chương trình.

2. Tính năng của chương trình dịch được cải tiến. Bộ phân tích từ vựng có thể cải tiến để nâng cao tốc độ phân tích, bộ phân tích cú pháp phân tích được các cấu trúc phức tạp hơn. Ngoài ra, có một số công cụ trợ giúp chỉ có thể thêm vào khi hai bộ phận này đã được tách rời nhau.

Ví dụ 3.2: Trong Turbo Pascal có chức năng High Light hiện các từ khoá, biến, chữ số, chú thích, các chỗ viết sai với các màu khác nhau là nhờ một số chức năng của phần phân tích này.

3. Cho phép chương trình dịch có thể chuyển đổi. Ví dụ như khi thay đổi bảng mã ký tự, quy ước lại các ký hiệu... thì việc sửa đổi ít, dễ dàng và chính xác hơn.

Phân tích từ vựng là phần đơn giản nhất của chương trình dịch. Tuy vậy, nó *lại chiếm một phần lớn trong tổng số thời gian dịch* (thường chiếm từ 20 đến 40%) do phải làm việc trực tiếp với chương trình nguồn ghi trên các thiết bị lưu trữ ngoài có tốc độ chậm (như ổ đĩa), điều đó cũng có nghĩa tác dụng cải tiến của nó đặc biệt quan trọng.

Từ tố (token), từ vị (lexeme) và mẫu (pattern)

Khi làm việc với phần phân tích từ vựng, ta thường phải dùng các thuật ngữ trên chỉ một số khái niệm khác nhau.

Đầu vào của bộ phân tích từ vựng là một chuỗi các ký tự. Một nhóm các ký tự kề nhau có thể tuân theo một quy ước (mẫu hay luật) nào đó và tạo thành một từ vị. *Từ vị (lexeme)* là các thành phần nhỏ nhất của một văn bản chương trình nguồn còn giữ một ý nghĩa nào đó.

Các từ vị có cùng mẫu (luật mô tả) lại được nhóm lại với nhau thành các *từ tố (token)*. Như vậy, một từ tố có thể chỉ nhiều từ vị khác nhau.

Mỗi một từ tố đều được mô tả bằng một *mẫu* (luật mô tả), do đó ta có cơ sở để phân biệt và nhận dạng các từ tố khác nhau.

Từ tố	Từ vị	Mẫu (luật mô tả)
const	const	const
if	if	If
quanhệ	<, <=, =, <>, >, >=	<, <=, =, <>, >, >=
tên	pi, count, i, d2	chữ cái theo sau là các chữ cái hoặc các số
số	3.14, 0, 6.02E23	bắt đầu một hằng số nào đó
xâu	"Xin chao cac ban"	bắt đầu chữ nào đặt trong cặp dấu ", trừ dấu "

Bảng 3.1. Ví dụ về từ tố, từ vị và mẫu.

Về mặt nào đó, ta có thể coi từ vị giống các từ cụ thể trong từ điển như *nhà, xe, ăn, com*. Còn từ tố gần giống khái niệm từ loại như *danh từ, động từ...* Như vậy, có thể có một tập các xâu từ chương trình nguồn có cùng một từ tố. Tập xâu đó cùng thỏa mãn luật mô tả. Ta cũng có thể coi mỗi từ tố là tên riêng của một luật mô tả và từ vị là các ví dụ thỏa mãn luật này.

Nhiệm vụ của chúng ta là phải thiết kế được một bộ phân tích từ vựng để nó có thể tự động phát hiện các từ vị ứng với các mẫu và biết được từ tố của nó.

Thuộc tính của các từ tố (attribute)

Như trên cho thấy, một từ tố có thể ứng với một tập các từ vị khác nhau, ta buộc phải thêm một số thông tin nữa để khi cần có thể biết được cụ thể đó là từ vị nào.

Ví dụ, các chữ số 15 và 267 đều là một chuỗi số, có từ tố là *num*, nhưng khi đến bộ sinh mã phải được biết cụ thể đó là số 15 hay 267.

Như vậy, bộ phân tích từ vựng phải cho biết các thông tin về từ tố cùng với các thuộc tính cần thiết. Từ tố sẽ tác động đến việc điều khiển của bộ phân tích cú pháp, thuộc tính lại ảnh hưởng đến việc sinh mã.

Trong thực tế, thông thường một từ tố sẽ chứa một con trỏ, trỏ đến một vị trí trên bảng ký hiệu có chứa các thông tin về nó.

Ví dụ 3.3: Cho biểu thức

*position := initial + rate * 60;*

thì dãy từ tố nhận được bao gồm:

<đại lượng, con trỏ đến *position* trên bảng ký hiệu >
<phép_gán, >
<đại lượng, con trỏ đến *initial* trên bảng ký hiệu >
<toán_tử_cộng, >
<đại lượng, con trỏ đến *rate* trên bảng ký hiệu >
<toán_tử_nhân, >
<sốnguyên, giá trị số nguyên 60>

II. XÁC ĐỊNH TỪ TỐ

1. Biểu diễn từ tố

Như trên đã nói, các từ tố khác nhau có các luật mô tả (mẫu) khác nhau, tức là các luật mô tả này là cơ sở để nhận dạng được từ tố.

Cách biểu diễn các luật này đơn giản và thông dụng nhất là bằng lời. Tuy nhiên, nếu bằng lời ta sẽ gặp các hiện tượng như nhập nhằng (cùng một lời nói có thể hiểu theo nhiều ý khác nhau), phát biểu theo nhiều cách khác nhau, khó đưa vào máy tính... Ví dụ, ta có thể phát biểu luật đặt tên trong ngôn ngữ lập trình Pascal theo nhiều cách như sau:

- *Tên là một xâu có một chữ cái và theo sau không hoặc nhiều chữ cái hoặc số.*
- *Tên là một xâu bắt đầu là một chữ cái, tiếp theo có thể có các chữ cái hoặc số.*
- *Tên là một xâu có ký hiệu đầu tiên là chữ, các ký hiệu còn lại phải là chữ hoặc số.*
- ...

Do đó, ta cần tìm hiểu cách hình thức hoá và biểu diễn các luật trên trong máy tính để làm sao chương trình có thể xử lý được. Việc này có thể thực hiện được nhờ biểu thức chính quy và ôtômat hữu hạn. Điều đó cũng có nghĩa là, chỉ cần lớp ngôn ngữ nhỏ nhất, đơn giản nhất - lớp ngôn ngữ chính quy (ứng với văn phạm chính quy) là đủ để biểu diễn các luật mô tả từ tố - vôn cung rất đơn giản. Ngoài ra, ta có thể mượn cách biểu diễn trực quan của VPPNC - đồ thị chuyển, để dùng biểu diễn các luật đó.

a. Biểu thức chính quy (Regular Expression)

Biểu thức chính quy có các ký hiệu quy ước ta đã học trong chương trước. Với bộ ký hiệu của biểu thức chính quy, ta biểu diễn lại các khái niệm về chữ cái, chữ số, tên, phép quan hệ trong Pascal như sau:

chữ cái $\rightarrow A | B | \dots | Z | a | b | \dots | z$

chữ số $\rightarrow 0 | 1 | \dots | 9$

tên \rightarrow *chữ cái* (*chữ cái* | *chữ số*)^{*}

quan hệ $\rightarrow < | \leq | = | \neq | > | \geq$

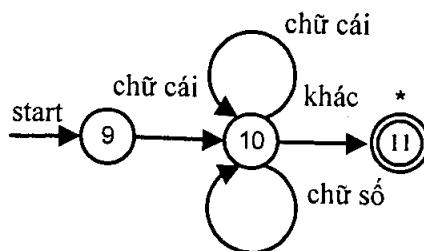
Ngoài những quy ước của biểu thức chính quy như trên, người ta còn đưa thêm vào những quy ước nhằm viết gọn hơn như sau:

 + lặp lại một hoặc nhiều lần

 ? lặp lại không hoặc một lần

b. Đồ thị chuyển (transition diagram)

Phương pháp đơn giản và hiệu quả để xây dựng một bộ phân tích từ vựng là xây dựng một lược đồ minh họa cấu trúc của các từ tố của ngôn ngữ nguồn. Sau đó, ta chuyển chúng thành chương trình tìm từ tố.



Hình 3.2. Đồ thị chuyển cho khái niệm trên

Đồ thị chuyển miêu tả vị trí các ký tự đầu vào của bộ phân tích từ vựng. Ví dụ ở hình 3.2 biểu diễn khái niệm *tên* trong Pascal. Khi ở trạng thái *vẽ* bằng vòng tròn kép có nghĩa là đồ thị chuyển này đã chấp nhận xâu vào là một cái *tên*.

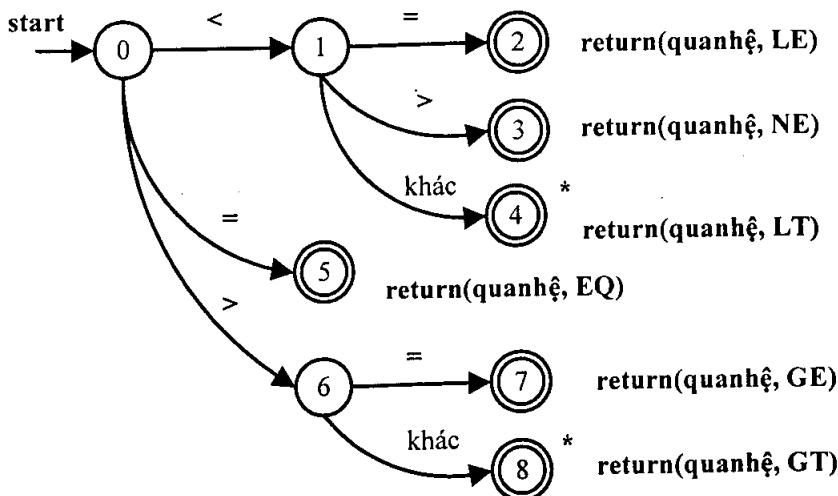
Chúng ta sẽ coi các đồ thị này là luôn *xác định*, có nghĩa là không thể có hai cạnh từ một trạng thái *s* lại có cùng nhãn. Nói cách khác, đồ thị chuyển này sẽ được dùng cho máy đoán nhận ngôn ngữ - ôtômát hữu hạn đơn định (DFA).

Ở một số trạng thái, ta thêm một dấu * để nói rằng, đồ thị chuyển đã xử lý quá một ký hiệu của phần khác. Ký hiệu này sẽ được lưu và trả lại khi đoán nhận phần khác. Ta có thể nhận thấy là các dấu * này phần lớn được đánh dấu ở trạng thái có cạnh hướng đến với nhãn *khác*.

Khi hợp các đồ thị con lại với nhau ta sẽ được một đồ thị chuyển thống nhất.

Ví dụ 3.4: Các toán tử quan hệ trong C hoặc Pascal được biểu diễn bằng biểu thức chính quy như phần trên và đồ thị chuyển kết hợp như hình 3.3.

Các chú thích như return (quan hệ, LE) cho biết trạng thái đó trả về là một phép tính quan hệ và giá trị thuộc tính của nó là LE (viết tắt của phép quan hệ nhỏ hơn hoặc bằng).



Hình 3.3. Đồ thị chuyển cho các phép quan hệ

2. Viết chương trình cho đồ thị chuyển đổi

a. Tập hợp tất cả các mẫu của từ tố

Ta biểu diễn chúng bằng các biểu thức chính quy và đồ thị chuyển.

Ví dụ 3.5: Ngoài các biểu thức chính quy và đồ thị chuyển của các mẫu tên và quan hệ như trên, ta có các khái niệm khác trong Pascal như sau:

số thực \rightarrow chữ số⁺ (.chữ số⁺) ? (E (+ | -) ? chữ số⁺) ?

số thực \rightarrow chữ số⁺. chữ số⁺

số nguyên \rightarrow chữ số⁺

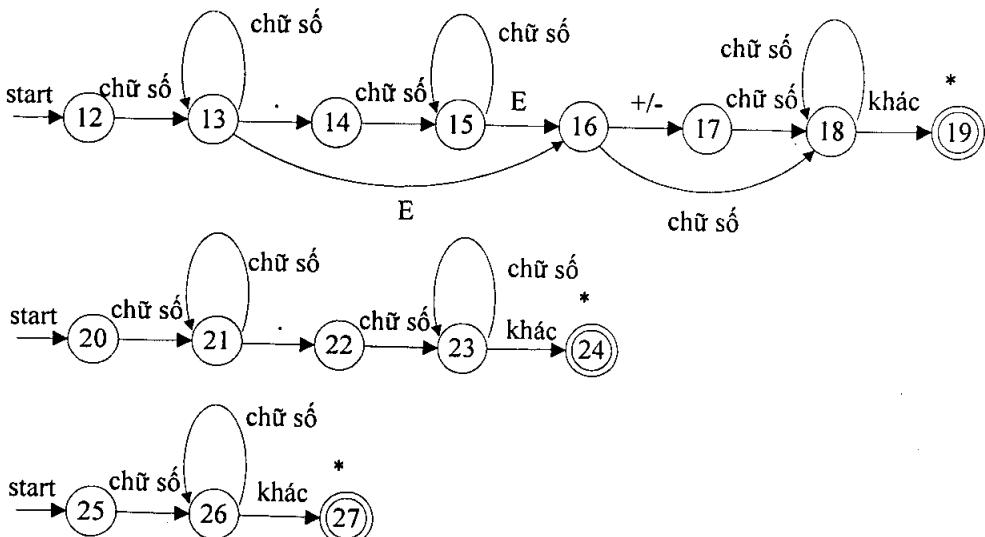
Đồ thị chuyển của chúng như hình 3.4 với các trạng thái bắt đầu là 12, 20 và 25.

b. Lập bộ phân tích từ vựng bằng phương pháp diễn giải đồ thị chuyển

Ta thấy, tại các nút hay các trạng thái có thể có một hoặc nhiều nhánh. Do vậy ta có thể dùng các lệnh *if* (khi số lựa chọn ít) và *case* (khi số lựa chọn lớn) ứng với mỗi chỏ rẽ nhánh này. Cả chương trình ta dùng *while* hoặc *repeat*.

Dưới đây là một ví dụ viết bằng Pascal thực hiện nhiệm vụ phân tích chương trình nguồn để tìm từ tố. Ta dùng một biến *state* để theo dõi các trạng thái chuyển đổi trên đồ thị chuyển và *start* là trạng thái bắt đầu. Chương trình bao gồm hai hàm chính làm các nhiệm vụ như sau:

Hàm *fail* dùng để trả lại đầu vào của một đồ thị chuyển tiếp theo. Nói cách khác, nó sẽ lần lượt trả lại các con số 0, 9, 12, 20, 25 là trạng thái đầu của các đồ thị chuyển ở trên. Nếu đến hết đồ thị cuối nghĩa là gặp lỗi thì nó gọi hàm *recover* để khôi phục lỗi.



Hình 3.4. Đồ thị chuyển cho các loại số

Hàm *nexttoken* với kiểu trả về là *token* (tù tố). Hàm *nextchar* dùng để đọc một ký hiệu vào từ chương trình nguồn. *blank*, *tab*, *newline* là các hằng số đã được định nghĩa trước của các ký hiệu vào như trắng, dấu tab, xuống dòng. *isalpha* và *isdigit* là các hàm dùng để kiểm tra ký tự vào là chữ hoặc là số (bạn có thể tự viết các hàm này hoặc khai thác thư viện).

Khi *state* đến được 11 thì có nghĩa là xâu vào là một cái tên. Tuy vậy, để kết luận từ tố của cái tên đó là gì, nó còn phải gọi hàm *install_id* để kiểm tra xem đó là từ khoá hay là một tên biến hoặc tên chương trình con. Nếu đó là biến hoặc tên chương trình con thì nó lại kiểm tra xem đã có trong bảng ký hiệu chưa. Nếu chưa có nó sẽ thêm vào bảng này. Hàm *gettoken* lại căn cứ vào thông tin từ *install_id* để trả về từ tố là mã của một từ khoá nào đó hoặc *tên*. Hàm *install_num* xử lý tương tự đối với các con số. Hàm *retract* xử lý các trạng thái có đánh dấu *, tức là các trạng thái xử lý quá một ký tự, thực chất nó cho lui con trỏ trên xâu vào một ký tự.

```
int state, start;

int fail()
{
    switch (start) {
        case 0: start = 9; break;
        case 9: start = 12; break;
        case 12: start = 20; break;
        case 20: start = 25; break;
        case 25: recover(); /* chương trình nguồn có lỗi */
        default:
            /*bản thân chương trình dịch có lỗi và xử lý ở đây */
    }
    return start;
}

token nexttoken()
{
    int exitflag;

    exitflag = 0;
    state = 0; start = 0;
    do {
        switch (state) {
```

```
case 0: c = nextchar(); /* c là ký hiệu nhìn trước */
    switch (c) {
        case blank:
        case tab:
        case newline:
            state = 0; break;
        case '<': state = 1; break;
        case '=': state = 5; break;
        case '>': state = 6; break;
        default: state = fail;
    }
    break;
```

/ case 1..8: xử lý tương tự ở đây */*

```
case 9: c = nextchar ();
    if (isalpha(c)) state = 10; else state = fail;
    break;
```

```
case 10: c = nextchar ();
    if (isalpha (c)) state = 10;
    else if (isdigit(c)) then state = 10;
    else state = 11;
    break;
```

```
case 11: retract(1); install_id();
    nexttoken = gettoken(); exitflag = 1;
    break;
```

/ case 12..24: xử lý tương tự ở đây */*

```
case 25: c = nextchar ();
    if (isdigit(c)) state = 26; else state = fail();
    break;
```

```
case 26:
    c = nextchar ();
    if (isdigit(c)) state = 26; else state = 27;
    break;
```

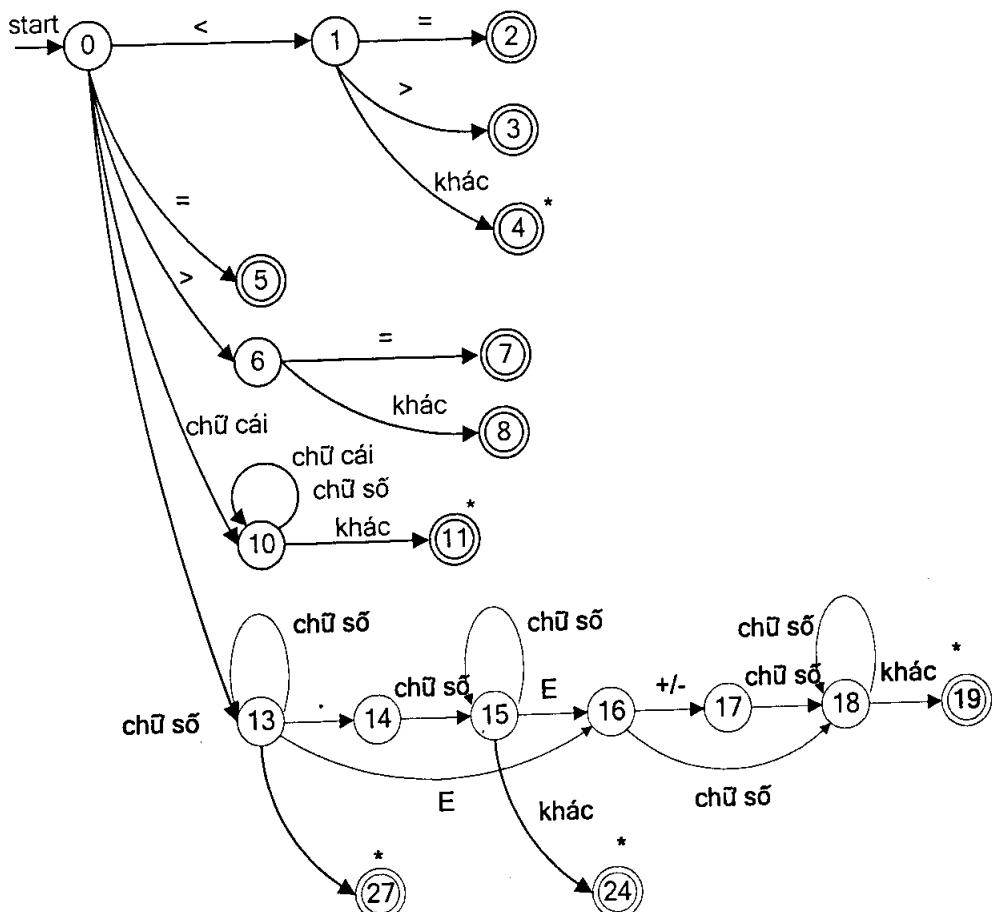
```
case 27: retract(1); install_num();
```

```

        nexttoken = sô_tutô; exitflag = 1;
        break;
    } /* switch */
    while (exitflag==0);
}

```

Ưu điểm của phương pháp này là dễ hiểu, dễ viết. Còn nhược điểm là ta đã gắn kết câu đố thị chuyển vào trong chương trình. Khi thay đổi đố thị thì phải viết lại chương trình. Do đó khó bảo trì.



Hình 3.5. Kết hợp các đồ thị chuyển

c. Lập bộ phân tích từ vựng điều khiển bằng bảng (mô phỏng ôtômát hữu hạn đơn định)

Ta để ý rằng, các đồ thị chuyển ở trên có tất cả 27 trạng thái khác nhau, ta có thể kết hợp lại thành một đồ thị chuyển duy nhất như hình 3.5. Nhờ kết

hợp này, ta có thể lược bỏ đi nhiều trạng thái bắt đầu và một số trạng thái trung gian - đồ thị chuyển tổng hợp chỉ còn có 20 trạng thái.

Nếu nhìn kĩ đồ thị chuyển này, ta thấy chúng hoạt động như sau: từ một trạng thái nào đó (gọi là trạng thái hiện tại) đọc một ký tự vào và tùy theo ký tự này là loại gì chương trình sẽ chuyển đến một trạng thái mới. Như vậy ta có thể nảy sinh ý tưởng dùng một cái bảng có số dòng tương ứng với số trạng thái, số cột tương ứng với loại ký tự đọc vào và mỗi ô chính là trạng thái mới. Và chương trình lúc này chỉ đơn thuần dõi theo các thay đổi của trạng thái nhờ tra bảng. Khi chương trình đến được các trạng thái vòng kép ta sẽ biết được xâu vào là gì.

Nếu trừ đi các trạng thái số 2, 3, 4, 5, 7, 8, 11, 19, 24, 27 là các trạng thái mà ở đó ta có thể kết luận xâu vào đã phân tích xong có từ tố là gì thì chỉ còn 10 trạng thái. Các ký tự vào từ chương trình nguồn được chia thành các loại: **chữ cái**, **chữ số**, E, ., <, =, >, +/- (tức là có 8 loại khác nhau). Do đó, ta lập một bảng 10×8 với nội dung như dưới đây, trong đó, các ô trống là các vị trí báo chương trình nguồn có lỗi và phải gọi phần khôi phục lỗi vào xử lý.

Trạng thái	Loại ký tự vào							
	<	=	>	chữ cái	chữ số	E	.	+/ -
0	1	5	6	10	13			
1	4	2	3	4	4	4	4	4
6	8	7	8	8	8	8	8	8
10	11	11	11	10	10	11	11	11
13	27	27	27	27	13	16	14	27
14					15			
15	24	24	24	24	15	16	24	24
16					18			17
17					18			
18	19	19	19	19	18	19	19	19

Còn dưới đây là chương trình hoạt động với bảng trên. Bảng này có tên là Lưu đồ và được gán các giá trị nhờ khai báo hằng số ban đầu:

```

int Lưuđò[10][8] = { ... };

LoạiTùTó: Tùtô()
{
    trạngthái: int;
    trạngthái = 0;
    do {
        ĐọcKýTự (NextChar);
        trạngthái = Lưuđò [trạngthái, LoạiKýTự (NextChar)];
        while (trạngthái != ' ') || (trạngthái in {2,3,4,5,7,8,11, 19,24,27});

        switch (trạngthái) {
            case 2: Tùtô = quanhệ; thuộctính = LE; break;
            case 3: Tùtô = quanhệ; thuộctính = NE; break;
            case 4: Tùtô = quanhệ; thuộctính = LT; break;
            case 5: Tùtô = quanhệ; thuộctính = EQ; break;
            case 7: Tùtô = quanhệ; thuộctính = GE; break;
            case 8: Tùtô = quanhệ; thuộctính = GT; break;
            case 11: Tùtô = tên;
            case 19: Tùtô = sóthựccmũ;
            case 24: Tùtô = sóthực;
            case 27: Tùtô = sốnguyên;
        default:
            Gọi hàm xử lý lỗi ở đây;
        }
    }
}

```

Ưu điểm của phương pháp trên là ta đã tách dữ liệu độc lập với chương trình, do đó rất dễ biến đổi mà không cần phải sửa lại chương trình. Còn nhược điểm là khó hiểu hơn và khó lập bảng.

Cơ cấu hoạt động của chương trình trên chính là cơ cấu hoạt động của một ôtômát hữu hạn đơn định (ta đã học ở chương trước).

III. XÁC ĐỊNH LỖI TRONG PHÂN PHÂN TÍCH TỪ VỰNG

Chỉ có rất ít lỗi được phát hiện trong lúc phân tích từ vựng, vì bộ phân tích từ vựng chỉ quan sát chương trình nguồn một cách rất cục bộ.

Ví dụ 3.6: Khi bộ phân tích từ vựng lần đầu tiên gặp xâu f trong biểu thức viết trong ngôn ngữ Pascal:

f $a = b$ then ...

thì bộ phân tích từ vựng không thể cho biết rằng *f* là từ viết sai của từ khoá *if* hoặc là một tên không khai báo. Nó sẽ nghiêm nhiên cho *f* là một tên đúng và trả lại một từ tố tên. Lỗi này chỉ được các phần phân tích sau (phân tích cú pháp) phát hiện ra.

Các lỗi mà bộ phân tích từ vựng phát hiện được là các lỗi về từ vị không đúng (ví dụ trong Pascal bạn viết nhầm lệnh gán thay cho := lại trở thành !=). Lỗi này xảy ra khi bộ phân tích không thể tìm được một luật từ tố nào đúng với phần còn lại của xâu.

Khi gặp lỗi này, cách xử lý đơn giản nhất là hệ thống sẽ ngừng hoạt động và báo lỗi cho người sử dụng.

Cách xử lý tốt hơn và hiệu quả hơn là bộ phân tích từ vựng sẽ ghi lại các lỗi này và cố gắng bỏ qua chúng để hệ thống có thể tiếp tục làm việc, nhằm phát hiện đồng thời thêm nhiều lỗi khác. Mặt khác, nó có thể tự sửa (hoặc gọi ý sửa lại chương trình nguồn).

Các cách khắc phục có thể có:

1. Xoá hoặc nhảy qua các ký tự mà bộ phân tích không tìm được từ tố;
2. Thêm một ký tự bị thiếu;
3. Thay một ký tự sai bằng một ký tự đúng;
4. Tráo hai ký tự đứng cạnh nhau.

IV. CÁC BƯỚC ĐỂ XÂY DỰNG MỘT BỘ PHÂN TÍCH TỪ VỰNG

Sau đây là các bước tuần tự nên tiến hành để xây dựng được một bộ phân tích từ vựng tốt, hoạt động chính xác và dễ cải tiến, bảo hành, bảo trì:

1. Sưu tầm tất cả các luật từ vựng, các luật này thường được mô tả bằng lời.
2. Vẽ đồ thị chuyển cho từng luật một. Trước đó có thể mô tả chúng bằng các biểu thức chính quy để tiện theo dõi và chỉnh sửa, và làm dễ cho việc dựng đồ thị.
3. Kết hợp các luật này thành một đồ thị chuyển duy nhất.
4. Chuyển đồ thị này thành bảng.
5. Thêm phần chương trình ở trên để thành bộ phân tích hoạt động được.
6. Thêm phần báo lỗi để thành bộ phân tích từ vựng hoàn chỉnh.

Bài tập

1. Phân tích các chương trình Pascal và C sau thành các từ tố và thuộc tính tương ứng.

a. Pascal

```
function max(i, j: integer): integer;
{ trả lại số lớn nhất trong hai số nguyên i và j }

begin
    if i > j then max := i
    else max := j;
end;
```

b. C

```
int max(int i, int j)
/* trả lại số lớn nhất trong hai số nguyên i và j */
{
    return i > j ? i : j;
}
```

Hãy cho biết có bao nhiêu từ tố được đưa ra và chia thành bao nhiêu loại?

2. Yêu cầu giống bài tập trên đối với các đoạn chương trình sau:

a. Pascal

```
var i, j;
begin
    for i = 0 to 100 do j = i;
    write('i:=' , i, 'j:=' , j);
end;
```

b. C

```
int i, j;
main (void
```

```

    {
        for (i0; i=100; i++)
            printf("i= %d;", i, " j=%d", j ==i);
    }

```

3. Mô tả các ngôn ngữ chỉ định bởi các biểu thức chính quy sau:

- a. $0(0|1)^*0$
- b. $((\epsilon|0)1^*)^*$

4. Viết biểu thức chính quy cho: tên, số nguyên, số thực, char, string... trong Pascal. Dụng đồ thị chuyển cho chúng. Sau đó, kết hợp chúng thành đồ thị chuyển duy nhất.

5. Dụng đồ thị chuyển cho các mô tả dưới đây:

a. Tất cả các xâu chữ cái có 6 nguyên âm *a, e, i, o, u, y* theo thứ tự. Ví dụ:

"abeiptowwrungfhy"

b. Tất cả các xâu số không có một số nào bị lặp.

c. Tất cả các xâu số có ít nhất một số nào đó bị lặp.

d. Tất cả các xâu chỉ bao gồm các chữ số 0 và 1, với tổng số chữ số 0 là chẵn, số chữ số 1 là lẻ.

e. Tất cả các xâu chỉ bao gồm các chữ số 0 và 1, nhưng không chứa xâu con 011.

f. Tất cả các xâu chỉ bao gồm các chữ số 0 và 1, nhưng không chứa liên tục các xâu con 011.

Bài tập thực hành

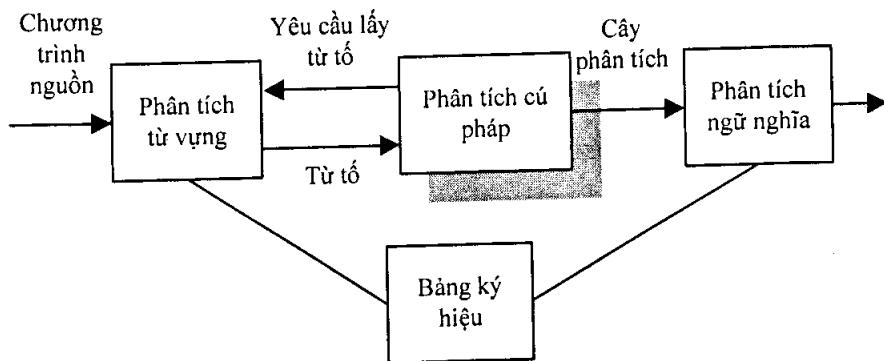
1. Hoàn chỉnh chương trình mục II.2.b và II.2.c.
2. Viết chương trình trong Pascal hoặc C dùng để phân tích từ vựng, tìm tất cả các từ tố và thuộc tính của nó theo đồ thị chuyển ra trong bài tập 3 và 4.
3. Thêm phần phát hiện, thông báo và khôi phục lỗi vào chương trình trên.
4. Viết chương trình đếm số từ trong một văn bản tiếng Anh hoặc tiếng Việt.

Chương 4

PHÂN TÍCH CÚ PHÁP VÀ CÁC PHƯƠNG PHÁP PHÂN TÍCH CƠ BẢN

I. MỤC ĐÍCH

Phân tích cú pháp tách chương trình nguồn (mà bây giờ chưa toàn các từ tố) thành các phần theo văn phạm và biểu diễn cấu trúc này bằng một cây (gọi là cây phân tích) hoặc theo một cấu trúc nào đó tương đương với cây.



Hình 4.1. Vị trí khối Phân tích cú pháp

II. HOẠT ĐỘNG CỦA BỘ PHÂN TÍCH

1. Văn phạm

Mọi ngôn ngữ lập trình đều có các luật mô tả các cấu trúc cú pháp. Một chương trình nguồn viết đúng phải tuân theo các luật mô tả này - tức là viết đúng văn phạm (hay đúng ngữ pháp). Ví dụ trong ngôn ngữ lập trình C hoặc Pascal, một chương trình có cấu tạo từ các khái niệm, các khái niệm lại từ các câu lệnh, các câu lệnh lại từ các biểu thức, các biểu thức lại từ các hạng thức...

Văn phạm đem lại nhiều ưu điểm hơn cho các nhà thiết kế ngôn ngữ lẫn người viết chương trình dịch với các lý do sau đây:

- Một văn phạm làm cho cú pháp của một ngôn ngữ lập trình trở nên chính xác, dễ hiểu.
- Từ các lớp rõ ràng của văn phạm có thể xây dựng tự động một bộ phân tích làm việc hiệu quả đối với một chương trình nguồn có cấu trúc cú pháp tốt. Một ưu điểm nữa là bộ phân tích này có thể chỉ ra các cấu trúc cú pháp nhập nhằng và các cấu trúc khó phân tích mà chúng có thể không được phát hiện trong các pha thiết kế trước của cả ngôn ngữ và chương trình dịch của nó.
- Một văn phạm thiết kế tốt làm cho cấu trúc của ngôn ngữ lập trình tốt và có thể chuyển đổi chính xác từ chương trình nguồn thành mã đích và phát hiện được các lỗi. Các công cụ chuyển các mô tả dựa trên cơ sở văn phạm sang chương trình làm việc là có thể xây dựng được.
- Ngôn ngữ lập trình luôn phát triển, luôn cần thêm các cấu trúc mới và thực hiện thêm các nhiệm vụ mới. Các cấu trúc này có thể thêm vào một ngôn ngữ dễ dàng khi đã có một cơ sở mô tả văn phạm của ngôn ngữ tốt.

2. Mô tả văn phạm

Văn phạm của một ngôn ngữ lập trình nói chung có cấu trúc có thể mô tả bằng văn phạm phi ngữ cảnh và biểu diễn theo ký pháp BNF hoặc đồ thị chuyển.

3. Các phương pháp phân tích

Như chương trước đã đề cập, cơ sở của phân tích cú pháp đối với lớp VPPNC là định lý **Bài toán thành viên với ngôn ngữ phi ngữ cảnh**. Người ta đã chứng minh được định lý này bằng cách đưa ra các giải thuật cài đặt trên thực tế, ví dụ như:

- Thuật toán phân tích Top-down.
- Thuật toán phân tích Bottom - Up.
- Thuật toán phân tích CYK (Coke-Younger-Kasami).
- Thuật toán phân tích Earley.

Việc phân tích một câu là xây dựng suy dẫn sinh ra nó, do đó ta sẽ dụng được cây suy dẫn. Thông thường trong các thuật toán phân tích, ta hay tiến hành từ một phía của câu, kiểm tra lần lượt các thành phần của câu đó cho đến hết (đa số từ trái sang phải).

Có nhiều chiến lược phân tích, ta có thể phân ra làm hai chiến lược chính:

- Chiến lược phân tích **top-down** (trên xuống): cho một văn phạm phi ngữ cảnh $G = (\Sigma, \Delta, P, S)$ và một câu cần phân tích w . Ta xuất phát từ điểm khởi đầu, nghĩa là từ S , áp dụng các suy dẫn trái, tiến từ trái qua phải thử tạo ra câu đưa vào phân tích w .
- Chiến lược phân tích **bottom-up** (dưới lên): Quá trình ngược lại với phân tích top-down, xuất phát từ chính câu vào phân tích w , bằng cách áp dụng thu gọn các suy dẫn phải, tiến hành từ trái qua phải để đi tới ký hiệu đầu S của văn phạm.

Điều kiện để các thuật toán trên là đúng như sau:

- Phân tích top-down (phân tích trái) là đúng khi và chỉ khi G không có đệ quy trái.
- Phân tích bottom-up (phân tích phải) là đúng khi và chỉ khi G không chứa suy dẫn $A \xrightarrow{*} A$ và không có sản xuất $B \rightarrow \epsilon$ (sản xuất rỗng).

Trong quá trình phân tích các phương pháp trên sẽ phải đối mặt với khó khăn khi gặp các luật có nhiều lựa chọn ở về phải như:

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \dots \mid \alpha_k, \quad k \geq 2$$

Vấn đề ở đây là cần thay thế A bởi α_i ($1 \leq i \leq k$) nào để phân tích được câu vào?

Giải quyết vấn đề này có nhiều phương pháp, song xét tổng quát có hai chiến lược:

1. Phân tích quay lui (backtrack): ta thử lần lượt các α_i ($1 \leq i \leq k$) để tìm α_i thích hợp. Rõ ràng cách này rất tốn thời gian.
2. Phân tích không quay lui (without-backtrack). Trong việc tìm sản xuất thích hợp, ta biết cách xác định sản xuất duy nhất thích hợp mà không cần phải thử các sản xuất khác. Các phương pháp phân tích như vậy gọi là *phân tích tất định* (deterministic parsing).

Ta sẽ nghiên cứu các phương pháp phân tích không quay lui ở chương sau. Còn dưới đây là hai phương pháp phân tích quay lui, có tên trùng với chiến lược phân tích mà chúng dùng.

Phân tích Top-down

Tên *Phân tích top-down* xuất phát từ ý tưởng cố gắng tạo ra một cây phân tích cho câu vào bắt đầu từ đỉnh và đi xuống cho đến lá.

Với một VPPNC cho trước, trước tiên chúng ta sẽ đánh dấu mọi lựa chọn trong từng sản xuất. Chẳng hạn, nếu các sản xuất dạng $S \rightarrow aSbS \mid aS \mid c$ thì $aSbS$ là lựa chọn thứ nhất, aS là lựa chọn thứ hai và c là lựa chọn cuối của sản xuất S .

Bây giờ ta sẽ xem bộ phân tích top-down hoạt động như thế nào.

Trước tiên, ta dùng một con trỏ chỉ đến xâu vào. Ký hiệu trên xâu vào do con trỏ chỉ đến gọi là *ký hiệu vào hiện tại*. Vị trí đầu tiên của con trỏ là ký hiệu bên trái nhất của xâu vào.

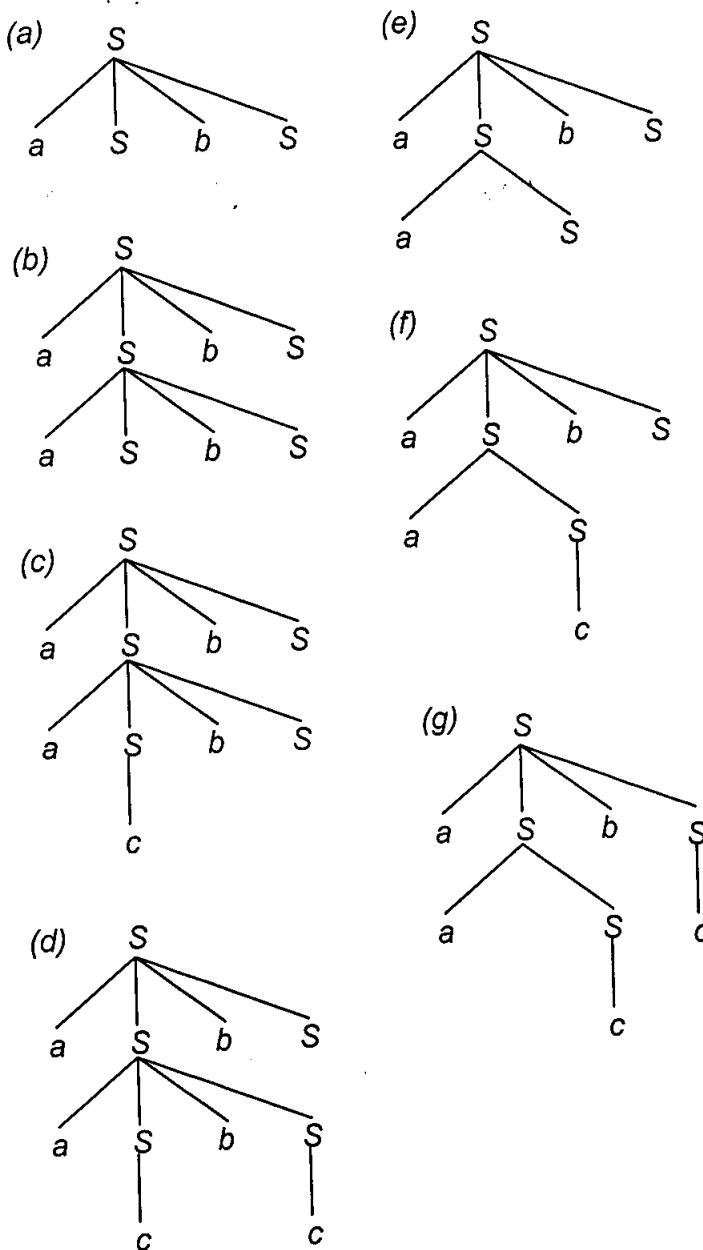
Ta bắt đầu công việc với một cây phân tích chỉ có một nút duy nhất là nút gốc S . S trở thành nút đang xét. Tiến hành các bước đệ quy sau:

1. Nếu nút đang xét là một nút ký hiệu không kết thúc A thì ta lấy lựa chọn đầu tiên. Ta ký hiệu là $X_1 \dots X_k$. Lại lấy nút X_1 làm nút đang xét. Trường hợp $k = 0$ (sản xuất ϵ) thì lấy nút ngay bên phải A làm nút đang xét.
2. Nếu nút đang xét là nút ký hiệu kết thúc a , thì so sánh nó với ký hiệu vào hiện tại. Nếu giống nhau thì lấy nút ngay bên trái a làm nút đang xét và chuyển con trỏ xâu vào sang bên phải một ký hiệu. Nếu a không giống thì quay lại nút do sản xuất trước tạo ra, điều chỉnh lại con trỏ xâu vào nếu cần thiết, sau đó ta lại thử lựa chọn tiếp theo. Nếu không còn lựa chọn nào nữa thì lại qua lại nút trước đó và cứ như vậy.

Ví dụ 4.1: Cho một VPPNC $G=(\Sigma, \Delta, P, S)$ với các sản xuất là $S \rightarrow aSbS \mid aS \mid c$. Xâu vào là $aacbc$. Ta tìm cây phân tích trái. Quá trình dựng cây được minh họa ở hình 4.2.

Trong ví dụ của chúng ta, cây suy diễn lúc đầu chỉ có một nút S . Ta áp dụng lựa chọn đầu tiên của sản xuất S , nghĩa là $S \rightarrow aSbS$ để mở rộng cây (thành cây hình a). Vì nút đầu tiên trong lựa chọn của S là a lại là một ký hiệu kết thúc và trùng khớp với ký hiệu đầu tiên của xâu vào, ta chuyển sang ký hiệu ngay bên phải của a làm nút đang xét là nút S và tăng con trỏ xâu vào lên một, tức là trỏ vào ký hiệu a thứ hai. Ta lại sử dụng lựa chọn đầu tiên trong sản xuất của S để mở rộng cây thành cây hình b . Với nút S dưới cùng này, ta áp dụng lựa chọn thứ nhất, thứ hai đều không được do các sản xuất này tạo ra ký hiệu kết thúc không trùng với ký hiệu vào hiện tại. Chỉ có lựa chọn thứ ba

của sản xuất S mới cho ký hiệu cuối thích hợp. Cứ như vậy, ta áp dụng lần lượt thuật toán đệ quy trên và nhận được các cây suy dẫn từ a đến g .



Hình 4.2. Quá trình dựng cây suy dẫn theo phương pháp top-down

Hình 4.3 là một ví dụ vui, bạn hãy tưởng tượng nếu ta cần tìm đường từ Hà Nội đến TP Hồ Chí Minh mà không biết đường, phải đi dò đường bằng phương pháp top-down.



Hình 4.3. Bản đồ giao thông Việt Nam. Nếu đi theo phương pháp Top - Down hoặc Bottom - Up thì để tìm đường từ Hà Nội vào Tp HCM người đi có thể phải đi “thử” ngược lên Lạng Sơn, Cao Bằng, Hà Giang... và rất nhiều đường nhánh khác cho đến khi tìm được đường đi đến đích.

Phân tích Bottom-up

Phương pháp phân tích bottom-up về tư tưởng là ngược lại với phương pháp top-down. Phương pháp này lại bắt đầu từ lá (tức là từ chính các ký hiệu vào) và cố gắng xây dựng thành cây bằng cách hướng lên gốc.

Phân tích bottom-up được gọi là phân tích gạt thu gọn (shift-reduce parsing). Quá trình phân tích này sử dụng bộ phân tích phải duyệt các suy diễn phải có thể được, tương ứng với xâu vào. Một hành động của bộ phân tích bao gồm việc quét xâu trên đỉnh của danh sách đầy xuồng để tìm xem có sản xuất nào có vé phải nào đó đúng với các ký hiệu trên đỉnh của danh

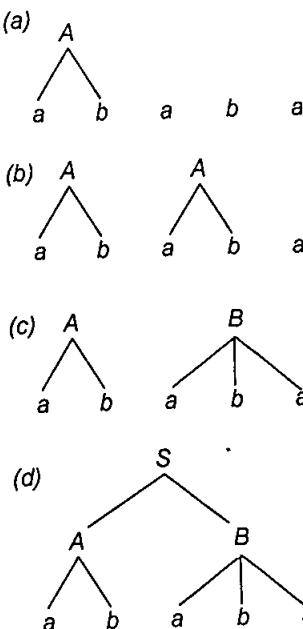
sách này hay không. Nếu có thì ta thực hiện một thu gọn bằng cách thay các ký hiệu này bằng các ký hiệu về trái của sản xuất đó. Nếu có nhiều lựa chọn thì ta đánh thứ tự chúng để thử lần lượt. Nếu không thể có một thu gọn nào thì ta gạt thêm một ký hiệu vào để lại tiếp tục như trên. Chúng ta sẽ luôn luôn cố gắng thực hiện các thu gọn trước khi phải gạt thêm. Nếu đi đến cuối xâu mà không có một thu gọn nào được thực hiện thì chúng ta quay lại bước chuyển dịch trước mà đã thực hiện thu gọn. Nếu còn một thu gọn khác thì ta lại thử tiếp thu gọn này.

Ví dụ 4.2: Một văn phạm có các sản xuất:

$$S \rightarrow AB, A \rightarrow ab, B \rightarrow aba$$

và một xâu vào là *ababa*. Quá trình dựng cây minh họa ở hình 4.4.

Trước tiên, ta gạt *a* vào danh sách đầy xuống. Không có một thu gọn nào thực hiện được. Ta gạt tiếp *b* vào danh sách đầy xuống. Bây giờ có thể thu gọn *ab* đang nằm trên đỉnh của danh sách đầy xuống bằng *A*. Vì *A* không thể tiếp tục thu gọn được, ta lại phải gạt *a* rồi *b* vào danh sách đầy xuống. Ta lại thu gọn được xâu *ab* thành *A*. Gạt nốt *a* vào danh sách đầy xuống và thấy rằng không thể thu gọn được tiếp. Ta quay lại bước chuyển dịch cuối đã thay *ab* thành *A* và bỏ bước thu gọn này. Bây giờ danh sách đầy xuống có xâu *Aaba*. Ta thu gọn *aba* bằng *B*. Cuối cùng thay *AB* bằng *S* và như vậy ta đã phân tích xong. Cây phân tích như hình 4.4.



Hình 4.4. Quá trình dựng cây theo phương pháp bottom-up.

Thời gian, bộ nhớ và độ phức tạp của phân tích top-down và bottom-up.

Người ta đã chứng minh rằng, tồn tại một hằng số c đối với thuật toán phân tích top-down hoặc bottom-up sao cho với xâu vào w có độ dài $n \geq 1$, thì thuật toán không cần quá c^n phép tính. Nói cách khác, độ phức tạp của thuật toán có dạng hàm số mũ.

Để hình dung tốc độ tính toán bằng các phương pháp trên như thế nào, ta giả sử $c = 3$. Chú ý là con số n có thể lớn đến vài chục. Ví dụ một câu lệnh có thể gấp trong Pascal như sau:

```
if (a+b+c) > (d-e) and (3*d-2) < (2*n) then a:= (a+b+c+d+e) / 5 - n;
```

sẽ được phân tích thành 47 từ tố ($n = 47$).

Chỉ với $n = 20$ ta đã có $3^n = 3^{20}$, là trên 3.10^9 (trên 3 tỷ). Nếu máy tính có khả năng thực hiện được 1000 phép xét từ tố mỗi giây, thì tổng thời gian cần để phân tích mỗi một câu lệnh này là $3 \text{ tỷ} / 1000 / 60 / 60 > 80$ giờ. Nếu máy tính có khả năng xử lý gấp 10 lần (10.000 từ tố mỗi giây) thì cũng cần đến 8 giờ để xử lý xong câu lệnh đó.

Điều này chứng tỏ là các phương pháp phân tích trên quá chậm, cần phải có những phương pháp khác nhanh hơn.

4. Phát hiện lỗi

Nếu một chương trình dịch chỉ phải dịch các chương trình máy tính viết đúng thì thiết kế và hoạt động của nó sẽ rất đơn giản. Nhưng những người lập trình lại rất thường xuyên tạo ra những chương trình viết sai. Một chương trình dịch tốt phải phát hiện, định vị, phân loại được tất cả các lỗi để giúp đỡ người viết. Phần lớn các ngôn ngữ lập trình không cho biết một chương trình dịch phải xử lý lỗi như thế nào. Người thiết kế chương trình dịch phải tự giải quyết vấn đề này. Ta phải thiết kế các chương trình dịch xử lý lỗi một cách đúng đắn ngay từ khi bắt đầu hoạt động.

Giai đoạn phân tích cú pháp phát hiện và khắc phục được khá nhiều lỗi. Một trong các lý do là nhiều loại lỗi khác nhau đồng thời cũng là lỗi cú pháp. Ví dụ lỗi do chuỗi từ tố từ bộ phân tích từ vựng không theo thứ tự của luật văn phạm của ngôn ngữ lập trình. Một lý do khác là nhờ sự chính xác của các phương pháp phân tích, chúng có thể phát hiện lỗi cú pháp rất hiệu quả. Tất nhiên việc phát hiện chính xác lỗi trong thời gian dịch là một nhiệm vụ khó khăn cần nhiều nỗ lực.

Trong phần này, chúng ta sẽ xem xét một vài kỹ thuật cơ bản để phát hiện và phục hồi lỗi cú pháp. Bộ bắt lỗi trong phần phân tích cú pháp có một số mục đích sau:

- Phát hiện, chỉ ra vị trí và mô tả chính xác, rõ ràng các lỗi;
- Phục hồi quá trình phân tích sau khi gặp lỗi đủ nhanh để có thể phát hiện được các lỗi tiếp theo;
- Không làm giảm đáng kể thời gian xử lý các chương trình viết đúng.

Đáp ứng các yêu cầu trên là một thách thức lớn. May mắn là các lỗi thông thường cũng là các lỗi đơn giản và một cơ chế bắt lỗi không phức tạp lắm cũng thường là đủ dùng. Trong một số trường hợp, một lỗi có thể xảy ra rất lâu trước khi nó được phát hiện, và khó có thể chỉ được chính xác đặc điểm của lỗi đó. Trong các trường hợp phức tạp, bộ bắt lỗi dành phải phán đoán xem người lập trình suy nghĩ gì khi viết chương trình.

Một số phương pháp phân tích, ví dụ như các phương pháp LL và LR, phát hiện lỗi rất nhanh. Chúng có cơ chế *tiền tố* nghĩa là chúng phát hiện được lỗi ngay khi chúng phát hiện tiền tố của một đầu vào không đúng với bất cứ xâu nào trong ngôn ngữ mà không cần phải phân tích hết.

Người ta đã tính được rằng các lỗi không phải xuất hiện thường xuyên. 60% số chương trình khi dịch đúng về mặt cú pháp và ngữ nghĩa (có thể có lỗi ở phần khác). Còn khi có lỗi, gần 80% số câu lệnh đó có một lỗi, 13% có hai lỗi. Cuối cùng, ta lại thấy phần lớn lỗi lại là đơn giản: 90% lỗi là lỗi từ tố đơn.

Lỗi lại có thể phân loại đơn giản: 60% là lỗi dấu chấm, chấm phẩy, phẩy (nếu trong Pascal thì đa số lỗi loại này là dấu chấm phẩy), 20% là lỗi phép toán, 15% lỗi từ khóa, và phần còn lại 5% là lỗi loại khác.

Trong Pascal: Một lý do số lỗi sai nhiều do dùng sai dấu chấm phẩy (;) là nó được dùng rất nhiều để tách câu lệnh. Một lỗi khác thường gặp là thiếu dấu hai chấm (:) trong phép gán $t :=$. Việc viết sai từ khóa ít gấp hơn. Nhiều chương trình dịch Pascal có thể phát hiện và khắc phục dễ dàng các lỗi trên.

Một số lỗi khác thì khó khắc phục hơn. Ví dụ như lỗi thiếu mất từ khóa *begin* hay *end*. Phần lớn các chương trình dịch không làm tốt việc phát hiện và khắc phục các lỗi này.

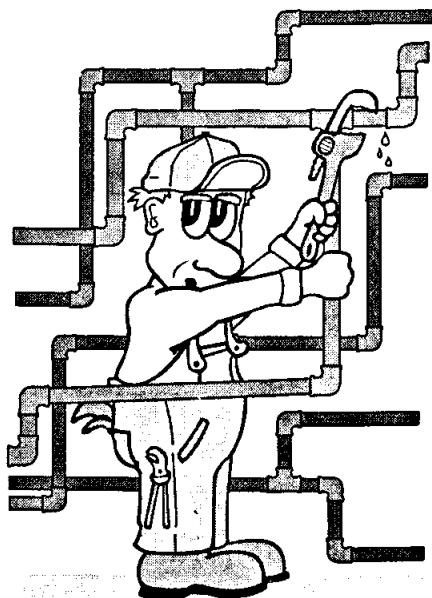
Khi bộ phân tích phát hiện lỗi, thông thường nó sẽ in ra dòng gặp lỗi. Nếu có thể nó sẽ in thêm rằng đó là loại lỗi gì, ví dụ như: *lỗi thiếu dấu hai chấm*.

Các chiến lược phục hồi lỗi

Có nhiều chiến lược mà một bộ phân tích có thể dùng để phục hồi quá trình phân tích sau khi gặp một lỗi cú pháp. Mặc dù không có một chiến lược nào tổng quát và hoàn hảo, có một số phương pháp được dùng rộng rãi. Sau đây là một vài chiến lược:

Phục hồi kiểu trùng phẹt: Đây là phương pháp đơn giản nhất và được áp dụng trong đa số các bộ phân tích. Mỗi khi phát hiện lỗi, bộ phân tích lại bỏ qua một hoặc một số ký hiệu vào mà không kiểm tra cho đến khi nó gặp một ký hiệu trong tập từ tố đồng bộ. Các từ tố đồng bộ này thường đã được xác định trước, ví dụ như *end* và dấu *chấm phẩy*. Khi gặp lỗi, chương trình sẽ bỏ qua các ký hiệu cho đến khi nó gặp từ khóa *end* hoặc dấu chấm phẩy. Dĩ nhiên là người thiết kế chương trình dịch phải tự chọn các từ tố đồng bộ, thích hợp với từng ngôn ngữ lập trình. Phương pháp này có ưu điểm: đơn giản; không sợ rơi vào vòng lặp vô hạn và rất hiệu quả khi gặp một câu lệnh có nhiều lỗi.

Khôi phục cụm từ: Mỗi khi phát hiện lỗi, bộ phân tích cố gắng phân tích cục bộ phần còn lại của câu lệnh. Nó có thể phải thay thế phần đầu của phần còn lại này bằng một xâu nào đó cho phép bộ phân tích làm việc tiếp. Ví dụ, nó có thể thay dấu phẩy bằng dấu chấm phẩy, xóa dấu chấm phẩy thừa hoặc thêm một dấu chấm phẩy bị thiếu. Những việc này là do người thiết kế chương trình dịch nghĩ ra.



Phân tích cú pháp là phần khó nhất của chương trình dịch. Bạn cần phải làm việc kiên nhẫn, tỉ mỉ và cẩn thận.

Sản xuất lỗi: Nếu bạn có hiểu biết tốt về các lỗi phổ biến có thể sẽ gặp, bạn có thể gia cố văn phạm của ngôn ngữ này tại các luật hay sinh ra các cấu trúc lỗi. Bạn có thể dùng văn phạm được gia cố này để xây dựng bộ phân tích. Nếu bộ phân tích dùng một luật lỗi, bạn có thể tạo ra các chuẩn đoán lỗi tương ứng để chỉ ra các cấu trúc lỗi có thể phát hiện ở đầu vào.

Chỉnh lý toàn cục: Trong ý thức, chúng ta mong muốn có một chương trình dịch thay đổi càng ít càng tốt quá trình xử lý một chuỗi vào. Có các thuật toán cho phép chọn được chuỗi thay đổi nhỏ nhất để đạt được sự sửa

toàn thể là ít nhất. Rất có thể là một chương trình sau khi sửa xong rất gần với chương trình bị sai nhưng lại không đúng với ý định của người lập trình. Điều không tốt nữa là các thuật toán này lại rất tốn thời gian và bộ nhớ, do vậy chúng chỉ được quan tâm về phương diện lý thuyết.

Ngoài ra, người ta lại có những cách bắt lỗi cụ thể hơn phụ thuộc vào đặc thù của từng phương pháp phân tích khác nhau (ta cũng sẽ học một số loại ở chương sau).

Bài tập

1. Mục đích và hoạt động của bộ phân tích cú pháp?
2. Đầu vào và đầu ra của phân tích cú pháp. So sánh với phân tích từ vựng?
3. Hãy trình bày các phương pháp phân tích cơ bản.
4. Hãy trình bày các phương pháp xử lý lỗi của phần phân tích cú pháp.
5. Tại sao cây phân tích nhận được từ phép phân tích top-down lại được gọi là cây phân tích trái?
6. Tại sao cây phân tích nhận được từ phép phân tích bottom-up lại được gọi là cây phân tích phải?

Bài tập thực hành

1. Bạn hãy tự thống kê các lỗi hay mắc (loại lỗi, số lỗi) khi lập trình bằng ngôn ngữ thường dùng. Ghi lại và lý giải.
2. Cài đặt các phương pháp phân tích top-down, bottom-up và CYK (CYK sẽ được trình bày trong phần tham khảo dưới). Có thể tự nghĩ ra các luật dùng để thử nghiệm hoặc dựa vào một ngôn ngữ nào đó (ngôn ngữ tự nhiên hoặc lập trình đều được). Thủ nghiệm và so sánh kết quả với những xâu vào có độ dài khác nhau. Phương pháp nào nhanh nhất?
3. Bạn hãy lấy một chương trình bất kì trong ngôn ngữ thường dùng, hãy tính thời gian để phân tích xong nó bằng phương pháp top-down hoặc bottom-up. Giả sử máy bạn có khả năng xét 10.000 từ tố một giây, $c = 3$.

Tham khảo

Phương pháp phân tích bảng CYK (Cocke-Younger-Kasami)

Giải thuật CYK làm việc với tất cả các VPPNC. Giải thuật này chỉ đòi hỏi thời gian phân tích $O(n^3)$ (n là độ dài của xâu vào cần phân tích). Nếu văn phạm không nhập nhằng thì chỉ cần $O(n^2)$ thời gian. Như vậy, để phân tích xâu vào với độ dài 20 từ tố bằng máy tính có khả năng xét 1000 từ tố /giây thì chỉ cần đến 8 giây (và chỉ nửa giây với văn phạm không nhập nhằng) thay cho 80 giờ nếu dùng phương pháp Top-down hoặc Bottom - up.

Điều kiện của thuật toán là văn phạm phi ngữ cảnh $G=(\Sigma, \Delta, P, S)$ ở dạng chuẩn Chomsky (CNF) và không có ϵ -sản xuất (sản xuất có dạng $A \rightarrow \epsilon$) và các sản xuất vô ích.

Dạng chuẩn Chomsky

Một VPPNC ở dạng chuẩn Chomsky là văn phạm trong đó mọi sản xuất đều có dạng $A \rightarrow BC$ hoặc $A \rightarrow a$.

Người ta đã chứng minh rằng, mọi VPPNC không chứa ϵ trong các sản xuất đều có thể chuyển đổi về dạng chuẩn Chomsky theo phương pháp sau:

Cứ với mỗi sản xuất chưa ở dạng chuẩn dạng $A \rightarrow B_1B_2\dots B_m$ trong P' với $m \geq 3$, ta đưa thêm các biến mới D_1, D_2, \dots, D_{m-2} và thay sản xuất trên bởi tập các sản xuất:

$$\{A \rightarrow B_1D_1, D_1 \rightarrow B_2D_2, \dots, D_{m-3} \rightarrow B_{m-2}D_{m-2}, D_{m-2} \rightarrow B_{m-1}B_m\}$$

Như thế ta thu được một tập các biến mới Δ'' và một tập các sản xuất mới P'' . Cho $G' = (\Sigma, \Delta'', P'', S)$. G' là VPPNC và ở dạng chuẩn Chomsky.

Giải thuật CYK

Để áp dụng thuật toán CYK, ta phải tạo ra một bảng dữ liệu hình tam giác (như hình vẽ ở dưới) có mỗi chiều là n (n là độ dài của xâu vào cần phân tích) và thực hiện thuật toán sau để điền vào các ô của bảng.

begin

for $i := 1$ **to** n **do**

$\Delta_{ii} := \{A \mid A \rightarrow a \text{ là một sản xuất và } a \text{ là ký hiệu thứ } i \text{ trong } w\};$

for $j := 2$ **to** n **do**

for $i := 1$ **to** $n-j+1$ **do** **begin**

$\Delta_{ij} := \emptyset;$

for $k := 1$ **to** $j-1$ **do**

$\Delta_{ij} := \Delta_{ij} \cup \{A \mid A \rightarrow BC \text{ là một sản xuất, } B \in \Delta_{ik} \text{ và } C \in \Delta_{i+k, j-k}\}$

end;

end;

Ví dụ: Xét văn phạm dạng chuẩn Chomsky:

$$S \rightarrow AB \mid BC$$

$$A \rightarrow BA \mid a$$

$$B \rightarrow CC \mid b$$

$$C \rightarrow AB \mid a$$

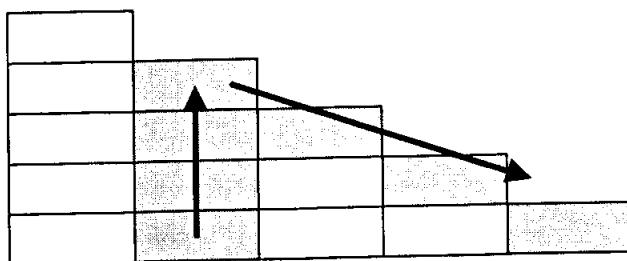
và xâu vào là $baaba$. Bảng các Δ_{ij} cho ở hình dưới. Dòng đầu tiên trong bảng được cho bởi các bước (1) và (2) trong giải thuật. Vì $w_{11} = w_{41} = b$, nên $\Delta_{11} = \Delta_{41} = \{B\}$. Còn $w_{21} = w_{31} = w_{51} = a$, suy ra $\Delta_{21} = \Delta_{31} = \Delta_{51} = \{A, C\}$

	1	2	3	4	5
5	S, A, C				
4	\emptyset	S, A, C			
3	\emptyset	B	B'		
2	S, A	B	S, C	S, A	
1	B	A, C	A, C	B	A, C

$i \longrightarrow$

Bảng các Δ_{ij}

Để tính Δ_{ij} với $j > 1$, ta phải thực hiện vòng lặp FOR ở các dòng (6) và (7). Ta phải đổi chiều Δ_{ik} với $\Delta_{i+k,j-k}$ với $k = 1, 2, \dots, j-1$, để tìm biến D trong Δ_{ik} và biến E trong $\Delta_{i+k,j-k}$ sao cho DE là vẽ phải của một hay nhiều sản xuất. Các vẽ trái của những sản xuất đó được đưa vào trong Δ_{ij} . Quá trình đổi chiều đó diễn ra bằng cách đi lên dần trên cột i , đồng thời trượt dần xuống theo đường chéo qua Δ_{ij} về phía phải như các mũi tên vẽ ở hình sau:



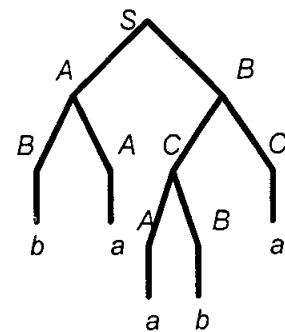
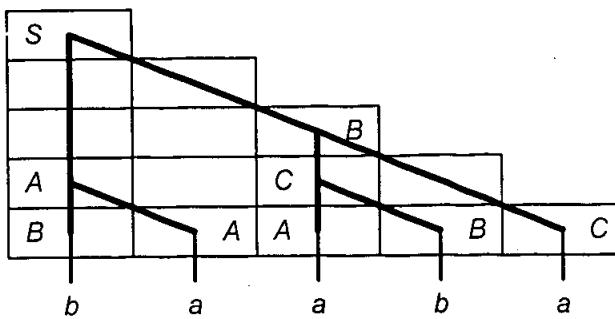
Quá trình tính Δ_{ij}

Chẳng hạn để tính Δ_{24} . Đầu tiên đổi chiều $\Delta_{21} = \{A, C\}$ với $\Delta_{33} = \{B\}$. Ta có $\Delta_{21}\Delta_{33} = \{AB, CB\}$. Vì có các sản xuất $S \rightarrow AB$ và $C \rightarrow AB$, nên S và C được đưa vào Δ_{24} . Tiếp đến lại xét $\Delta_{22}\Delta_{42} = \{B\}\{S, A\} = \{BS, BA\}$. Vì có sản xuất $A \rightarrow BA$, vậy đưa

thêm A và Δ_{24} . Cuối cùng xét $\Delta_{23}\Delta_{51} = \{B\}\{A,C\} = \{BA,BC\}$ gặp lại các vé phải đã xét, vậy không thêm gì vào Δ_{24} . Vậy $\Delta_{24} = \{S,AC\}$. Cuối cùng vì $S \rightarrow \Delta_{15}$, vậy xâu $baaba$ là thuộc ngôn ngữ sinh ra bởi văn phạm đã cho.

Nếu S có ở trong ô trên cùng thì ta kết luận là xâu vào phân tích thành công và có thể dựng cây phân tích cho nó. Số lượng cây phân tích chính bằng số lượng S trong ô này (nếu có trên một cây thì xâu vào là nhập nhằng).

Dưới đây là ví dụ dựng cây phân tích dựa vào bảng. Chú ý rằng, chỉ trừ những nút sinh trực tiếp ra lá, các nút trong khác đều có hai con. Do đó, cây nhận được nếu bỏ lá chính là cây nhị phân.



Cây phân tích suy từ bảng và vẽ lại

Một phương pháp phân tích bảng khác là phương pháp Earley cũng có thời gian phân tích hiệu quả dạng $O(n^3)$. Đặc điểm của phương pháp này là có thể áp dụng cho VPPNC viết ở dạng thông thường, không cần phải chuyển sang dạng Chomsky chuẩn nên tiện hơn. Phương pháp này tương đối dài và phức tạp nên không được trình bày trong giáo trình này.

Chương 5

CÁC PHƯƠNG PHÁP PHÂN TÍCH HIỆU QUẢ

Trong chương trước, ta đã đề cập đến các bộ phân tích theo các phương pháp *top-down*, *bottom-up*, ... Các bộ phân tích này làm việc không hiệu quả lắm vì chúng cần thời gian phân tích rất lớn, có thể đến c^n , trong đó n là chiều dài xâu vào cần phân tích. Rõ ràng khi chiều dài xâu vào tăng, thời gian cần để phân tích tăng rất nhanh. Các phương pháp phân tích bảng như CYK và Earley tuy thời gian phân tích chỉ còn c^3 nhưng vẫn còn quá chậm.

Trong chương này, ta sẽ đề cập đến các bộ phân tích làm việc hiệu quả hơn. Đó là các bộ phân tích chỉ cần thời gian phân tích là cn để xử lý các xâu có độ dài n . Nói cách khác, *thời gian cần thiết để phân tích tỷ lệ tuyến tính với độ dài xâu vào* ($O(n)$). Tất nhiên phải trả giá cho tính hiệu quả đó: chỉ làm việc được với một số lớp con của văn phạm phi ngữ cảnh (tuy vậy là khá đủ cho đa số ngôn ngữ lập trình). Các lớp này luôn có khả năng phân tích được.

Các thuật toán phân tích có đặc điểm chung là xâu vào được quét từ trái sang phải và quá trình phân tích là hoàn toàn xác định, do đó ta còn gọi là các bộ *phân tích xác định*. Ta chỉ đơn thuần giới hạn trong lớp văn phạm phi ngữ cảnh nên luôn có khả năng xây dựng được bộ phân tích trái hoặc phải xác định cho văn phạm này.

Giả sử ta có các sản xuất nhiều lựa chọn dạng $A \rightarrow \alpha_1 | \alpha_2 | \alpha_3 | \alpha_4 \dots$. Trong chiến lược phân tích top-down, để biết phải dùng lựa chọn nào ta có thể thử lần lượt tất cả các lựa chọn cho đến khi nhận được lựa chọn thích hợp. Việc thử lần lượt này được gọi là phân tích có quay lui - là lý do làm việc phân tích chậm đi nhiều.

Ý tưởng cơ bản của phương pháp phân tích tất định là ta lựa chọn cẩn thận các luật văn phạm như thế nào đó để tránh việc quay lui rất mất thời gian. Tức là phải có một cách nào đó xác định được ngay lựa chọn đúng mà không phải thử các lựa chọn khác. *Thông tin để xác định lựa chọn dựa vào những gì đã biết: trạng thái và ký hiệu kết thúc hiện thời.*

Ví dụ, một câu lệnh của Pascal có thể bao gồm: câu lệnh gán, khối bắt đầu bằng *begin*, các câu lệnh *if*, *while*, ... Nếu bộ phân tích gấp từ tố đầu tiên là từ khoá *if*, rõ ràng nó xác định được ngay đây là câu lệnh *if* và chắc chắn không cần phải kiểm tra xem nó có phải là câu lệnh gán hay *while* hay không.

Nhược điểm của các lớp văn phạm này là bé hơn lớp VPPNC (tức là lớp con của VPPNC) nên ngôn ngữ của chúng bị giới hạn nhiều hơn. Rõ ràng ta khó dùng chúng để tả cảnh, kể chuyện như vẫn dùng với tiếng Việt, tiếng Anh.

Các lớp văn phạm được đề cập đến trong chương này bao gồm:

1. Văn phạm LL (*k*) - văn phạm cho phép xây dựng các bộ phân tích làm việc tất định nếu bộ phân tích này được phép nhìn *k* ký hiệu vào nằm ngay ở bên phải của vị trí vào hiện thời.
2. Văn phạm LR (*k*) - văn phạm cho phép xây dựng các bộ phân tích làm việc tất định nếu bộ phân tích này được phép nhìn *k* ký hiệu vào nằm vượt quá vị trí vào hiện thời.

Trong cả hai phương pháp trên, các luật văn phạm đều được tính toán cài đặt trong các bảng phân tích. Để phân tích những văn phạm khác nhau ta chỉ cần sửa lại nội dung các bảng này mà không cần sửa chương trình. Do đó, sau khi hiểu xong cơ chế hoạt động của các phương pháp đó, ta sẽ bỏ nhiều công sức để tìm hiểu các cách nhằm đưa các luật văn phạm vào bảng (từ đó, ta cũng có thể xây dựng những chương trình lập bảng tự động).

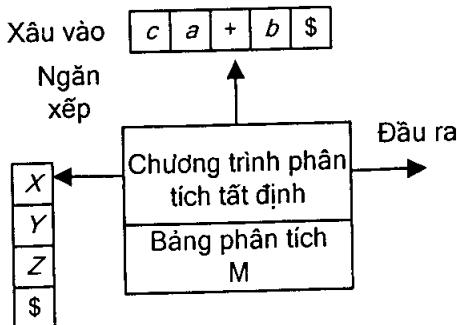
Các bộ phân tích có thể dùng để xây dựng nên toàn bộ cây phân tích với đỉnh là *S* hoặc chỉ một cây con nào đó (có đỉnh khác *S*). Để tổng quát, các ví dụ dưới đây đều giả thiết ta phân tích để dựng một cây con có đỉnh là *E* (*E* là chữ cái đầu của từ expression - biểu thức, một dạng cây con rất gấp).

I. PHÂN TÍCH LL

1. Mô tả

Cơ sở của phân tích LL (*k*) dựa trên phương pháp phân tích top-down và máy ôtômat đầy xuống. Mô hình của bộ phân tích này có:

- Một xâu vào chứa xâu cần phân tích, kết thúc bằng một dấu kết thúc ký hiệu là \$;
- Một ngăn xếp (hay một danh sách đầy xuồng) chứa chuỗi các ký hiệu văn phạm và đáy của nó cũng được đánh dấu bằng \$;
- Một bộ phận điều khiển chứa bảng phân tích là một mảng hai chiều $M[A, a]$ với A là ký hiệu không kết thúc, a là ký hiệu kết thúc. Các phần tử của mảng M xác định bởi hàng là ký hiệu không kết thúc A và cột là ký hiệu kết thúc a , chính là các luật sản xuất có vé trái là A , vé phải là các ký hiệu có thể suy dẫn ra được một xâu mà xâu này bắt đầu bằng a ;
- Một đầu ra để thông báo các luật sản xuất đã dùng hoặc tình trạng phân tích thành công hay không.



Hình 5.1. Mô hình của bộ phân tích tất định.

Bộ phân tích được điều khiển bằng một chương trình hoạt động như dưới đây. Hoạt động của bộ điều khiển được xác định từ ký hiệu X trên đỉnh của ngăn xếp, và ký hiệu vào hiện tại a . Các khả năng xảy ra như sau:

1. Nếu $X = a = \$$, bộ phân tích dừng và tuyên bố phân tích thành công.
Như vậy, lúc này cả xâu vào lẫn ngăn xếp đều trở thành rỗng.
2. Nếu $X = a \neq \$$, bộ phân tích lấy X ra khỏi ngăn xếp và dịch con trỏ vào sang ký hiệu vào tiếp theo. Bộ phân tích đã khai triển đến lá và lá này khớp với tình trạng xâu vào.
3. Nếu X là một ký hiệu không kết thúc, chương trình điều khiển sẽ xét vị trí $M[X, a]$ của bảng phân tích. Vị trí này có thể là một sản xuất X của văn phạm hoặc là một vị trí đánh dấu là lỗi. Nếu $M[X, a] = \{X \rightarrow UVW\}$ thì bộ phân tích thay X đang nằm trên đỉnh ngăn xếp bằng UVW (U sẽ nằm trên đỉnh). Nó đã thực hiện được một phép mở rộng

cây. Lúc này, bộ phân tích có thể đưa ra sản xuất vừa được sử dụng. Nếu $M[X, a] = \text{lỗi}$ (vị trí lỗi), bộ phân tích sẽ gọi hàm khôi phục lỗi.

Ta mô tả thuật toán phân tích trên như dưới đây.

Thuật toán 5.1. Phân tích LL không đệ quy.

Vào: Một xâu w và một bảng phân tích M của văn phạm G .

Ra: Đưa ra suy diễn trái nhất của w nếu $w \in L(G)$, nếu không sẽ là một thông báo lỗi.

Thuật toán:

Ở trạng thái khởi đầu, ngăn xếp được đặt các ký hiệu $\$E$ (E là đỉnh cây phân tích), còn xâu vào là $w\$$.

Đặt con trỏ ip chỉ đến ký tự đầu tiên của xâu $w \$$

do {

 Giả sử X là ký hiệu đỉnh của ngăn xếp và a là ký hiệu vào tiếp theo;

if (X là một ký hiệu kết thúc hoặc $\$$) {
 if ($X == a$)

 pop X từ đỉnh ngăn xếp và loại bỏ a khỏi xâu vào;

else

 ERROR();

 } **else** /* X không phải là ký hiệu kết thúc */

if ($M[X, a] == X \rightarrow Y_1 Y_2 \dots Y_k$) {

 pop X từ ngăn xếp;

 push $Y_k Y_{k-1} \dots Y_1$ vào ngăn xếp, với Y_1 ở đỉnh;
 đưa ra sản xuất $X \rightarrow Y_1 Y_2 \dots Y_k$;

 } **else**

 ERROR();

 } **while** ($X != \$$); /* ngăn xếp rỗng */

Ta có thể coi bảng M như một cái cẩm nang tìm đường. Để tìm đường từ Hà Nội đến Tp HCM ta sẽ tra bảng 5.1 tại dòng Hà Nội (nơi xuất phát) và cột Tp HCM (đích đi đến). Nếu ở ô đó lại là Hà Nội \rightarrow Nam Định thì ta hiểu phải chọn đường đi từ Hà Nội đến Nam Định đã. Khi đến Nam Định rồi, ta lại tiếp tục tra cẩm nang ở điểm hiện tại Nam Định, vẫn đích đến TpHCM và lại nhận được đường phải đi. Cứ như thế, ta đi đến đích. Toàn bộ quá trình này bạn không cần phải đi thử các đường khác như ngược lên Lạng Sơn hay ra Hải Phòng để xem đường đó có đến được Tp HCM không.

Xuất phát	Điểm đến			
	Tp HCM	Hải Phòng	Nam Định	...
Hà Nội	Hà Nội - Nam Định	Hà Nội - Hải Dương	Hà Nội - Nam Định	...
Nam Định	Nam Định - Thanh Hoá	Nam Định - Hà Nội		...
Thanh Hoá	Thanh Hoá - Vinh	Thanh Hoá - Nam Định	Thanh Hoá - Nam Định	...
...
Phan Thiết	Phan Thiết - Tp HCM	Phan Thiết - Phan Rang	Phan Thiết - Phan Rang	...
TP HCM		Tp HCM - Phan Thiết	Tp HCM - Phan Thiết	...

Bảng 5.1. "Cẩm nang tìm đường"

Ví dụ 5.1: Cho bảng phân tích $M[A, a]$ (bảng 5.2).

Ký hiệu cú pháp	Ký hiệu vào					
	a	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow a$			$F \rightarrow (E)$		

Bảng 5.2. Bảng phân tích M.

Ta sẽ dùng thuật toán 5.1 để phân tích xâu vào $a+a*a$. Ta sẽ thấy rằng, suy diễn nhận được là suy diễn trái nhất, quá trình phân tích là top - down như bảng 5.3.

Để tạo nên "cẩm nang tìm đường" như trên, ta cần phải biết những thông tin như: từ Hà Nội và những thành phố khác có thể đến được những thành phố nào trong cả nước (bảng đường bộ)... Đó là lý do của tính FIRST và FOLLOW ở phần sau.

Ngăn xếp	Đầu vào	Đầu ra
\$E	$a+a^*a\$$	
\$E'T	$a+a^*a\$$	$E \rightarrow TE'$
\$E'T'F	$a+a^*a\$$	$T \rightarrow FT'$
\$E'T'a	$a+a^*a\$$	$F \rightarrow a$
\$E'T'	$+a^*a\$$	
\$E'	$+a^*a\$$	$T' \rightarrow \epsilon$
\$E'T+	$+a^*a\$$	$E' \rightarrow +TE'$
\$E'T	$a^*a\$$	
\$E'T'F	$a^*a\$$	$T \rightarrow FT'$
\$E'T'a	$a^*a\$$	$F \rightarrow a$
\$E'T'	$*a\$$	
\$E'T'F*	$*a\$$	$T' \rightarrow *FT'$
\$E'T'F	$a\$$	
\$E'T'a	$a\$$	$F \rightarrow a$
\$E'T'	$\$$	
\$E'	$\$$	$T' \rightarrow \epsilon$
\$	$\$$	$E' \rightarrow \epsilon$

Bảng 5.3. Quá trình phân tích.

2. FIRST và FOLLOW¹

Bây giờ ta sẽ đề cập đến việc xây dựng bảng phân tích nhờ hai hàm liên quan đến văn phạm G : $FIRST$ và $FOLLOW$. Các hàm này sẽ tạo ra các thông tin để điền vào bảng.

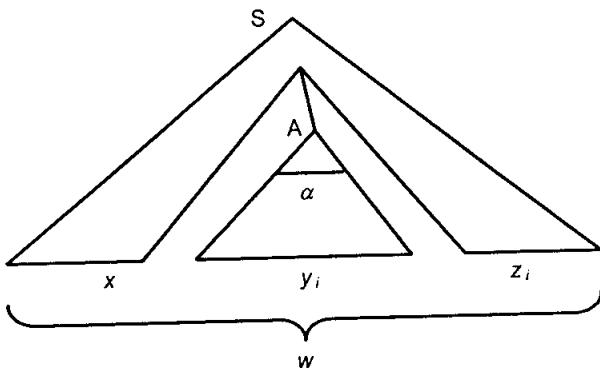
Định nghĩa FIRST (α) - với α là một xâu nào đó, là tập các ký hiệu kết thúc bắt đầu các xâu được suy dẫn từ α . Nếu $\alpha \Rightarrow \epsilon$ thì $\epsilon \in FIRST(\alpha)$.

Định nghĩa FOLLOW (A), với A là ký hiệu không kết thúc, là tập các ký hiệu kết thúc a mà chúng có thể xuất hiện ngay bên phải của A ở trong một số dạng câu, tức là tập các ký hiệu kết thúc a sao cho tồn tại một suy dẫn dạng $E \Rightarrow \alpha A a \beta$ đối với α và β bất kỳ. Nếu A là ký hiệu bên phải nhất trong một số dạng câu thì ta thêm \$ vào FOLLOW (A).

¹ Trong trường hợp tổng quát, để xây dựng bộ phân tích LL (k) hoặc LR (k) cần phải tính FIRST (k) và FOLLOW (k). Nhưng trong thực hành người ta thường chỉ dùng $k \leq 1$ nên ta cũng chỉ học cách tính FIRST (1), FOLLOW(1) và viết tắt là FIRST, FOLLOW.

Hàm FIRST (α) cho biết một xâu c α hiện tại có thể suy dẫn đến tận cùng thành một xâu bắt đầu bằng các ký hiệu kết thúc nào. Hàm FOLLOW (A) cho biết các ký hiệu kết thúc ở đầu phần câu do các phần tử sau A tạo ra.

Trong hình 5.2, từ xâu α có thể suy được xâu con y . Do có nhiều lựa chọn khác nhau trong các bước suy dẫn từ A đến y , nên thực chất có thể suy ra được các xâu y khác nhau mà ta đánh số là $y_1, y_2, y_3\dots$. Tập tất cả các ký hiệu kết thúc nằm đầu các xâu con trên chính là FIRST (α).



Hình 5.2.

Tương tự, ta cũng có thể thấy, các đỉnh nằm sau A tạo thành các xâu con $z_1, z_2, z_3\dots$. Tập tất cả các ký hiệu đầu tiên của các xâu đó chính là FOLLOW (A).

Tính FIRST

Để tính FIRST (α), ta phải tính dựa trên hàm FIRST (X). Cách tính như sau:

- **FIRST(X)**

Ta sử dụng các quy tắc sau cho đến khi không còn ký hiệu kết thúc hoặc ký hiệu ϵ còn có thể thêm được vào một tập FIRST nào đó:

1. Nếu X là ký hiệu kết thúc thì $\text{FIRST}(X) = \{X\}$;

2. Nếu $X \rightarrow \epsilon$ là một sản xuất thì thêm ϵ vào $\text{FIRST}(X)$;

3. Nếu $X \rightarrow Y_1 Y_2 \dots Y_k$ là một sản xuất, và nếu với một i nào đó thì ϵ có trong mọi $\text{FIRST}(Y_i)$, $\text{FIRST}(Y_2), \dots, \text{FIRST}(Y_{i-1})$ (nghĩa là $Y_1 Y_2 \dots Y_{i-1} \Rightarrow \epsilon$) thì ta thêm mọi ký hiệu kết thúc có trong $\text{FIRST}(Y_i)$ vào $\text{FIRST}(X)$. Nếu $i = k$ (tức là ϵ có trong mọi $\text{FIRST}(Y_i)$ với $i = 1, 2, \dots, k$) thì thêm ϵ vào $\text{FIRST}(X)$.

- **FIRST(α)**

Bây giờ ta có thể tính được FIRST (α) cho mọi xâu c α có dạng $X_1X_2 \dots X_n$ như sau: Thêm vào FIRST ($X_1X_2 \dots X_n$) tất cả các ký hiệu không phải ε của FIRST (X_1). Ta cũng thêm các ký hiệu không phải ε của FIRST (X_2) nếu ε thuộc FIRST (X_1), các ký hiệu không phải ε của FIRST (X_3) nếu ε thuộc cả FIRST (X_1) và FIRST (X_2), cứ như vậy... Cuối cùng, thêm ε vào FIRST ($X_1X_2\dots X_n$) nếu với mọi i mà FIRST (X_i) có chứa ε , hoặc nếu $n = 0$.

Tính FOLLOW (A)

Ta dùng các quy tắc sau cho đến khi không thể thêm gì vào tập FOLLOW:

1. Đặt $\$$ vào FOLLOW (A), với A là ký hiệu bắt đầu (đỉnh cây), $\$$ là ký hiệu đánh dấu kết thúc xâu vào (chú ý là A không nhất thiết phải trùng với S do ta đang tính FOLLOW cho cây con);
2. Nếu có một sản xuất dạng $B \rightarrow \alpha A \beta$ (với $\beta \neq \varepsilon$), thì mọi phần tử thuộc FIRST (β) trừ ε đều được cho vào FOLLOW (A);
3. Nếu có một sản xuất dạng $B \rightarrow \alpha A$ (hoặc một sản xuất $B \rightarrow \alpha A \beta$ với FIRST (β) chứa ε , nghĩa là $\beta \Rightarrow \varepsilon$), thì mọi phần tử của FOLLOW (B) cũng cho vào FOLLOW (A).

Ví dụ 5.2: Cho văn phạm có các sản xuất sau:

$$E \rightarrow TE'; E' \rightarrow +TE' \mid \varepsilon; T \rightarrow FT'; T' \rightarrow *FT' \mid \varepsilon; F \rightarrow (E) \mid a$$

Trong các sản xuất này, E , E' , T , T' và F là các ký hiệu không kết thúc. Các ký hiệu còn lại là a , $+$, $*$, $($, $)$, ε là các ký hiệu kết thúc. Ta tính được kết quả như sau:

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, a\}$$

$$\text{FIRST}(E') = \{ +, \varepsilon \}$$

$$\text{FIRST}(T') = \{ *, \varepsilon \}$$

$$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{), \$ \}$$

$$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{ +,), \$ \}$$

$$\text{FOLLOW}(F) = \{ +, *,), \$ \}$$

Để tính FIRST (F), ta áp dụng quy tắc 3 của cách tính FIRST (X) vào các sản xuất $F \rightarrow (E)$ và $F \rightarrow a$ ta có $\text{FIRST}(F) = \{ (, a\}$. Từ sản xuất $T \rightarrow FT'$

ta thấy không có ε trong FIRST của phần tử đầu tiên về phải (tức là FIRST(F)), do đó theo quy tắc 3 ta có FIRST(T) = FIRST(F) = $\{ \text{, } a \}$. Tương tự đối với sản xuất $E \rightarrow TE'$, ta có FIRST(E) = FIRST(T) = $\{ \text{, } a \}$. Từ sản xuất $E' \rightarrow +TE'$ áp dụng quy tắc 3, sản xuất $E' \rightarrow \varepsilon$ áp dụng quy tắc 2 ta có FIRST(E') = $\{ +, \varepsilon \}$. Hoàn toàn tương tự đối với hai sản xuất $T' \rightarrow *FT'$ và $T' \rightarrow \varepsilon$, ta thu được FIRST(T') = $\{ *, \varepsilon \}$.

Đối với phép tính FOLLOW(E), trước hết ta có $\$$ thuộc tập này theo quy tắc 1. Quy tắc này sẽ không được áp dụng cho bất kì ký hiệu nào khác nữa. Sau đó, ta áp dụng quy tắc 2 vào sản xuất $F \rightarrow (E)$ (sản xuất này có dạng $B \rightarrow \alpha A \beta$, trong đó xâu β chỉ bao gồm một phần tử duy nhất là dấu đóng ngoặc đơn) ta thu được FOLLOW(E) = FIRST()) = $\{ \text{) } \}$. Không còn luật nào có E về phải nữa nên phép tính FOLLOW(E) dừng ở đây. Tổng hợp kết quả: FOLLOW(E) = $\{ \text{, } \$ \}$. Theo quy tắc 3 áp dụng cho luật $E \rightarrow TE'$ ta nhận được FOLLOW(E') = FOLLOW(E) = $\{ \text{, } \$ \}$. Cũng quy tắc này có thể áp dụng cho luật 2 nhưng ta không thu thêm được gì mới...

3. Lập bảng phân tích

Thuật toán sau được dùng để lập bảng phân tích tất định cho một văn phạm G . Thuật toán rất đơn giản. Cho $A \rightarrow \alpha$ là một sản xuất với a thuộc FIRST(α). Mỗi khi bộ phân tích gấp A ở trên đỉnh của *ngăn xếp* và a là ký hiệu vào hiện tại thì bộ phân tích sẽ mở rộng A bằng α . Chỉ có một sự rắc rối khi $\alpha = \varepsilon$ hoặc $\alpha \Rightarrow \varepsilon$. Trong trường hợp này, chúng ta cũng có thể mở rộng A bằng α nếu như ký hiệu vào hiện tại thuộc FOLLOW(A), hoặc nếu con trỏ đầu vào chỉ đến $\$$ và $\$$ thuộc FOLLOW(A).

Thuật toán 5.2. Xây dựng bảng phân tích tất định LL.

Vào: Văn phạm G .

Ra: Bảng phân tích M .

Thuật toán:

1. Đối với mỗi sản xuất $A \rightarrow \alpha$, thực hiện bước 2 và bước 3.
2. Đối với mỗi ký hiệu kết thúc a thuộc FIRST(α), thêm $A \rightarrow \alpha$ vào $M[A, a]$.
3. Nếu ε thuộc FIRST(α), thêm $A \rightarrow \alpha$ vào $M[A, \$]$ đối với mỗi b thuộc FOLLOW(A). Nếu ε thuộc FIRST(α) và $\$$ là thuộc FOLLOW(A), thêm $A \rightarrow \alpha$ vào $M[A, \$]$.
4. Đặt tất cả các vị trí chưa được định nghĩa còn lại của bảng là *lỗi*.

Ví dụ 5.3: Áp dụng thuật toán 5.2 cho ví dụ trên. Vì FIRST (TE') = FIRST(T) = {(, a} nên cả hai ô M[E, ()] và v [E, a] đều được đặt là E → TE'. Tương tự ô M[E', +] được đặt là E' → +TE'... Cuối cùng bảng phân tích như ví dụ 5.1.

4. Văn phạm LL (k) và LL (1)

Chữ “L” đầu tiên của LL (k) viết tắt cho việc quét từ trái sang phải (left-to-right), chữ “L” tiếp theo chỉ suy dẫn đưa ra là suy dẫn trái nhất (leftmost derivation), và k là một số chỉ việc nhìn trước k ký tự để quyết định hành động phân tích ở mỗi bước. Trong thực tế hay dùng $k = 1$, bộ phân tích là LL (1) và thường được viết tắt là LL nếu không sợ hiểu nhầm.

Khi áp dụng thuật toán 5.2 cho một số văn phạm, có thể có một số vị trí trong bảng phân tích có trên một giá trị (tức là số sản xuất nhiều hơn 1). Ví dụ, nếu văn phạm G là đệ quy trái hoặc nhập nhằng thì ít nhất phải có một vị trí trong bảng như vậy. Trường hợp mọi vị trí trong bảng chỉ có nhiều nhất một thì ta có định nghĩa của LL (1) như dưới đây.

Định nghĩa: Văn phạm LL (1) là các văn phạm xây dựng được bằng phân tích theo thuật toán 5.2 có các ô chỉ được định nghĩa nhiều nhất là một lần.

Như vậy văn phạm LL (1) không bị nhập nhằng và không có đệ quy trái.

Điều kiện để một văn phạm là LL (1)

Nếu $A \rightarrow \alpha | \beta$ là hai sản xuất phân biệt của G thì các điều kiện sau phải thỏa mãn:

1. Không có một ký hiệu kết thúc a nào mà cả α và β có thể suy dẫn các xâu bắt đầu bằng a .
2. Nhiều nhất là chỉ một trong α hoặc β có thể suy dẫn ra xâu rỗng.
3. Nếu $\beta \Rightarrow^* \epsilon$ thì α không suy dẫn được một xâu nào bắt đầu bằng một ký hiệu kết thúc b thuộc FOLLOW (A).

Một cách đơn giản khác để kiểm tra xem một văn phạm có là LL (1) hay không là lập bảng phân tích và dựa vào định nghĩa 5.3 kiểm tra.

5. Khôi phục lỗi trong phân tích tất định

Ngăn xếp của bộ phân tích LL chứa lẩn lộn các ký hiệu kết thúc và không kết thúc. Mong muốn của ta là các ký hiệu này sẽ khớp với việc phân

tích phần còn lại của xâu vào. Một lỗi sẽ được phát hiện trong quá trình phân tích tất định khi:

- Ký hiệu kết thúc ở trên đỉnh ngăn xếp không đúng với ký hiệu vào tiếp theo;
- Vị trí trong bảng phân tích là $M[A, a]$ với biến A ở trên đỉnh ngăn xếp và ký hiệu vào tiếp theo a , lại là rỗng hay đã diễn dấu hiệu báo lỗi (error).

Chiến lược khôi phục lỗi thường theo kiểu trùng phạt là bỏ qua các ký hiệu trên xâu vào cho đến khi xuất hiện một từ tố thuộc tập từ khoá đã được xác định trước (gọi là tập từ khoá đồng bộ). Tính hiệu quả của phương pháp này phụ thuộc vào việc chọn tập đồng bộ. Các tập này được chọn sao cho bộ phân tích vượt qua được lỗi nhanh chóng. Một vài cách như sau:

1. Tại thời điểm bắt đầu, ta có thể đưa tất cả các ký hiệu trong FOLLOW (A) vào tập đồng bộ của ký hiệu không kết thúc A . Nếu khi có lỗi chúng ta bỏ qua các từ tố cho đến khi một phần tử của FOLLOW (A) xuất hiện và lấy A khỏi ngăn xếp, lúc này quá trình phân tích có thể tiếp tục.
2. Nếu chỉ dùng FOLLOW (A) cho tập đồng bộ thì không đủ. Ví dụ, nếu dấu chấm phẩy kết thúc các câu lệnh (như trong ngôn ngữ C) thì các từ khoá bắt đầu câu lệnh có thể không xuất hiện trong tập FOLLOW của ký hiệu không kết thúc tạo ra các biểu thức. Một dấu chấm phẩy bị thiếu sau một phép gán có thể là lý do làm cho từ khoá bắt đầu câu lệnh tiếp theo bị bỏ qua. Đôi khi các ngôn ngữ lập trình có các cấu trúc phân cấp: một biểu thức có thể xuất hiện trong một câu lệnh, các câu lệnh lại tạo thành các khối... thì chúng ta có thể thêm vào tập đồng bộ của một cấu trúc thấp hơn các ký hiệu bắt đầu của các lớp cao hơn. Ví dụ, chúng ta có thể thêm các từ khoá bắt đầu các câu lệnh vào các tập đồng bộ của các ký hiệu không kết thúc sinh ra các biểu thức.
3. Nếu chúng ta thêm các ký hiệu trong FIRST (A) vào tập đồng bộ của ký hiệu không kết thúc A thì nó có thể tiếp tục phân tích theo A nếu một ký hiệu trong FIRST (A) xuất hiện ở đầu vào.
4. Nếu một ký hiệu không kết thúc có thể sinh ra một xâu rỗng thì sản xuất suy dẫn ra có thể được sử dụng như ngầm định. Thực hiện điều này thì có thể phát hiện ra lỗi nhưng không làm lỗi mất được. Phương pháp này dùng để giảm số lượng các ký hiệu không kết thúc phải xem xét trong lúc khắc phục lỗi.

5. Nếu một ký hiệu kết thúc nằm trên đỉnh ngăn xếp không đúng, thì ý tưởng đơn giản là loại bỏ ký hiệu kết thúc này, đưa ra một thông báo và tiếp tục phân tích. Phương pháp này tạo nên tập đồng bộ của một từ tố có chứa tất cả các từ tố khác.

Ví dụ 5.4: Ta lập tập đồng bộ synch dựa vào các kết quả tính FOLLOW trong ví dụ 5.2 ở trên. Cụ thể ta có: $\text{synch} = \{+, *, (), \$\}$. Dựa vào tập này, ta lập bảng phân tích như bảng 5.4. Bảng 5.5 cho thấy quá trình phân tích xâu vào có lỗi $) a * + a$.

Ký hiệu cú pháp	Ký hiệu vào					
	a	$+$	$*$	()	$\$$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	<i>synch</i>	<i>synch</i>
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	<i>synch</i>		$T \rightarrow FT'$	<i>synch</i>	<i>synch</i>
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow a$	<i>synch</i>	<i>synch</i>	$F \rightarrow (E)$	<i>synch</i>	<i>synch</i>

Bảng 5.4. Bảng phân tích có *synch*.

Ngăn xếp	Đầu vào	Lưu ý
$\$E$	$) a * + a \$$	lỗi, bỏ qua)
$\$E$	$A * + a \$$	
$\$E'T$	$A * + a \$$	
$\$E'T'F$	$A * + a \$$	
$\$E'T'a$	$A * + a \$$	
$\$E'T'$	$* + a \$$	
$\$E'T'F^*$	$* + a \$$	
$\$E'T'F$	$+ a \$$	lỗi, $M[F, +] = \text{synch}$, bỏ F
$\$E'T'$	$+ a \$$	
$\$E'$	$+ a \$$	
$\$E'T^+$	$+ a \$$	
$\$E'T$	$a \$$	

$SE'T'F$	$a\$$	
$SE'T'a$	$a\$$	
$SE'T'$	$\$$	
SE'	$\$$	
$\$$	$\$$	

Bảng 5.5. Quá trình phân tích một xâu vào lỗi.

II. PHÂN TÍCH LR (tùy chọn)¹

1. Giới thiệu

Chúng ta sẽ tiếp tục nghiên cứu một phương pháp phân tích khác cũng rất hiệu quả, dựa trên chiến lược phân tích bottom-up. Phương pháp này có thể dùng để phân tích một lớp lớn của VPPNC. Nó được gọi là phân tích LR (k). Chữ "L" chỉ việc quét xâu vào từ trái sang phải (left-to-right), chữ "R" chỉ suy dẫn nhận được là suy dẫn phải nhất (rightmost), còn k là một số chỉ số ký hiệu vào được xem trước để quyết định việc phân tích. Trong thực tế, k thường là 1 và phân tích LR (1) được viết tắt là LR nếu không sợ hiểu nhầm.

Phân tích LR có các ưu điểm sau:

- Các bộ phân tích LR có thể được xây dựng để phân tích hầu hết các ngôn ngữ lập trình xây dựng từ VPPNC.
- Phương pháp phân tích LR là phương pháp phân tích gạt - thu gọn không quay lui tổng quát nhất, và nó lại có thể hoạt động hiệu quả như các phương pháp gạt - thu gọn khác.
- Lớp các văn phạm có thể phân tích được bằng phương pháp LR là tập rất lớn của lớp các văn phạm có thể được phân tích bằng các phương pháp phân tích tất định.
- Một bộ phân tích LR có thể phát hiện lỗi cú pháp rất nhanh, ngay trong khi quét xâu vào từ trái sang phải.

Nhược điểm cơ bản của phương pháp phân tích này là phải làm rất nhiều việc để xây dựng nên một bộ phân tích LR cho một văn phạm của một ngôn

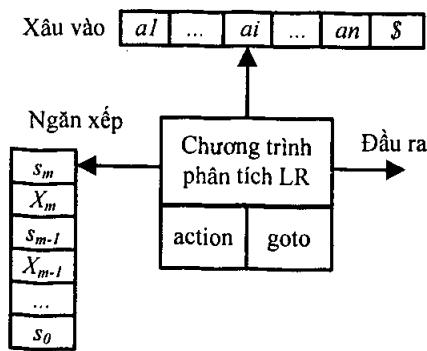
¹ Bạn có thể học hoặc bỏ phần này. Một cách chọn khác là chỉ tập trung vào phần lập bảng SLR và bỏ qua phần LR chuẩn và LALR (bỏ qua các phần II.5 và II.6)

ngữ lập trình thông thường. Chính vì vậy, ta cần phải có thêm một công cụ đặc biệt - một bộ sinh ra bộ phân tích LR. Thật may mắn là đã có nhiều bộ phân tích như vậy (ví dụ bộ Yacc có trong hệ điều hành UNIX. Với một bộ sinh như vậy có thể đưa ra một VPPNC, rồi dùng bộ sinh để tạo ra bộ phân tích cho văn phạm đó. Nếu văn phạm này có chứa các nhập nhằng hoặc các cấu trúc khó phân tích thì bộ sinh sẽ định vị lại các cấu trúc đó để báo cho người thiết kế chương trình dịch biết xử lý).

Sau khi đề cập đến hoạt động của bộ phân tích LR, chúng ta sẽ đề cập tiếp đến việc xây dựng các bảng phân tích LR khác nhau cho một văn phạm. Phương pháp đầu tiên gọi là LR đơn giản (viết tắt là SLR - Simple LR), là phương pháp đơn giản nhất nhưng cũng kém nhất. Nó có thể thất bại khi xây dựng bảng phân tích cho một văn phạm cho trước, trong khi các phương pháp khác lại không bị, nói cách khác, lớp văn phạm dùng được cho nó bé nhất. Phương pháp thứ hai, gọi là LR chuẩn (canonical LR), là phương pháp mạnh nhất nhưng cũng tốn kém nhất. Phương pháp thứ ba, gọi là LR nhìn vượt (LALR-Lookahead LR) là phương pháp trung gian về sức mạnh và chi phí giữa hai phương pháp trên. Phương pháp LALR làm việc với hầu hết các văn phạm của các ngôn ngữ lập trình, và về một số khía cạnh nào đó, nó hoạt động rất hiệu quả.

2. Thuật toán phân tích LR

Cơ sở của phương pháp phân tích LR dựa vào chiến lược phân tích bottom-up và ôtômát đầy xuống.



Hình 5.3. Mô hình của bộ phân tích LR

Sơ đồ của một bộ phân tích LR được cho như hình 5.3. Nó bao gồm một xâu vào, một ngăn xếp (hay một danh sách đầy xuống), một đầu ra, một đầu đọc. Đầu đọc này được điều khiển bằng một chương trình gọi là chương trình phân tích LR, một bảng phân tích có hai phần (*action* và *goto*).

Chương trình điều khiển đầu đọc thì giống nhau đối với mọi bộ phân tích LR, chỉ khác nhau ở bảng phân tích.

Chương trình phân tích mỗi lần chỉ đọc một ký hiệu của xâu vào. Nó dùng ngăn xếp để lưu các xâu có dạng $s_0X_1s_1X_2s_2\dots X_ms_m$, với s_m nằm trên đỉnh. Các X_i là các ký hiệu văn phạm (các ký hiệu kết thúc và không kết thúc), các s_i là các ký hiệu trạng thái. Mỗi một ký hiệu trạng thái là một tóm tắt về thông tin chứa trong phần ngăn xếp ở dưới nó. Kết hợp ký hiệu trạng thái đang nằm trên đỉnh của ngăn xếp và ký hiệu vào hiện được dùng để tham khảo đến bảng phân tích để xác định hành động phân tích gạt - thu gọn.

Bảng phân tích có hai phần, một hàm hành động phân tích *action* và một hàm nhảy *goto*. Chương trình phân tích LR hoạt động như sau. Nó xác định s_m , trạng thái hiện tại nằm trên đỉnh của ngăn xếp, và a_i , ký hiệu vào hiện tại. Sau đó, nó tham khảo giá trị *action*[s_m, a_i] để biết phải làm gì, giá trị của bảng này có thể là một trong bốn giá trị dưới đây:

1. *gạt s* (vào ngăn xếp), với s là một trạng thái.
2. *thu gọn* (trong ngăn xếp) bằng một sản xuất dạng $A \rightarrow \beta$.
3. chấp nhận.
4. *lỗi*.

Hàm *goto* có hai tham số đầu vào là trạng thái và ký hiệu văn phạm và đưa ra ở đầu ra một trạng thái. Ta sẽ thấy hàm *goto* thực chất là một hàm chuyển trạng thái của một ôtômát hữu hạn đơn định (DFA) dùng để nhận ra các phần tiền tố khác nhau của G . Các phần tiền tố khác nhau của G chính là các tiền tố của dạng câu phải, xuất hiện trong ngăn xếp của bộ phân tích gạt - thu gọn. Trạng thái khởi đầu của DFA này là trạng thái ban đầu được đặt ở đỉnh của ngăn xếp.

Một hình trạng (configuration) của một bộ phân tích LR là một cặp bao gồm các thành phần nằm trên ngăn xếp và phần xâu vào còn lại chưa được thu gọn:

$$(s_0X_1s_1X_2s_2\dots X_ms_m, a_i a_{i+1} \dots a_n \$)$$

Cấu hình này thể hiện dạng câu phải:

$$X_1X_2\dots X_ma_ia_{i+1}\dots a_n$$

về cơ bản thì giống như dạng câu phải trong phân tích gạt - thu gọn, có thêm các trạng thái.

Bước chuyển tiếp của bộ phân tích này được xác định bởi ký hiệu đọc vào hiện tại a_i và trạng thái trên đỉnh của ngăn xếp s_m nhờ xét giá trị bảng *action*[s_m, a_i]. Các hình trạng của thuật toán sau bốn kiểu chuyển như trên sẽ có các dạng như sau:

1. Nếu $action[s_m, a_i] = gat\ s$, bộ phân tích sẽ thực hiện một bước gạt, và chuyển sang hình trạng sau:

$$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$$

ở đây, bộ phân tích đã gạt cả ký hiệu vào hiện tại a_i và trạng thái tiếp theo s , căn cứ vào $action[s_m, a_i]$ vào ngăn xếp. Lúc này a_{i+1} trở thành ký hiệu vào hiện tại.

2. Nếu $action[s_m, a_i] = thu\ gọn\ A \rightarrow \beta$ thì bộ phân tích sẽ thực hiện một bước thu gọn, chuyển sang hình trạng sau:

$$(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$$

với $s = goto[s_{m-r}, a_i]$, r là độ dài của β - vé bên phải của sản xuất. Ở đây, bộ phân tích đã loại bỏ $2r$ ký hiệu nằm ở phần định của ngăn xếp (bao gồm r ký hiệu trạng thái và r ký hiệu văn phạm), trạng thái s_{m-r} trở thành định. Sau đó bộ phân tích đẩy vào ngăn xếp ký hiệu văn phạm A của vé trái sản xuất và trạng thái s căn cứ vào bảng goto. Ký hiệu vào không hề bị đụng chạm đến.

3. Nếu $action[s_m, a_i] = chấp\ nhận$, phân tích hoàn thành.

4. Nếu $action[s_m, a_i] = lõi$, bộ phân tích sẽ tìm cách khôi phục lõi bằng cách gọi thủ tục khôi phục lõi.

Thuật toán 5.3. Thuật toán phân tích LR.

Vào: Một xâu vào w và một bảng phân tích LR với các hàm $action$ và $goto$ cho một văn phạm G.

Ra: Nếu w thuộc L (G), một phân tích bottom-up cho w ; còn nếu không là một thông báo lỗi.

Thuật toán:

Lúc đầu, s_0 là trạng thái khởi đầu nằm trên đỉnh ngăn xếp, $w\$$ trong xâu vào.

while true begin

Coi s là trạng thái trên đỉnh ngăn xếp và a là ký hiệu vào chỉ bởi con trỏ vào ip

if $action[s, a] = gat\ s'$ **then begin**

Đặt a , rồi đặt s' vào đỉnh của ngăn xếp;
dịch con trỏ vào ip sang vị trí vào tiếp theo;

end

else

if $action[s, a] = thu\ gọn, A \rightarrow \beta$ **then begin**

Lấy $2^* | \beta$ | ký hiệu khỏi ngăn xếp;
 s' là trạng thái hiện tại trên đỉnh ngăn xếp;
 Đặt A , rồi đặt $goto[s', A]$ vào đỉnh của ngăn xếp;
 đưa ra sản xuất $A \rightarrow \beta$;

```

end
else   if  $action[s, a] = \text{chấp nhận}$  then
            return
        else  $\text{error}();$ 
end;

```

Ví dụ 5.5: Cho văn phạm dưới đây là các biểu thức số học của phép + và *, cho bảng phân tích 5.6:

$$1) E \rightarrow E + T$$

$$2) E \rightarrow T$$

$$3) T \rightarrow T * F$$

$$4) T \rightarrow F$$

$$5) F \rightarrow (E)$$

$$6) F \rightarrow a$$

1. si có nghĩa là gạt một ký hiệu vào và trạng thái i vào ngăn xếp,
2. rj có nghĩa là thu gọn bằng sản xuất đánh số j ,
3. acc có nghĩa là chấp nhận,
4. các ô trống có nghĩa là lỗi.

Trạng thái	action						goto		
	a	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

7	s5			s4					10
8		s6			s11				
9		r1	r7		r1	R1			
10		r3	r3		r3	R3			
11		r5	r5		r5	R5			

Bảng 5.6. Bảng phân tích.

Với xâu vào a^*a+a , quá trình phân tích như bảng 5.7 dưới đây.

	Ngăn xếp	Xâu vào	Action
1	0	$a^* a + a \$$	Gạt
2	0 a 5	$* a + a \$$	thu gọn bởi sản xuất $F \rightarrow a$
3	0 F 3	$* a + a \$$	thu gọn bởi sản xuất $T \rightarrow F$
4	0 T 2	$* a + a \$$	Gạt
5	0 T 2 * 7	$a + a \$$	Gạt
6	0 T 2 * 7 a 5	$+ a \$$	thu gọn bởi sản xuất $F \rightarrow a$
7	0 T 2 * 7 F 10	$+ a \$$	thu gọn bởi sản xuất $T \rightarrow T^*F$
8	0 T 2	$+ a \$$	thu gọn bởi sản xuất $E \rightarrow T$
9	0 E 1	$+ a \$$	Gạt
10	0 E 1 + 6	$a \$$	Gạt
11	0 E 1 + 6 a 5	$\$$	thu gọn bởi sản xuất $F \rightarrow a$
12	0 E 1 + 6 F 3	$\$$	thu gọn bởi sản xuất $T \rightarrow F$
13	0 E 1 + 6 T 9	$\$$	thu gọn bởi sản xuất $E \rightarrow E + T$
14	0 E 1	$\$$	chấp nhận

Bảng 5.7. Quá trình phân tích.

3. Văn phạm LR

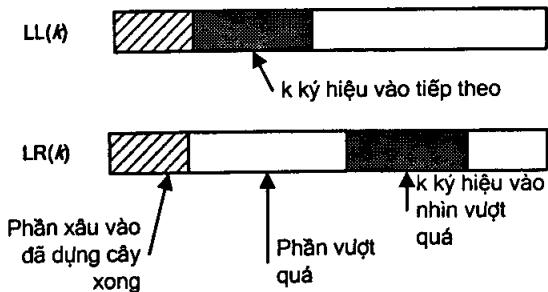
Vấn đề bây giờ là làm sao xây dựng được bảng phân tích LR từ một văn phạm cho trước. Một văn phạm mà ta có thể xây dựng được bảng phân tích được gọi là một văn phạm LR. Có những VPPNC không phải là LR nhưng nói chung các cấu trúc của các ngôn ngữ lập trình thông thường lại là LR.

Để một văn phạm là LR thì bộ phân tích LR phải có khả năng nhận ra được các thành phần ở đỉnh của ngăn xếp. Nó không cần phải quét toàn bộ ngăn xếp để phát hiện về phải của một sản xuất có nằm ở đỉnh ngăn xếp hay không mà chỉ cần xem ký hiệu trạng thái ở đỉnh ngăn xếp là có đủ thông tin cần thiết. Do đó, ta có thể dùng một ôtômát hữu hạn đọc các ký hiệu văn phạm từ đỉnh ngăn xếp xuống đáy là có thể xác định được có về phải của một sản xuất nào đó ở đỉnh ngăn xếp hay không. Hàm *goto* của một bảng phân tích LR về nguyên tắc chính là một ôtômát hữu hạn. Ôtômát này không cần phải đọc ngăn xếp ở mọi bước chuyển. Ký hiệu trạng thái nằm ở đỉnh ngăn xếp chính là trạng thái của ôtômát này nếu nó đọc các ký hiệu văn phạm của ngăn xếp từ đáy tới đỉnh và chính là trạng thái báo nó đã nhận ra được về phải của một sản xuất nào đó. Do vậy, bộ phân tích LR có thể xác định từ trạng thái ở đỉnh của ngăn xếp mọi thứ mà nó cần biết về những gì đang ở trong ngăn xếp.

Một nguồn thông tin khác mà một bộ phân tích LR có thể sử dụng trong các quyết định gạt - thu gọn là k ký hiệu vào tiếp theo. Các trường hợp $k = 0$ hoặc $k = 1$ được chú ý nhiều trong thực hành và ta cũng chỉ quan tâm đến các bộ phân tích LR với $k \leq 1$. Ví dụ ở bảng phân tích trên chỉ nhìn quá 1 ký hiệu ($k = 1$). Một văn phạm có thể phân tích bằng một bộ phân tích LR xét trước k ký hiệu vào cho mỗi bước chuyển thì được gọi là một văn phạm LR (k).

Có sự khác nhau đáng kể giữa các văn phạm LL và LR. Đối với một văn phạm LR (k), ta phải có khả năng nhận biết về phải của một sản xuất nào đó, bằng cách xem tất cả những gì suy dẫn được từ về phải qua k ký hiệu vào được nhìn vượt quá. Điều này kém chặt chẽ hơn nhiều so với các văn phạm LL (k) mà ta phải nhận biết được sản xuất nào được dùng chỉ với k ký hiệu đầu tiên mà về phải của nó suy dẫn ra. Nói cách khác, LL chỉ là một trường hợp đặc biệt của LR. Do đó, văn phạm LR có thể mô tả được nhiều ngôn ngữ hơn là văn phạm LL.

Người ta đã chứng minh được rằng, mọi văn phạm LL (1) đều là văn phạm LR (1), nhưng điều ngược lại thì không đúng.



Hình 5.4. Giải thích sự khác nhau giữa LL và LR.

4. Xây dựng bảng phân tích SLR

Đây là phương pháp kém nhất trong ba phương pháp nếu so sánh về số lượng văn phạm xây dựng được bảng phân tích nhưng lại là phương pháp dễ thực hiện nhất. Một văn phạm mà có thể xây dựng được một bộ phân tích SLR được gọi là văn phạm SLR.

Một mục LR(0) (LR(0) item - và gọi tắt là mục) của một văn phạm G là một sản xuất của G có một dấu chấm nằm ở đâu đó bên vệ phải. Do vậy sản xuất $A \rightarrow XYZ$ sẽ có bốn mục như sau:

$$A \rightarrow \bullet XYZ$$

$$A \rightarrow X \bullet YZ$$

$$A \rightarrow XY \bullet Z$$

$$A \rightarrow XYZ \bullet$$

Sản xuất $A \rightarrow \epsilon$ chỉ có một mục là $A \rightarrow \bullet$. Một mục có thể được biểu diễn bằng một cặp số nguyên, số đầu tiên là số thứ tự của sản xuất, số thứ hai là vị trí của dấu chấm.

Về trực quan, một mục cho biết một sản xuất trong quá trình phân tích có thể suy dẫn như thế nào. Ví dụ, mục đầu tiên ở trên nói rằng, ta hi vọng thấy một xâu con tiếp theo trên đầu vào có thể suy dẫn được từ XYZ . Mục thứ hai chỉ rằng ta đã thấy trên đầu vào một xâu suy dẫn từ X và ta hi vọng thấy một xâu tiếp theo có thể suy dẫn từ YZ .

Tư tưởng chủ đạo của phương pháp SLR là ta sẽ xây dựng từ văn phạm đã cho một ôtômát hữu hạn đơn định DFA dùng để nhận dạng các tiền tố có thể có. Ta nhóm các mục với nhau thành các tập và các tập này được lấy làm các trạng thái của bộ phân tích SLR.

Một tập các mục LR (0) được gọi là *tập LR (0) chuẩn*, là cơ sở cho việc xây dựng các bộ phân tích SLR. Để xây dựng tập LR (0) chuẩn cho một văn phạm ta định nghĩa *văn phạm mở rộng* và hai hàm: *closure* (bao đóng) và *goto* (nhảy tới).

Văn phạm mở rộng

Nếu G là một văn phạm với ký hiệu bắt đầu là E thì G' là văn phạm mở rộng của G bằng cách thêm vào G một ký hiệu bắt đầu E' và thêm vào đó một sản xuất $E' \rightarrow E$. Lý do của việc sửa đổi này là nhằm chỉ cho bộ phân tích biết khi nào nên dùng phân tích và thông báo chấp nhận xâu vào. Do

đó, việc chấp nhận sẽ xảy ra khi và chỉ khi bộ phân tích thu gọn bằng sản xuất $E' \rightarrow E$.

Hàm closure

Nếu I là tập các mục của một văn phạm G thì bao đóng $\text{closure}(I)$ là tập các mục được xây dựng từ I bằng hai luật:

1. Khởi đầu, mọi mục trong I đều được thêm vào $\text{closure}(I)$.
2. Nếu $A \rightarrow \alpha \bullet B\beta$ có trong $\text{closure}(I)$ và $B \rightarrow \gamma$ là một sản xuất thì thêm mục $B \rightarrow \bullet\gamma$ vào I nếu nó chưa có sẵn trong đó. Áp dụng luật này cho đến khi không thêm được một mục nào nữa vào $\text{closure}(I)$.

Trực quan, $A \rightarrow \alpha \bullet B\beta$ trong $\text{closure}(I)$ chỉ ra rằng, tại một điểm nào đó trong quá trình phân tích, có thể thấy tiếp một xâu con suy dẫn từ $B\beta$ ở đầu vào. Nếu $B \rightarrow \gamma$ là một sản xuất, ta cũng mong muốn thấy một xâu con suy dẫn từ γ tại điểm này. Vì những lý do đó ta thêm $B \rightarrow \bullet\gamma$ vào $\text{closure}(I)$.

Ví dụ 5.6: Xem xét văn phạm mở rộng của các biểu thức như sau:

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

Nếu I là tập mục chỉ có đúng một mục $\{[E' \rightarrow \bullet E]\}$ thì $\text{closure}(I)$ bao gồm các mục:

$$\begin{array}{lll} E' \rightarrow \bullet E & E \rightarrow \bullet E + T & E \rightarrow \bullet T \\ T \rightarrow \bullet T * F & T \rightarrow \bullet F & \\ F \rightarrow \bullet(E) & F \rightarrow \bullet a & \end{array}$$

ở đây $E' \rightarrow \bullet E$ được đặt vào $\text{closure}(I)$ theo luật 1. Khi có một E đi ngay sau một chấm thì bằng luật 2 ta thêm các sản xuất E với các chấm nằm ở đầu trái, tức là $E \rightarrow \bullet E + T$ và $E \rightarrow \bullet T$. Bây giờ lại có một T đi ngay sau một chấm và ta lại thêm $T \rightarrow \bullet T * F$ và $T \rightarrow \bullet F$. Tiếp theo, tương tự với F ta thêm được $F \rightarrow \bullet(E)$ và $F \rightarrow \bullet a$. Và bây giờ không còn thêm được một mục nào nữa vào $\text{closure}(I)$ qua luật 2.

Hàm *closure* có thể được tính như dưới đây. Một cách thuận tiện thực hiện hàm này là dùng một mảng boolean *added*, được đánh chỉ số theo các

ký hiệu không kết thúc của G , ví dụ như $added[B]$ được đặt là **true** nếu và khi ta thêm các mục $B \rightarrow \bullet\gamma$ đối với mỗi sản xuất $B \rightarrow \gamma$.

```

function closure( $I$ );
begin
     $J := I;$ 
    repeat
        for với mỗi mục  $A \rightarrow \alpha \bullet B\beta$  trong  $J$  và
            với mỗi sản xuất  $B \rightarrow \gamma$  của  $G$  mà  $B \rightarrow \bullet\gamma$  không trong  $J$  do
                thêm  $B \rightarrow \bullet\gamma$  vào  $J$ 
    until không còn mục nào được thêm vào  $J$ ;
    Trả lại  $J$ ;
end;

```

Chú ý khi một sản xuất B được thêm vào bao đóng của I với dấu chấm ở đầu bên trái, thì tất cả các sản xuất B sẽ được thêm tương tự vào bao đóng này. Thật ra, trong một số hoàn cảnh thực tế không cần thiết liệt kê các mục $B \rightarrow \bullet\gamma$ thêm vào I bằng hàm *closure*. Ta có thể chia tất cả các tập mục mà ta quan tâm thành hai lớp mục:

- Mục nhân*: bao gồm mục khởi đầu $E' \rightarrow \bullet E$ và tất cả các mục mà các chấm không ở đầu bên trái.
- Mục không phải nhân*: các mục có các chấm nằm ở đầu bên trái.

Ta thấy rằng, mỗi tập mục được hình thành bằng cách lấy bao đóng của một tập mục nhân, dĩ nhiên các mục được thêm vào bao đóng không còn là mục nhân nữa. Do đó có thể biểu diễn các tập mục ta thật sự quan tâm với lưu trữ ít nhất bằng cách loại bỏ tất cả các mục không phải nhân đi và nếu cần có thể được sinh lại bằng hàm tính bao đóng.

Hàm goto

Hàm thứ hai là $goto(I, X)$ với I là một tập các mục và X là một ký hiệu văn phạm $goto(I, X)$ được xác định là bao đóng closure của tập tất cả các mục $[A \rightarrow \alpha X \bullet \beta]$ mà $[A \rightarrow \alpha \bullet X \beta]$ là thuộc I . Ta có thể thấy, nếu I là tập các mục hợp lệ đối với một tiền tố có thể có γ nào đó, thì $goto(I, X)$ là tập các mục hợp lệ với tiền tố có thể có γX .

Ví dụ 5.7: Nếu I là tập hai mục $\{[E' \rightarrow E\bullet], [E \rightarrow E\bullet + T]\}$ thì $goto(I, +)$ bao gồm:

$$E \rightarrow E + \bullet T$$

$$T \rightarrow \bullet T * F$$

$$T \rightarrow \bullet F$$

$$F \rightarrow \bullet(E)$$

$$F \rightarrow \bullet a$$

Ta tính $goto(I, +)$ bằng cách kiểm tra I cho các mục với dấu $+$ ở ngay bên phải dấu chấm. $E' \rightarrow E\bullet$ không phải mục như vậy, nhưng $E \rightarrow E\bullet + T$ thì đúng. Ta chuyển dấu chấm qua bên kia dấu $+$ để nhận được $\{E \rightarrow E + \bullet T\}$ và sau đó tính *closure* cho tập này.

Xây dựng tập mục

Thuật toán dưới đây dùng để xây dựng C là một tập chuẩn LR (0) cho một văn phạm mở rộng G' :

procedure items(G');

begin

$$C := \{ \text{closure}(\{[E' \rightarrow \square E]\}) \};$$

repeat

for với mỗi tập mục I của C và mỗi ký hiệu văn phạm X

mà $goto(I, X)$ không rỗng và không thuộc C **do**

Thêm $goto(I, X)$ vào C ;

until không thêm được tập mục nào nữa vào C ;

end;

Ví dụ 5.8:

Tập chuẩn của tập các mục LR (0) cho văn phạm ở ví dụ 5.6 như sau. Hàm *goto* cho tập mục này giống như đồ thị chuyển của một DFA như hình dưới.

$$I_0: \quad E' \rightarrow \bullet E$$

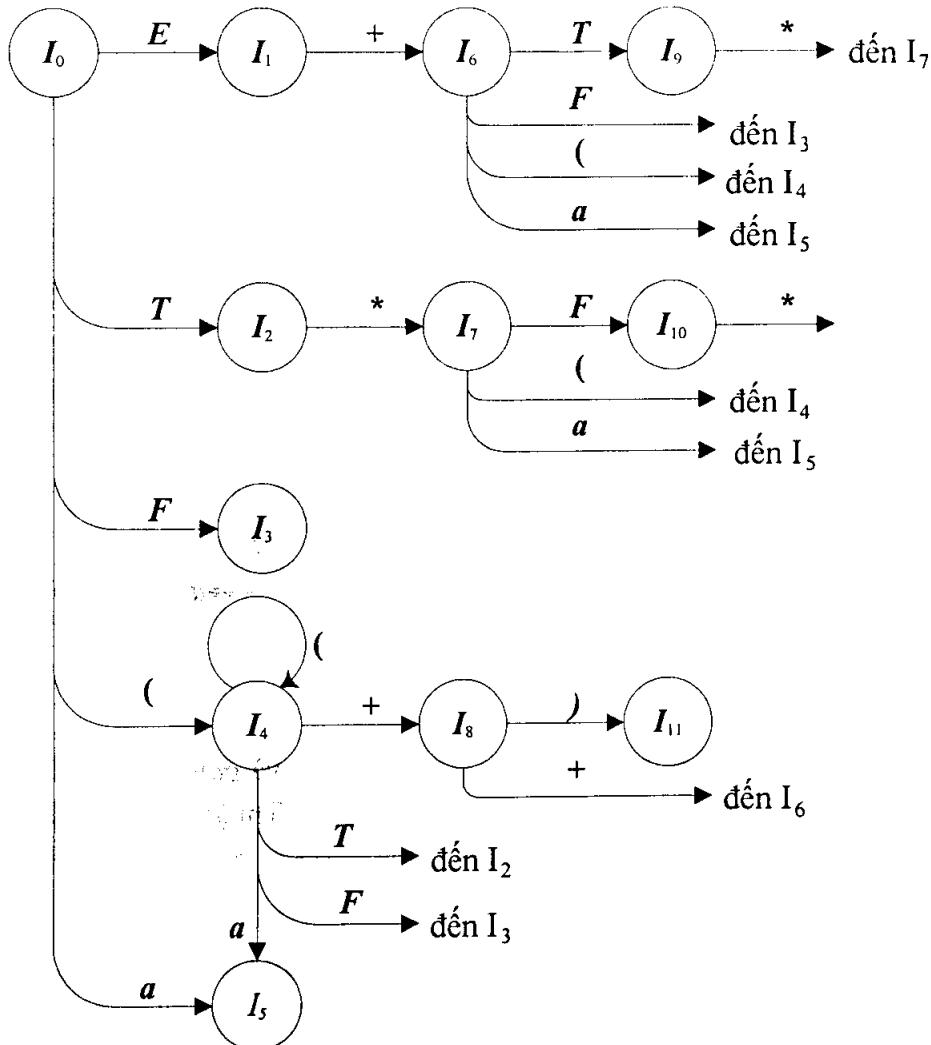
$$I_5: \quad F \rightarrow a\bullet$$

$$E' \rightarrow \bullet E + T$$

$E' \rightarrow \bullet T$	$I_6:$	$E \rightarrow E + \bullet T$	
$T \rightarrow \bullet T * F$		$T \rightarrow \bullet T * F$	
$T \rightarrow \bullet F$		$T \rightarrow \bullet F$	
$F \rightarrow \bullet(E)$		$F \rightarrow \bullet(E)$	
$F \rightarrow \bullet a$		$F \rightarrow \bullet a$	
$I_1:$	$E' \rightarrow E \bullet$	$I_7:$	$T \rightarrow T * \bullet F$
	$E' \rightarrow E \bullet + T$		$F \rightarrow \bullet(E)$
			$F \rightarrow \bullet a$
$I_2:$	$E \rightarrow T \bullet$	 	
	$T \rightarrow T * \bullet F$	$I_8:$	$F \rightarrow (E \bullet)$
			$E \rightarrow E \bullet + T$
$I_3:$	$T \rightarrow \bullet F$	 	
$I_4:$	$F \rightarrow (\bullet E)$	$I_9:$	$E \rightarrow E + T \bullet$
	$E \rightarrow \bullet E + T$		$T \rightarrow T \bullet * F$
	$E \rightarrow \bullet T$	$I_{10}:$	$T \rightarrow T * F \bullet$
	$T \rightarrow \bullet T * F$	 	
	$T \rightarrow \bullet F$	$I_{11}:$	$F \rightarrow \bullet(E)$
	$F \rightarrow \bullet(E)$		
	$F \rightarrow \bullet a$		

Mục hợp lệ: Ta nói mục $A \rightarrow \beta_1 \bullet \beta_2$ là *hợp lệ* cho một tiền tố có thể có $\alpha\beta_1$ nếu có một suy diễn phải nhất $E' \xrightarrow{rm} \alpha Aw \xrightarrow{rm} \alpha\beta_1\beta_2w$. Nói chung, một mục sẽ hợp lệ đối với nhiều tiền tố có thể có. Việc $A \rightarrow \beta_1 \bullet \beta_2$ là hợp lệ đối với $\alpha\beta_1$ cho ta biết nhiều thứ như lúc nào thì gạt hay thu gọn khi ta tìm thấy $\alpha\beta_1$ trong ngăn xếp đang phân tích. Cụ thể, nếu $\beta_2 \neq \epsilon$ thì nó cho biết là ta chưa gạt đủ cho vé phải của một sản xuất nào đó vào ngăn xếp, do đó bước chuyển của ta là gạt. Nếu $\beta_2 = \epsilon$ cho thấy ngăn xếp chứa vé phải của $A \rightarrow \beta_1$, và ta cần phải thu gọn bằng sản xuất này. Dĩ nhiên là hai mục hợp lệ có thể

thực hiện những việc khác nhau cho cùng tiền tố có thể có. Một số xung đột có thể giải quyết bằng phương pháp ở phần sau, nhưng ta không nên cho rằng mọi xung đột phân tích có thể được giải quyết nếu dùng phương pháp LR để xây dựng một bảng phân tích cho một văn phạm nhập nhằng.



Hình 5.5. Đồ thị chuyển.

Ta có thể dễ dàng tính tập các mục hợp lệ cho mỗi tiền tố có thể có mà có thể xuất hiện trong ngăn xếp của một bộ phân tích LR. Đây chính là trung tâm của lý thuyết phân tích LR mà tập các mục hợp lệ cho tiền tố có thể có γ đến được chính xác từ tập các mục bắt đầu từ trạng thái khởi đầu và theo một đường được đánh nhãn γ trong DFA được cấu trúc từ tập chuẩn LR (0) với các chuyển dịch cho bằng *goto*. Về nguyên tắc, tập các mục hợp lệ chứa mọi thông tin có ích có thể thấy trong ngăn xếp.

Ví dụ 5.9: Ta xét lại văn phạm trong ví dụ 5.6 lần nữa mà tập mục và goto đã được tính. Rõ ràng là xâu $E + T^*$ là tiền tố có thể có của văn phạm này. Ôtômát trong hình vẽ sẽ ở trạng thái I_7 sau khi đọc $E + T^*$. Trạng thái I_7 bao gồm các mục:

$$T \rightarrow T^* \bullet F$$

$$F \rightarrow \bullet(E)$$

$$F \rightarrow \bullet a$$

đều là các mục hợp lệ đối với $E + T^*$. Để thấy điều này ta xem ba suy dẫn phải ở dưới đây:

$$E \Rightarrow E$$

$$E' \Rightarrow E$$

$$E' \Rightarrow E$$

$$\Rightarrow E + T$$

$$\Rightarrow E + T$$

$$\Rightarrow E + T$$

$$\Rightarrow E + T^* F$$

$$\Rightarrow E + T^* F$$

$$\Rightarrow E + T^* F$$

$$\Rightarrow E + T^* (E) \Rightarrow E + T^* a$$

Suy dẫn đầu tiên cho thấy mục hợp lệ là $T \rightarrow T^* \bullet F$, suy dẫn tiếp theo cho thấy mục hợp lệ thứ hai là $F \rightarrow \bullet(E)$, và thứ ba là $F \rightarrow \bullet a$ đối với tiền tố có thể có $E + T^*$.

Bảng phân tích SLR

Bây giờ ta sẽ xem cách xây dựng các hàm *action* và *goto* của phân tích SLR từ DFA dùng để nhận dạng các tiền tố có thể có. Thuật toán này không phải luôn đưa ra được các bảng *action* cho mọi văn phạm, nhưng nó thành công đối với nhiều ngôn ngữ lập trình.

Cho một văn phạm G , ta mở rộng nó thành G' và từ G' ta xây dựng C là tập chuẩn của các tập mục cho G' . Ta xây dựng hàm phân tích *action* và hàm nhảy *goto* từ C bằng thuật toán dưới đây. Nó đòi hỏi ta tính FOLLOW (A) cho mọi ký hiệu không kết thúc của văn phạm.

Thuật toán 5.4: Xây dựng một bảng phân tích SLR

Vào: một văn phạm mở rộng G' .

Ra: Bảng phân tích SLR có các hàm *action* và *goto* cho G' .

Phương pháp:

1. Xây dựng $C = \{I_0, I_1, \dots, I_n\}$ là tập các tập mục LR (0) cho G' .

2. Trạng thái i được xây dựng từ I_i . Các hoạt động phân tích cho trạng thái i được xác định như sau:
- Nếu $[A \rightarrow \alpha \bullet a\beta]$ thuộc I_i và $goto(I_i, a) = I_j$ thì đặt $action[i, a]$ thành "gạt j ". Ở đây a phải là một ký hiệu kết thúc.
 - Nếu $[A \rightarrow \alpha \bullet]$ thuộc I_i thì đặt $action[i, a]$ là "thu gọn $A \rightarrow \alpha$ " cho mọi a thuộc FOLLOW(A); ở đây A phải khác E' .
 - Nếu $[E' \rightarrow E \bullet]$ thuộc I_i thì đặt $action[i, \$]$ là "chấp nhận".

Nếu có bất kì đụng độ nào được sinh ra bằng các luật trên thì ta nói văn phạm đã cho không phải là SLR (1). Thuật toán thông báo không sinh được bộ phân tích trong trường hợp này.

- Các bước chuyển $goto$ cho trạng thái i được xây dựng cho mọi ký hiệu không kết thúc A bằng luật: nếu $goto(I_i, A) = I_j$ thì $goto[i, A] = j$.
- Mọi vị trí không được xác định bằng luật 2 và 3 sẽ được đặt là "lỗi".
- Trạng thái khởi đầu của bộ phân tích được xây dựng từ tập mục có chứa $[E' \rightarrow \bullet E]$.

Bảng phân tích ở trên được gọi là bảng SLR (1) cho văn phạm G và gọi tắt là SLR.

Ví dụ 5.10: Ta sẽ xây dựng bảng SLR cho văn phạm trong ví dụ 5.6. Trước hết ta xét tập mục I_0 :

$$E' \rightarrow \bullet E$$

$$E \rightarrow \bullet E + T$$

$$E \rightarrow \bullet T$$

$$T \rightarrow \bullet T^* F$$

$$T \rightarrow \bullet F$$

$$F \rightarrow \bullet (E)$$

$$F \rightarrow \bullet a$$

Mục $F \rightarrow \bullet (E)$ giúp ta thông tin để đặt vào bảng tại vị trí $action[0, ()] =$ gạt 4, mục $F \rightarrow \bullet a$ đặt vị trí $action[0, a] =$ gạt 5. Các mục khác trong I_0 không gây hành động gì cả. Bây giờ ta tính I_1 :

$$E' \rightarrow E \bullet$$

$$E \rightarrow E \bullet + T$$

Nhờ mục đầu tiên ta đặt được $action[1, \$] = \text{chấp nhận}$, cái thứ hai làm $action[1, +] = \text{gạt } 6$. Tiếp theo I_2 :

$$E \rightarrow T \bullet$$

$$T \rightarrow T \bullet^* F$$

Ta tính được FOLLOW (E) = $\{\$, +, \}\},$ mục đầu tiên làm $action[2, \$] = action[2, +] = action[2,]\} = thu gọn E \rightarrow T$. Mục thứ 2 làm $action[2, *] = gạt 7$. Cứ thế tiếp tục ta sẽ nhận được các bảng $action$ và $goto$ như ví dụ đầu 5.5.

5. Xây dựng bảng phân tích LR chuẩn

Bây giờ ta sẽ nghiên cứu một kỹ thuật tổng quát nhất để xây dựng một bảng phân tích LR cho một văn phạm. Cũng giống như trong phương pháp SLR, trạng thái i gây thu gọn nhờ $A \rightarrow \alpha$ nếu tập mục I_i có chứa mục $[A \rightarrow \alpha \bullet]$ và a là trong FOLLOW (A). Trong một số tình huống, trạng thái i nằm trên đỉnh ngăn xếp, tiền tố có thể có $\beta\alpha$ trong ngăn xếp thì βA không thể có a đi theo trong dạng câu trái. Do đó, việc thu gọn bằng $A \rightarrow \alpha$ sẽ không thích hợp đối với đầu vào a .

Có khả năng ta xét nhiều thông tin hơn đối với các trạng thái cho phép ta không chế được các thu gọn không hợp bằng các sản xuất dạng $A \rightarrow \alpha$. Bằng cách tách các trạng thái ra khi cần thiết, ta có thể sắp xếp để có môi trường trạng thái của bộ phân tích LR chỉ được chính xác các ký hiệu vào có thể đi theo một vế trái sản xuất α có khả năng thu gọn về A hay không.

Thông tin thêm này được hợp nhất vào trạng thái bằng cách định nghĩa lại các mục để thêm được các ký hiệu kết thúc làm thành phần thứ hai. Dạng tổng quát của mục trở thành $[A \rightarrow \alpha \bullet \beta, a]$, với $A \rightarrow \alpha\beta$ là một sản xuất và a là một kết thúc hoặc dấu kết thúc phải $\$$. Ta gọi đó là một mục LR (1). Số 1 là độ dài của thành phần thứ hai, và gọi là *nhìn quá* (lookahead) của các mục. Nhìn quá này không tác động trong mục dạng $[A \rightarrow \alpha \bullet \beta, a]$ với β không phải là ϵ , nhưng một mục của dạng $[A \rightarrow \alpha \bullet, a]$ dùng cho thu gọn $A \rightarrow \alpha$ chỉ nếu ký hiệu vào là a . Do vậy, ta hoàn thành các thu gọn bằng $A \rightarrow \alpha$ chỉ khi các ký hiệu vào là a và mục $[A \rightarrow \alpha \bullet, a]$ là một mục LR (1) có trong trạng thái nằm trên đỉnh ngăn xếp. Tập các a như vậy luôn luôn là tập con của FOLLOW (A).

Hình thức, ta cũng nói mục LR (1) $[A \rightarrow \alpha \bullet \beta, a]$ là hợp lệ đối với một tiền tố có thể có γ nếu có một suy diễn $E \xrightarrow{r_m} \delta A w \xrightarrow{r_m} \delta \alpha \beta w$ với:

1. $\gamma = \delta\alpha$

2. hoặc a là ký hiệu đầu tiên của w hoặc w là ϵ và a là \$.

Ví dụ 5.11: Ta có văn phạm

$$E \rightarrow BB$$

$$B \rightarrow aB \mid b$$

Có một suy dẫn phải nhất $E \xrightarrow{r_m^*} aaBab \xrightarrow{r_m^*} aaaBab$. Ta thấy mục $[B \rightarrow a \bullet B, a]$ là hợp lệ với một tiền tố có thể có $\gamma = aaa$ bằng cách coi $\delta = aa$, $A = B$, $w = ab$, $\alpha = a$ và $\beta = B$ theo định nghĩa ở trên.

Còn có một suy dẫn nữa $E \xrightarrow{r_m^*} BaB \xrightarrow{r_m^*} BaaB$. Đối với suy dẫn này ta thấy mục $[B \rightarrow a \bullet B, \$]$ là hợp với tiền tố có thể có Baa .

Phương pháp xây dựng bộ các tập các mục hợp lệ LR (1) về cơ bản giống như tập chuẩn LR (0). Ta chỉ cần sửa đổi hai hàm *closure* và *goto*.

Để đánh giá được định nghĩa mới của *closure*, ta xem một mục dạng $[A \rightarrow \alpha \bullet B\beta, a]$ trong tập mục hợp lệ đôi với một tiền tố có thể có γ nào đó. Có một suy dẫn trái $E \xrightarrow{r_m^*} \delta Aax \xrightarrow{r_m^*} \delta\alpha B\beta ax$, với $\gamma = \delta\alpha$. Giả sử βax dẫn đến một xâu kết thúc bằng by . Với mỗi sản xuất dạng $B \rightarrow \eta$ với η nào đó ta có suy dẫn $E \xrightarrow{r_m^*} \gamma Bby \xrightarrow{r_m^*} \gamma\eta by$. Do vậy, $[B \rightarrow \bullet\eta, b]$ là hợp lệ với γ . Lưu ý là b có thể là ký hiệu kết thúc đầu tiên được suy dẫn từ β hoặc có thể β suy dẫn ra ϵ trong suy dẫn $\beta ax \xrightarrow{*} by$, và do vậy b có thể là a . Để tổng kết cho cả hai, ta nói b có thể là một ký hiệu kết thúc bất kì trong FIRST (βax). Chú ý là x không thể chứa kết thúc đầu của by , do đó FIRST (βax) = FIRST(βa).

Thuật toán 5.5: Xây dựng các tập mục LR (1)

Vào: Một văn phạm mở rộng G' .

Ra: Các tập mục hợp lệ LR (1) cho một hoặc nhiều tiền tố có thể có.

Thuật toán: Các hàm *closure*, *goto* và thủ tục *items* dùng để xây dựng các tập mục.

```

function closure( $I$ );
begin
    repeat
        for với mỗi mục  $[A \rightarrow \alpha \bullet B\beta, a]$  trong  $I$  và
            với mỗi sản xuất  $B \rightarrow \gamma$  của  $G'$  và
            với mỗi ký hiệu kết thúc  $b$  trong FIRST( $\beta a$ )
            mà  $B \rightarrow \bullet \gamma$  không trong  $I$  do
                Thêm  $[B \rightarrow \bullet \gamma, b]$  vào  $I$ ;
        until không còn mục nào được thêm vào  $I$ ;
        Trả lại  $I$ ;
    end;

```

```

function goto( $I, X$ );
begin
    Đặt  $J$  là tập mục  $[A \rightarrow \alpha X \bullet \beta, a]$  mà  $[A \rightarrow \alpha \bullet X\beta, a]$  trong  $I$ 
    Trả lại closure( $J$ );
end;
procedure items( $G'$ );
begin
     $C := \{ \text{closure}(\{[E' \rightarrow \square E, \$]\}) \};$ 
    repeat
        for với mỗi tập mục  $I$  của  $C$  và với mỗi ký hiệu văn phạm  $X$ 
            mà  $\text{goto}(I, X)$  không rỗng và không thuộc  $C$  do
            Thêm  $\text{goto}(I, X)$  vào  $C$ ;
        until không thêm được tập mục nào nữa vào  $C$ ;
    end;

```

Ví dụ 5.12: Xem xét văn phạm mở rộng sau:

$$E' \rightarrow E$$

$$E \rightarrow CC$$

$$C \rightarrow cC \mid d$$

Ta tính được kết quả như sau:

$$I_0: \quad E' \rightarrow \bullet E, \$$$

$$E \rightarrow \bullet CC, \$$$

$$C \rightarrow \bullet cC, c/d$$

$$C \rightarrow \bullet d, c/d$$

$$I_1: \quad E' \rightarrow E \bullet, \$$$

$$I_3: \quad C \rightarrow c \bullet C, c/d$$

$$C \rightarrow \bullet cC, c/d$$

$$C \rightarrow \bullet d, c/d$$

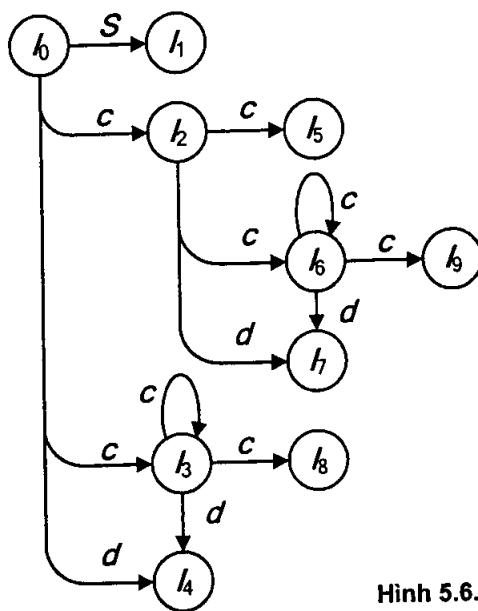
$$I_4: \quad C \rightarrow d \bullet, c/d$$

$$I_5: \quad E \rightarrow CC \bullet, \$$$

$$I_6: \quad C \rightarrow c \bullet C, \$$$

$$C \rightarrow \bullet cC, \$$$

$$C \rightarrow \bullet d, \$$$



Hình 5.6.

Thuật toán 5.6: Xây dựng một bảng phân tích LR chuẩn

Vào: Một văn phạm mở rộng G'

Ra: Bảng phân tích LR chuẩn có các hàm *action* và *goto* cho G' .

Phương pháp:

1. Xây dựng $C = \{I_0, I_1, \dots, I_n\}$ là bộ các tập mục LR (1) cho G' .
2. Trạng thái i được xây dựng từ I_i . Các hoạt động phân tích cho trạng thái i được xác định như sau:
 - a. Nếu $[A \rightarrow \alpha \bullet a\beta, b]$ thuộc I_i và $goto(I_i, a) = I_j$ thì đặt $action[i, a]$ thành "*gạt j*". Ở đây a phải là một ký hiệu kết thúc.
 - b. Nếu $[A \rightarrow \alpha \bullet, a]$ thuộc I_i , $A \neq E'$ thì đặt $action[i, a]$ là "*thu gọn $A \rightarrow \alpha$* ".
 - c. Nếu $[E' \rightarrow E \bullet, \$]$ thuộc I_i thì đặt $action[i, \$]$ là "*chấp nhận*".
- Nếu có bất kì đụng độ nào được sinh ra khi dùng các luật trên thì ta nói văn phạm không phải là LR (1). Thuật toán thông báo không sinh được bộ phân tích trong trường hợp này.
3. Các bước chuyển *goto* cho trạng thái i được xây dựng như sau: nếu $goto(I_i, A) = I_j$ thì $goto[i, A] = j$.
4. Mọi vị trí không được xác định bằng luật 2 và 3 sẽ được đặt là "*lỗi*".
5. Trạng thái khởi đầu của bộ phân tích được xây dựng từ tập mục có chứa $[E' \rightarrow \bullet E, \$]$.

Bảng *action* và *goto* được xây dựng từ thuật toán trên được gọi là bảng phân tích LR (1) chuẩn. Một bộ phân tích LR dùng bảng này gọi là một bộ phân tích chuẩn LR (1). Nếu các vị trí không bị xác định nhiều lần thì văn phạm đã cho gọi là một văn phạm LR (1). Thông thường ta hay bỏ qua chữ "(1)" thành LR nếu không sợ hiểu nhầm.

Ví dụ 5.13: Bảng 5.8 là phân tích chuẩn cho văn phạm ở trên. Sản xuất 1, 2, 3 là $E \rightarrow CC$, $C \rightarrow cC$ và $C \rightarrow d$.

Mọi văn phạm SLR (1) là văn phạm LR (1), nhưng với cùng một văn phạm SLR (1) thì bộ phân tích LR có thể có nhiều trạng thái hơn bộ phân tích SLR. Như các ví dụ trên cho thấy, cùng văn phạm thì bộ phân tích SLR chỉ có 7, còn ở đây có tới 10 trạng thái.

Trạng thái	Action			goto	
	c	d	\$	E	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Bảng 5.8. Bảng phân tích chuẩn.

6. Xây dựng bảng phân tích LALR

Bây giờ ta sẽ nghiên cứu phương pháp cuối cùng để xây dựng bộ phân tích kỹ thuật LALR (Lookahead LR - LR nhìn vượt quá). Phương pháp này được dùng nhiều trong thực hành do xây dựng được bảng phân tích nhỏ hơn đáng kể bảng LR chuẩn và thể hiện thuận tiện các cấu trúc văn phạm của nhiều ngôn ngữ lập trình bằng văn phạm LALR. Điều này hầu như cũng đúng với các văn phạm SLR, nhưng lại có vài cấu trúc không thể thể hiện bằng kĩ thuật SLR. Nói cách khác, lớp văn phạm LALR lớn hơn SLR một chút.

So sánh về kích thước bộ phân tích, các bảng SLR và LALR cho một văn phạm luôn có cùng số trạng thái. Đối với các ngôn ngữ như Pascal, chúng thường có vài trăm trạng thái, trong khi đó bảng chuẩn LR lại phải có vài ngàn trạng thái cho cùng ngôn ngữ đó. Do đó, việc xây dựng các bảng SLR và LALR dễ và tiết kiệm hơn rất nhiều so với các bảng chuẩn LR.

Ta lại tiếp tục làm việc với văn phạm cho ở trên. Xét hai trạng thái I_4 và I_7 . Mỗi một trạng thái này chỉ có các mục với thành phần đầu $C \rightarrow d$. Trong I_4 , ký hiệu nhìn quá là c hoặc d ; trong I_7 chỉ có $\$$.

Để thấy vai trò khác nhau của I_4 và I_7 trong bộ phân tích, chú ý là văn phạm này tạo tập chính quy c^*dc^*d . Khi đọc một xâu vào $cc...cdcc...cd$, bộ phân tích gạt nhóm c và theo sau chúng là d vào ngăn xếp, chuyển sang

trạng thái 4 sau khi đọc d . Sau đó bộ phân tích thu gọn bằng sản xuất $C \rightarrow d$ và ký hiệu vào tiếp theo là c hoặc d . Yêu cầu là c hoặc d phải theo một thứ tự như thế nào đó để là đầu của các xâu trong c^*d . Nếu \$ theo sau kết thúc đầu d , ta có một xâu vào có dạng ccd không có trong ngôn ngữ, và trạng thái 4 sẽ thông báo lỗi nếu \$ lại là ký hiệu vào tiếp theo.

Bộ phân tích chuyển sang trạng thái 7 sau khi đọc d thứ hai. Sau đó, bộ phân tích phải nhìn thấy \$ trong đầu vào, hoặc nó bắt đầu một xâu không ở dạng c^*dc^*d . Nó tạo ra tình huống là trạng thái 7 cần phải thu gọn bằng $C \rightarrow d$ với đầu vào \$ và thông báo lỗi với đầu vào c hoặc d .

Bây giờ ta thay I_4 và I_7 bằng I_{47} - là kết hợp của I_4 và I_7 , với tập ba mục được biểu diễn bằng $[C \rightarrow d, c/d/$]$. Trước đây d nhảy vào trạng thái I_4 hoặc I_7 từ I_0, I_2, I_3 và I_6 thì bây giờ d nhảy vào I_{47} . Hành động của trạng thái 47 là thu gọn một đầu vào nào đó. Bộ phân tích được thay đổi ở trên về cơ bản sẽ hành động giống như bản gốc mặc dù nó có thể thu gọn d về C trong một số tình huống khi bộ gốc thông báo lỗi như khi gặp các xâu vào như ccd hoặc $cdcdc$. Các lỗi này sẽ được bắt giữ, thật ra, nó sẽ bị bắt giữ trước khi bắt cứ ký hiệu vào nào được gạt thêm.

Tổng quát hơn, ta có thể tìm các tập mục LR (1) có cùng nhân đó là các thành phần đầu tiên và ta có thể kết hợp các tập này với các nhân chung trong tập mục. Ví dụ đồ thị chuyển ở hình trên có I_4 và I_7 như là một cặp với nhân $\{C \rightarrow d\}$. Tương tự, I_3 và I_6 là một cặp khác với nhân $\{C \rightarrow cC, C \rightarrow cC, C \rightarrow d\}$. Một cặp nữa I_8 và I_9 có nhân $\{C \rightarrow cC\}$. Chú ý rằng, tổng quát một cặp là một tập mục LR (0) của văn phạm và một văn phạm LR (1) có thể tạo ra nhiều hơn hai tập mục với cùng nhân.

Khi nhân của $goto(I, X)$ chỉ phụ thuộc vào nhân của I , thì $goto$ của các tập kết hợp bản thân chúng có thể được kết hợp lại. Như vậy, không có vấn đề gì trong việc biến đổi hàm $goto$ khi ta kết hợp hai tập mục. Các hàm *action* được biến đổi để phản ánh các *action* không có lỗi của tất cả các tập mục trong bộ kết hợp.

Giả sử ta có một văn phạm LR (1) và một tập mục LR (1) của nó tạo ra các xung đột trong bảng *action*. Nếu ta thay tất cả các trạng thái có chung nhân với liên kết của chúng, thì dường như kết quả liên kết sẽ có xung đột, nhưng không phải như thế do có các lý do sau: Giả sử việc kết hợp có một xung đột ở lookahead a do có một mục $[A \rightarrow \alpha \bullet, a]$ dùng để thu gọn bằng $A \rightarrow \alpha$ và có một mục khác $[B \rightarrow \beta \bullet a\gamma, b]$ dùng cho một phép gạt. Do vậy tập mục nào đó từ liên kết này tạo ra có mục $[A \rightarrow \alpha \bullet, a]$ và khi các nhân của tất

cả các trạng thái đó như nhau, nó phải có một mục $[B \rightarrow \beta \bullet a\gamma, b]$ đối với c nào đó. Nhưng sau đó, trạng thái này có cùng xung đột gạt / thu gọn với a , và văn phạm này không là LR (1) như chúng ta giả thuyết. Do vậy việc kết hợp các trạng thái với các nhân chung có thể không bao giờ tạo ra một xung đột gạt / thu gọn mà nó không có trong một trong các trạng thái gốc, vì các hành động gạt phụ thuộc chỉ vào nhân, không vào ký hiệu nhìn vượt quá.

Nhưng có khả năng là bộ kết hợp sẽ tạo ra một xung đột thu gọn / thu gọn như ví dụ sau.

Ví dụ 5.14: Ta xét văn phạm sau:

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow aAd \mid bBd \mid bAe \\ A &\rightarrow c \\ B &\rightarrow c \end{aligned}$$

và bốn xâu vào được sinh ra là acd , ace , bcd và bce . Văn phạm này là LR (1). Khi ta xây dựng tập mục, ta sẽ thấy các mục $\{[A \rightarrow c \bullet, d], [B \rightarrow c \bullet, e]\}$ hợp lệ với tiền tố có thể có ac và $\{[A \rightarrow c \bullet, e], [B \rightarrow c \bullet, d]\}$ hợp lệ với bc . Không có tập nào bị xung đột và nhân của chúng là như nhau. Chỉ có kết hợp của chúng:

$$\begin{aligned} A &\rightarrow c \bullet, d/e \\ B &\rightarrow c \bullet, d/e \end{aligned}$$

tạo ra một xung đột thu gọn / thu gọn, khi có thể thu gọn bằng cả $A \rightarrow c$ và $B \rightarrow c$ cho các đầu vào d và e .

Bây giờ ta chuẩn bị cho thuật toán xây dựng bảng LALR đầu tiên. Tư tưởng chính là xây dựng các tập mục LR (1), và nếu có xung đột, kết hợp các tập với các nhân chung. Sau đó, ta xây dựng bảng phân tích từ tập các tập mục được kết hợp. Phương pháp này được coi như định nghĩa của văn phạm LALR (1). Việc xây dựng toàn bộ tập mục LR (1) đòi hỏi rất nhiều khoảng trống bộ nhớ và thời gian tính toán.

Thuật toán 5.7: Xây dựng bảng LALR theo cách dễ nhưng tốn không gian.

Vào: Văn phạm mở rộng G' .

Ra: Hàm bảng phân tích LALR *action* và *goto* cho G' .

Phương pháp:

1. Xây dựng $C = \{I_0, I_1, \dots, I_n\}$ là bộ các tập mục LR (1) cho G' .
2. Đối với mỗi nhân trong tập mục LR (1), ta tìm tất cả các tập có nhân này và thay thế bằng kết hợp của chúng.
3. Cho $C = \{J_0, J_1, \dots, J_m\}$ là các tập kết quả của các mục LR (1). Các hoạt động phân tích cho trạng thái i được xác định từ J_i cùng phương pháp với thuật toán 5.6. Nếu có một đựng độ xảy ra thì thuật toán không sinh được bộ phân tích và văn phạm này không phải là LALR (1).
4. Bảng goto được xây dựng như sau. Nếu J là kết hợp của một hoặc nhiều tập mục LR (1) tức là $J = I_1 \cup I_2 \cup \dots \cup I_k$, thì các nhân của $goto(I_1, X)$, $goto(I_2, X), \dots, goto(I_k, X)$ là như nhau nếu I_1, I_2, \dots, I_k có cùng nhân. Nếu K là kết hợp của tất cả các tập mục có cùng nhân đó thì $goto(J, X) = K$.

Bảng được tạo ra bằng thuật toán trên được gọi là bảng phân tích LALR cho G . Nếu không có xung đột thì văn phạm đã cho được nói là văn phạm LALR (1). Tập các mục được xây dựng trong bước 3 được gọi là tập LALR (1).

Ví dụ 5.15: Ta lại tiếp tục xem xét văn phạm 5.6 có đồ thị goto ở trên. Như ta đã thấy, có ba cặp mục có thể kết hợp lại. I_3 và I_6 được thay thế bằng liên kết của chúng:

$$I_{36}: \quad C \xrightarrow{?} c \bullet C, c/d/\$$$

$$C \xrightarrow{?} \bullet c C, c/d/\$$$

$$C \xrightarrow{?} \bullet d, c/d/\$$$

I_4 và I_7 được thay bằng

$$I_{47}: \quad C \xrightarrow{?} d \bullet, c/d/\$$$

I_8 và I_9 được thay bằng

$$I_{89}: \quad C \xrightarrow{?} c C \bullet, c/d/\$$$

Các hàm *action* và *goto* LALR cho các tập mục được cô đọng hơn như bảng 5.9 dưới đây:

Trạng thái	action			goto	
	c	d	\$	E	C
0	s36	s74		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Bảng 5.9. Bảng cho LALR.

Để thấy các *goto* được tính như thế nào, ta tính *goto*(I_{36} , C). Trong tập mục LR (1) gốc, $goto(I_3, C) = I_8$ bây giờ là I_{89} , nên ta đặt $goto(I_{36}, C)$ là I_{89} . Ta có thể đi đến cùng kết luận nếu ta xem xét I_6 - thành phần khác của I_{36} . Do đó $goto(I_6, C) = I_9$ và I_9 bây giờ là một phần của I_{89} . Tương tự, ta xét $goto(I_2, C)$ và một đầu vào sau khi gạt c của I_2 . Trong các tập mục LR (1) gốc $goto(I_3, c) = I_6$. Khi kết hợp thành I_{36} , $goto(I_2, c)$ trở thành I_{36} . Do vậy bảng trên cho trạng thái 2 và đầu vào c được đổi thành s36 nghĩa là gạt và đẩy trạng thái 36 vào ngăn xếp.

Khi đầu vào là xâu c^*dc^*d thì cả hai bộ phân tích LR chuẩn và LALR như bảng trên cùng tạo ra chuỗi hành động gạt và thu gọn giống nhau, mặc dù các trạng thái trong ngăn xếp có thể khác nhau. Ví dụ, nếu bộ phân tích LR chuẩn đặt I_3 hoặc I_6 vào ngăn xếp thì bộ phân tích LALR sẽ đẩy I_{36} vào ngăn xếp. Nói chung các bộ phân tích LR chuẩn và LALR hoạt động giống nhau đối với các xâu vào đúng.

Chỉ khi có xâu vào sai, bộ phân tích LALR có thể được xử lý để thực hiện một số thu gọn sau khi bộ phân tích LR chuẩn thông báo lỗi, mặc dù sau đó bộ phân tích LALR cũng không hề gạt thêm bất cứ ký hiệu vào nào nữa.

Ví dụ 5.16: Với đầu vào ccd\$, bộ phân tích LR sẽ đặt chuỗi:

0 c 3 c 3 d 4

trong ngăn xếp và qua trạng thái 4 sẽ phát hiện lỗi do gặp \$. Ngược lại LALR sẽ tạo ra các bước chuyển tương ứng trong ngăn xếp.

0 c 36 c 36 d 47

Thê nhưng trạng thái 47 trên đầu vào \$ có một phép thu gọn $C \rightarrow d$. LALR này sẽ chuyển ngắn xếp về:

$0 c 36 c 36 C 89$

Bây giờ hoạt động của trạng thái 89 với đầu vào \$ là thu gọn $C \rightarrow c$. Ngắn xếp trở thành

$0 c C 89$

tương tự ta có một thu gọn nữa về:

$0 c 2$

Cuối cùng thì trạng thái 2 báo có một lỗi trên đầu vào \$ và do vậy lỗi được phát hiện.

7. Khôi phục lỗi trong phân tích LR

Một bộ phân tích LR sẽ phát hiện lỗi thông qua bảng phân tích *action* và những vị trí lỗi trong bảng. Các lỗi không bao giờ được xác định nhờ bảng *goto*. Các bộ phân tích LR phát hiện và báo lỗi rất nhanh ngay khi không có sự tiếp tục hợp lệ với một đầu vào. Bộ phân tích LR chuẩn không phải thực hiện bất kì một thu gọn nào khi đã phát hiện có lỗi. Các bộ phân tích SLR và LALR có thể còn phải thực hiện một vài thu gọn trước khi báo lỗi nhưng cũng không hề gạt một ký hiệu vào nào vào ngắn xếp.

Trong phân tích LR ta có thể thực hiện việc phục hồi theo chiến lược trùng phạt như sau: Ta quét xuống theo ngắn xếp, cho đến khi tìm thấy một trạng thái s với một *goto* trên ký hiệu không kết thúc A . Không hoặc nhiều ký hiệu đầu vào sẽ bị bỏ đi cho đến khi tìm được một ký hiệu a mà có thể theo A . Bộ phân tích sẽ đặt ngắn xếp về trạng thái $goto[s, A]$ và coi như phân tích lại được tiến hành bình thường. Có thể có nhiều lựa chọn ký hiệu không kết thúc A . Thông thường chúng có thể là những phần các biểu diễn không kết thúc chính của chương trình như một biểu thức, câu lệnh, hoặc khối. Ví dụ, nếu A là không kết thúc *stmt* thì a có thể là chấm phẩy hoặc end.

Phương pháp này cố gắng cô lập các cụm từ có chứa lỗi văn phạm. Bộ phân tích xác định một xâu suy dẫn được từ A có một lỗi. Một phần của xâu đã được xử lý, và kết quả là một chuỗi các trạng thái trong định ngắn xếp. Phần còn lại của xâu vẫn nằm trên đầu vào, và bộ phân tích cố gắng bỏ đi phần còn lại này bằng cách tìm một ký hiệu đầu vào có thể theo sau A một cách hợp pháp. Khi bỏ các trạng thái khỏi ngắn xếp, nó bỏ qua đầu vào đó

và đẩy $goto[s, A]$ vào ngăn xếp, bộ phân tích coi như nó có A và phân tích tiếp hành bình thường.

Bài tập

1. Dựa vào bảng phân tích trong ví dụ 5.1, hãy phân tích các xâu vào sau:

- $(a+a)^*a$
- $a+a^*a^*a$
- $(a+a)^*(a+a)$

2. Cho một văn phạm với các luật sản xuất như sau:

$$E \rightarrow aEbE \mid bEaE \mid \epsilon$$

Văn phạm này có là văn phạm LL (1) không? Tại sao?

3. Cho một văn phạm với các luật sản xuất như sau:

$$E \rightarrow TE'; E' \rightarrow +E \mid \epsilon; T \rightarrow FT'; T' \rightarrow T \mid \epsilon;$$

$$F \rightarrow PF'; F' \rightarrow *F' \mid \epsilon; P \rightarrow (E) \mid a \mid b \mid \epsilon$$

trong đó, P là đỉnh của cây con.

- Tính các FIRST và FOLLOW;
- Văn phạm này là LL (1) ?
- Xây dựng bảng phân tích nếu nó là LL1. Dùng bảng phân tích để phân tích xâu $(a+b)^*b$

4. Cho một văn phạm của các biểu thức logic như sau:

$$bexpr \rightarrow bexpr \text{ or } bterm \mid bterm$$

$$bterm \rightarrow bterm \text{ and } bfactor \mid bfactor$$

$$bfactor \rightarrow \text{not } bfactor \mid (bexpr) \mid \text{true} \mid \text{false}$$

với quy ước là các từ viết đậm, dấu mở và đóng ngoặc là các ký hiệu kết thúc, các từ còn lại là các ký hiệu không kết thúc. $bexpr$ là ký hiệu đỉnh cây con.

Hãy xây dựng bộ phân tích SLR và dùng nó phân tích xâu vào **not (true or false)**

5. Cho văn phạm sau:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T F \mid F$$

$$F \rightarrow F^* \mid a \mid b$$

a. Xây dựng bảng phân tích SLR cho văn phạm này.

b. Xây dựng bảng phân tích LALR.

6. Hãy cho thấy văn phạm sau:

$$E \rightarrow Aa \mid bAc \mid dc \mid bda$$

$$A \rightarrow d$$

là LALR (1) mà không phải là SLR (1).

Bài tập thực hành

1. Viết chương trình tính FIRST và FOLLOW.

2. Sửa chương trình trên để nó tự động phát hiện văn phạm có là LL (1) hay không.

3. Cài đặt bộ phân tích LL (1).

4. Cài đặt bộ phân tích LR (1).

5. Cài đặt chương trình lập bảng SLR.

6. Cài đặt chương trình lập bảng LR chuẩn.

7. Cài đặt chương trình lập bảng LALR.

Bài đọc

Lược sử phát triển phân tích cú pháp

Trong bản báo cáo có tầm ảnh hưởng rất lớn cho tương lai về Algol 60 (Naur 1963), các tác giả đã dùng ký pháp Backus - Naur Form (BNF) để định nghĩa cú pháp của một trong những ngôn ngữ lập trình quan trọng nhất. Sau đó, các nhà nghiên cứu đã nhanh chóng chỉ ra sự tương đương giữa BNF và văn phạm phi ngữ cảnh. Vào những năm 1960, lý thuyết các ngôn ngữ hình thức đã bắt đầu được quan tâm nhiều. Hopcroft và Ullman (1979) là những người đã đặt cơ sở cho lĩnh vực này.

Sau khi phát triển các văn phạm phi ngữ cảnh, các lý thuyết phân tích đã trở nên hệ thống hơn nhiều. Người ta đã phát minh ra một số kĩ thuật phân tích cho mọi văn phạm phi ngữ cảnh. Một trong các kĩ thuật lập trình sớm nhất đã được các tác giả J.Cocke, Younger (1967) và Kasami (1965) khám phá một cách độc lập với nhau - đó là phương pháp phân tích bảng CYK. Trong luận án tiến sĩ, Earley (1970) cũng đã phát triển một thuật toán phân tích tổng quát cho mọi văn phạm phi ngữ cảnh. Aho và Ullman (1972, 1973) đã đề cập chi tiết đến chúng và các phương pháp phân tích khác.

Nhiều phương pháp phân tích khác cũng đã được dùng trong các chương trình dịch. Sheridan (1959) mô tả phương pháp phân tích được dùng trong chương trình dịch Fortran đầu tiên có đưa thêm các dấu ngoặc đơn bao quanh các toán hạng để có thể phân tích được các biểu thức. Floyd (1963) đã đưa ra tư tưởng về trật tự phép toán và sử dụng các hàm trật tự. Vào những năm 1960, có một lượng lớn các chiến lược phân tích bottom - up đã được đưa ra. Trong đó có trật tự đơn giản (Wirth và Weber -1966), ngữ cảnh buộc (Floyd 1964, Graham 1964), trật tự chiến lược hỗn hợp (McKeeman, Horning, và Wortman 1970), và trật tự yếu (Ichbian và Morse 1970).

Hai phương pháp phân tích đệ quy đi xuống và tất định được dùng rộng rãi trong thực hành. Do tính mềm dẻo của nó, phân tích đệ quy đi xuống được dùng trong nhiều hệ thống dịch ban đầu như META (Schorre 1964) và TMG (McClure 1965). Pratt (1973) đưa ra phương pháp phân tích trật tự toán tử top - down.

Lewis và Stearns (1968) bắt đầu nghiên cứu các văn phạm LL và sau đó chúng được Rosenkrantz và Stearns (1970) phát triển tiếp. Knuth (1971) đã nghiên cứu mở rộng các bộ phân tích tất định. Lewis, Rosenkrantz và Stearns (1976) đã đề cập đến việc sử dụng các bộ phân tích tất định trong chương trình dịch. Các thuật toán dùng để chuyển đổi các văn phạm thành dạng LL (1) đã được Foster (1968), Wood (1969), Stearns (1971), Soisalon-Soininen và Ukkonen (1979) đưa ra.

Các văn phạm LR và các bộ phân tích được Knuth (1965) giới thiệu đầu tiên, ông đã đưa ra cấu trúc của các bảng phân tích LR chuẩn. Phương pháp LR không được dùng trong thực hành cho đến khi Korenjak (1969) cho thấy có thể xây dựng được những bộ phân tích có kích thước chấp nhận được cho các văn phạm ngôn ngữ lập trình. Khi DeRemer (1969, 1971) đưa ra các phương pháp SLR và LALR đơn giản hơn của Korenjak, thì kĩ thuật LR trở thành phương pháp được chọn cho các chương trình tự động sinh bộ phân tích. Ngày nay, các chương trình sinh bộ phân tích LR rất phổ biến trong các môi trường xây dựng chương trình dịch.

Một tư tưởng lớn đã được phát triển cho kĩ thuật phân tích LR. Đó là sử dụng các văn phạm nhập nhằng trong phân tích LR, qua Aho, Johnson, và Ullman (1975) và Earley (1975). Việc xoá các giám thiêu bởi các sản xuất đơn đã được đề cập bởi Anderson, Eve, Horning (1973), Aho và Ullman (1973), Demer (1975),

Backhouse (1976), Joliat (1976), Pager (1977), Soisalon-Soinien (1980), và Tokuda (1981).

Các kĩ thuật tính các tập nhìn vượt quá của LALR (1) đã được LaLonde (1971), Anderson, Eve, Horning (1973), Pager (1977), Kristensen và Madsen (1981), DeRemer và Pennello (1982), và Park, Choe, Chang (1985) đưa ra. Họ cũng cung cấp thêm một số so sánh dựa vào kinh nghiệm.

Aho và Johnson (1974) đưa ra một số nghiên cứu tổng quát về phân tích LR và đề cập đến một số thuật toán dùng cho chương trình sinh bộ phân tích Yacc, cùng với việc dùng các sản xuất lỗi để khôi phục lỗi. Aho và Ullman (1972 và 1973) đưa ra một số nghiên cứu mở rộng về phân tích LR và các lý thuyết cốt lỗi của nó.

Đã có nhiều kĩ thuật khôi phục lỗi cho bộ phân tích được đưa ra. Ciesinger (1979), Sippu(1981), Irrons (1963) đã nghiên cứu những kĩ thuật này. Các sản xuất lỗi đã được Wirth (1968) dùng trong chương trình dịch PL360. Leinius (1970) đưa ra chiến lược khôi phục mức cụm từ. Aho và Peterson (1972) cho thấy khôi phục lỗi toàn cục giá thấp có thể đạt được bằng cách dùng các sản xuất lỗi kết hợp với các thuật toán phân tích tổng quát cho các văn phạm phi ngữ cảnh. Mauney và Fischer (1982) mở rộng các tư tưởng này cho các cặp cục bộ giá thấp cho các bộ phân tích LL và LR dùng các kĩ thuật phân tích của Graham, Harrison và Ruzzo (1980). Graham và Rhodes (1975) đề cập đến khôi phục lỗi trong ngữ cảnh phân tích trật tự.

Horning (1976) bàn về chất lượng các thông báo lỗi tốt phải nêu như thế nào. Sippu và Soisalon - Soinien (1983) so sánh hoạt động của kĩ thuật khôi phục lỗi trong Bộ xử lý ngôn ngữ Helsinki với kĩ thuật khôi phục "bước chuyển trước" của Pennello và DeRemer (1978), kĩ thuật khôi phục lỗi LR của Graham, Haley và Joy (1979) và khôi phục lỗi "ngữ cảnh toàn cục" của Pai và Kieburtz (1980).

Chữa lỗi trong khi phân tích được Conway và Maxwell (1963), Moulton và Muller (1967), Conway và Wilcox (1973), Levy (1975), Tai (1978) và Rohrich (1980), Aho và Peterson (1972) đề cập đến.

Chương 6

ĐIỀU KHIỂN DỰA CÚ PHÁP (Xác định ngữ nghĩa dựa trên cú pháp)

I. MỤC ĐÍCH

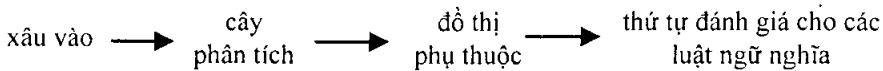
Phần lớn các chương trình dịch hiện đại ngày nay đều là dạng *điều khiển dựa cú pháp* (syntax-directed), nghĩa là các hành động dịch phụ thuộc rất nhiều vào cú pháp của chương trình nguồn cần dịch. Nói cách khác, quá trình dịch được điều khiển theo cấu trúc cú pháp của chương trình nguồn, mà cú pháp này được xác định thông qua bộ phân tích cú pháp. Hầu như mọi khối sau phân tích cú pháp như phân tích ngữ nghĩa, sinh mã trung gian và sinh mã đều thực hiện nhiệm vụ của mình dựa trên cơ sở cấu trúc cú pháp.

Nhằm điều khiển các phần hoạt động theo cú pháp cách thường dùng là "gia cố" các luật sản xuất (mà ta đã biết cụ thể những luật nào và thứ tự được dùng ra sao thông qua cây phân tích) bằng cách thêm các luật xử lý đi kèm với từng luật sản xuất. Các luật này được gọi là các *luật ngữ nghĩa*. Ta lại cần một cơ chế duyệt / thực hiện các luật này dựa theo cây phân tích.

Có hai phương pháp gắn các luật ngữ nghĩa với các sản xuất: các *định nghĩa điều khiển dựa cú pháp* và *lược đồ chuyển đổi*. Các định nghĩa điều khiển dựa cú pháp là điều khiển ở mức cao. Chúng dấu đi nhiều chi tiết thực hiện và cho phép người dùng không cần phải chỉ ra một cách rõ ràng thứ tự thực hiện. Ngược lại, lược đồ chuyển đổi lại chỉ ra thứ tự mà các luật ngữ nghĩa được đánh giá, do đó chúng cho phép thấy một số chi tiết thực hiện.

Với cả hai phương pháp trên, ta có thể duyệt cây cú pháp (là cây được biến đổi từ cây phân tích) để đánh giá (lượng giá) các luật ngữ nghĩa tại các nút của cây. Việc đánh giá các luật ngữ nghĩa có thể nhằm sinh mã, lưu các thông tin trong bảng ký hiệu, đưa ra các thông báo lỗi hoặc thực hiện các nhiệm vụ khác.

Hình 6.1 biểu diễn quá trình một xâu từ chương trình nguồn vào được biến đổi qua nhiều bước cho đến đánh giá ngữ nghĩa ra sao.



Hình 6.1. Quá trình duyệt và đánh giá cây phân tích.

Như vậy, các kiến thức trong phần này không nằm trong một khối chức năng riêng rẽ của chương trình dịch mà sẽ được dùng làm cơ sở cho toàn bộ các khối nằm sau khối phân tích cú pháp.

II. ĐỊNH NGHĨA ĐIỀU KHIỂN DỰA CÚ PHÁP

Một *định nghĩa điều khiển dựa cú pháp* (syntax-directed definition) là một sự tổng quát hoá của văn phạm phi ngữ cảnh. Lúc này mỗi một ký hiệu văn phạm sẽ có một tập các thuộc tính tương ứng đi theo. Các thuộc tính này lại có thể chia thành hai loại: các *thuộc tính tổng hợp* và các *thuộc tính kế thừa*.

Nếu ta coi mỗi một nút ký hiệu văn phạm trên cây phân tích là một bản ghi với các trường lưu thông tin thì một thuộc tính chính là tên của một trường.

Một thuộc tính có thể là mọi thứ ta muốn: một xâu, một con số, một kiểu, một khoảng nhỡ... Giá trị của một thuộc tính tại một nút của cây phân tích được xác định bằng một luật ngữ nghĩa tương ứng với sản xuất dùng tại nút đó. Giá trị của một thuộc tính tổng hợp tại một nút được tính từ các giá trị thuộc tính tại các nút con của nó; giá trị của một thuộc tính kế thừa được tính từ các giá trị thuộc tính tại các nút anh em và cha của nó. Các thuộc tính tổng hợp dùng để chuyển thông tin đi lên trong một cây ngữ pháp, còn các thuộc tính kế thừa thì lại chuyển thông tin đi xuống.

Các luật ngữ nghĩa tạo nên các phụ thuộc giữa các thuộc tính và sẽ được biểu diễn bằng một đồ thị gọi là *đồ thị phụ thuộc*. Từ đồ thị này ta sẽ biết được thứ tự để đánh giá các luật ngữ nghĩa. Việc đánh giá các luật ngữ nghĩa nhằm xác định các giá trị thuộc tính tại các nút trong cây phân tích của một xâu vào. Một luật ngữ nghĩa có thể thực hiện được các nhiệm vụ như in ra một thông báo (gây tác động ra bên ngoài) hoặc đơn giản hơn - cập nhật một biến toàn cục (không gây tác động ra bên ngoài).

Việc thực hiện đánh giá không nhất thiết phải có cây phân tích hoặc đồ thị phụ thuộc có cấu trúc rõ ràng mà có thể dựa trên bất cứ dạng cây tương đương. Nó chỉ cần tạo ra được cùng kết quả đối với một xâu vào là được.

Một cây phân tích có các giá trị thuộc tính tại mỗi nút được gọi là *cây phân tích đánh dấu*. Quá trình tính giá trị thuộc tính tại các nút được gọi là *đánh dấu* hoặc *trang hoàng* cây phân tích.

1. Dạng của định nghĩa điều khiển dựa cú pháp

Trong một định nghĩa điều khiển dựa cú pháp, mỗi một sản xuất $A \rightarrow \alpha$ sẽ có tương ứng một tập các luật ngữ nghĩa có dạng $b = f(c_1, c_2, \dots, c_k)$ với f là một hàm. Có thể ở một trong hai dạng sau:

1. b là một thuộc tính tổng hợp của A và c_1, c_2, \dots, c_k là các thuộc tính theo các ký hiệu văn phạm của một sản xuất.
2. b là một thuộc tính kế thừa của một trong các ký hiệu văn phạm nằm bên vé phải của sản xuất và c_1, c_2, \dots, c_k là các thuộc tính theo các ký hiệu văn phạm của một sản xuất.

Ta nói là thuộc tính b phụ thuộc vào các thuộc tính c_1, c_2, \dots, c_k .

Một *văn phạm thuộc tính* là một định nghĩa điều khiển dựa cú pháp mà các luật ngữ nghĩa không gây tác động ra bên ngoài.

Các hàm trong các luật ngữ nghĩa thường được viết như những biểu thức. Đôi khi, kết quả của một luật ngữ nghĩa là tạo ra một hành động (gây tác động ra bên ngoài). Các luật ngữ nghĩa như vậy được viết như các lời gọi thủ tục hay các đoạn chương trình.

Ví dụ 6.1: Định nghĩa điều khiển dựa cú pháp như bảng dưới đây cho một chương trình bàn tính tay (máy tính con cầm tay dùng để thực hiện các phép tính đơn giản). Mỗi một ký hiệu không kết thúc E, T và F đều có một thuộc tính tổng hợp val kèm theo. Đối với các sản xuất E, T và F, luật ngữ nghĩa tính giá trị thuộc tính val cho các ký hiệu không kết thúc nằm ở vé trái từ các giá trị val của các ký hiệu không kết thúc nằm ở vé phải.

Sản xuất	Luật ngữ nghĩa
$L \rightarrow E \text{ n}$	$\text{print}(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} := E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} := T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} := T_1.\text{val} \times F.\text{val}$
$T \rightarrow F$	$T.\text{val} := F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} := E.\text{val}$
$F \rightarrow \text{số}$	$F.\text{val} := \text{số}.lexval$

Từ **tổ số** (ký hiệu kết thúc) có một thuộc tính tổng hợp *lexval* chính là giá trị của con số đó được tính nhờ bộ phân tích từ vựng. Ký hiệu **n** là ký hiệu dấu xuống dòng (enter), thủ tục *print* dùng để in ra màn hình kết quả.

Thuộc tính tổng hợp

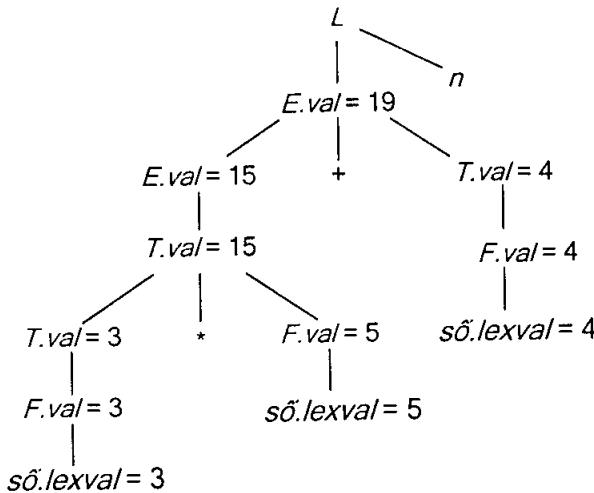
Các thuộc tính tổng hợp được dùng rộng rãi trong thực tế. Một định nghĩa điều khiển dựa cú pháp dùng các thuộc tính tổng hợp được gọi riêng là *định nghĩa thuộc tính -S*. Một cây phân tích cho một định nghĩa thuộc tính -S có thể luôn được đánh dấu (trang hoàng) bằng đánh giá các luật ngữ nghĩa đối với các thuộc tính tại mỗi nút từ dưới lên, từ lá đến gốc.

Ví dụ 6.2: Các định nghĩa điều khiển dựa cú pháp trong ví dụ 6.1 chính là các định nghĩa thuộc tính -S. Áp dụng cho biểu thức $3*5+4n$ ta sẽ được cây phân tích và được đánh giá như hình 6.2.

Thuộc tính kế thừa

Một thuộc tính kế thừa là thuộc tính mà giá trị của nó tại một nút trong cây phân tích được xác định từ các thuộc tính từ cha, các anh, em của nó hoặc kết hợp từ các nút đó. Thuộc tính kế thừa rất thuận tiện để nhấn mạnh sự phụ thuộc của một cấu trúc ngôn ngữ lập trình vào ngữ cảnh mà nó xuất hiện trong đó. Ví dụ, ta có thể dùng một thuộc tính kế thừa để theo dõi một tên xuất hiện trong vé trái hoặc phải của một phép gán nhằm biết là cần lấy địa chỉ hay giá trị của tên này. Mặc dù là ta có thể viết một định nghĩa điều khiển dựa cú pháp chỉ dùng toàn các thuộc tính tổng hợp thì đôi khi việc dùng các thuộc tính kế thừa lại dễ dàng và tự nhiên hơn.

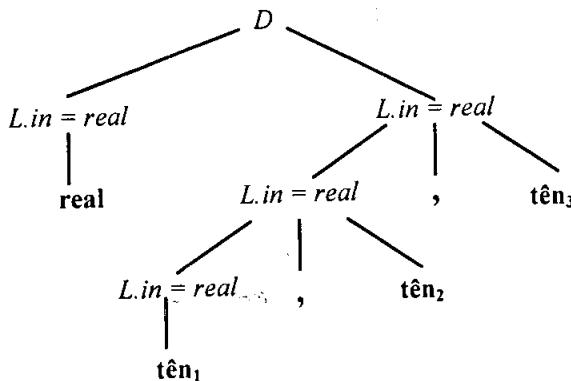
Ví dụ 6.3: Trong ví dụ này, một thuộc tính kế thừa cung cấp thông tin về kiểu cho các tên biến trong một khai báo. Ký hiệu không kết thúc D sinh ra một khai báo theo định nghĩa điều khiển dựa cú pháp trong bảng dưới bao gồm từ khóa **int** hoặc **real**, tiếp theo bảng một danh sách các tên. Ký hiệu không kết thúc T có một thuộc tính tổng hợp type, có giá trị được xác định bằng từ khóa trong khai báo. Luật ngữ nghĩa $L.in = T.Type$ ứng với sản xuất $D \rightarrow TL$, sẽ đặt thuộc tính kế thừa $L.in$ thành kiểu được khai báo. Sau đó, luật này sẽ được chuyển xuống theo cây phân tích qua thuộc tính kế thừa $L.in$. Các luật liên quan đến các sản xuất của L gọi thủ tục *addtype* để thêm kiểu của mỗi tên vào bảng ký hiệu.



Hình 6.2. Ví dụ về điều khiển dựa cú pháp.

Sản xuất	Luật ngữ nghĩa
$D \rightarrow TL$	$L.in = T.Type$
$T \rightarrow \text{int}$	$T.Type = \text{integer}$
$T \rightarrow \text{real}$	$T.Type = \text{real}$
$L \rightarrow L_1, \text{tên}$	$L_1.in = L.in$ $\text{addtype}(\text{tên}, \text{entry}, L.in)$
$L \rightarrow \text{tên}$	$\text{addtype}(\text{tên}, \text{entry}, L.in)$

Còn hình 6.3 áp dụng các thuộc tính kế thừa ở trên cho xâu vào phân tích real $\text{tên}_1, \text{tên}_2, \text{tên}_3$.



Hình 6.3. Áp dụng các thuộc tính kế thừa.

2. Đồ thị phụ thuộc

Nếu như một thuộc tính b tại một nút trong cây phân tích phụ thuộc vào một thuộc tính c , thì luật ngữ nghĩa cho b tại nút này phải được đánh giá sau luật xác định c . Các phụ thuộc lẫn nhau này giữa các thuộc tính kế thừa và tổng hợp tại các nút có thể biểu diễn bằng một đồ thị có hướng gọi là đồ thị phụ thuộc. Đồ thị này có một nút cho mỗi một luật ngữ nghĩa và một cạnh đến nút b từ nút c nếu thuộc tính b phụ thuộc vào thuộc tính c .

Chi tiết hơn, đồ thị phụ thuộc cho một cây phân tích cho trước được xây dựng như sau:

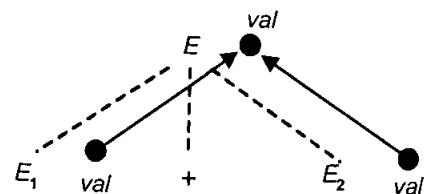
```

for mỗi nút  $n$  trong cây phân tích do
    for mỗi thuộc tính  $a$  của ký hiệu văn phạm tại  $n$  do
        dựng một nút của đồ thị phụ thuộc cho  $a$ ;
for mỗi nút  $n$  trong cây phân tích do
    for mỗi luật ngữ nghĩa  $b := f(c_1, c_2, \dots, c_k)$  ứng với sản xuất được
        dùng tại  $n$  do
        for  $i := 1$  to  $k$  do
            dựng một cạnh từ nút  $c_i$  đến nút  $b$ ;

```

Ví dụ 6.4: Dựa vào cây phân tích (nét đứt đoạn) và luật ngữ nghĩa ứng với sản xuất ở bảng dưới, ta thêm các nút và cạnh thành đồ thị phụ thuộc và được hình 6.4.

Sản xuất	Luật ngữ nghĩa
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$



Hình 6.4. Đồ thị phụ thuộc.

3. Thứ tự đánh giá thuộc tính

Có vài phương pháp khác nhau để đánh giá các luật ngữ nghĩa:

1. *Phương pháp cây phân tích:* Tại thời điểm dịch, chương trình có được thứ tự đánh giá dựa trên cấu trúc hình học của đồ thị phụ thuộc (được xây dựng từ cây phân tích cho mỗi xâu vào). Phương pháp này sẽ đưa ra thứ tự đánh giá sai nếu đồ thị phụ thuộc lại có một vòng lặp.

2. *Phương pháp dựa trên luật*: Lúc xây dựng chương trình dịch, các luật ngữ nghĩa ứng với các sản xuất được phân tích bằng tay hoặc một công cụ nào đó. Đối với mỗi sản xuất, thứ tự các thuộc tính tương ứng được đánh giá lại được xác định trước trong thời gian xây dựng chương trình dịch này.

3. *Phương pháp quên lãng*: Chương trình sẽ chọn thứ tự đánh giá mà không cần chú ý đến các luật ngữ nghĩa. Ví dụ, nếu việc chuyển đổi thực hiện trong lúc phân tích thì thứ tự đánh giá bị bắt buộc theo phương pháp phân tích và độc lập với các luật ngữ nghĩa. Phương pháp này chỉ cài đặt được trên một lớp hạn chế của điều khiển dựa cú pháp.

Các phương pháp dựa trên luật và quên lãng không cần đến cấu trúc đồ thị phụ thuộc trong lúc dịch nên chúng có thể tiết kiệm không gian và thời gian hơn trong khi dịch.

Sau đây ta sẽ xem xét một số phương pháp cụ thể.

3.1 Các bộ đánh giá thuộc tính dựa theo cây phân tích

Phương pháp duyệt cây phân tích từ trái sang phải

Giả sử cây phân tích đã được tạo rồi và đã được đánh nhãn. Ký hiệu khởi đầu có các thuộc tính kế thừa và tất cả các ký hiệu kết thúc có các thuộc tính tổng hợp. Phương pháp này sẽ duyệt cây theo một thứ tự nào đó, cho đến khi tất cả các thuộc tính đều được lượng giá. Thông thường thứ tự theo chiều sâu và từ trái sang phải. Nếu cần thiết thì cây có thể duyệt đi duyệt lại vài lần.

Dưới đây là thuật toán này:

```
while còn các thuộc tính phải đánh giá do
    Thăm_nút ( $S$ );      {  $S$  là nút khởi đầu }

procedure Thăm_nút (nút  $N$ )
begin
    if  $N$  là một ký hiệu không kết thúc then
        begin
            { Giả sử là có sản xuất  $N \rightarrow X_1..X_m$  }
            for  $i = 1$  to  $m$  do
                if  $X_i$  là một ký hiệu hoạt động hoặc không kết thúc then
                    begin
                        Đánh giá tất cả các thuộc tính kết thừa có thể được của  $X_i$ ;
                        Thăm_nút ( $X_i$ );
                    end;
            end;
    end;
end;
```

3.2 Các bộ đánh giá thuộc tính quên lãng

Thuộc tính -L

Một trường hợp thú vị là duyệt cây của những bộ phân tích duyệt một lần. Các văn phạm thuộc tính cho việc duyệt một lần từ trái sang phải luôn luôn cho phép đánh giá được tất cả các thuộc tính thì được gọi là *thuộc tính -L*. Chú ý văn phạm LL (1) là trường hợp này.

Một định nghĩa điều khiển dựa cú pháp là thuộc tính -L nếu mỗi thuộc tính kế thừa X_j , $1 \leq j \leq n$ trong vé phải của sản xuất $A \rightarrow X_1 X_2 \dots X_n$, chỉ phụ thuộc vào:

- Các thuộc tính của các ký hiệu X_1, X_2, \dots, X_{j-1} nằm bên trái X_j trong sản xuất.
- Các thuộc tính kế thừa của A .

Đánh giá thuộc tính "trên cùng chuyến bay"

Không giống như phép duyệt cây, đánh giá thuộc tính "trên cùng chuyến bay" thực hiện cùng với bộ phân tích chứ không phải sau nó. Do đó, đây là mô hình tốt cho các trình biên dịch duyệt một lần.

Do bộ đánh giá này liên kết với bộ phân tích nên nó có hai vấn đề cần quan tâm: bộ phân tích nào được sử dụng và kiểu thuộc tính được dùng.

a. Bộ đánh giá thuộc tính -L LL(1)

Lớp thuộc tính -L LL(1) rất phổ biến và mạnh. Thực hiện các luật thuộc tính -L ta có thể đánh giá được các thuộc tính kế thừa bên vé trái, sau đó các thuộc tính của vé phải. Công việc này hoạt động tốt với bộ phân tích LL (1) mà đầu tiên nó xác định được vé trái của một sản xuất, sau đó xác định được và khớp các ký hiệu bên vé phải của sản xuất này.

Ta giả sử là bộ đánh giá sẽ dùng một ngăn xếp chứa thuộc tính. Khi một ký hiệu không kết thúc được xác định, các thuộc tính kế thừa được đẩy vào ngăn xếp. Khi vé phải của sản xuất được xử lý, các thuộc tính kế thừa và tổng hợp của mỗi ký hiệu vé phải này cũng được đẩy vào. Cuối cùng, khi toàn bộ vé phải được xử lý xong thì toàn bộ các thuộc tính của vé phải được lấy ra, và các thuộc tính tổng hợp của vé trái được đẩy vào.

Như vậy, nếu một luật $X \rightarrow YZ$ được dùng trong lúc phân tích, ngăn xếp thuộc tính được tính như sau, trong đó ký hiệu *Inh* chỉ thuộc tính kế thừa, *Syn* chỉ thuộc tính tổng hợp:

(1) Đẩy các thuộc tính kế thừa của X :

Stack = ... Inh(X)

(2) Đẩy các thuộc tính kế thừa của Y :

Stack = ... Inh(X) Inh(Y)

(3) Đẩy các thuộc tính tổng hợp của Y (sau khi phân tích Y):

Stack = ... Inh(X) Inh(Y) Syn(Y)

(4) Đẩy các thuộc tính kế thừa của Z :

Stack = ... Inh(X) Inh(Y) Syn(Y) Inh(Z)

(5) Đẩy các thuộc tính tổng hợp của Z (sau khi phân tích Z):

Stack = ... Inh(X) Inh(Y) Syn(Y) Inh(Z) Syn(Z)

(6) Lấy ra các thuộc tính về phải và đặt các thuộc tính tổng hợp của X :

Stack = ... Inh(X) Syn(X)

Do đây là thuộc tính -L nên toàn bộ các giá trị thuộc tính sẽ được đặt lên ngăn xếp và vị trí của chúng đều xác định được.

b. Đánh giá dưới lên của các định nghĩa thuộc tính tổng hợp

Các thuộc tính -S có thể được đánh giá bằng một bộ phân tích dưới lên (bottom-up) - như dùng trong phân tích LR, giống như khi phân tích xâu vào. Bộ phân tích này có thể lưu các giá trị thuộc tính tổng hợp tương ứng với các ký hiệu văn phạm trong ngăn xếp của nó. Mỗi khi thực hiện một thu gọn, các giá trị thuộc tính tổng hợp mới lại được tính từ các thuộc tính xuất hiện trong ngăn xếp cho các ký hiệu văn phạm ở về phải sản xuất thu gọn.

Mở rộng ngăn xếp bộ phân tích cho thuộc tính tổng hợp

Một bộ phân tích bottom -up dùng một ngăn xếp để lưu thông tin về các cây con vừa được phân tích. Ta sẽ thêm các trường trong ngăn xếp để lưu các giá trị của các thuộc tính tổng hợp. Hình 6.5 cho thấy ví dụ của một ngăn xếp của bộ phân tích với khoảng trống chứa cho một giá trị thuộc tính. Giả sử ngăn xếp này được thực hiện bằng một cặp các mảng *state* và *val*. Mỗi một vị trí trong *state* là một con trỏ (hoặc chỉ số) đến một bảng phân tích LR (1) (chú ý là ký hiệu văn phạm là không chỉ rõ trong trạng thái và không cần phải lưu trong ngăn xếp). Nếu ký hiệu trạng thái là A thì *val*[i] sẽ lưu giá trị của thuộc tính tương ứng với nút cây phân tích ứng với A này.

state	val
...	...
X	X.x
Y	Y.y
Z	Z.z
...	...

Hình 6.5. Ngăn xếp.

Định hiện tại của ngăn xếp được chỉ bằng con trỏ *top*. Ta coi rằng các thuộc tính tổng hợp được đánh giá ngay trước khi thu gọn. Giả sử luật ngữ nghĩa $A.a := f(X.x, Y.y, Z.z)$ là ứng với sản xuất $A \rightarrow XYZ$. Trước khi XYZ được thu gọn về A , giá trị thuộc tính $Z.z$ nằm tại $val[top]$, còn $Y.y$ nằm tại $val[top-1]$ và $X.x$ nằm tại $val[top-2]$. Nếu một ký hiệu không có thuộc tính thì vị trí tương ứng trong mảng *val* sẽ không được xác định. Sau khi thu gọn *top* sẽ bị giảm đi 2. Trạng thái *A* được đặt vào *state[top]* (tức là vị trí của *X*) và giá trị của thuộc tính tổng hợp $A.a$ được đặt vào *val[top]*.

III. LƯỢC ĐỒ CHUYỂN ĐỔI

Một lược đồ chuyển đổi là một văn phạm phi ngữ cảnh có các thuộc tính ứng với các ký hiệu văn phạm và các hoạt động ngữ nghĩa được đóng trong cặp dấu ngoặc nhọn {} và được chèn vào về phải của các sản xuất.

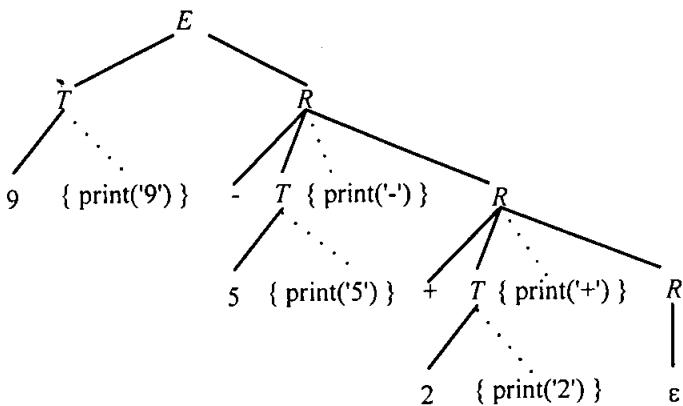
Lược đồ chuyển đổi là những ký pháp hữu ích cho việc chuyển đổi trong lúc đang phân tích. Nó có thể bao gồm các thuộc tính tổng hợp và kế thừa.

Ví dụ 6.5: Đây là một lược đồ chuyển đổi đơn giản để chuyển đổi một biểu thức cộng trừ ở dạng thông thường thành dạng hậu tố.

$$E \rightarrow TR$$

$$\begin{aligned} R &\rightarrow addop T \{ print (addop.lexeme) \} R_1 \mid \epsilon \\ T &\rightarrow num \{ print (num.val) \} \end{aligned}$$

Hình 6.6 cho thấy cây phân tích đối với xâu vào 9-5+2. Kết quả in ra là 9 5 - 2 +.



Hình 6.6. Cây phân tích cho $9-5+2$ thực hiện hành động.

Khi thiết kế một lược đồ chuyển đổi, ta phải khắc phục một số hạn chế để khăng định là một giá trị thuộc tính là dùng được khi một hành động cần đến nó. Các hạn chế này, được khắc phục bằng định nghĩa thuộc tính $-L$, khăng định một hoạt động sẽ không dùng thuộc tính chưa được tính.

Trường hợp đơn giản nhất là khi chỉ dùng các thuộc tính tổng hợp, ta có thể xây dựng được lược đồ chuyển đổi bằng cách tạo một hành động chứa một lệnh gán cho mỗi luật ngữ nghĩa, và thay hành động này tại cuối về phải của sản xuất tương ứng. Ví dụ, luật sản xuất và ngữ nghĩa:

Sản xuất	Luật ngữ nghĩa
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

cho sản xuất và hoạt động ngữ nghĩa như sau:

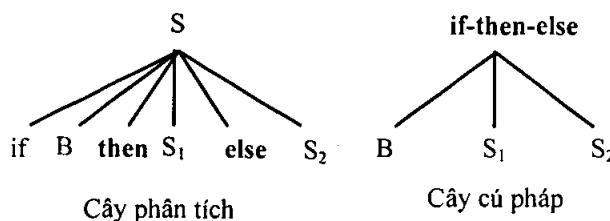
$$T \rightarrow T_1 * F \{ T.val := T_1.val \times F.val \}$$

IV. DỤNG CÂY CÚ PHÁP

Trong phần này, ta xem định nghĩa điều khiển dựa cú pháp có thể dùng để xây dựng một cây cú pháp và các biểu diễn đồ họa khác của các cấu trúc ngữ nghĩa như thế nào.

Việc dùng cây cú pháp làm một biểu diễn trung gian cho phép việc diễn giải được tách rời khỏi phân tích. Các thủ tục chuyển đổi đòi hỏi trong lúc phân tích phải bị hai kiểu giới hạn. Đầu tiên, một văn phạm thích hợp cho phân tích có thể không phản ánh cấu trúc phân cấp tự nhiên của các cấu trúc trong ngôn ngữ. Ví dụ, văn phạm của Fortran có thể cho thấy các thủ tục con chỉ đơn giản là một danh sách các câu lệnh. Có thể việc phân tích các thủ tục con này dễ hơn nếu ta dùng một biểu diễn cây phản ánh được các

Vòng lặp do. Thứ hai, phương pháp phân tích cưỡng ép thứ tự xét các nút trong cây. Thứ tự này có thể không khớp với thứ tự mà thông tin về cấu trúc trở thành dùng được. Vì lý do này các chương trình dịch cho ngôn ngữ C thường xây dựng cây cú pháp cho các khai báo.



Hình 6.7. So sánh hai loại cây.

Cây cú pháp

Một cây cú pháp là một dạng cô đọng của cây phân tích nhằm giúp cho việc biểu diễn các cấu trúc ngữ nghĩa. Sản xuất $M \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$ có thể đặt trong một cây phân tích và cây cú pháp như hình 6.7 (trong đó, if, then, else là các ký hiệu kết thúc).

Trong một cây cú pháp, các toán tử và từ khoá không xuất hiện như là các lá mà lại trở thành các nút trong ứng với cha của các lá liên quan. Một khác biệt quan trọng nữa trong cây cú pháp là chuỗi các sản xuất tạo cây có thể bị mất đi. Hình 6.7 cho thấy một cây cú pháp của cây phân tích trong ví dụ 6.2.

Xây dựng cây cú pháp cho biểu thức

Xây dựng cây cú pháp cho một biểu thức tương tự như chuyển đổi biểu thức thành dạng hậu tố. Ta xây dựng các cây con cho các biểu thức con bằng cách tạo ra một nút ứng với toán tử và toán hạng. Các con của một toán tử lại là các gốc của các nút biểu diễn các biểu thức con cấu trúc nên các toán hạng của toán tử này.

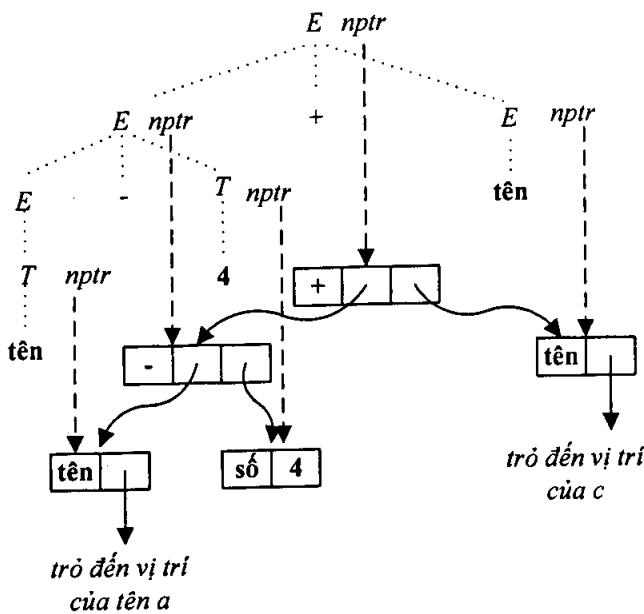
Mỗi nút trong một cây cú pháp có thể được xây dựng như một bản ghi với vài trường. Đôi với nút cho toán tử, có một trường xác định phép toán và các trường còn lại trỏ đến các nút cho toán hạng. Toán tử này đôi khi được gọi là nhãn của nút. Các nút trong một cây cú pháp có thể có thêm các trường nữa để lưu các giá trị (hoặc con trỏ đến giá trị) của các thuộc tính gắn với nút này. Trong phần này ta sẽ dùng các hàm sau để tạo ra các nút của cây cú pháp cho biểu thức dùng các toán tử hai ngôi.

Mỗi hàm sẽ trả lại một con trỏ đến một nút mới được tạo ra.

1. $mknode(op, left, right)$ tạo một nút toán tử với nhãn op và hai trường chứa các con trỏ left và right.
2. $mkleaf(id, entry)$ tạo một nút tên với nhãn id và một trường chứa con trỏ entry trỏ đến tên này đặt trong bảng ký hiệu.
3. $mkleaf(num, val)$ tạo một nút tên với nhãn num và một trường chứa val là giá trị của số đó.

Ví dụ 6.6: Sau đây là một định nghĩa điều khiển dựa cú pháp để xây dựng cây cú pháp cho các biểu thức dùng các phép + và -. Đây đều là các thuộc tính tổng hợp với một thuộc tính nptr của E và T là chứa con trỏ do các hàm mknnode và mkleaf trả về.

Sản xuất	Luật ngữ nghĩa
$E \rightarrow E_1 + T$	$E.nptr := mknode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr := mknode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow (E)$	$T.nptr := E.nptr$
$T \rightarrow \text{tên}$	$T.nptr := mkleaf(\text{tên}, \text{tên}.entry)$
$T \rightarrow \text{số}$	$T.nptr := mkleaf(\text{số}, \text{số}.val)$



Hình 6.8. Xây dựng cây cú pháp cho biểu thức $a - 4 + c$.

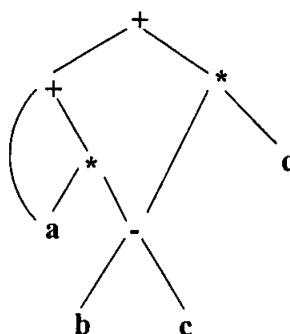
Hình 6.8 là một cây cú pháp được xây dựng từ cây phân tích của biểu thức $a - 4 + c$.

Đồ thị DAG

Đồ thị Dag (Directed Acyclic graph - đồ thị định hướng không chu trình) cho một biểu thức cũng giống như cây cú pháp, nó có một nút ứng với các biểu thức con trong biểu thức; một nút trong biểu diễn một phép toán và con của nó biểu diễn các toán hạng. Điểm khác là một nút trong dag biểu diễn một biểu thức con chung có thể có nhiều cha.

Hình 6.9 là đồ thị dag cho biểu thức $a + a * (b - c) + (b - c) * d$.

Lá a có hai cha chung vì a là chung trong hai biểu thức a và $a * (b - c)$. Tương tự như vậy $b - c$ cũng có hai cha chung.



Hình 6.9. Dag của biểu thức $a+a*(b-c)+(b-c)*d$.

Bài tập

1. Dựa vào ví dụ 6.1, hãy dựng một cây phân tích cho biểu thức vào $(4 * 7 + 1) * 2$.

2. Dựng cây phân tích và cây cú pháp cho biểu thức $((a) + (b))$ dựa theo:

- Định nghĩa cú pháp - điều khiển như ví dụ 6.6.

- Lược đồ chuyển đổi.

3. Xây dựng đồ thị dag cho biểu thức $a + a + (a + a + a(a + a + a + a))$.

4. Vấn phạm sau dùng để xây dựng các biểu thức sử dụng các thuật toán + giữa các hằng số nguyên (int) và thực (real)). Nếu hai số nguyên được cộng với nhau thì kiểu kết quả là một số nguyên, nếu khác vậy thì là số thực.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow \text{số} \cdot \text{số} \mid \text{số}$$

- Hãy đưa ra một định nghĩa điều khiển dựa cú pháp để xác định kiểu của mỗi biểu thức con.
- Mở rộng câu trên để chuyển biểu thức đó về dạng ký pháp hậu tố ngay trong khi xác định kiểu. Sử dụng phép toán inttoreal để chuyển đổi một giá trị nguyên thành giá trị thực tương ứng để thực hiện phép + với các toán hạng cùng kiểu.

5. Sau đây là một văn phạm dùng cho các khai báo đầu chương trình:

$$D \rightarrow \text{tên } L$$

$$L \rightarrow , \text{tên } L \mid : T$$

$$T \rightarrow \text{integer} \mid \text{real}$$

Hãy dụng một lược đồ chuyển đổi để nhập kiểu của các tên vào bảng ký hiệu.

Bài tập thực hành

Chọn một phương pháp gắn điều khiển dựa cú pháp với các luật cú pháp và viết chương trình đánh giá chúng.

Chương 7

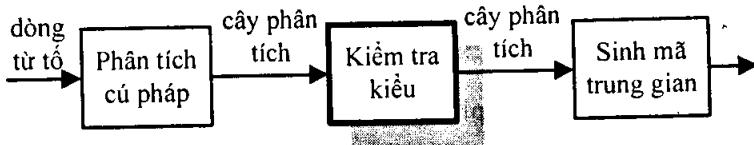
PHÂN TÍCH NGỮ NGHĨA

I. NHIỆM VỤ

Một chương trình dịch phải kiểm tra xem chương trình nguồn có theo các quy định về cú pháp và ngữ nghĩa của ngôn ngữ nguồn hay không. Việc kiểm tra này, gọi là kiểm tra tĩnh (phân biệt với kiểm tra động thực hiện trong quá trình chương trình đích chạy), đảm bảo sẽ phát hiện và mô tả các lỗi về kiểu. Ví dụ, kiểm tra tĩnh có thể gồm:

1. *Kiểm tra kiểu.* Một chương trình dịch phải báo lỗi nếu một toán tử được dùng cho các toán hạng không tương thích; ví dụ, khi ta cộng một xâu với một số nguyên trong ngôn ngữ lập trình C.
2. *Kiểm tra dòng điều khiển.* Các câu lệnh làm thay đổi dòng điều khiển từ một khối phải có một vài nơi để chuyển điều khiển đến đó. Ví dụ, lệnh *break* trong C làm dòng điều khiển thoát khỏi vòng lặp *while*, *for*, hoặc *switch* gần nhất: sẽ có lỗi nếu như các vòng lặp đó lại không có.
3. *Kiểm tra tính nhất quán.* Có những hoàn cảnh mà một đối tượng được định nghĩa chỉ đúng một lần. Ví dụ, trong Pascal, một định danh (tên) phải được khai báo là duy nhất, các nhãn trong *case* phải khác nhau và các phần tử trong một kiểu vô hướng có thể không được lặp lại.
4. *Kiểm tra quan hệ tên.* Đôi khi, cùng một tên phải xuất hiện từ hai lần trở lên. Ví dụ, trong Assembly, một chương trình con có một tên mà chúng phải xuất hiện ở đầu và cuối của chương trình con này. Chương trình dịch sẽ kiểm tra tên đó phải được dùng ở cả hai nơi.

Trong chương này chúng ta chỉ xem xét kiểm tra kiểu. Nhiều chương trình dịch như Pascal kết hợp phần kiểm tra tĩnh và phần sinh mã trung gian vào phần phân tích cú pháp. Một số chương trình dịch khác như Ada thì lại tách riêng phần kiểm tra kiểu làm việc như hình 7.1.



Hình 7.1. Vị trí của khối Kiểm tra kiểu.

Một bộ kiểm tra kiểu sẽ kiểm tra xem kiểu của một cấu trúc có đúng với kiểu mong muốn có theo ngữ cảnh của nó hay không. Ví dụ, phép toán số học *mod* trong Pascal cần hai toán tử là số nguyên, nên bộ kiểm tra kiểu phải kiểm tra các toán hạng trong *mod* phải là số nguyên. Tương tự, bộ kiểm tra kiểu phải kiểm tra các giá trị tham biến phải là con trỏ...

Các thông tin về kiểu có sau khi kiểm tra kiểu có thể cần thiết để sinh mã. Ví dụ, các phép toán số học như phép cộng thường được dùng cho cả số nguyên, số thực, hoặc một số kiểu khác, và chúng ta phải xét xem ngữ cảnh của phép cộng để biết được mục đích dùng nó. Một ký hiệu mà có thể đưa ra các phép toán khác nhau trong những hoàn cảnh khác nhau được gọi *định nghĩa chồng* "overloaded".

II. CÁC HỆ THỐNG KIỂU

1. Các hệ thống kiểu

Thiết kế một bộ phân tích kiểu cho một ngôn ngữ phải bắt nguồn từ thông tin về các cấu trúc cú pháp của ngôn ngữ này, các ký hiệu của các kiểu, và các luật để gán các kiểu vào các cấu trúc của ngôn ngữ. Ví dụ trong Pascal và C:

- Nếu cả hai toán hạng trong các phép toán số học cộng, trừ và nhân là các số nguyên (*integer*) thì kết quả là số nguyên.
- Kết quả của một phép toán đơn & là một con trỏ đến một đối tượng. Nếu kiểu của toán hạng là '...' thì kiểu của kết quả là 'con trỏ trỏ đến ...'.

Trong cả Pascal và C, các kiểu có thể là kiểu cơ sở hoặc do người lập trình xây dựng nên. Các kiểu cơ sở là các kiểu nguyên tố không có cấu trúc nội tại như các kiểu xây dựng nên. Nhiều ngôn ngữ lập trình chia phép xây dựng các kiểu mới từ các kiểu cơ sở hoặc từ các kiểu đã xây dựng khác.

Kiểu biểu thức

Kiểu của một cấu trúc ngôn ngữ sẽ được gọi là "kiểu biểu thức". Một kiểu biểu thức có thể là một kiểu cơ sở hoặc dùng các toán tử gọi là toán tử

xây dựng kiều để lập thành. Tập các kiều cơ sở và xây dựng tuỳ thuộc vào từng ngôn ngữ và sẽ được kiểm tra.

Chương này sử dụng các định nghĩa sau đây của kiều biểu thức:

1. Một kiều cơ sở là một kiều biểu thức. Các kiều cơ sở bao gồm *boolean*, *char*, *integer*, *real*... Một kiều cơ sở đặc biệt *type_error* được dùng để đánh dấu lỗi trong khi kiểm tra kiều. Cuối cùng là kiều *void* ký hiệu cho "một giá trị thiếu" cho phép kiểm tra được các câu lệnh.
2. Kiểu biểu thức có thể được đặt tên khác, một kiểu tên cũng là kiểu biểu thức.
3. Một toán tử xây dựng kiểu áp dụng vào một kiểu biểu thức là kiểu biểu thức. Các toán tử này bao gồm:
 - a. Mảng (array). Nếu T là một kiểu biểu thức, thì $array(I..T)$ là một kiểu biểu thức ký hiệu cho kiểu của một mảng với các phần tử có kiểu T và tập chỉ số I . I thường là một đoạn số nguyên. Ví dụ, trong Pascal khai báo như sau:

```
var A:array[1..10] of integer;
```

thì kiểu biểu thức của A là $array(1..10, integer)$.

- b. Tích. Nếu T_1 và T_2 là các kiểu biểu thức, thì tích Đề các của chúng $T_1 \times T_2$ là một kiểu biểu thức.
- c. Bản ghi (record). Kiểu của một bản ghi chính là trường hợp tích của các kiểu trong trường của nó. Sự khác nhau giữa một bản ghi và một tích là các miền của bản ghi thì có tên. Ví dụ, trong Pascal:

```
type row = record
```

```
    address: integer;
```

```
    lexeme: array[1..15] of char;
```

```
end;
```

```
var table: array[1..101] of row;
```

khai báo một tên kiểu row có kiểu biểu thức là:

```
record((address <integer>) <lexeme >array(1..15, char)))
```

và biến $table$ là một mảng các bản ghi có kiểu này.

- d. Con trỏ (pointer). Nếu T kiểu biểu thức, thì $\text{pointer}(T)$ là một kiểu biểu thức ký hiệu cho kiểu "con trỏ tới một đối tượng T ". Ví dụ trong Pascal, khai báo sau:

```
var p: ^row;
```

khai báo biến p có kiểu con trỏ pointer (row).

- e. Hàm (function). Về mặt toán học, một hàm ánh xạ các phần tử của một tập (miền-domain) vào một tập khác (phạm vi -range). Như vậy chúng ta có thể coi các hàm trong một ngôn ngữ lập trình như là một ánh xạ từ một kiểu miền (domain type) D vào một kiểu phạm vi (range type) R . Kiểu của một hàm như vậy sẽ được ký hiệu bằng một kiểu biểu thức $D \rightarrow R$. Ví dụ, hàm xây dựng sẵn mod của Pascal có miền là $\text{int} \times \text{int}$, nghĩa là một cặp số nguyên, và kiểu phạm vi là int . Như vậy, ta nói hàm mod có kiểu là:

$$\text{int} \times \text{int} \rightarrow \text{int}$$

Một ví dụ khác, khai báo:

```
function f(a, b: char): ^integer;
```

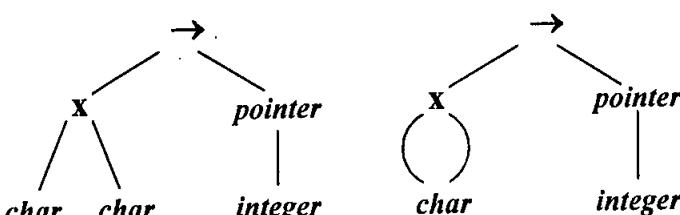
nói rằng kiểu miền của f được ký hiệu là $\text{char} \times \text{char}$ và kiểu phạm vi là pointer (integer). Như vậy, kiểu của f được ký hiệu bởi một kiểu biểu thức như sau:

$$\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$$

Đôi khi vì lý do kỹ thuật, kiểu trả về của một hàm bị hạn chế trong một số ngôn ngữ.

4. Kiểu biểu thức có thể chứa các biến mà giá trị của chúng là các kiểu biểu thức.

Một cách thuận tiện để biểu diễn kiểu biểu thức là dùng đồ thị cú pháp hoặc dag. Các đồ thị này có các nút chính là kiểu hợp thành, các lá là các kiểu cơ sở, tên kiểu hoặc biến kiểu. Ví dụ về cây cú pháp và dag ở hình 7.2.



Hình 7.2. Cây cú pháp và dag cho biểu thức
 $\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$.

Kiểu hệ thống

Một *kiểu hệ thống* là một tập các luật để gán kiểu biểu thức vào các phần khác nhau của chương trình. Một bộ kiểm tra kiểu thực hiện một kiểu hệ thống. Kiểu hệ thống thực chất chỉ là các định nghĩa cú pháp điều khiển mà ta đã xét trong chương trước. Do đó ta đã nắm được các kỹ thuật xử lý này. Ở đây, ta chỉ đi sâu vào một số chi tiết thực hiện.

Các kiểu hệ thống khác nhau có thể được dùng trong các chương trình dịch khác nhau hay trong các bộ xử lý khác nhau của cùng ngôn ngữ. Ví dụ, trong Pascal, kiểu của một mảng cùng với tập chỉ số của mảng, nên một hàm có một tham số mảng có thể được dùng cho mảng với các chỉ số đó. Nhiều chương trình dịch Pascal, cho phép không phải chỉ ra các tập chỉ số khi một mảng được truyền như một tham số. Như thế, các chương trình dịch này dùng một kiểu hệ thống hơi khác định nghĩa của ngôn ngữ Pascal. Tương tự, trong UNIX dùng với C, lệnh *lint* được dùng với nhiều kiểu hệ thống hơn là chương trình dịch C dùng.

Kiểm tra kiểu theo phương pháp tĩnh và động

Việc kiểm tra của chương trình dịch được gọi là **tĩnh**, còn việc kiểm tra thực hiện trong khi chương trình đích chạy được gọi là **động**. Về nguyên tắc, mọi phép kiểm tra đều có thể là động, nếu như mã chương trình nguồn có chứa kiểu của một phần tử cùng với giá trị của nó.

Một kiểu hệ thống đúng đắn sẽ xoá bỏ sự cần thiết kiểm tra động vì nó cho phép chúng ta xác định tĩnh rằng, các lỗi sẽ không xuất hiện trong lúc chương trình đích chạy. Tức là, nếu kiểu hệ thống đúng đắn được gán một kiểu khác *type_error* ở trong một đoạn chương trình thì lỗi kiểu không thể xuất hiện khi ta chạy đoạn chương trình đó. Một ngôn ngữ gọi là **định kiểu mạnh** nếu chương trình dịch của nó có thể bảo đảm rằng các chương trình mà nó dịch tốt sẽ hoạt động không có lỗi về kiểu.

Trong thực tế, một số kiểm tra chỉ có thể thực hiện động. Ví dụ, nếu chúng ta khai báo:

```
table: array[0..255] of char;  
i: integer;
```

và sau đó tính *table[i]*, thì một chương trình đích không thể bảo đảm rằng, trong quá trình hoạt động, giá trị của *i* sẽ chỉ nằm trong khoảng từ 0 đến 255 mà thôi.

Khắc phục lỗi

Điều quan trọng khi bộ kiểm tra kiểu phát hiện được lỗi, nó sẽ phải khắc phục lỗi để tiếp tục kiểm tra phần chương trình nguồn còn lại. Trước hết, nó phải thông báo về lỗi, mô tả và vị trí lỗi. Lỗi xuất hiện gây ảnh hưởng đến các luật kiểm tra lỗi, do vậy phải thiết kế kiểu hệ thống như thế nào để các luật có thể đương đầu với các lỗi này.

2. Ví dụ về một bộ kiểm tra kiểu đơn giản

Trong phần này, chúng ta xem xét một bộ kiểm tra kiểu cho một ngôn ngữ đơn giản mà kiểu của các biến phải được khai báo trước khi dùng. Bộ kiểm tra kiểu này là một cú pháp điều kiện dạng lược đồ chuyển đổi nhằm tổng hợp kiểu của từng biểu thức từ các kiểu của các biểu thức con của nó. Bộ kiểm tra kiểu này có thể làm việc với các mảng, các con trỏ, câu lệnh và các hàm.

Một ngôn ngữ đơn giản

Một văn phạm dưới đây sinh ra các chương trình, biểu diễn bởi biến P , bao gồm một chuỗi các khai báo D theo sau một biểu thức đơn E , các kiểu T .

$$P \rightarrow D ; E$$

$$D \rightarrow D ; D \mid \text{tên} : T$$

$$T \rightarrow \text{char} \mid \text{integer} \mid \text{array [số s] of } T \mid {}^T$$

$$E \rightarrow \text{chữ cái} \mid \text{số} \mid \text{tên} \mid E \text{ mod } E \mid E [E] \mid E {}^{\wedge}$$

Một chương trình (hoàn chỉnh) có thể sinh ra từ văn phạm trên như sau:

key: integer;

key mod 1999

Trước khi đề cập đến các biểu thức, ta xét các kiểu trong ngôn ngữ. Ngôn ngữ này bản thân có hai kiểu cơ sở, *char* và *integer*; một kiểu thứ ba là *type_error* được dùng để đánh dấu lỗi. Để đơn giản, chúng ta coi rằng tất cả các mảng bắt đầu bằng 1.

Ví dụ 7.1:

array[256] of char

Giống như trong Pascal, tiền tố ${}^{\wedge}$ là khai báo để xây dựng kiểu con trỏ, nên

${}^{\wedge}\text{integer}$

dẫn đến kiểu biểu thức *pointer(integer)*.

Ta có lược đồ chuyển đổi như sau:

$$P \rightarrow D ; E$$

$$D \rightarrow D ; D$$

$$D \rightarrow \text{tên: } T \quad \{ \text{addtype}(\text{tên.entry}, T.type) \}$$

$$T \rightarrow \text{char} \quad \{ T.type := \text{char} \}$$

$$T \rightarrow \text{integer} \quad \{ T.type := \text{integer} \}$$

$$T \rightarrow {}^{\wedge}T_1 \quad \{ T.type := \text{pointer}(T_1.type) \}$$

$$T \rightarrow \text{array [số s] of } T_1 \{ T.type := \text{array}(s\delta.\text{val}, T_1.type) \}$$

hành động ứng với sản xuất $D \rightarrow \text{tên: } T$ lưu vào bảng ký hiệu một kiểu cho một tên. Hàm *addtype(tên.entry, T.type)* có nghĩa là ta cất một thuộc tính *T.type* vào bảng ký hiệu ở vị trí *entry*.

Kiểm tra kiểu của các biểu thức

Trong các luật sau, kiểu của thuộc tính tổng hợp của *E* cho kiểu biểu thức được gán bằng kiểu hệ thống để sinh biểu thức bởi *E*. Các luật ngữ nghĩa sau cho thấy các hằng số biểu diễn bằng từ tố **chữ cái** và **số** có kiểu *char* và *integer*:

$$E \rightarrow \text{chữ cái} \quad \{ E.type := \text{char} \}$$

$$E \rightarrow \text{số} \quad \{ E.type := \text{integer} \}$$

Ta dùng một hàm *lookup(e)* để lấy kiểu cất trong bảng ký hiệu trả bởi *e*. Khi một tên xuất hiện trong một biểu thức, kiểu khai báo của nó được lấy và gán cho thuộc tính kiểu:

$$E \rightarrow \text{tên} \quad \{ E.type := \text{lookup}(\text{tên.entry}) \}$$

Một biểu thức được tạo bởi áp dụng lệnh mod cho hai biểu thức con có kiểu *integer* thì nó cũng có kiểu là *integer*; nếu không là kiểu *type_error*:

$$E \rightarrow E_1 \text{ mod } E_2 \quad \{ E.type := \text{if } E_1.type = \text{integer and} \}$$

E₂.type = integer then integer

else type_error }

Đối với một mảng *E₁ [E₂]*, biểu thức chỉ số *E₂* phải có kiểu là số *integer*, các phần tử của mảng có kiểu *t* chính là kiểu *array(s, t)* của *E₁*:

$$E \rightarrow E_1 [E_2] \quad \{ E.type := \text{if } E_2.type = \text{integer} \text{ and } E_1.type = \text{array}(s, t) \\ \text{then } t \text{ else type_error } \}$$

Đối với thuật toán lấy giá trị con trỏ:

$$E \rightarrow E_1^{\wedge} \quad \{ E.type := \text{if } E_1.type = \text{pointer}(t) \text{ then } t \text{ else type_error } \}$$

Kiểm tra kiểu của các câu lệnh

Đối với các câu lệnh không có giá trị, ta có thể gán cho nó một kiểu cơ sở đặc biệt *void*. Nếu có một lỗi được phát hiện trong câu lệnh thì kiểu của câu lệnh được gán là *type_error*.

Các câu lệnh bao gồm các câu lệnh gán, điều kiện và vòng lặp while. Chuỗi các câu lệnh được phân cách nhau bằng dấu chấm phẩy. Một chương trình hoàn chỉnh có luật dạng $P \rightarrow D ; S$ cho biết một chương trình bao gồm các khai báo và sau là các câu lệnh; các luật kiểm tra biểu thức ở trên vẫn cần ở đây do các câu lệnh có thể có các biểu thức ở trong.

$$S \rightarrow \text{tên} := E \quad \{ S.type := \text{if } \text{tên}.type = E.type \text{ then void} \\ \text{else type_error } \}$$

$$S \rightarrow \text{if } E \text{ else } S_1 \quad \{ S.type := \text{if } E.type = \text{boolean} \text{ then} \\ S_1.type \text{ else type_error } \}$$

$$S \rightarrow \text{while } E \text{ do } S_1 \quad \{ S.type := \text{if } E.type = \text{boolean} \text{ then} \\ S_1.type \text{ else type_error } \}$$

$$S \rightarrow S_1 ; S_2 \quad \{ S.type := \text{if } S_1.type = \text{void} \text{ and} \\ S_2.type = \text{void} \text{ then void} \\ \text{else type_error } \}$$

Kiểm tra kiểu của các hàm

Các hàm với các tham số có sản xuất dạng:

$$E \rightarrow E (E)$$

Các luật ứng với kiểu biểu thức của ký hiệu không kết thúc T có thể làm thừa số theo các sản xuất sau.

$$T \rightarrow T_1 \rightarrow T_2 \quad \{ T.type := T_1.type \rightarrow T_2.type \}$$

Luật kiểm tra kiểu của một hàm là:

$$E \rightarrow E_1 (E_2) \quad \{ E.type := \text{if } E_2.type = s \text{ and}$$

$E_1.type = s \rightarrow t$ then t

else type_error }

Luật này cho biết trong một biểu thức được tạo bằng cách áp dụng E_1 vào E_2 , kiểu của $s \rightarrow t$ phải là một hàm từ kiểu của s vào kiểu nào đó t . Kiểu E_1 (E_2) là t .

III. MỘT SỐ VẤN ĐỀ KHÁC CỦA KIỂM TRA KIỂU

1. Sự tương đương của kiểu biểu thức

Nhiều luật kiểm tra trong phần cuối có dạng " if kiểu của hai biểu thức giống nhau then trả về kiểu đó else trả về type_error". Như vậy điều quan trọng là xác định chính xác khi nào thì hai kiểu biểu thức là tương đương. Vấn đề khó khăn hơn do một kiểu có thể được đặt bằng một tên, sau đó tên này được gán cho các biểu thức nhỏ hơn. Chìa khoá ở đây là một tên trong một biểu thức kiểu viết tắt cho bản thân nó hoặc là một viết tắt cho một kiểu biểu thức khác.

Hàm sau đây dùng để kiểm tra sự tương đương về cấu trúc của kiểu biểu thức:

```

function sequiv(s, t): boolean;
begin
    if s và t là cùng kiểu cơ sở then
        return true;
    else if s = array(s1, s2) and t = array(t1, t2) then
        return sequiv(s1, t1) and sequiv(s2, t2);
    else if s = s1 x s2 and t = t1 x t2 then
        return sequiv(s1, t1) and sequiv(s2, t2);
    else if s = pointer(s1) and t = pointer(t1) then
        return sequiv(s1, t1);
    else if s = s1 → s2 and t = t1 → t2 then
        return sequiv(s1, t1) and sequiv(s2, t2);
    else
        return false;
end;

```

2. Đổi kiểu

Ta xem xét một biểu thức dạng $x+i$ với x có kiểu là *real* và i là *integer*. Biểu diễn của *real* và *integer* trong máy tính là khác nhau, đồng thời, cách thực hiện phép cộng đối với số *integer* và đổi với số *real* cũng khác nhau.

Trong thực tế, để thực hiện được phép cộng này, trước tiên, chương trình dịch sẽ đổi cả hai toán tử này về cùng một kiểu (cụ thể ở đây là kiểu *real*), sau đó thực hiện phép cộng.

Trong các ngôn ngữ, việc chuyển kiểu như trên là rất cần thiết. Bộ kiểm tra kiểu trong một chương trình dịch có thể được dùng để chèn thêm các phép toán vào các biểu diễn trung gian của chương trình nguồn. Ví dụ, ký hiệu cho $x+i$ có thể được chèn thêm phép toán *inttoreal* dùng để chuyển một số *integer* thành *real* rồi mới thực hiện phép cộng số thực *real+* như sau:

$x \ i \ inttoreal \ real+$

Ép kiểu

Một phép đổi kiểu từ một kiểu sang kiểu khác được gọi là *không rõ* (*ẩn*) nếu nó thực hiện một cách tự động bởi chương trình dịch. Các phép đổi kiểu này cũng được gọi là ép kiểu và có trong nhiều ngôn ngữ. Các phép đổi kiểu nói chung dễ làm mất thông tin nên các chương trình dịch thường không tự động thực hiện các phép đổi kiểu này. Ví dụ một số nguyên đổi tự động được thành số thực nhưng ngược lại thì mất thông tin.

Một phép đổi kiểu được gọi là *rõ* nếu người lập trình phải viết một số mã lệnh để thực hiện phép đổi này.

Ví dụ 7.2

Sản xuất	Luật ngữ nghĩa
$E \rightarrow \text{số}$	$E.type := \text{integer}$
$E \rightarrow \text{số} . \text{số}$	$E.type := \text{real}$
$E \rightarrow \text{tên}$	$E.type := \text{lookup}(\text{tên.entry})$
$E \rightarrow E_1 \text{ op } E_2$	$E.type := \begin{cases} \text{if } E_1.type=\text{integer} \text{ and } E_2.type=\text{integer} \\ \quad \text{then integer} \\ \text{else if } E_1.type=\text{integer} \text{ and } E_2.type=\text{real} \\ \quad \text{then real} \\ \text{else if } E_1.type=\text{real} \text{ and } E_2.type=\text{integer} \\ \quad \text{then real} \\ \text{else if } E_1.type=\text{real} \text{ and } E_2.type=\text{real} \\ \quad \text{then real} \\ \text{else type_error} \end{cases}$

3. Định nghĩa chồng (overloading) của hàm và các phép toán

Một ký hiệu chồng là một ký hiệu có nhiều nghĩa khác nhau phụ thuộc vào ngữ cảnh của nó. Trong toán, toán tử $+$ là định nghĩa chồng, bởi vì dấu $+$ trong $A+B$ có các ý nghĩa khác nhau khi A và B là số nguyên, thực, số phức, ma trận, xâu... Trong Ada cặp dấu ngoặc $()$ cũng được định nghĩa chồng, biểu thức $A(I)$ có thể là phần tử thứ I của mảng A , cũng có thể là gọi hàm A với tham số I . Các toán tử số học được định nghĩa chồng trong hầu hết ngôn ngữ.

Nói cách khác *định nghĩa chồng cho phép tạo ra nhiều hàm khác nhau nhưng có cùng một tên*. Thông thường các hàm này làm những công việc có ý nghĩa gần giống nhau. Để xác định thực sự định nghĩa chồng nào cần dùng ta phải căn cứ vào ngữ cảnh lúc áp dụng. Điều kiện để thực hiện các toán tử chồng là phải có sự khác nhau về kiểu hoặc số tham số. Do đó, ta có thể dựa vào các luật ngữ nghĩa ở phần trên để kiểm tra kiểu và gọi các hàm xử lý thích hợp.

Bài tập

1. Viết kiểu biểu thức cho các kiểu sau:

- Một mảng con trỏ trả đến số thực, chỉ số của mảng từ 1 đến 100.
- Một mảng hai chiều (nghĩa là mảng của mảng) với các dòng có các chỉ số 0 đến 9, cột từ -10 đến 10.

2. Có một khai báo trong C như sau:

```
typedef struct {  
    int     a, b;  
} CELL, *PCELL;  
  
CELL foo[100];  
  
PCELL bar (x, y) int x; CELL y {...}
```

Viết kiểu biểu thức cho các kiểu của foo và bar.

3. Văn phạm sau dùng để định nghĩa danh sách list.

$P \rightarrow D ; E$

$D \rightarrow D ; D | \text{tên} : T$

$T \rightarrow \text{char} | \text{integer} | \text{list of } T$

$E \rightarrow \text{chữ cái} \mid \text{số} \mid \texttên \mid (L)$

$L \rightarrow E, L \mid E$

Viết các luật dựa theo bài để xác định kiểu của biểu thức (E) và danh sách (L).

Bài tập thực hành

1. Thiết kế chi tiết cú pháp điều khiển để kiểm tra kiểu.
2. Cài đặt khối phân tích ngữ nghĩa.

Chương 8

BẢNG KÝ HIỆU

I. MỤC ĐÍCH, NHIỆM VỤ

Một chương trình dịch cần phải thu thập và sử dụng các thông tin về các tên xuất hiện trong chương trình nguồn. Các thông tin này được lưu trong một cấu trúc dữ liệu gọi là một bảng ký hiệu. Các thông tin bao gồm tên (là một chuỗi ký tự tạo nên), kiểu của nó (nghĩa là số nguyên, thực, xâu), dạng của nó (nghĩa là một biến hay một cấu trúc), vị trí của nó trong bộ nhớ, và các thuộc tính khác phụ thuộc vào ngôn ngữ lập trình.

Mỗi lần một tên cần xem xét, chương trình dịch sẽ tìm trong bảng ký hiệu xem đã có tên đó chưa. Nếu tên là mới thì thêm tên đó vào bảng ký hiệu. Các thông tin về tên được tìm và đưa vào bảng trong giai đoạn phân tích từ vựng và cú pháp.

Các thông tin có trong bảng ký hiệu được dùng ở một số quá trình dịch. Nó được dùng trong lúc phân tích ngữ nghĩa, như kiểm tra xem việc dùng các tên này có khớp với khai báo của chúng hay không. Nó cũng được dùng trong giai đoạn sinh mã, ví dụ để biết kích thước, loại bộ nhớ phải cấp phát cho một tên.

Cũng có một số nhu cầu dùng bảng ký hiệu theo cách khác như để phát hiện và khắc phục lỗi.

II. CÁC YÊU CẦU ĐỐI VỚI BẢNG KÝ HIỆU

Chúng ta cần một số khả năng làm việc với bảng như sau:

1. Phát hiện một tên cho trước có ở trong bảng hay không
2. Thêm một tên mới vào bảng
3. Lấy thông tin tương ứng với tên cho trước

4. Thêm các thông tin mới vào một tên cho trước

5. Xoá một tên hoặc nhóm tên trong bảng

Các thông tin đưa vào bảng ký hiệu có thể bao gồm:

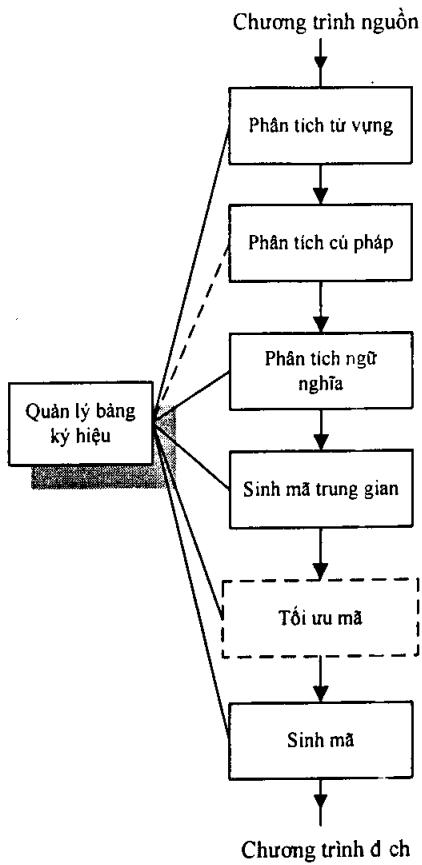
1. Xâu ký tự tạo nên tên.

2. Thuộc tính của tên

3. Các tham số, như số chiều của mảng

4. (Có thể) con trỏ đến tên cấp phát

Các thông tin đưa vào bảng thường không cùng một lúc mà trong những thời điểm khác nhau.



Hình 8.1. Vị trí bảng ký hiệu

Đầu vào của bảng

Mỗi một vị trí trong bảng ký hiệu là một cặp có dạng (tên, thuộc tính), tức là một bảng ký hiệu có hai trường, một trường tên và một trường thuộc

tính. Trường tên là khoá của bảng, tức là trong toàn bảng không được tồn tại hai dòng (hai bản ghi) khác nhau có cùng một tên. Trong trường hợp một ngôn ngữ đặc biệt cho phép hai bản ghi khác nhau có cùng một tên thì ta phải bổ sung thêm một trường nữa vào khoá.

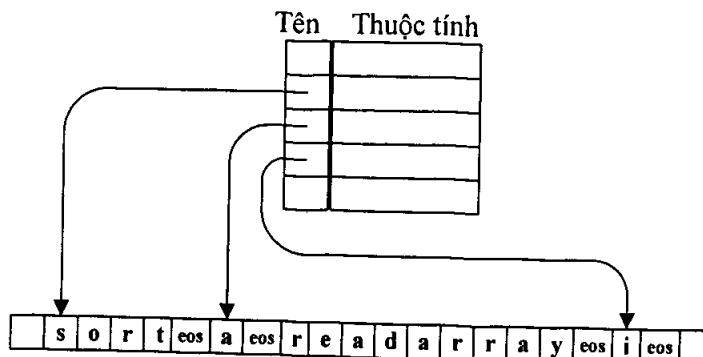
Tên	Thuộc tính
s o r t	
a	
r e a d a r r a y	
i	

Hình 8.2. Đầu vào của bảng.

Giảm kích thước của bảng

Nếu ghi trực tiếp tên trong trường tên của bảng thì ưu điểm là:

- Đơn giản
- Nhanh



Hình 8.3. Giảm kích thước bảng dùng con trỏ.

Còn nhược điểm là:

- Độ dài tên bị giới hạn. Đó là giới hạn kích thước trường. Ví dụ như trong ngôn ngữ Standard Pascal (Pascal chuẩn), độ dài của tên chỉ cho phép đến 8 ký tự.
- Hiệu quả sử dụng bộ nhớ không cao. Ở hình 8.2, ta thấy trường tên còn thừa rất nhiều chỗ trống. Thông thường, chỉ dùng khoảng 10-20% kích thước thực tế của bảng.

Một phương pháp tận dụng bộ nhớ triệt để hơn là dùng con trỏ như hình 8.3. Các tên được đặt liên tiếp nhau trong bộ nhớ và được phân cách với

nhau bằng một ký hiệu đặc biệt **eos**. Trường tên lúc này chỉ là con trỏ trả đến xâu tên tương ứng trong bộ nhớ.

III. CẤU TRÚC DỮ LIỆU CỦA BẢNG KÝ HIỆU

Có rất nhiều cách tổ chức bảng ký hiệu khác nhau, như có thể tách bảng riêng rẽ ứng với tên biến, nhãn, hằng số, tên hàm và các kiểu tên khác,... phụ thuộc vào từng ngôn ngữ.

Về cách tổ chức dữ liệu, có nhiều lựa chọn: danh sách tuyến tính, bảng băm, cây tìm kiếm với nhiều loại khác nhau. Cài đặt danh sách tuyến tính đơn giản nhưng chậm; bảng băm nhanh hơn nhưng phức tạp hơn, cây tìm kiếm chiếm mức phức tạp và chất lượng vừa phải.

Trong khi dịch, chương trình sẽ làm việc liên tục với bảng ký hiệu. Do đó chúng ta phải thiết kế bảng ký hiệu sao cho đạt hiệu quả cao. Như trên đã nói, có ba cách tổ chức dữ liệu. Chúng ta sẽ đánh giá các phương pháp này qua thời gian hệ thống phải thêm n vị trí mới và trả lời m yêu cầu thông tin.

a. Danh sách

Danh sách là cấu trúc dữ liệu đơn giản nhất và dễ thực hiện nhất. Bảng ký hiệu sẽ trở thành một danh sách tuyến tính của các bản ghi. Chúng ta chỉ cần dùng một hoặc vài mảng (array) để lưu các tên và các thông tin tương ứng. Tên được thêm vào mảng theo thứ tự gấp trong chương trình nguồn.

Để tìm kiếm một tên, ta tìm từ đầu mảng lần lượt cho đến vị trí tên cuối cùng trong bảng. Như vậy, nếu bảng có n tên thì ta có thể phải tìm n lần, còn trung bình là $n/2$. Như vậy độ phức tạp cỡ $O(n)$.

tên ₁	thôngtin ₁
tên ₂	thôngtin ₂
tên _{n-1}	thôngtin _{n-1}
tên _n	thôngtin _n

Để thêm n tên và trả lời m câu hỏi, tổng số công việc cần làm là $cn(n+m)$, với c là một hằng số thời gian phụ thuộc vào máy. Trong một chương trình cỡ trung bình, chúng ta có thể có $n=100$ và $m=1000$, do đó chương trình có thể phải thực hiện hàng trăm ngàn thao tác.

Các cách cải tiến danh sách:

- Danh sách cấp phát động, dùng mốc nối một chiều hoặc hai chiều. Nhược điểm của mảng danh sách là kích thước mảng phải cố định, do đó kích thước mảng phải lớn hơn và bằng số bản ghi lớn nhất mà một chương trình có thể đạt tới. Còn bình thường, số bản ghi tương đối nhỏ, gây lãng phí bộ nhớ rất lớn (hiệu quả đạt được thường chỉ 20-30%).

40%). Dùng danh sách cấp phát động sẽ tận dụng được bộ nhớ hơn nhiều. Mặt khác, nó cũng giúp cho quá trình sắp xếp danh sách đơn giản và nhanh hơn.

- Danh sách có sắp xếp theo trường tên (trường khoá). Lúc này ta chỉ cần tốn lượng thời gian tìm kiếm dạng $O(\log_2 n)$ mà thôi.

b. Cây tìm kiếm

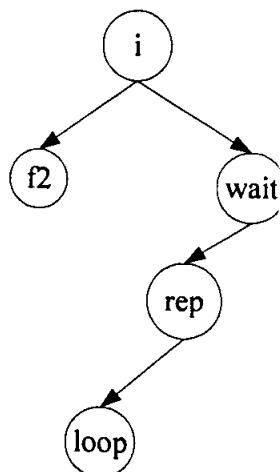
Cây tìm kiếm là một tổ chức bảng ký hiệu hiệu quả hơn. Một trong các dạng cây tìm kiếm tốt là cây tìm kiếm nhị phân. Các nút của cây nhị phân này có khoá là các tên của bản ghi và hai con trỏ LEFT và RIGHT. Đối với mọi nút trên cây, các tính chất sau luôn được thỏa mãn:

- Mọi khoá thuộc cây con trái của một nút đều nhỏ hơn khoá ứng với nút đó
- Mọi khoá thuộc cây con phải nút đó đều lớn hơn khoá ứng với nó.

Ví dụ 8.1: Giải thuật tìm kiếm đối với cây nhị phân để tìm xem một khoá X nào đó có trên cây hay không như sau:

So sánh X với khoá ở gốc, có các tình huống:

1. Không có gốc (cây rỗng): X không có trên cây;
2. X trùng với khoá ở gốc: phép tìm kiếm thỏa
3. X nhỏ hơn khoá ở gốc: thực hiện tiếp việc tìm kiếm bằng cách xét cây con trái của gốc.
4. X lớn hơn khoá ở gốc: thực hiện tiếp việc tìm kiếm bằng cách xét cây con phải của gốc.



Hình 8.4. Cây tìm kiếm.

Độ dài trung bình của mỗi nhánh cây tìm kiếm là $\log_2 n$, do đó, thời gian tìm kiếm là $O(\log_2 n)$. Để đảm bảo thời gian tìm kiếm luôn là $\log_2 n$ ta có thể thay cây tìm kiếm nhị phân bằng cây nhị phân cân đối AVL. Thời gian cần thiết để nhập n tên và trả lời m yêu cầu tỷ lệ với $(n+m) \log_2 n$. Nếu n lớn hơn 50, thì cây tìm kiếm có ưu điểm hơn hẳn danh sách tuyến tính ở trên.

c. Bảng băm (hash table)

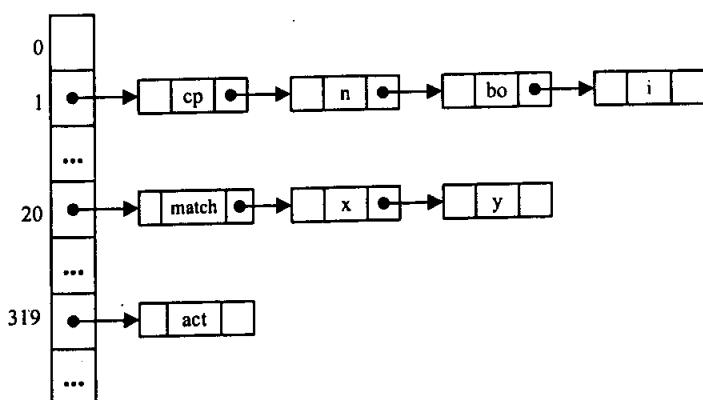
Trong các kỹ thuật bảng ký hiệu thì bảng băm là một kỹ thuật được dùng rất nhiều, ở nhiều hình thức khác nhau. Sau đây chúng ta xem xét một trường hợp đơn giản của bảng băm là “bảng băm mở”. Chữ “mở” ở đây nghĩa là ta không giới hạn số lượng mục có thể nhập vào bảng.

Nguyên tắc làm việc của bảng băm không dựa trên phép so sánh giá trị khoá mà dựa vào chính bản thân giá trị từng khoá. Bằng một quy tắc biến đổi nào đó, từ giá trị của khoá ta tính ra một địa chỉ (tương đối). Địa chỉ này sẽ được dùng để lưu trữ bản ghi tương ứng và đồng thời cũng để tìm kiếm bản ghi đó. Nghĩa là ta đã thiết lập một *hàm địa chỉ h(k)* thực hiện phép ánh xạ tập các giá trị của k lên tập các địa chỉ tương đối.

Nếu có m yêu cầu trên n tên thì thời gian tỷ lệ với $n(n+m)/k$, với một hằng số k nào đó do ta chọn. Như vậy k càng lớn càng tốt, và tốt nhất là đạt đến giới hạn của nó là n .

Một bảng băm cơ sở được minh họa ở hình 8.5, cấu trúc dữ liệu bao gồm hai phần:

- Một mảng có kích thước k cố định chứa các con trỏ chỉ đến các đầu của bảng.
- Các vị trí của bảng được tổ chức thành các danh sách liên kết riêng rẽ. Mỗi một bản ghi của bảng ký hiệu được chứa trong một phần tử của danh sách liên kết.



Hình 8.5. Kết hợp bảng băm với danh sách liên kết.

Để xác định vị trí của một xâu s nằm ở đâu trong bảng, trước tiên chúng ta gọi hàm băm với tham số s , tức là $h(s)$ và nhận được giá trị trả về nằm giữa 0 và $k-1$. Sau đó, duyệt tiếp danh sách liên kết để tìm s . Nếu s không có trong danh sách liên kết, ta thêm một nút mới vào cuối danh sách đó.

Tính sơ lược, trung bình một danh sách liên kết có n/k bản ghi nếu cả bảng có n bảng ghi và bảng băm có kích thước k . Nếu ta chọn tỷ số n/k là một hằng số nhỏ như 2 thì thời gian truy nhập vào một vị trí bảng cơ bản là hằng số.

Khó khăn khi thiết kế một bảng băm là làm sao phân bổ đều được các địa chỉ tính trên bảng địa chỉ.

Một phương pháp tính hàm băm như sau:

- Xác định một số nguyên dương h từ chuỗi các ký tự $c_1, c_2, c_3\dots, c_k$ trong xâu s . Các hàm xác định này thường đã có sẵn trong nhiều ngôn ngữ lập trình, ví dụ hàm *ord* trong Pascal.
- Biến đổi số nguyên dương h xác định ở trên thành số của một danh sách trong bảng băm, nghĩa là một số nguyên trong khoảng 0 và $k-1$. Cách đơn giản nhất là chia h cho k và lấy phần dư.

Bài tập

Giả sử chúng ta có một bảng băm có 10 vị trí và ta mong muốn nhập các tên biến loại số nguyên, hàm băm là $h(i) = i \bmod 10$, tức là phần dư khi chia i cho 10. Hãy biểu diễn các mối liên kết được tạo thành trong bảng băm khi ta nhập vào mười số nguyên tố đầu tiên 2, 3, 5, ..., 29 theo thứ tự. Nếu bạn nhập thêm các số nguyên tố nữa, bạn có mong muốn chúng sẽ rải ngẫu nhiên trên bảng hay không?

Bài tập thực hành

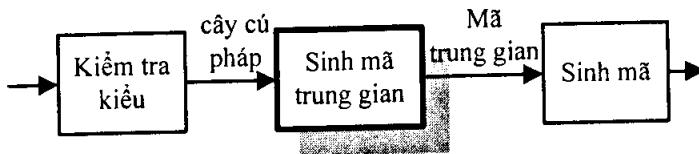
Thiết kế và cài đặt bảng ký hiệu cho chương trình dịch.

Chương 9

SINH MÃ TRUNG GIAN

I. MỤC ĐÍCH, NHIỆM VỤ

Để chuyển đổi cấu trúc của một ngôn ngữ lập trình, một chương trình dịch có thể cần biết nhiều thông tin phụ. Ví dụ, chương trình dịch có thể cần biết kiểu của cấu trúc, hoặc vị trí của mã lệnh đầu tiên trong mã đích, hoặc số mã lệnh sinh ra. Ta sẽ đề cập đến một khái niệm trùu tượng là các thuộc tính liên quan đến các cấu trúc. Một thuộc tính có thể cho biết một vài thông tin như kiểu, xâu, vị trí cấp phát trong bộ nhớ...



Hình 9.1. Khối sinh mã trung gian.

Tuy ta có thể sinh mã máy trực tiếp ngay sau khối kiểm tra ngữ nghĩa, việc sinh mã trung gian có những lợi ích như sau:

- Dễ thiết kế từng phần.
- Sinh được mã độc lập với từng máy tính cụ thể. Từ đó làm giảm độ phức tạp của phần sinh mã thật sự.
- Dễ tối ưu mã.

Các vấn đề của bộ sinh mã trung gian là

- Nên dùng mã trung gian nào.
- Thuật toán sinh mã trung gian.

Hành động sinh mã trung gian thực hiện qua các định nghĩa cú pháp điều khiển mà ta đã xét ở chương 6. Do đó các kĩ thuật ta đã nắm được. Ở đây ta chỉ đi sâu vào các chi tiết cài đặt cụ thể.

II. CÁC NGÔN NGỮ TRUNG GIAN

Ngôn ngữ trung gian là một ngôn ngữ nằm giữa ngôn ngữ nguồn và ngôn ngữ đích. Chương trình viết bằng ngôn ngữ trung gian sẽ vẫn tương đương với chương trình viết bằng ngôn ngữ nguồn về chức năng và nhiệm vụ. Ngôn ngữ trung gian có thể là mọi kiểu biểu diễn thỏa mãn được yêu cầu này, nhưng thông thường người ta hay dùng một trong các kiểu ngôn ngữ trung gian dưới đây.

1. Ký pháp hậu tố

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

- Nếu E và một biến hoặc một hằng số, thì ký pháp hậu tố của E là bản thân E .
- Nếu E và một biểu thức dạng $E_1 op E_2$, với op là một toán tử hai ngôi thì ký pháp hậu tố của E là $E_1' E_2' op$, với E_1' , E_2' là các ký pháp hậu tố của E_1 và E_2 tương ứng.
- Nếu E và một biểu thức dạng (E_1), thì ký pháp hậu tố của E_1 cũng là ký pháp hậu tố của E .

Ví dụ 9.1:

Ký pháp hậu tố của $(9-5)+2$ là $9\ 5\ -\ 2\ +$;

Ký pháp hậu tố của $9- (5+2)$ là $9\ 5\ 2\ +\ -$.

Ký pháp hậu tố của câu lệnh

if a then if $c-d$ then $a+c$ else $a*c$ else $a+b$
là $a ? (c-d ? a+c : a*c) : a+b$ tức là $acd-ac+ac^*? ac+?$

Định nghĩa cú pháp điều khiển tạo mã hậu tố

Tạo mã hậu tố cho mã trung gian của một biểu thức rất đơn giản như ví dụ dưới. Ở đây, $E.CODE$ là một xâu dùng để chứa mã hậu tố kết quả.

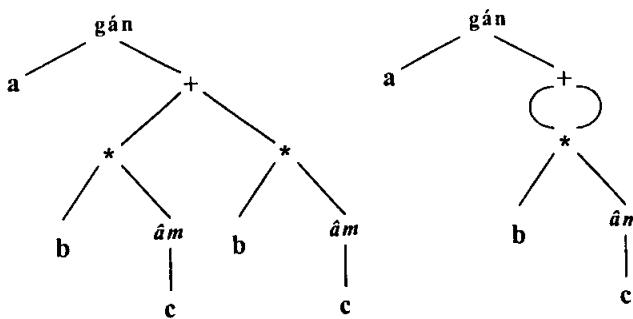
Sản xuất	Luật ngữ nghĩa
$E \rightarrow E^{(1)} op E^{(2)}$	$E.CODE := E^{(1)}.CODE\ E^{(2)}.CODE\ 'op'$
$E \rightarrow (E^{(1)})$	$E.CODE := E^{(1)}.CODE$
$E \rightarrow - E^{(1)}$	$E.CODE := E^{(1)}.CODE\ '-'$
$E \rightarrow \text{tên}$	$E.CODE := \text{tên}$

2. Đồ thị

Cây phân tích bản thân nó là một biểu diễn tốt của ngôn ngữ trung gian cho ngôn ngữ nguồn, đặc biệt khi tối ưu hoá. Một dạng của cây phân tích là cây cú pháp, một cây mà mỗi lá biểu diễn một toán hạng và mỗi một nút trong là một toán tử. Trong chương 7 ta đã biết cách chuyển cây phân tích nhận được từ bộ phân tích cú pháp thành cây cú pháp hoặc DAG.

Một cây cú pháp miêu tả cấu trúc phân cấp tự nhiên của một chương trình nguồn. Một đồ thị DAG cũng cho cùng thông tin tương tự nhưng có kích thước tiết kiệm hơn vì dùng chung những phần giống nhau.

Ví dụ 9.2: Hình 9.2 là một cây cú pháp và một đồ thị DAG của câu lệnh $a := b * -c + b * -c$:



Hình 9.2. Cây cú pháp và đồ thị DAG.

3. Mã ba địa chỉ (three-address code)

Mã ba địa chỉ là một chuỗi các câu lệnh, thông thường có dạng

$x := y \ op \ z$

với x, y, z là các tên, hằng định nghĩa bởi người lập trình, hoặc tên trung gian do bộ sinh mã sinh ra; op là một phép toán nào đó, như các phép toán số học, logic... Tên gọi "mã ba địa chỉ" nghĩa là mỗi một câu lệnh thường chứa ba địa chỉ, hai cho toán hạng và một cho kết quả. Chú ý rằng mã ba địa chỉ không cho phép thể hiện trực tiếp các biểu thức số học phức tạp. Thay vào đó các biểu thức phức tạp phải được phân tách thành các biểu thức đơn giản có đúng một phép toán.

Như vậy, một biểu thức dạng $X + Y * Y$ phải được viết lại thành:

$t_1 := y * z$

$t_2 := x + t_1$

với t_1, t_2 là các tên trung gian do bộ sinh mã tạo ra.

Có một số loại câu lệnh ba địa chỉ cần ít hơn ba địa chỉ, tức là một trong các địa chỉ không có tên. Sau đây là liệt kê các câu lệnh ba địa chỉ:

- 1) Câu lệnh gán dạng $x := y op z$ với op là phép toán số học hai ngôi hoặc phép toán logic.
- 2) Câu lệnh gán dạng $x := op y$ với op là phép toán không ngôi. Các phép toán không ngôi cơ sở bao gồm phép lấy âm, đảo logic, các phép toán dịch chuyển, các phép toán đổi kiểu như phép đổi kiểu từ char sang integer.
- 3) Câu lệnh sao chép dạng $x := y$, gán y vào x .
- 4) Nhảy không điều kiện $goto L$. Câu lệnh ba địa chỉ sẽ đánh nhãn L là điểm hoạt động tiếp theo.
- 5) Các lệnh nhảy có điều kiện như $if x relop y goto L$. Câu lệnh này sẽ dùng các toán tử quan hệ ($<$, $=$, \geq , ...) cho x và y và thực hiện câu lệnh tiếp theo tại nhãn L nếu biểu thức quan hệ đúng, nếu không câu lệnh tiếp theo câu lệnh if sẽ được thực hiện.
- 6) Các câu lệnh **param A** và **Call P, n**. Các cấu trúc này được dùng để thực hiện một lời gọi hàm. Thường được dùng ở dạng một chuỗi các câu lệnh mã ba địa chỉ như sau:

param A₁

param A₂

...

param A₁

call P, n

sẽ sinh ra lời gọi hàm P (A₁, A₂, ..., A_n). Tham số n trong **call P, n** là một số nguyên cho biết số tham số thật sự trong lời gọi. Tham số n này có thể coi là thừa vì chương trình có thể tự tính lấy từ các tham số trước lời gọi.

- 7) Câu lệnh gán lấy chỉ số dạng $x = y[i]$ và $x[i] = y$.
- 8) Các phép gán địa chỉ và giá trị của con trỏ dạng $x = &y$, $x = *y$, $*x = y$.

Cú pháp điều khiển sinh mã ba địa chỉ

Bảng sau là một ví dụ dùng các định nghĩa cú pháp điều khiển tổng hợp để sinh các mã ba địa chỉ cho các câu lệnh gán.

Ký hiệu không kết thúc E có hai thuộc tính:

- $E.place$ - nơi giữ giá trị của E .
- $E.code$ - chuỗi các mã ba địa chỉ đánh giá tại E .

Hàm $newtemp$ trả lại một chuỗi các tên tách nhau $t_1, t_2\dots$ Hàm gen dùng để sinh mã ba địa chỉ. Trong thực tế thì hàm gen sẽ đưa mã ba địa chỉ ra file chứ không cần phải lưu trong trường $code$ nữa.

Sản xuất	Luật ngữ nghĩa
$S \rightarrow \text{tên} := E$	$S.code := E.code \parallel gen(\text{tên}.place ':=' E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place ':=' E_1.place '+' E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place ':=' E_1.place '*' E_2.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow \text{tên}$	$E.place := \text{tên}.place;$ $E.code := ''$

Còn dưới đây là định nghĩa cú pháp điều khiển sinh mã ba địa chỉ cho câu lệnh while. Trong đó có dùng thêm các thuộc tính *begin* và *after* để đánh dấu câu lệnh đầu tiên cho E và câu lệnh sau mã của S . Hàm $newlabel$ sẽ trả về một nhãn mới.

Sản xuất	Luật ngữ nghĩa
$S \rightarrow \text{while } E \text{ do } S_1$	$S.begin := newlabel;$ $S.after := newlabel;$ $S.code := gen(S.begin ':') $ $E.code $ $gen('if' E.place '=' '0' 'goto' S.after) $ $S_1.code $ $gen('goto' S.begin) $ $gen(S.after ':')$

Biểu diễn mã ba địa chỉ

a. Biểu diễn bằng bộ bốn (quadruples)

Chúng ta có thể dùng một cấu trúc có bốn trường mà ta gọi là OP, ARG1, ARG2 và RESULT. OP chứa mã nội bộ của toán tử. Câu lệnh ba địa chỉ $A := B \text{ op } C$ sẽ đặt B vào ARG1, C vào ARG2 và A vào RESULT. Câu lệnh không ngôi $A := B$ hay $A := -B$ sẽ không dùng ARG2.

Ví dụ 9.3: Câu lệnh $A := -B * (C + D)$ sẽ được chuyển đổi về các câu lệnh mã ba địa chỉ như sau:

T1:=-B

T2:=C+D

T3:=T1*T2

A:=T3

và biểu diễn bằng bộ bốn như sau:

	OP	ARG1	ARG2	RESULT
0	uminus	B	-	T1
1	+	C	D	T2
2	*	T1	T2	T3
3	:=	T3	-	A

b. Biểu diễn bằng bộ ba

Để tránh phải đưa các tên tạm thời vào bảng ký hiệu, có một cách cho phép tính một giá trị tạm thời. Nếu chúng ta làm như vậy, câu lệnh mã ba địa chỉ sẽ được biểu diễn bằng một cấu trúc chỉ có ba miền OP, ARG1 và ARG2. Loại mã trung gian loại này được gọi là triple.

Ta dùng một số dấu ngoặc để biểu diễn các con trỏ trong cấu trúc triple. Trong thực tế, các thông tin cần thiết để diễn tả các loại miền tham số có thể được mã hoá trong miền OP hoặc một số miền bổ sung.

Ví dụ 9.4: Ví dụ 9.3 sẽ được biểu diễn dạng triple như sau:

	OP	ARG1	ARG2
0	uminus	B	-
1	+	C	D
2	*	(1)	(2)
3	:=	A	(2)

Còn sau đây là một số ví dụ khác về các câu lệnh dạng $A[I]:=B$ hay $A:=B[I]$

	OP	ARG1	ARG2
0	[]:=	A	I
1	-	B	-
0	=[]	B	I
1	:=	(0)	A

Bài tập

1. Chuyển biểu thức số học $a * -(b+c)$ thành:

- Một cây cú pháp.
- Ký pháp hậu tố.
- Mã ba địa chỉ.

2. Chuyển biểu thức $-(a+b)* (c+d)+(a+b+c)$ thành mã ba địa chỉ.

3. Chuyển đoạn chương trình sau trong C:

```
main()
{
    int i;
    int a[10];
    i = 1;
    while (i <= 10) {
        a[i] = 0; i = i+1;
    }
}
```

thành:

- Một cây cú pháp.
- Ký pháp hậu tố.
- Mã ba địa chỉ.

Bài tập thực hành

1. Viết chương trình chuyển đổi một biểu thức thành dạng mã hậu tố.
2. Thiết kế và cài đặt bộ sinh mã trung gian.

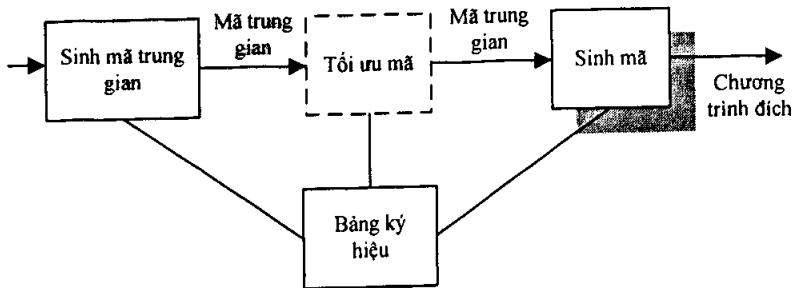
Chương 10

SINH MÃ

I. MỤC ĐÍCH, NHIỆM VỤ

Phần sinh mã là giai đoạn cuối của quá trình dịch. Sinh mã tốt rất khó và mã sinh ra thường gắn với một loại máy tính cụ thể nào đó. Một thuật toán sinh mã được nghiên cứu kỹ sẽ sinh ra mã đích có thể chạy nhanh hơn nhiều lần các mã sinh bởi các thuật toán sinh mã không cần nhắc kỹ.

Sơ đồ khái của các phần này như Hình 10.1. Đầu vào của bộ sinh mã là mã trung gian, đầu ra là một chương trình viết ở dạng mã đối tượng nào đó và gọi là chương trình đích.



Hình 10.1. Sơ đồ khái bô sinh mã.

Rõ ràng tiêu chuẩn quan trọng nhất đối với bộ sinh mã là phải tạo ra mã đúng. Việc sinh mã đúng thực sự là thách thức do có nhiều trường hợp ngoại lệ trong lúc sinh mã. Đặt tiêu chuẩn sinh mã đúng lên hàng đầu giúp cho việc thiết kế, xây dựng và bảo trì bộ sinh mã trở nên định hướng tốt hơn, dễ thực hiện và kiểm tra hơn.

Trong thực tế, việc sinh mã máy đòi hỏi nhiều kinh nghiệm và nhiều khi phải dùng "mẹo" mới tạo được mã máy tốt.

II. CÁC DẠNG MÃ ĐÓI TƯỢNG (object code)

1. Mã máy định vị tuyệt đối

Một chương trình mã máy tuyệt đối có các lệnh mã máy được định vị tuyệt đối (trong chương trình đích). Chương trình đích xác định hoàn toàn chương trình đối tượng này.

Mã có thể được một chương trình dịch thực sự tạo ra và đặt vào các vị trí này và do vậy nó có thể hoạt động được ngay tức khắc.

Ưu điểm cơ bản của việc sinh ra mã máy tuyệt đối là dịch nhanh và có tổng số thời gian cần để dịch một chương trình nguồn thành dạng mã nạp chạy được thấp. Bản thân mã đối tượng được tạo ra cũng thường được các hệ điều hành nạp chạy nhanh hơn các loại mã khác. Nhược điểm chính là toàn bộ chương trình nguồn phải được dịch từ các đoạn mà mỗi thời điểm chỉ một phần của nó được chọn. Thêm nữa, chương trình đối tượng hầu như không thể di chuyển được, không thể chạy được trong một máy tính có cấu trúc khác.

2. Mã đối tượng có thể định vị lại được

Đa số các chương trình dịch thương mại ngày nay sinh ra các mã đối tượng ở dạng các mô đun có thể định vị lại được. Với các mô đun như vậy khi được lưu ở dạng tĩnh thì địa chỉ câu lệnh của chúng không được xác định (các địa chỉ đó là các biểu thức cho biết quan hệ giữa các địa chỉ với các địa chỉ khởi đầu - mà lúc này cũng không được xác định).

Một hoặc nhiều môđun mã đối tượng sẽ được biến đổi thành các chương trình đối tượng tuyệt đối nhờ một quá trình gọi là liên kết (linking) và nạp (loading). Liên kết được tạo bởi một chương trình (thường được gọi là bộ liên kết - linker), chương trình này sẽ xác định kích thước lưu trữ cần thiết cho phần dữ liệu và các chỉ thị lệnh của cả môđun cũng như các địa chỉ bắt đầu của phần dữ liệu và các lệnh đó. Quá trình tải thường được tiến hành bằng một chương trình gọi là bộ tải (loader) sử dụng các thông tin để xác lập lại quan hệ giữa các địa chỉ của mã đối tượng và hình thành chương trình mã tuyệt đối sẵn sàng để chạy.

Ưu điểm của việc sinh mã có thể định vị lại được và sau đó liên kết chúng lại là các môđun chương trình nguồn độc lập và có thể dịch đi dịch lại độc lập với nhau và sau cùng kết hợp lại với nhau thành một chương trình

đối tượng hoàn chỉnh nhờ việc liên kết và nạp. Các thủ tục trong các thư viện (thực hiện các công việc đã được chuẩn hoá như vào /ra) có thể được đặt và liên kết với chương trình đối tượng mà không cần phải dịch lại. Thư viện và nhiều chương trình đối tượng có thể được viết và dịch trong nhiều ngôn ngữ lập trình khác nhau vẫn có thể liên kết được với nhau.

Nhược điểm chính của việc sinh mã kiểu này là quá trình dịch - liên kết - nạp rất tốn thời gian và chậm hơn nhiều việc sinh mã tuyệt đối.

3. Mã đối tượng thông dịch

Trong một chương trình mã đối tượng thông dịch chuỗi lệnh được biểu diễn không phải bằng các chỉ thị lệnh máy hoạt động trực tiếp mà bằng các câu lệnh thông dịch trừu tượng (ở một dạng mã hoá nào đó). Trong trường hợp này chương trình biên dịch sẽ chuyển chương trình nguồn thành một chương trình đối tượng với các lệnh ảo trên. Chương trình đối tượng sau đó sẽ được cho hoạt động nhờ một chương trình thông dịch. Nó sẽ đọc vào từng mã lệnh, giải mã và gọi một thủ tục tương ứng để thực hiện các hành động theo yêu cầu.

Những câu lệnh trừu tượng xác lập một máy tính ảo. Máy này thông thường được thiết kế để đáp ứng lại ngôn ngữ bậc cao đang được xử lý. Bộ thông dịch do vậy sẽ là một bộ mô phỏng hiệu quả đối với máy tính ảo đó.

Việc dùng mã đối tượng kiểu này cho các ưu điểm đáng chú ý trong các hoàn cảnh và môi trường như sau:

- Dễ viết chương trình dịch. Đa số các mã đối tượng thông dịch được thiết kế tương ứng với các đặc tính của ngôn ngữ bậc cao do vậy nói chung là dễ sinh.
- Chương trình đối tượng nhỏ gọn. Một hành động cần nhiều mã lệnh máy thực hiện có thể được mã hoá thành một lệnh ảo duy nhất với kích thước chỉ một word (4 byte) hoặc nhỏ hơn.
- Các phương tiện đối thoại và gỡ rối trong khi chạy có thể được thêm vào bộ thông dịch mà không cần phải thêm vào bản thân mã đối tượng.
- Tính khả chuyển của ngôn ngữ thực hiện - nếu các thủ tục thông dịch được viết trong một ngôn ngữ bậc cao, thì chương trình đối tượng có thể chạy được trên một máy tính bất kỳ chạy được ngôn ngữ đó.

Một trong các chương trình dịch phổ biến nhất tạo ra mã thông dịch là Java. Nhờ dùng loại mã này (trong Java gọi là bytecode) ngôn ngữ lập trình Java mới có thể mệnh danh là “viết một lần, chạy mọi nơi”.

Nhược điểm chính của việc dùng mã thông dịch là chúng chạy chậm. Nói chung chúng chạy chậm hơn từ 5 đến 100 lần mã máy tuyệt đối. Điều này cũng giải thích được tại sao các chương trình Java thường chạy chậm hơn các chương trình tương đương viết bằng C hoặc C++. Cách khắc phục việc chạy chậm chỉ duy nhất bằng cách nâng cấp phần cứng: phải dùng các máy tính chạy nhanh hơn.

III. CÁC VẤN ĐỀ THIẾT KẾ CỦA BỘ SINH MÃ

1. Đầu vào

Đầu vào của bộ sinh mã là một chương trình trong ngôn ngữ trung gian và bảng ký hiệu mà nó được dùng để xác định các địa chỉ chạy của các đối tượng dữ liệu ký hiệu bằng các tên trong mã trung gian.

Nhu chương 9 đã đề cập, mã trung gian thường có các dạng:

- Biểu diễn đồ thị
- Ký pháp hậu tố
- Mã ba địa chỉ (three-address code).

2. Đầu ra

Đầu ra của bộ sinh mã là một chương trình đối tượng. Nó có thể có nhiều dạng:

- Một chương trình hoàn toàn bằng ngôn ngữ máy, chạy được ngay
- Một chương trình bằng ngôn ngữ máy có thể định vị lại được (relocatable)
- Một chương trình bằng ngôn ngữ Assembly.
- Một chương trình bằng một ngôn ngữ lập trình nào khác.

Việc tạo ra một chương trình hoàn toàn bằng ngôn ngữ máy có các ưu điểm như nó có thể được đặt trong các vị trí bộ nhớ cố định và chạy được ngay. Một chương trình cỡ nhỏ có thể được dịch và chạy trong vài giây.

Tạo ra một chương trình bằng ngôn ngữ máy có thể định vị được (các object module) cho phép các chương trình con có thể dịch độc lập với nhau. Một tập các chương trình định vị được có thể liên kết (link) và nạp chạy bằng bộ liên kết - nạp. Mặc dù phải trả giá cho việc liên kết - nạp này, chúng ta có được sự tiện lợi lớn nhờ việc dịch các chương trình con độc lập

với nhau và dùng lại các file đã được dịch từ trước. Phần lớn các chương trình thường mại tạo ra các chương trình loại này.

Tạo ra chương trình bằng ngôn ngữ Assembly giúp cho bộ sinh mã làm việc dễ hơn. Ta có thể sinh ra các lệnh ký hiệu và dùng các tiện ích macro của assembler để giúp cho bộ sinh mã. Cái già phải trả chính là việc phải dịch chương trình hợp ngữ sau đó. Thông thường đây là một lựa chọn của chương trình dịch, sử dụng khi người lập trình muốn sinh ra chương trình assembler để học tập hay để tối ưu nó, hoặc dùng trong trường hợp máy có bộ nhớ nhỏ và chương trình dịch phải dùng cách duyệt nhiều lần.

Tạo ra một chương trình bằng một ngôn ngữ lập trình bậc cao nào khác cũng làm nhẹ bộ sinh mã. Các ưu, nhược điểm cũng tương tự như sinh ra mã assembly. Ví dụ, chương trình dịch chuyển đổi như từ Visual Basic sang Delphi.

3. Quản lý bộ nhớ

Việc chuyển đổi tương ứng từ tên (biến, hàm...) trong chương trình nguồn thành địa chỉ dữ liệu trong lúc chạy của chương trình đích được thực hiện nhờ sự phối hợp giữa các phần trước và phần sinh mã. Trong chương trước, chúng ta đã thấy trong mã ba địa chỉ có các con trỏ trỏ đến các biến nằm trong bảng ký hiệu. Từ các thông tin trong bảng ký hiệu, có thể xác định được một địa chỉ tương đối cho một tên ở trong miền dữ liệu.

Khi mã máy được sinh ra, các nhãn trong câu lệnh ba địa chỉ sẽ được chuyển đổi thành địa chỉ mã lệnh.

4. Chọn chỉ thị lệnh

Tính chất của tập các chỉ thị lệnh xác định mức phức tạp của việc chọn chỉ thị lệnh. Tính thống nhất và tính đầy đủ là các nhân tố quan trọng. Nếu máy đích không có các kiểu dữ liệu ở dạng thống nhất, thì mỗi một ngoại lệ lại đòi hỏi một quá trình xử lý đặc biệt.

Tốc độ chỉ thị lệnh và các biểu diễn là các nhân tố quan trọng khác. Nếu chúng ta không quan tâm đến tính hiệu quả của chương trình thì việc chọn chỉ thị lệnh là chân phương. Đối với mỗi câu lệnh ba địa chỉ, chúng ta có thể thiết kế một bộ khung mã hình thành nên mã đích cho cấu trúc đó. Ví dụ, mỗi một câu lệnh mã ba địa chỉ dạng $x := y + z$, với x , y và z là các biến được cấp phát tĩnh thì nó có thể được dịch thành chuỗi mã như sau:

MOV y, R0

ADD z, R0

MOV R0, x

Không may là nếu ta cũng làm như vậy cho hai câu lệnh:

a := b + c

d := a + e

thì mã sinh ra là:

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

Ở đây, câu lệnh thứ tư là dư thừa, và nếu a không dùng về sau thì câu lệnh thứ ba cũng trở thành dư thừa.

Chất lượng của mã sinh ra được xác định bởi tốc độ và kích thước của nó. Một máy đích có bộ lệnh giàu có có thể cho phép có vài cách để sinh mã cho cùng một hành động. Bộ sinh mã sẽ chọn các mã hiệu quả và nhỏ gọn nhất. Ví dụ, câu lệnh tăng một có thể dùng phép cộng hoặc phép tăng INC thì bộ sinh mã sẽ chọn phép INC.

Việc tính thời gian để chọn mã chạy nhanh nhất còn là một vấn đề rất khó.

5. Sử dụng thanh ghi

Các câu lệnh tính toán chỉ với thanh ghi thì thường là nhanh và gọn hơn tính toán với bộ nhớ. Do đó, chúng ta cũng cần lưu ý đến điều này. Việc sử dụng thanh ghi có thể chia làm hai vấn đề:

- Trong khi dùng thanh ghi, ta chọn tập các biến sẽ được đặt trong các thanh ghi tại một vị trí của chương trình.
- Trong giai đoạn gán thanh ghi, ta chỉ định một thanh ghi xác định để lưu một biến.

6. Thứ tự làm việc

Thứ tự tính toán có thể ảnh hưởng đến hiệu quả của mã đích. Một số thứ tự tính toán cần nhiều thanh ghi giữ kết quả trung gian hơn một số thứ tự khác. Tất nhiên, xác định được một thứ tự tốt nhất là công việc rất khó.

IV. MÁY ĐÍCH

Cấu tạo của nhiều máy tính có thể được mô tả như sau: có một bộ nhớ được đánh địa chỉ theo từng byte, bốn byte tạo thành một word và có n thanh ghi đa năng R0, R1,...Rn-1. Các câu lệnh có hai địa chỉ dạng:

op nguồn, đích

trong đó *op* là một mã phép toán, *nguồn* và *đích* là các trường dữ liệu. Sau đây là một số mã phép toán:

MOV (chuyển nguồn vào đích)

ADD (cộng nguồn vào đích)

SUB (trừ nguồn với đích)

Ví dụ 10.1:

ADD R2, R1 (cộng thanh ghi R1 với R1 và đặt kết quả vào R1)

MOV R0, M (chuyển giá trị thanh ghi R0 vào bộ nhớ vị trí M)

V. MỘT BỘ SINH MÃ ĐƠN GIẢN

Trong phần này, chúng ta sẽ nghiên cứu một bộ sinh mã đơn giản tạo ra mã máy từ mã trung gian ba địa chỉ.

Để đơn giản, ta coi mỗi toán tử trong các câu lệnh sẽ có một toán tử tương ứng trong mã máy. Ta cũng coi kết quả tính toán có thể đặt được trên thanh ghi, và chỉ cần chúng vào bộ nhớ nếu ta cần các thanh ghi này cho việc tính toán khác hoặc trước các lời gọi hàm, nhảy, hoặc câu lệnh đánh nhăn.

Chúng ta có thể tạo ra một mã tốt hơn cho câu lệnh ba địa chỉ $a := b + c$ nếu ta chỉ tạo ra một mã lệnh đơn ADD Rj, Ri và đặt kết quả a trong thanh ghi Ri. Lệnh trên thực hiện được chỉ nếu thanh ghi Ri chứa b, Rj chứa c và b sẽ không còn cần đến nữa sau câu lệnh này.

Nếu Ri chứa b nhưng c là một vị trí trong bộ nhớ, thì chúng ta có thể tạo ra chuỗi:

ADD c, Ri

hoặc

MOV c, Rj

ADD Rj, Ri

Bộ diễn tả thanh ghi và địa chỉ

Thuật toán sinh mã dùng bộ diễn tả này để theo dõi nội dung các thanh ghi và địa chỉ của các tên:

- *Bộ diễn tả thanh ghi* theo dõi những gì xuất hiện trên các thanh ghi. Nó được dùng mỗi khi cần một thanh ghi mới. Ta coi là khởi đầu tất cả các thanh ghi đều rỗng. Theo quá trình sinh mã, mỗi một thanh ghi sẽ giữ một giá trị không hoặc nhiều tên tại một thời điểm nào đó.
- *Bộ diễn tả địa chỉ* theo dõi việc cấp phát khi giá trị hiện tại của tên có thể gấp trong lúc chạy. Vị trí có thể là một thanh ghi, một ngăn xếp, một địa chỉ bộ nhớ hoặc một tập nào đó, khi được sao chép một giá trị cũng nằm đúng ở nơi thích hợp. Thông tin này có thể được lưu trong bảng ký hiệu và được dùng để xác định phương pháp truy nhập đối với một tên.

Thuật toán sinh mã

Thuật toán sinh mã có đầu vào là chuỗi các câu lệnh mã ba địa chỉ dùng để tạo khôi cơ bản. Đối với mỗi mã ba địa chỉ dạng $x := y op z$ chúng ta thực hiện các bước sau:

1. Gọi hàm *gettreg()* để xác định vị trí L nơi phép tính $y op z$ sẽ được thực hiện. L thường là một thanh ghi nhưng cũng có thể là một vị trí nhớ. Hàm *gettreg* được cho như phần dưới.
2. Gọi bộ mô tả địa chỉ cho y để xác định y' là vị trí hiện thời của y . Ta chọn y' là thanh ghi nếu giá trị y hiện tại vừa trong thanh ghi vừa trong bộ nhớ. Nếu giá trị y không ở L thì sinh ra câu lệnh *MOV y', L* để đặt một bản sao của y vào L.
3. Sinh ra lệnh *OP z'*, L với z' là vị trí hiện thời của z . Tiếp tục, ta lại chọn thanh ghi nếu z nằm ở cả thanh ghi lẫn bộ nhớ. Cập nhật lại địa chỉ của x để chỉ rằng x nằm tại vị trí L. Nếu L là một thanh ghi, cập nhật bộ mô tả của nó để chỉ rằng nó chứa giá trị của x và loại x khỏi các bộ mô tả thanh ghi khác.

4. Nếu giá trị hiện thời của y hoặc z (hoặc cả hai) không được dùng tiếp nữa, không tồn tại khi kết thúc khối, và đang nằm trên các thanh ghi, thì chọn bộ mô tả thanh ghi để chỉ rằng sau khi thực hiện $x := y \text{ op } z$ thì các thanh ghi đó sẽ không còn chứa y, z nữa.

Hàm Getreg

Hàm này trả lại vị trí L để giữ giá trị x cho phép gán $x := y \text{ op } z$.

1. Nếu tên y có trong một thanh ghi và y không được dùng tiếp sau khi thực hiện $x := y \text{ op } z$, thì trả lại y cho L. Cập nhật bộ mô tả địa chỉ cho y để chỉ rằng y không còn tồn tại trong L nữa.
2. Quay lại 1, trả lại một thanh ghi rỗng cho L nếu có một.
3. Quay lại 2 nếu x được dùng tiếp trong khối, hoặc op là một phép toán như lấy chỉ số (đòi hỏi thanh ghi), tìm một thanh ghi dùng được R. Lưu giá trị R tại một nơi nào đó trong bộ nhớ (bằng lệnh MOV R, M) nếu nó không sẵn trong vị trí bộ nhớ thích hợp M thì cập nhật bộ mô tả địa chỉ cho M và trả lại R. Nếu R giữ giá trị của vài biến, cần phải sinh ra một lệnh MOV cho mỗi giá trị cần lưu. Một thanh ghi thích hợp có thể là cái trỏ đến các dữ liệu xa nhất về sau này, hoặc cái mà dữ liệu của nó cũng có trong bộ nhớ.
4. Nếu x không được dùng trong khối, hoặc không tìm được thanh ghi thích hợp, thì chọn vị trí trong bộ nhớ của x cho L.

Ví dụ 10.2: Lệnh gán $d := (a-b)+(a-c)+(a-c)$ có thể được chuyển thành chuỗi các mã ba địa chỉ như sau:

$t := a - b$

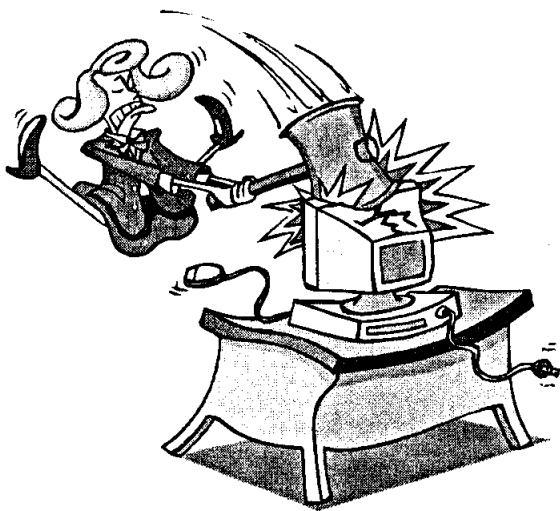
$u := a - c$

$v := t + u$

$d := v + u$

với d sẽ sống cho đến cuối. Thuật toán sinh mã ở trên có thể sinh ra mã như ở bảng dưới.

Câu lệnh	Mã được sinh	Bộ mô tả thanh ghi	Bộ mô tả địa chỉ
		Các thanh ghi rỗng	
$t := a - b$	MOV a, R0 SUB b, R0	R0 chứa t R0 chứa a R0 chứa b	t trong R0
$u := a - c$	MOV a, R1 SUB c, R1	R0 chứa t R1 chứa u R1 chứa a R1 chứa c	t trong R0 u trong R1
$v := t + u$	ADD R1, R0	R0 chứa v R1 chứa u R1 chứa t R1 chứa u	u trong R1 v trong R0
$d := v + u$	ADD R1, R0 MOV R0, d	R0 chứa d R0 chứa v R0 chứa u R0 chứa t R0 chứa u	d trong R0 d trong R0 và trong bộ nhớ



Hình 10.2. Hãy cố kiên nhẫn, bạn sắp hoàn thành công việc rồi.

Phần II

THỰC HÀNH

Chương 1

ĐẶT VẤN ĐỀ

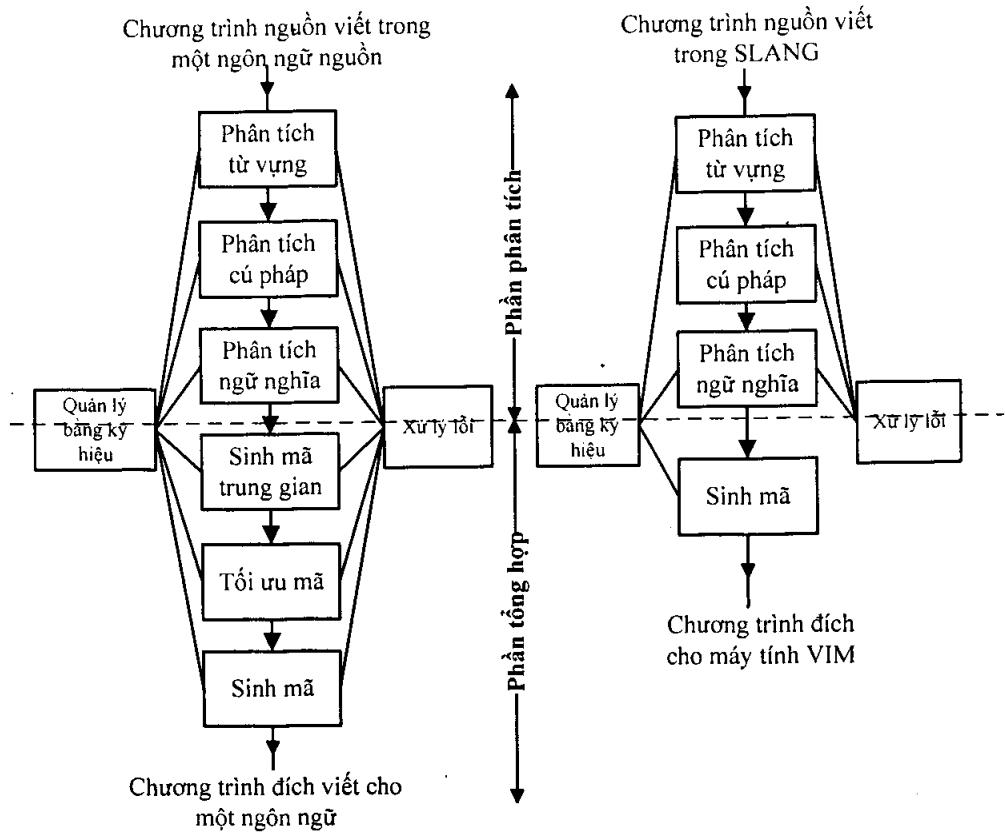
Lý thuyết môn học *Chương trình dịch* là một trong những lý thuyết chuyên ngành khó của công nghệ thông tin. Để nắm vững và đi sâu vào lý thuyết này thì không cách nào hơn là tự mình xây dựng lấy một chương trình dịch hoàn chỉnh, qua đó ta có thể vỡ ra nhiều vấn đề về lý thuyết và thực hành. Các kĩ thuật của chương trình dịch có nhiều điều bổ ích và rất lý thú. Nó có thể áp dụng rất tốt sang các bài toán khác.

Nhằm giúp các bạn thực hành môn học, chúng tôi biên soạn phần thực hành này. Nhiệm vụ thực hành của chúng ta như sau:

Viết một chương trình biên dịch hoàn chỉnh. Chương trình này sẽ dịch các chương trình nguồn viết trong một ngôn ngữ lập trình bậc cao và cho ra chương trình đích trong một ngôn ngữ lập trình bậc thấp, chạy được trên một loại máy tính nào đó.

Chương trình dịch loại biên dịch (sau này sẽ gọi đơn giản là chương trình dịch nếu không sợ hiểu lầm) sẽ dịch toàn bộ chương trình nguồn thành chương trình đích rồi mới chạy. Nó khác với loại thông dịch là dịch rồi cho chạy lần lượt từng câu lệnh. Đa số các chương trình dịch ngày nay là loại biên dịch.

Như phần lý thuyết đã cho thấy, chương trình dịch có nhiều giai đoạn, mỗi giai đoạn phải áp dụng những lý thuyết và những kĩ thuật khác nhau. Đồng thời, các chương trình dịch cho cùng cặp ngôn ngữ vào /ra có thể có nhiều mức độ phức tạp và hoàn chỉnh khác nhau. Do đó, vấn đề đặt ra với chúng ta là biết chọn độ phức tạp và hoàn chỉnh vừa đủ để thực hành và hoàn thành được nó.



Hình 1.1. Sơ đồ khái của một chương trình dịch đầy đủ và chương trình ta sẽ thực hành.

Chúng ta sẽ không thực hiện mọi khái chức năng của chương trình dịch mà chỉ các chức năng cơ bản và bỏ đi hai khái (bạn có thể tự phát triển chúng): sinh mã trung gian và tối ưu mã. Ta cũng bỏ đi một số đường nối ít dùng giữa các khái: trong các chương trình dịch bình thường thì khái phân tích cú pháp không cần làm việc với bảng ký hiệu và bộ sinh mã không tạo lỗi nên có thể bỏ các đường nối giữa chúng. Sơ đồ khái của một chương trình dịch đầy đủ và chương trình dịch ta sẽ thực hành được trình bày ở hình 1.1 để bạn đọc tiện so sánh.

Đầu tiên, ta sẽ nghiên cứu về một ngôn ngữ lập trình bậc cao (tên là SLANG) nhưng nhỏ và vừa đủ phức tạp mà ta sẽ viết chương trình dịch cho nó (chương 2, phần A). Căn cứ vào định nghĩa của ngôn ngữ lập trình này, ta sẽ thiết kế và lập trình cho ba khái đầu: phân tích từ vựng, phân tích cú pháp và phân tích ngữ nghĩa.

Các kĩ thuật dùng trong phần phân tích từ vựng (chương 3) là kĩ thuật viết chương trình dựa vào đồ thị chuyển theo phương pháp diễn giải - một phương pháp thực hiện tương đối chân phương, dễ làm.

Trong phần phân tích cú pháp (chương 4), ta sẽ học cách viết bộ phân tích đệ quy đi xuống. Việc xây dựng bộ phân tích được tiến hành như một công nghệ: áp dụng một cách máy móc các luật xây dựng. Do đó có thể viết chương trình phân tích một cách đơn giản và chính xác (hoặc tự tạo ra các công cụ sinh tự động chương trình nguồn của bộ phân tích cú pháp).

Phản tiếp theo (chương 5) ta sẽ học về cách thiết kế và thực hiện bảng ký hiệu qua kĩ thuật cây nhị phân kết hợp bảng băm (hash table). Phần này cũng trình bày về kĩ thuật cài đặt cú pháp điều khiển thông qua kĩ thuật "trên cùng chuyến bay" - hay kĩ thuật cho phép duyệt các luật ngữ nghĩa trong lúc đang phân tích. Nhờ việc duyệt các luật ngữ nghĩa mà có thể kiểm tra kiểu (tức là thực hiện nhiệm vụ chính của bộ phân tích ngữ nghĩa) và sinh ra mã cho máy tính ảo VIM trong hai chức năng cuối (chương 5 và chương 6).

Đầu ra của chương trình dịch là các chương trình đích viết trong một ngôn ngữ bậc thấp, dùng để chạy trên một máy tính ảo VIM. Máy tính ảo VIM chỉ là một cái máy do ta tự thiết kế và lập trình mô phỏng. Do đó, ta phải nghiên cứu, đưa ra cú pháp và bộ lệnh cho máy tính này sao cho gần giống với các máy tính thông thường. Ta cũng học về cách thực hiện máy tính ảo này nhờ một chương trình dịch loại khác - loại thông dịch (chương 2, phần B). Tuy mất công xây dựng máy tính ảo chứ không dùng máy tính thật, nhưng sẽ giảm được độ phức tạp và công sức xây dựng phần sinh mã (vốn đòi hỏi rất nhiều công sức và kinh nghiệm) đến mức thực hiện được.

Ta sẽ học về cách xử lý lỗi của chương trình dịch (chương 7). Trong giáo trình này, phần này được giản lược nhiều nên rất đơn giản.

Trong chương cuối (chương 8), giáo trình sẽ đề cập đến một số cải tiến ngôn ngữ đầu vào và đầu ra cùng với hướng dẫn cách sửa chương trình cho phù hợp với các cải tiến đó. Sau khi cải tiến, ngôn ngữ đầu vào đã trở thành một ngôn ngữ bậc cao thật sự, có khả năng và mức độ phức tạp gần bằng các ngôn ngữ lập trình thông thường.

Thiết kế của các phần đều rất "mở", nghĩa là ta có thể dễ dàng thêm, sửa mà không làm rối, hỏng thiết kế tổng quát. Do đó, nhiều phần mở rộng được ra như một bài tập mà bạn có thể thực hiện dễ dàng.

Theo các chương và các bài tập, giáo trình đưa ra các gợi ý, các vấn đề mới, các hướng phát triển, các dự án để các bạn say mê có thể tự mình tìm tòi, nghiên cứu. Bạn có thể áp dụng những kiến thức và kinh nghiệm thu được để xây dựng chương trình dịch hoàn chỉnh cho một ngôn ngữ lập trình thật sự. Giáo trình có cung cấp thiết kế của một số ngôn ngữ lập trình (trong phụ lục A) để bạn tham khảo. Cũng theo các chương và bài tập, độ khó của

kiến thức và yêu cầu đối với bạn sẽ tăng dần. Trong các chương đầu, bạn sẽ được học và làm theo. Đến các chương cuối bạn phải tự mình giải quyết lấy một số vấn đề.

Ngoài nhiệm vụ chính là giúp bạn thực hành về chương trình dịch, giáo trình còn đưa ra một số chương trình ứng dụng có ích (trong phụ lục B). Các chương trình này đều phát triển từ các kĩ thuật của chương trình dịch.

Các chương trình trong giáo trình chủ yếu được viết trong ngôn ngữ C, một ngôn ngữ lập trình bậc cao rất mạnh và mềm dẻo, tiện cho các bài tập lớn. Tuy vậy, để thích hợp với thói quen của nhiều bạn, giáo trình cũng đưa ra các chương trình nguồn tương đương viết trong ngôn ngữ Pascal trong phần phụ lục C.

Hi vọng sau khi thực hành, bạn sẽ cải thiện thêm tay nghề thiết kế và lập trình, có cái nhìn logic, sâu sắc hơn vào lý thuyết và thực hành môn học này.

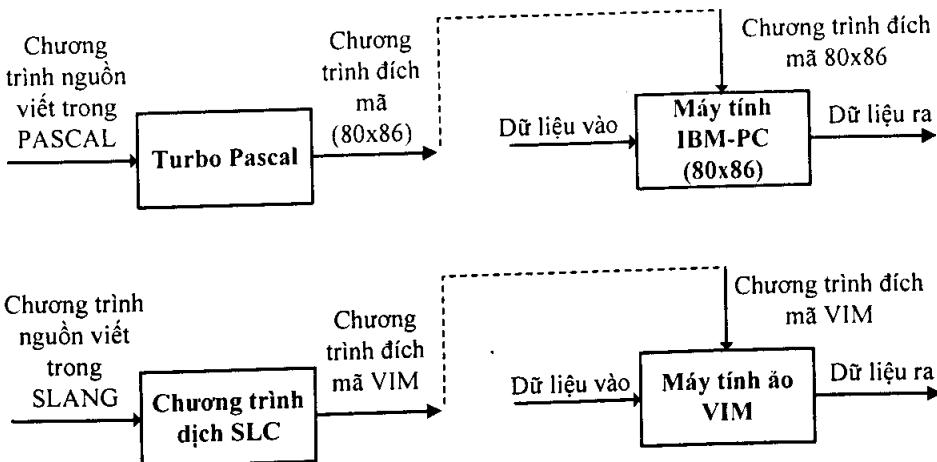
Tóm tắt những kĩ thuật chính sẽ thực hành:

<i>Chức năng</i>	<i>Kĩ thuật chính sẽ thực hành</i>
Phân tích từ vựng	Dựng đồ thị chuyển và viết chương trình cho nó theo phương pháp diễn giải.
Phân tích cú pháp	Phân tích theo phương pháp LL (1) với kĩ thuật đệ quy đi xuống - xây dựng chương trình phân tích cú pháp nhờ áp dụng luật xây dựng.
Cú pháp điều khiển	Kĩ thuật gắn các luật ngữ nghĩa (cú pháp điều khiển) với luật cú pháp và duyệt chúng trong lúc đang phân tích: kĩ thuật “trên cùng chuyến bay”.
Phân tích ngữ nghĩa	Kiểm tra kiểu nhờ cú pháp điều khiển.
Sinh mã	Sinh mã cho máy tính ảo nhờ cú pháp điều khiển.
Quản lý bảng ký hiệu	Tổ chức bảng ký hiệu thông qua cây nhị phân.
Xử lý lỗi	Phát hiện và khôi phục sau khi gặp lỗi trong các phần phân tích từ vựng, cú pháp và ngữ nghĩa. Khôi phục lỗi cú pháp theo chiến lược “khôi phục cụm từ”.

Chương 2

NGÔN NGỮ NGUỒN VÀ MÁY TÍNH ẢO

Trong phần này ta sẽ tìm hiểu về ngôn ngữ nguồn SLANG, ngôn ngữ đích VIM của chương trình dịch mà ta sẽ viết. Nói cách khác, đó là các ngôn ngữ của đầu vào và ra. Ta sẽ tìm hiểu và thiết kế máy tính chạy được VIM - máy tính ảo VIM. Quan hệ giữa SLANG, VIM và máy tính ảo như hình 2.1 (so sánh tương ứng với hệ thống dịch điển hình Turbo Pascal):



Hình 2.1. Chương trình dịch SLC dịch từ ngôn ngữ lập trình SLANG sang mã VIM và máy tính ảo chạy chương trình mã VIM. So sánh với chương trình dịch Turbo Pascal cho máy IBM – PC.

A. NGÔN NGỮ NGUỒN SLANG (bước 1)¹

Ngôn ngữ nguồn SLANG là một ngôn ngữ lập trình bậc cao nhỏ và đơn giản (SLANG có thể coi là viết tắt của các từ Source LANGuage, hoặc Small LANGuage, hoặc Simple LANGuage). Thiết kế SLANG trong bước

¹ Các mở rộng SLANG (bước 2) được trình bày trong chương 8.

đầu phải đạt được hai mục đích: Trước hết, ngôn ngữ này cần phải đủ phirc tạp để ta có thể thực hành đầy đủ các quá trình dịch. Do đó ngôn ngữ này phải có đủ các thành phần và cấu trúc cơ bản cũng như cần có thêm các thủ tục. Thứ hai, ngôn ngữ này lại phải đủ nhỏ và đơn giản. Do đó, các thủ tục lại không có tham số và chỉ có một kiểu dữ liệu (kiểu số nguyên). Ngoài ra, ngôn ngữ này phải dễ mở rộng để trong các bước sau ta có thể mở rộng như thêm các cấu trúc mới, thêm tham số cho thủ tục...

I. MÔ TẢ NGÔN NGỮ SLANG

Ngôn ngữ SLANG cho phép viết các biểu thức số học, trong đó dùng được các hằng số, biến và các biểu thức con nằm trong các cặp dấu ngoặc đơn cũng được coi như các toán hạng. Chỉ có một kiểu dữ liệu duy nhất là kiểu số nguyên (integer). Thứ tự ưu tiên các phép toán trong các biểu thức số học và quan hệ như sau: Phép toán lấy nghịch đảo một ngôi (biểu diễn bằng NEG) và lấy giá trị tuyệt đối (ABS) có độ ưu tiên cao nhất, các phép toán nhân (*), chia (/) và modulo v (|) có thứ tự tiếp theo, và sau đó là các phép toán cộng (+), trừ t (-) và cuối cùng là các phép toán quan hệ như v =, <>, <, <=, >, >=.

Ta không xem xét các biểu thức logic Boolean do việc dịch chúng cũng tương tự như các biểu thức số học (sẽ được coi như một bài tập).

SLANG là một ngôn ngữ lệnh nên nó có một lệnh gán. Nó cũng có các câu lệnh cho phép chọn, lặp, lời gọi các thủ tục và vào /ra. Ngôn ngữ cũng có một câu lệnh điều kiện gọi là câu lệnh -if, và câu lệnh lặp -while. Ngôn ngữ không có câu lệnh case, for và repeat do chúng chỉ là các biến thể khác của các câu lệnh -if và câu lệnh -while (sẽ được coi như một bài tập).

SLANG không có câu lệnh goto. Lý do là các câu lệnh goto rất dễ làm cho chương trình có cấu trúc tồi và quan trọng hơn là rất khó thực hiện đối với chương trình dịch.

Một chương trình SLANG sẽ bao gồm các khai báo và các câu lệnh. Các khai báo được dùng để mô tả dữ liệu, còn các câu lệnh được dùng để mô tả các hành động dựa trên dữ liệu.

Dữ liệu có thể là số, hằng, biến và các thủ tục. Các dữ liệu này thường chỉ được phép hoạt động trong một phần nào đó của chương trình và được gọi là phạm vi (scope). Các thủ tục lại có thể được đặt trong lòng của một phần khác (lồng nhau).

Bản thiết kế đầu của SLANG được giữ nhỏ và đơn giản nhất có thể được.

II. ĐỊNH NGHĨA NGÔN NGỮ SLANG

a. Từ tố (*token*)

Định nghĩa từ tố thường được biểu diễn ở dạng biểu thức chính quy.

Quy ước của biểu thức chính quy (Regular Expression) :

| nghĩa là hoặc (or)

() nhóm các thành phần

* lặp lại không hoặc nhiều lần

Những từ tố đơn giản được định nghĩa bằng cách đưa trực tiếp mẫu của nó (như các định nghĩa về từ khoá, dấu toán học...). Những từ tố phức tạp hơn sẽ được định nghĩa bằng biểu thức chính quy như tên và số.

Phân phân tích từ vựng phải có khả năng phân tích chương trình nguồn và đưa ra được các từ tố như sau:

Tên và số

Từ tố	Ký hiệu
Ident_token	Tên
Num_token	số

Bảng các ký hiệu đặc biệt

open_token	(
close_token)
list_token	,
period token	.
separator token	;
becomes token	:=
open comment token	{
close comment token	}
plus token	+
minus token	-
times token	*

over token	/
modulo token	
equal token	=
not equal token	\diamond
less than token	<
less or equal token	\leq
greater than token	>
greater or equal token	\geq

Tùy khoá

begin token	BEGIN
end token	END
int token	INT
var token	VAR
procedure token	PROC
call token	CALL
read token	READ
write token	WRITE
if token	IF
then token	THEN
else token	ELSE
fi token	FI
while token	WHILE
do token	DO
od token	OD
negate token	NEG
absolute token	ABS

Chú ý:

- Các từ khoá và tên không phân biệt chữ hoa, chữ thường
- FI là từ khoá đánh dấu kết thúc câu lệnh IF, OD là từ khoá đánh dấu kết thúc câu lệnh WHILE.

- Các chữ thích được xử lý đơn giản bằng cách bỏ đi (tương tự như các dấu xuống dòng, tab, khoảng trắng).

Từ vựng của SLANG được xác định từ các chữ cái, chữ số, các ký hiệu. Trong đó:

chữ cái → a | b | c | ... | z | A | B |...| Z

chữ số → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

và

tên → *chữ cái* (*chữ cái* | *chữ số*)*

số → *chữ số* (*chữ số*)*

b. Cú pháp và ngữ nghĩa

Cú pháp của ngôn ngữ SLANG là một văn phạm phi ngữ cảnh và được biểu diễn bằng ký pháp BNF.

Ký pháp BNF

BNF (Backus - Naur Form) thực chất chỉ là một biểu diễn khác của VPPNC và có quy ước như sau:

- Các ký hiệu viết hoa biểu diễn các ký hiệu không kết thúc (nonterminal), cũng có thể thay bằng một từ in nghiêng,
- Các ký hiệu viết chữ nhỏ biểu diễn các ký hiệu kết thúc (terminal), cũng có thể thay bằng một từ in đậm,
- Ký hiệu → là ký hiệu chỉ phạm trù cú pháp ở về trái được giải thích bởi về phải,
- Ký hiệu | chỉ sự lựa chọn.

Trong giáo trình này, ta cố gắng viết gộp các lệnh có cùng về trái lại với nhau thành dạng $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$, và dùng thêm các quy ước sau (mượn của quy ước biểu thức chính quy) để viết gọn hơn:

- Ký hiệu () nhóm các thành phần.
- Ký hiệu * chỉ lặp lại không hoặc nhiều lần.

Ngoài ra, ta có quy ước:

- Ký hiệu [] chỉ không hoặc có. Ta viết [A] có nghĩa là $(A | \epsilon)$, trong đó ϵ là một ký hiệu đặc biệt chỉ xâu rỗng hoặc ký hiệu rỗng.

- Ký hiệu $^+$ chỉ lặp lại một hoặc nhiều lần.

Để diễn tả phần câu lệnh bao gồm một hoặc nhiều câu lệnh phân cách nhau bằng dấu chấm phẩy (dấu chấm phẩy sẽ tạo ra từ tố phân cách **separator_token**) đáng lẽ viết bằng các luật cú pháp như sau:

phâncâulệnh \rightarrow *câulệnh*

phâncâulệnh \rightarrow *câulệnh separator_token câulệnh*

phâncâulệnh \rightarrow *câulệnh separator_token câulệnh separator_token câulệnh*

...

thì ta có thể viết gọn lại thành:

phâncâulệnh \rightarrow *câulệnh (separator_token câulệnh)**

Các biểu diễn văn phạm phi ngữ cảnh như trên đôi khi còn gọi là *văn phạm biểu thức chính quy về phải* hoặc *văn phạm phi ngữ cảnh mở rộng*. Chú ý là các văn phạm này hoàn toàn tương đương với VPPNC. Tức là mọi VPPNC đều chuyển được về dạng biểu diễn VPPNC mở rộng và ngược lại. Lợi ích của cách viết này như ta đã thấy là gọn hơn, nhưng chủ yếu là *dễ áp dụng luật xây dựng chương trình phân tích cú pháp theo phương pháp đê quy đi xuống* như sẽ trình bày trong chương 4.

Các luật cú pháp và ngữ nghĩa

Trong các luật cú pháp dưới đây, những từ được in đậm là các từ tố (và là ký hiệu kết thúc) do bộ phân tích từ vựng trả về. Đối với từng cấu trúc ta đều có phần giải thích ngữ nghĩa (luật ngữ nghĩa), một số có thêm các điều kiện ràng buộc. Các chú thích bằng tiếng Anh đặt trong cặp ngoặc đơn là tên các thủ tục ứng với các luật mà ta sẽ dùng viết chương trình cho tiện.

Chương trình (program_declaration)

khaibáochươngtrình \rightarrow *khối period_token*

Ngữ nghĩa: Một môi trường lập trình được tạo ra. Sau đó khối (block) được cho chạy và cuối cùng môi trường bị xoá đi.

Các khối (block)

khối \rightarrow *begin_token phânkhaibáo phâncâulệnh end_token*

Ngữ nghĩa: Phần khai báo được cho hoạt động, sau đó đến phần câu lệnh.

Các khai báo (declaration_part)

$phân_khai_báo \rightarrow (khai_báo_hằng \mid khai_báo_biến\ k)^* \ khai_báo_thủ_tục^*$

Ngữ nghĩa: Các khai báo được xử lý theo thứ tự chúng xuất hiện.

Các khai báo hằng (constant_declaration và type_declarer)

$khai_báo_hằng \rightarrow khai_báo_kiểu$

$ident_token \ equal_token \ num_token$

$(list_token \ ident_token \ equal_token \ num_token)^*$

$separator_token$

$khai_báo_kiểu \rightarrow int_token$

Ràng buộc: Mỗi tên trong khai báo hằng phải đại diện cho một con số. Phạm vi hoạt động của một hằng là trong khối có khai báo của nó. Một tên không được định nghĩa quá một lần trong cùng một khối.

Ngữ nghĩa: Mọi tên trong danh sách khai báo hằng đều được buộc với một con số tương ứng.

Các khai báo biến (variable_declaration)

$khai_báo_biến \rightarrow var_token$

$khai_báo_kiểu \ ident_token \ (list_token \ ident_token)^*$

$separator_token$

Ràng buộc: Mỗi tên trong khai báo biến phải ứng với một biến. Phạm vi của một tên là trong khối có khai báo nó. Một tên không được định nghĩa quá một lần trong cùng một khối.

Ngữ nghĩa: Đối với mỗi tên trong danh sách, một vị trí mới (gọi là *loc*) được tạo ra để lưu nó và *loc* được buộc với tên này.

Các khai báo thủ tục (procedure_declaration)

$khai_báo_thủ_tục \rightarrow procedure_token \ ident_token \ khối$
 $separator_token$

Ràng buộc: Mỗi tên trong khai báo thủ tục phải ứng với một thủ tục. Phạm vi của một tên là trong khối có khai báo nó. Một tên không được định nghĩa quá một lần trong cùng một khối.

Ngữ nghĩa: Một khối được buộc vào một tên.

Các câu lệnh (statement_part và statement)

$\text{phâncâulệnh} \rightarrow \text{câulệnh} (\text{ separator_token } \text{câulệnh})^*$

$\text{câulệnh} \rightarrow \text{câulệnh_gán}$
| câulệnh_if
| câulệnh_while
| câulệnh_call
| câulệnh_read
| câulệnh_write

Ngữ nghĩa: Các câu lệnh được xử lý theo thứ tự chúng xuất hiện.

Các câu lệnh gán (assignment_statement và left_part)

$\text{câulệnh_gán} \rightarrow \text{phântrái becomes_token} \text{biểuthức}$

$\text{phântrái} \rightarrow \text{ident_token}$

Ràng buộc: Câu lệnh gán phải xuất hiện trong phạm vi hoạt động của tên trong vế trái và tên này phải là một biến.

Ngữ nghĩa: Biểu thức được đánh giá và giá trị của nó được ghi vào vị trí buộc với tên.

Câu lệnh if (if_statement)

$\text{câulệnh_if} \rightarrow \text{if_token} \text{quanhé} \text{then_token} \text{phâncâulệnh}$
[$\text{else_token} \text{phâncâulệnh}$]
 fi_token

Ngữ nghĩa: Quan hệ được đánh giá. Nếu giá trị của nó đúng (true) thì phần câu lệnh sau từ tố then (then_token) được chạy. Nếu giá trị của nó không đúng (false) và câu lệnh lại có cấu trúc else thì phần câu lệnh sau từ tố else (else_token) được chạy. Nếu không có cấu trúc else thì không hành động nào được thực hiện. Chú ý là lệnh if đòi hỏi đánh dấu kết thúc bằng từ khoá FI.

Các câu lệnh while (while_statement)

$\text{câulệnh_while} \rightarrow \text{while_token} \text{quanhé} \text{do_token} \text{phâncâulệnh}$
 od_token

Ngữ nghĩa: Quan hệ được đánh giá. Nếu giá trị của nó đúng (true) thì phần câu lệnh được chạy. Sau đó câu lệnh while lại được kiểm tra liên tục,

nếu giá trị của quan hệ không đúng (false) thì không hành động nào được thực hiện. Chú ý là câu lệnh phải kết thúc bằng từ khoá OD.

Các câu lệnh call (call_statement)

câu lệnh call → **call_token** **ident_token**

Ràng buộc: Câu lệnh call phải xuất hiện trong phạm vi hoạt động của tên và tên này phải là tên của một thủ tục. Lời gọi chỉ được xuất hiện sau khi đã khai báo tên và thủ tục đó.

Ngữ nghĩa: Khỏi buộc với tên được cho chạy trong môi trường mà tên được khai báo. Khi hoạt động của câu lệnh call chấm dứt thì phần câu lệnh tại điểm ngắt sẽ được chạy tiếp tục.

Các câu lệnh read (read_statement)

câu lệnh read → **read_token** **open_token**
ident_token (**list_token** **ident_token**)*
close_token

Ràng buộc: Câu lệnh read phải xuất hiện trong phạm vi hoạt động của mọi tên trong danh sách và mọi tên đều phải là biến.

Ngữ nghĩa: Đôi với mỗi tên trong danh sách và theo thứ tự xuất hiện, giá trị đầu vào đầu tiên được đọc và ghi vào vị trí buộc với tên này và sau đó bị loại ra khỏi đầu vào.

Các câu lệnh write (write_statement)

câu lệnh write → **write_token** **open_token**
biểuthức (**list_token** **biểuthức**)*
close_token

Ngữ nghĩa: Đôi với mỗi biểu thức trong danh sách và theo thứ tự xuất hiện, biểu thức được đánh giá và giá trị thu được của biểu thức được thêm vào đầu ra.

Các biểu thức (expression, term, factor, operand, add_operator, multiply_operator, unary_operator)

biểuthức → **hạngthức** (**phéptoán** **côngtrù** **hangthức**)*
hạngthức → **thìratô** (**phéptoán** **nhâncchia** **thìratô**)*
thìratô → [**phéptoán** **mộtngôî**] **toánhang**

toán hạng → **ident_token** | **num_token** | **open_token** *biểu thức*
close_token

phép toán cộng trừ → **plus_token** | **minus_token**

phép toán nhân chia → **times_token** | **over_token** | **modulo_token**

phép toán một ngôi → **negate_token** | **absolute_token**

Ràng buộc: Biểu thức phải xuất hiện trong phạm vi hoạt động của mọi tên có trong toán hạng của biểu thức, và mọi tên phải là biến hoặc hằng số.

Ngữ nghĩa: Một biểu thức là một luật tính một giá trị số của kiểu số nguyên. Giá trị thu được bằng cách áp dụng các phép toán số học với các toán hạng.

Chú ý: Do SLANG không có dữ liệu số thực nên phép chia của nó chỉ là phép chia cắt phần phân.

Các phép tính quan hệ (relation và relational _operator)

quan hệ → *biểu thức toán tử quan hệ* *biểu thức*

toán tử quan hệ → **equal_token** | **not_equal_token**

| **less_than_token** | **less_or_equal_token**

| **greater_than_token** | **greater_or_equal_token**

Ngữ nghĩa: Một quan hệ là một luật tính giá trị chân lý bằng cách áp dụng các phép toán quan hệ vào các giá trị thật sự của biểu thức bên trái và biểu thức bên phải trong quan hệ đó.

Chú thích và khoảng trắng

chú thích → **open_comment_token** bất kì chữ gì trừ những chữ tạo thành từ tổ **close_comment_token**
close_comment_token

khoảng trắng → (*xuống dòng* | *tab* | *cách*)⁺

Ngữ nghĩa: Các chú thích, ký hiệu xuống dòng, tab, dấu cách có ở mọi nơi trong chương trình, không có ý nghĩa gì (đối với chương trình dịch) và được coi là phân cách giữa các từ tổ. Chương trình dịch sẽ chỉ đơn giản bỏ chúng đi.

Các chương trình ví dụ viết trong SLANG

Ví dụ 2.1: Chương trình sau trong ngôn ngữ SLANG đọc các số vào từ bàn phím cho đến khi nhập vào số 0, cộng chúng lại với nhau và cuối cùng in kết quả ra màn hình. Bạn hãy ghi thành file VIDU21.PRO để tiện thử nghiệm về sau.

```

BEGIN VAR INT number, sum;
    sum := 0;
    READ (number);
    WHILE number <> 0 DO
        sum := sum + number;
        READ (number);
    OD;
    WRITE (sum);
END.

```

Ví dụ 2.2: Chương trình sau đọc vào các số từ bàn phím cho đến khi nhập vào số 0, in các số này ra màn hình với các chữ số trong từng số được đặt theo thứ tự ngược lại. Bạn hãy ghi thành file VIDU22.PRO để tiện thử nghiệm về sau. Chú ý là chương trình này có vài lỗi nhỏ.

```

BEGIN
    VAR int number;
    proc reverse
        BEGIN WRITE (number | 10);
        IF number >= 10 THEN
            number := number / 10;
            CALL reverse
        FI
    END

    READ (number)
    WHILE number <> 0 DO
        CALL reverse
        READ (number)
    OD;
END.

```

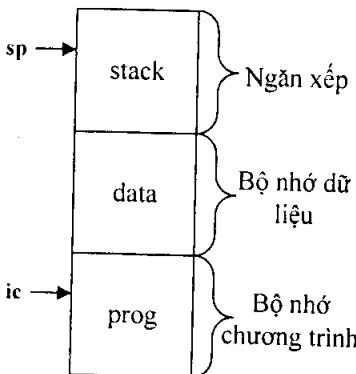
B. MÁY TÍNH ẢO (VIRTUAL MANICHE – VIM)¹

I. VERSION ĐƠN GIẢN ĐẦU TIÊN CỦA MÁY TÍNH ẢO VIM

Trong phần này, chúng ta sẽ nghiên cứu một version đơn giản ban đầu của một máy tính ảo VIM.

¹ Bạn có thể tạm bỏ qua phần này và chỉ cần đọc trước chương 6: Sinh mã.

Việc sinh mã cho một máy tính thực sự và cụ thể nào đó là công việc rất khó và không cần thiết. Ta sẽ tự xây dựng một máy tính đang chạy một bộ mã máy nào đó do ta thiết kế. Rõ ràng là máy tính này không có trong thực tế (máy ảo) nhưng ta sẽ đạt được các mục đích: tìm hiểu được việc sinh mã cho các máy tính nói chung và làm quá trình sinh mã đơn giản để thực hiện dễ dàng. Tuy nhiên, máy tính ảo này vẫn cần phải có các đặc điểm giống với các máy tính thông thường.



Hình 2.2. Cấu trúc máy tính VIM, bao gồm hai thanh ghi ic, sp và bộ nhớ chia thành ba vùng với các chức năng khác nhau.

1. Cấu trúc của máy tính ảo VIM

Máy tính ảo VIM chỉ có hai thanh ghi cùng với bộ nhớ và ngăn xếp. Nó có bộ nhớ chương trình và bộ nhớ dữ liệu nằm tách rời nhau. Mọi chỉ thị chiếm một ô trong bộ nhớ chương trình. Địa chỉ các ô lệnh này là các số tự nhiên. Một thanh ghi là con trỏ chỉ thị *ic* có chứa địa chỉ của lệnh sẽ được thực hiện. Mỗi con số cũng chiếm một ô trong bộ nhớ dữ liệu. Các địa chỉ ô trong bộ nhớ dữ liệu này cũng là các số tự nhiên. Định ngăn xếp được chỉ bằng thanh ghi thứ hai - *sp*.

Các chỉ thị bao gồm một toán tử và nhiều nhất một tham số mà đó có thể là một số hoặc một địa chỉ trong bộ nhớ chương trình hoặc dữ liệu.

Mã toán tử	Phần tham số (giá trị hoặc địa chỉ)
------------	-------------------------------------

Hình 2.3. Cấu trúc chỉ thị lệnh VIM. Phần mã toán tử là bắt buộc, phần tham số có hoặc không tùy theo mã toán tử.

Các toán tử số học và quan hệ không có các tham số. Chúng thực hiện trên một hoặc hai giá trị (toán hạng) đang nằm trên định ngăn xếp số học (gọi tắt là ngăn xếp - stack). Các toán hạng này sẽ được thay thế bằng kết quả thu được nhờ áp dụng phép toán cho các toán hạng đó.

Các chỉ thị nạp (load) và lưu (store), nhảy (jump) đều có một tham số. Lệnh nạp sẽ sao chép dữ liệu của một biến nằm trong bộ nhớ dữ liệu vào đinh của ngăn xếp, trong khi đó lệnh lưu lại loại bỏ giá trị trên đinh của ngăn xếp và đưa nó vào bộ nhớ dữ liệu. Các chỉ thị vào và ra (input/output) không có tham số. Lệnh vào (input) đọc một giá trị từ bàn phím và lưu nó trong ngăn xếp, trong khi đó lệnh ra (output) lại loại bỏ giá trị đang ở đinh ngăn xếp và ghi nó ra màn hình.

Nhằm thực hiện các lệnh điều kiện và while, ta cần chỉ thị nhảy (jump). Chỉ thị nhảy không điều kiện chuyển điều khiển đến lệnh nào đó có địa chỉ chính là tham số của nó. Các lệnh nhảy có điều kiện cũng chuyển điều khiển như vậy nhưng còn tùy thuộc vào giá trị chân lý đang nằm trên đinh ngăn xếp.

Cuối cùng, lệnh dừng (halt) không có tham số làm cho chương trình ngừng thực hiện.

Bộ chỉ thị

Các chỉ thị (instruction) của VIM được chia thành 10 lớp: số học không ngôi, số học hai ngôi, so sánh, nạp và lưu, nhảy không điều kiện, nhảy có điều kiện, gọi hàm và trả về, vào và ra, quản lý lưu trữ, dừng. Chỉ thị gọi (call), trả về (return) và các chỉ thị quản lý lưu trữ sẽ được giới thiệu sau. Định nghĩa của các lớp còn lại được cho trong các bảng dưới đây.

Các phép số học một ngôi

Các chỉ thị này đầu tiên loại bỏ giá trị số nguyên (gọi là intval) khỏi ngăn xếp và thực hiện các hành động dưới đây tùy theo chỉ thị.

chỉ thị	ý nghĩa
neg	Lưu giá trị số nguyên dạng -intval vào ngăn xếp
abs	Lưu giá trị tuyệt đối của intval vào ngăn xếp

Các phép số học hai ngôi

Các chỉ thị này đầu tiên sẽ loại bỏ hai giá trị (số nguyên) intval2 và intval1 theo thứ tự khỏi ngăn xếp và thực hiện các hành động dưới đây tùy theo chỉ thị.

chỉ thị	ý nghĩa
add	Lưu giá trị số nguyên intval1 + intval2 vào ngăn xếp
sub	Lưu giá trị số nguyên intval1 - intval2 vào ngăn xếp

mul	Lưu giá trị số nguyên intval1 * intval2 vào ngăn xếp
dvi	Lưu giá trị số nguyên intval1 / intval2 vào ngăn xếp
mdl	Lưu giá trị số nguyên intval1 intval2 vào ngăn xếp

Các phép so sánh

Các chỉ thị này đầu tiên loại bỏ hai giá trị (số nguyên) intval2 và intval1 theo thứ tự khỏi ngăn xếp và thực hiện các hành động dưới đây tùy theo chỉ thị.

chỉ thị	ý nghĩa
eq	Lưu giá trị chân lý intval1 = intval2 vào ngăn xếp
ne	Lưu giá trị chân lý intval1 ≠ intval2 vào ngăn xếp
lt	Lưu giá trị chân lý intval1 < intval2 vào ngăn xếp
le	Lưu giá trị chân lý intval1 ≤ intval2 vào ngăn xếp
gt	Lưu giá trị chân lý intval1 > intval2 vào ngăn xếp
ge	Lưu giá trị chân lý intval1 ≥ intval2 vào ngăn xếp

Nạp và lưu

chỉ thị	ý nghĩa
ldcon intval	Nạp giá trị số nguyên intval vào ngăn xếp
ldvar address	Sao chép giá trị số nguyên từ địa chỉ bộ nhớ theo address vào định ngăn xếp
stvar address	Loại bỏ giá trị số nguyên ở định ngăn xếp và ghi nó vào địa chỉ bộ nhớ theo address

Nhảy không có điều kiện

chỉ thị	ý nghĩa
jump address	Tiếp tục thực hiện chỉ thị tại địa chỉ address trong bộ nhớ chương trình

Các lệnh nhảy có điều kiện

chỉ thị	ý nghĩa
jift address	Loại bỏ giá trị chân lý (gọi là truthval) khỏi ngăn xếp; nếu giá trị của truthval là true thì tiếp tục thực hiện chỉ thị tại địa chỉ address trong bộ nhớ chương trình, nếu không chỉ thị tiếp theo sẽ được thực hiện
jiff address	Loại bỏ giá trị chân lý (gọi là truthval) khỏi ngăn xếp; nếu giá trị của truthval là false thì tiếp tục thực hiện chỉ thị tại địa chỉ address trong bộ nhớ chương trình, nếu không chỉ thị tiếp theo sẽ được thực hiện

Vào và ra

chỉ thị	ý nghĩa
rdint	Đọc một giá trị số nguyên từ bàn phím và lưu nó vào ngăn xếp
wrint	Loại giá trị nằm trên đỉnh ngăn xếp và ghi nó ra màn hình

Dùng

chỉ thị	ý nghĩa
halt	Ngừng việc thực hiện

Ví dụ 2.3: Ta có lệnh sau:

$$a := b + c$$

trong đó a, b là các biến, c là một hằng số. Giả sử bằng cách nào đó, lệnh này sẽ được chuyển sang mã máy của VIM như sau:

ldvar b

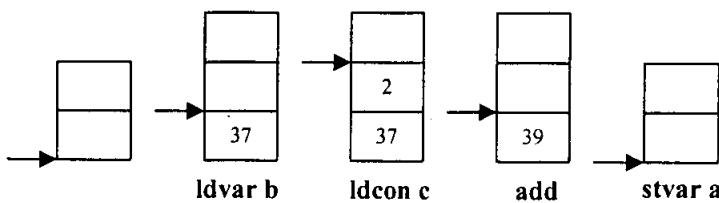
ldcon c

add

stvar a

Sau đây là các biến động trên ngăn xếp khi thực hiện lần lượt các chỉ thị với biến b có giá trị là 37 và hằng số c có giá trị 2. Chỉ thị đầu tiên ldvar b sẽ nạp giá trị của biến b vào ngăn xếp. Chỉ thị tiếp theo ldcon c sẽ nạp giá trị hằng số c vào ngăn xếp. Chỉ thị tiếp theo add sẽ loại hai giá trị số nguyên

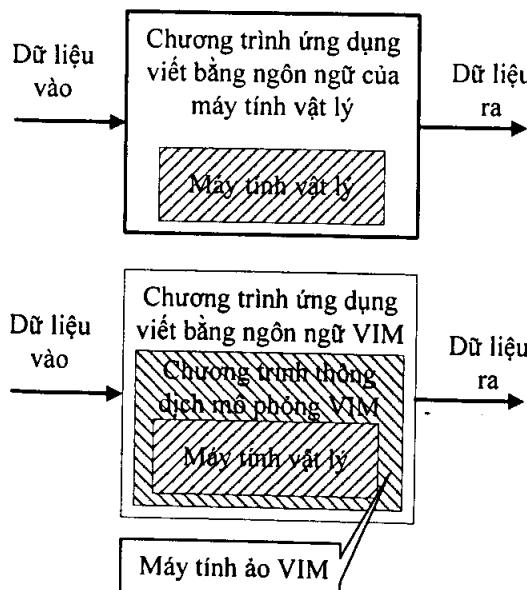
(vừa nạp vào) khỏi ngăn xếp, cộng chúng lại rồi sau đó đẩy lại vào ngăn xếp. Chỉ thị cuối cùng sẽ loại giá trị định ngăn xếp (mà lúc này chính là tổng $b + c$) và lưu nó vào vị trí cấp phát cho biến a .



Hình 2.4. Các thay đổi của ngăn xếp.

2. Trình thông dịch cho máy tính ảo VIM

Để chạy các chương trình viết bằng ngôn ngữ đích VIM - một ngôn ngữ máy không có trong thực tế, ta cần mô phỏng máy tính chạy ngôn ngữ này bằng chương trình thông dịch. Đây cũng là một loại chương trình đích, cho phép dịch và thực hiện lần lượt từng lệnh của chương trình đích VIM. Do cấu trúc của các chỉ thị VIM hết sức đơn giản nên chương trình này cũng rất đơn giản. Ta sẽ mô tả nó dưới đây.



Hình 2.5. So sánh máy tính thường và máy tính ảo VIM.

Máy tính ảo VIM có bộ nhớ chương trình và bộ nhớ dữ liệu khác nhau. Bộ nhớ chương trình được biểu diễn bằng một mảng *prog* có kiểu là *instruction* (chỉ thị). Mỗi một chỉ thị bao gồm phép toán (toán tử) và đối với một số chỉ thị còn có thêm một toán hạng mà đó có thể là *địa chỉ* hoặc giá

trị. Con đếm chỉ thị *ic* trả đến chỉ thị sẽ được thực hiện. Bộ nhớ dữ liệu được biểu diễn bằng một mảng *data* có kiểu là số nguyên.

Trình thông dịch này có một thủ tục *read_code* dùng để đọc chương trình VIM từ file và ghi nó vào mảng *prog*.

Các phép toán số học và quan hệ được thực hiện trong ngăn xếp. Ngăn xếp cũng được dùng cho các phép vào /ra. Ngăn xếp được biểu diễn bằng mảng *stack* với kiểu là số nguyên. Các thủ tục *push* và *pop* được dùng để lưu hoặc loại một giá trị ở đỉnh ngăn xếp và điều chỉnh con trỏ đỉnh ngăn xếp là *sp*. Các giá trị chân lý true và false được biểu diễn trong ngăn xếp là 1 và 0.

Các chỉ thị số học, quan hệ, nạp/tải, vào/ra được thực hiện trong ngăn xếp. Các chỉ thị nhảy làm thay đổi con đếm chỉ thị *ic*. Chương trình sẽ chạy cho đến khi giá trị boolean trong biến *stop* trở thành true. Chương trình nguồn của VIM được cho như dưới đây:

```
#include <stdio.h>
#include <conio.h>

#define max_prog 1000
#define max_data 100
#define max_stack 100

typedef enum {
    true = 1, false = 0
} boolean;

/* Liệt kê các chỉ thị của VIM và mã hoá chúng bằng cách gán cho mỗi lệnh một
mã khác nhau */
typedef enum {
    neg, abs_, add, sub, mul, dvi, mdl, eq, ne, lt, le, gt, ge,
    ldcon, ldvar, stvar, jump, jift, jiff, rdint, wrint, halt
} operation;

/* Khai báo cấu trúc của chỉ thị */
typedef struct {
    operation code; /* Phần mã cho biết đó là chỉ thị nào */
    union { /* Phần tham số, tùy thuộc vào chỉ thị có thể là giá trị (như lệnh
ldcon) hoặc địa chỉ (như cho các lệnh ldvar, stvar, jump, jift, jiff) */
        int value; /* for: ldcon */
        int address; /* for: ldvar, stvar, jump, jift, jiff */
    } u;
} instruction;
instruction prog[max_prog];
```

```

int          data[max_data];
int          stack[max_stack];
int          sp;

void read_code(void)
{
    /* phần này đọc chương trình mã máy VIM vào và đặt trong mảng prog */
}

void push(int x)
{
    stack[sp++] = x;
}

void pop(int *x)
{
    *x = stack[--sp];
}

void read_int(int *x)
{
    /* Đọc một số nguyên từ bàn phím và gán vào tham số x */
}

void write_int(int x)
{
    /* In ra màn hình giá trị số nguyên x */
}

void main(void)
{
    instruction    *instr;
    int            ic, next, left_operand, right_operand, operand, result;
    boolean        truthval, stop;

    read_code();
    ic = 0;
    sp = 0;
    stop = false;

    while (!stop) {
        next = ic + 1;
        instr = &prog[ic];

        switch (instr->code) {
            case neg:
            case abs_:
                pop (&operand);
                switch (instr->code) {
                    case neg:

```

```

        result = -operand;
        break;
case abs_:
    if (operand < 0)
        result = -operand;
    else
        result = operand;
    break;
}
push(result);
break;

case add:
case sub:
case mul:
case dvi:
case md1:
    pop(&right_operand);
    pop(&left_operand);
    switch (instr->code) {
        case add:
            result = left_operand + right_operand;
            break;

        case sub:
            result = left_operand - right_operand;
            break;

        case mul:
            result = left_operand * right_operand;
            break;

        case dvi:
            result = left_operand / right_operand;
            break;

        case md1:
            result = left_operand % right_operand;
            break;
    }
    push (result);
    break;

case eq:
case ne:
case lt:
case le:
case gt:
case ge:

```

```

pop(&right_operand);
pop(&left_operand);
switch (instr->code) {
    case eq:
        truthval = (left_operand == right_operand);
        break;

    case ne:
        truthval = (left_operand != right_operand);
        break;

    case lt:
        truthval = (left_operand < right_operand);
        break;

    case le:
        truthval = (left_operand <= right_operand);
        break;

    case gt:
        truthval = (left_operand > right_operand);
        break;

    case ge:
        truthval = (left_operand >= right_operand);
        break;
    }

    if (truthval)
        result = 1;
    else
        result = 0;
    push (result);
    break;

case ldcon:
    push (instr->u.value);
    break;

case ldvar:
    push (data[instr->u.address]);
    break;

case stvar:
    pop (&data[instr->u.address]);
    break;

case jump:
    next = instr->u.address;
    break;
}

```

```

case jift:
    pop (&operand);
    if (operand == 1)
        next = instr->u.address;
    break;

case jiff:
    pop (&operand);
    if (operand == 0)
        next = instr->u.address;
    break;

case rdint:
    read_int (&operand);
    push (operand);
    break;

case wrint:
    pop (&operand);
    write_int (operand);
    break;

case halt:
    stop = true;
    break;
}

ic = next;
}

```

Chương trình thông dịch ở trên còn chưa có phần phát hiện lỗi. Nếu chương trình VIM hoạt động đúng đắn thì chỉ có lỗi trong trường hợp ngăn xếp bị tràn. Ta có thể thêm phần kiểm tra vào thủ tục *push* để dễ dàng phát hiện được khi bị tràn.

3. Chuyển đổi từ SLANG sang VIM

Biểu thức và quan hệ

Trong phần trước, chúng ta đã thấy mọi phép tính số học và quan hệ đều được thực hiện thông qua sử dụng ngăn xếp. Nếu ta chuyển đổi một biểu thức từ dạng thông thường (dạng trung tố) thành dạng hậu tố thì việc chuyển đổi sang các chỉ thị dùng ngăn xếp hoàn toàn đơn giản. Ta minh họa điều này qua hai ví dụ sau đây.

Ví dụ 2.4: Dạng hậu tố của $a < b$ là $a\ b\ <$. Giả sử a và b là các ký hiệu của các biến thì các chỉ thị sau là tương ứng:

ldvar a

ldvar b

lt

Ví dụ 2.5: Biểu thức $(a+b)*c+d+e*f$ có dạng hậu tố là $a\ b\ +\ c\ *\ d\ +\ e\ f\ * \ +$ và các chỉ thị tương ứng là:

ldvar a

ldvar b

add

ldvar c

mul

ldvar d

add

ldvar e

ldvar f

mul

add

Trong cả hai ví dụ trên, kết quả thu được vẫn tiếp tục được đặt trong ngăn xếp. Ta có thể thấy các chỉ thị VIM được sinh ra lần lượt theo từng ký hiệu ở dạng trung tố.

Như vậy, nếu cho trước ký pháp hậu tố, có thể sinh trực tiếp mã ngăn xếp. Còn nhiệm vụ quan trọng và khó hơn của chương trình dịch là chuyển từ dạng thông thường (trung tố) sang hậu tố. Ta sẽ bắt đầu bằng cách xem xét việc chuyển đổi biểu thức và quan hệ sang dạng hậu tố.

Trong phần trước, ta định nghĩa ký hiệu không kết thúc *bieu_thuc* như sau:

bieu_thuc → *hang_thuc* (*phéptoán_cộngtrừ* *hang_thuc*)*

Sản xuất này cho phép viết các biểu thức bao gồm các phép cộng trừ ở dạng ký pháp trung tố (thông thường). Ta chuyển sang ký pháp hậu tố thành:

< *bieu_thuc* > → < *hang_thuc* > (< *hang_thuc* > < *phéptoán_cộngtrừ* >)*

Ta dùng cặp dấu ngoặc nhọn để ký hiệu rằng đó là một "*chuyển đổi hậu tố của*" (nói gọn là *chuyển đổi*). Ta có thể dựa vào sản xuất trên để nói như

sau: Chuyển đổi hậu tố của biểu thức bao gồm chuyển đổi của hạng thức và sau nó là không hoặc nhiều chuyển đổi của hạng thức với chuyển đổi của phép cộng trừ tiếp theo sau.

Ta có thể chuyển đổi cho các luật khác của SLANG như sau:

$< \text{hạng thức} > \rightarrow < \text{thùratô} > (< \text{thùratô} > < \text{phéptoánnhâncchia} >)^*$

$< \text{thùratô} > \rightarrow < \text{toán hạng} > < \text{phéptoánmôtnô} > | \varepsilon >$

$< \text{toán hạng} > \rightarrow \text{ldvar } tên$

$| \text{ldcon } tên$

$| \text{ldcon số}$

$| < \text{biểu thức} >$

$< \text{phéptoáncôngtrù} > \rightarrow \text{add} | \text{sub}$

$< \text{phéptoánnhâncchia} > \rightarrow \text{mul} | \text{dvi} | \text{mdl}$

$< \text{phéptoánmôtnô} > \rightarrow \text{neg} | \text{abs}$

Chú ý rằng, "ldvar tên" là chuyển đổi của tên ứng với biến, còn "ldcon tên" là chuyển đổi của tên ứng với hằng số, "ldcon số" là chuyển đổi của một số. Đây chính là các chỉ thị mà ta sẽ sinh ra ứng với sản xuất này.

Các phép tính quan hệ:

$< \text{quan hệ} > \rightarrow < \text{biểu thức} > < \text{biểu thức} > < \text{toán tử quan hệ} >$

$< \text{toán tử quan hệ} > \rightarrow \text{eq} | \text{ne} | \text{lt} | \text{le} | \text{gt} | \text{ge}$

Phép gán:

$< \text{câu lệnh_gán} > \rightarrow < \text{biểu thức} > < \text{phản trái} >$

$< \text{phản trái} > \rightarrow \text{stvar } tên$

Chọn và lặp:

$< \text{câu lệnh_if} > \rightarrow$

$< \text{quan hệ} >$

jiff fi

$< \text{phản câu lệnh} >$

fi:

$< \text{quan hệ} >$

jiff else

$< \text{phản câu lệnh} >$

jump fi

else: $< \text{phản câu lệnh} >$

fi:

< câu lệnh while > → while: < quan hệ >
 jiff od
< phần câu lệnh >
 jump while
od:

Ví dụ 2.6: Câu lệnh while

WHILE a >= b DO a := b - 1 OD
sẽ được chuyển thành các chỉ thị của VIM như sau:

while: ldvar a
 ldvar b
 ge
 jiff od
 ldvar b
 ldcon l
 sub
 stvar a
 jump while

od:

Chú ý: Các từ *while* và *od* trong chương trình VIM trên chỉ là nhãn.

Vào / ra

< câu lệnh read > → (rdint
 stvar tên
)+
< câu lệnh write > → (< biểu thức >
 wrint
)+

Các câu lệnh

< phần câu lệnh > → (< câu lệnh >)⁺

< câu lệnh > → < câu lệnh gán >

| < câu lệnh if >
| < câu lệnh while >

```

| < câu lệnh_call >
| < câu lệnh_read >
| < câu lệnh_write >

```

II. CẢI TIẾN MÁY TÍNH ẢO VIM – VERSION THẬT SỰ

Trong phần trên, chúng ta đã xây dựng một version đơn giản của máy tính ảo VIM. Máy này còn chưa sử dụng được với các lời gọi thủ tục. Trong phần này chúng ta sẽ tổ chức lại cách lưu trữ để có thể cho chạy được các thủ tục.

Trong lúc cho chạy chương trình SLANG, bộ nhớ dữ liệu phải được cấp phát cho các biến của chương trình. Đối với một chương trình không có các thủ tục thì tổng bộ nhớ nó cần có thể xác định được khi dịch chương trình này (với kiểu số nguyên thì một biến chiếm một ô nhớ). Do vậy, mỗi biến ứng với một chỉ số trong mảng data. Đối với chương trình có thêm thủ tục thì cách giải quyết này không ổn do phải cấp phát các ô bộ nhớ cho các biến cục bộ khi có một lời gọi thủ tục mà số lần gọi nó không thể biết trước. Điều này có nghĩa là không thể xác định trước lượng bộ nhớ cần thiết trước khi cho chạy chương trình này, tức là không thể xác định được trong lúc dịch.

Ví dụ 2.7: Một chương trình SLANG có nhiều chương trình con lồng nhau và cây (hình 2.6) biểu diễn sự lồng nhau đó:

BEGIN PROC p1	1
BEGIN var ...;	2
PROC p3 BEGIN ... END;	3
PROC p4 BEGIN VAR INT x; ...; call p3;... END;	4
...	
CALL p3;	5
CALL p4;	6
...	
END;	7
PROC p2	8
BEGIN var...;	9
PROC p5	10
BEGIN PROC p6	11
BEGIN ...; CALL p1; END;	12

...
CALL p6;

13

...
END;

14

...
CALL p5;

15

CALL p1;

16

...
END;

17

...
CALL p1;

18

CALL p2;

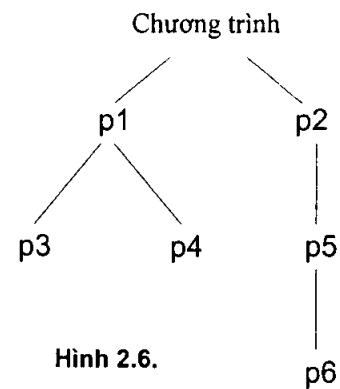
19

...
END;

20

Cách giải quyết là tạo ra một mảng bộ nhớ riêng biệt mỗi khi một thủ tục được gọi. Các biến cục bộ trong thủ tục được đánh địa chỉ nhờ một bộ hai: địa chỉ của chính mảng bộ nhớ đó và một chỉ số (vị trí tương đối trong mảng này). Ta gọi một mảng bộ nhớ kiểu này là một phân đoạn (segment). Ta sẽ xem cách đưa các phân đoạn vào bộ nhớ dữ liệu. Do thủ tục nào được gọi sau cùng thì cũng kết thúc đầu tiên, nên tập các phân đoạn cũng phải là kiểu vào-sau-ra-trước, nghĩa là phân đoạn được tạo cuối thì sẽ được xoá trước.

Phương pháp này đảm bảo là mỗi biến có một địa chỉ duy nhất. Chú ý là không thể gán một số cố định nào đó cho một phân đoạn vì trong thời gian dịch không thể biết được một thủ tục sẽ được gọi bao nhiêu lần và khi nào. Ta cũng không cần đánh địa chỉ tất cả các phân đoạn lúc dịch do các phân đoạn có trong khi chạy một thủ tục sẽ bao gồm phân đoạn ứng với lời gọi thủ tục, phân đoạn của các thủ tục lồng nhau có chứa khai báo của các thủ tục được gọi và phân đoạn của chương trình. Ta gọi tập các phân đoạn này là *nhóm phân đoạn hiện tại* (current segment group) và dùng một mảng CSG để lưu giá trị của các phân đoạn này. Chỉ số của biến trong một phân đoạn được gọi là *độ dịch chuyển* (displacement - *dpl*). Địa chỉ của biến *x*



(trong ví dụ 2.7) với độ dịch chuyển dpl_x trong thủ tục $p4$ có phân đoạn sn_{p4} trở thành CSG $[sn_{p4}] + dpl_x$.

Ví dụ 2.8: Bây giờ ta sẽ xem hoạt động của mảng CSG thay đổi như thế nào trong quá trình chạy chương trình ví dụ 2.7.

Khi bắt đầu chạy, một phân đoạn S0 được tạo ra. Địa chỉ của S0 sẽ được lưu trong CSG [1], và giá trị 1 được gán cho số phân đoạn hiện tại csn (current segment number). Các mối liên kết tĩnh và động của chương trình đều là 0 do chương trình chính là khôi ở ngoài cùng. Lúc này mới chỉ có duy nhất một nhóm phân đoạn S0.

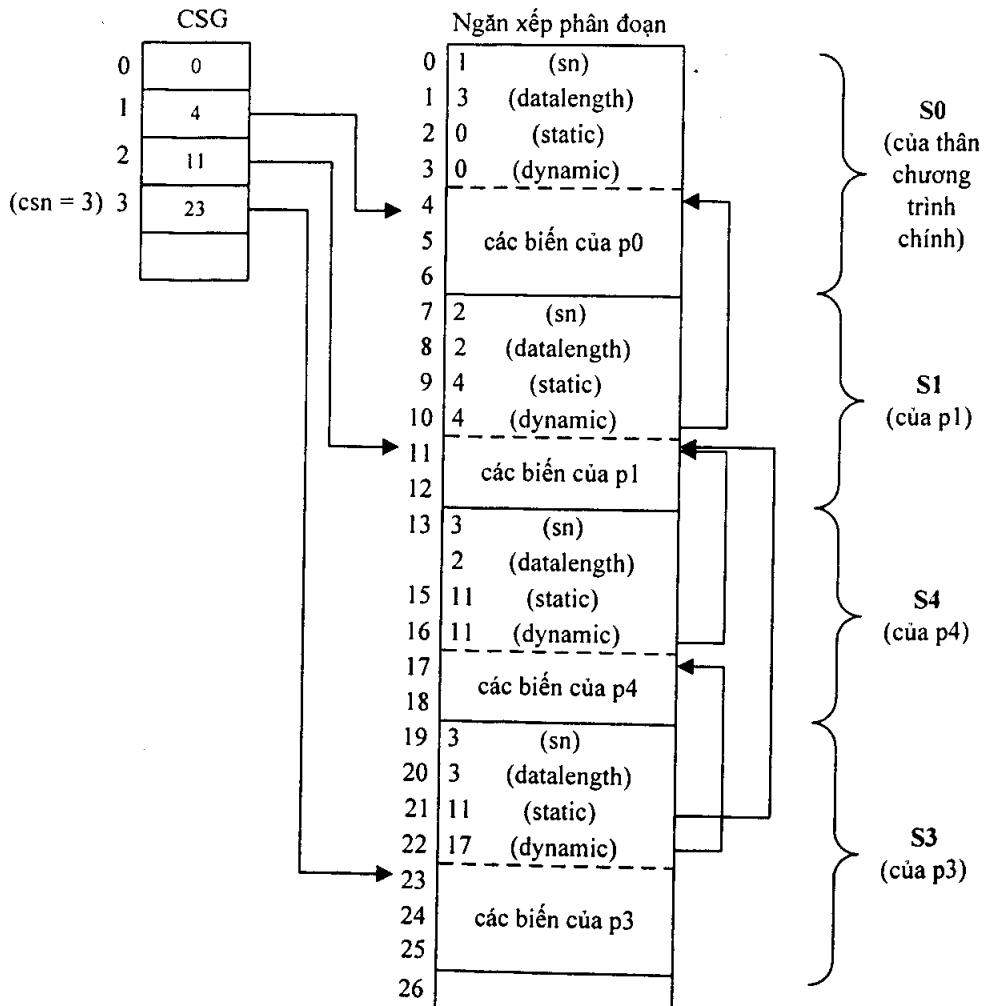
Bây giờ ta sẽ xem điều gì sẽ xảy ra khi gọi thủ tục p1 dòng 18. Những công việc dưới đây sẽ được thực hiện:

- Phân đoạn S1 của p1 được tạo ra và địa chỉ của S1 được lưu trong CSG [2]. Số phân đoạn hiện tại csn trở thành 2. Liên kết tĩnh và động của S1 đều là giá trị của CSG [1], tức là địa chỉ của phân đoạn thân chương trình chính S0.
- Gọi thủ tục p3 dòng 5 (trong thân p1). Phân đoạn S3 được tạo ra và thêm vào nhóm phân đoạn hiện tại. Như vậy cho đến lúc này vẫn chỉ có duy nhất một nhóm phân đoạn hiện tại và bao gồm các phân đoạn {S0, S1, S3}. Liên kết tĩnh và động của S3 đều trỏ vào S1. csn được tăng lên 3.
- Kết thúc p3. Phân đoạn của S3 được loại khỏi nhóm phân đoạn hiện tại, csn giảm thành 2.
- Gọi thủ tục p4 ở dòng 6. Phân đoạn S4 được tạo ra. Vẫn chỉ có một nhóm phân đoạn hiện tại và bao gồm {S0, S1, S4}. Liên kết tĩnh và động của S4 đều trỏ vào S1. csn được tăng lên 3.
- Gọi thủ tục p3 ở dòng 4. Bây giờ, phải tạo thêm một nhóm phân đoạn mới do số phân đoạn của p3 giống như của S4 (cùng cấp và đều là con của p1). Nhóm phân đoạn mới sẽ là {S0, S1, S3}. Giá trị của CSG [3] được gán bằng địa chỉ của S3. Liên kết tĩnh của S3 trỏ tới S1 còn liên kết động trỏ vào S4. csn được tăng lên 3.

Lúc này CSG và ngăn xếp phân đoạn có dạng như hình 2.7. Lưu ý là một phân đoạn bao gồm phần quản lý và phần dữ liệu. Phần quản lý lại bao gồm số phân đoạn (sn), độ dài dữ liệu (data length), và các liên kết tĩnh và động (static; dynamic). Phần dữ liệu có các biến trong thân thủ tục. Địa chỉ của phần dữ liệu được lấy làm địa chỉ của phân đoạn. Lý do làm như vậy là do ta muốn tính địa chỉ các biến sao cho nhanh nhất. Địa chỉ của một biến với số phân đoạn sn và chuyển dịch dpl được tính là CSG $[sn] + dpl$. Nếu ta

muốn tính địa chỉ của biến này từ đầu phân đoạn thì nó trở thành $\text{CSG}[\text{sn}]+\text{dpl}+4$. Rõ ràng phép tính cộng thêm 4 này làm việc tính toán chậm và cồng kềnh hơn.

Bạn hãy tự mình kiểm tra xem mảng CSG và ngăn xếp thay đổi như thế nào khi chương trình gọi tiếp thủ tục p2 nhờ lệnh gọi ở dòng thứ 19.



Hình 2.7. Ngăn xếp phân đoạn và CSG tại một thời điểm.

1. Cải tiến nhằm thêm thủ tục

Bộ nhớ dữ liệu vẫn tiếp tục nằm trong mảng data nhưng bây giờ hoạt động giống như một ngăn xếp của các phân đoạn dữ liệu. Con trỏ đến ngăn xếp - dữ liệu này - dsp trả đến vị trí trống đầu tiên của mảng data. Việc quản lý các phân đoạn và nhóm các phân đoạn thông qua chính các phân đoạn và

trong mảng CSG. Các thủ tục *create_segment* và *delete_segment* sẽ thực hiện nhiệm vụ của chỉ thị *crseg* và *dlseg*.

Chỉ thị *call* lưu địa chỉ trả về mà địa chỉ này được cất trong chỉ thị *return*. Địa chỉ trả về được cất trong ngăn xếp và biểu diễn bằng một mảng *return_stack* với kiểu số nguyên. Các thủ tục *push_return* và *pop_return* lưu hoặc lấy ra một địa chỉ trong ngăn xếp trả về này và được trả bằng con trỏ *rsp*.

Các chỉ thị *ldvar* và *stvar* được định nghĩa lại, và bây giờ hoạt động với cặp địa chỉ (*sn, dpl*). Số phân đoạn *sn* được dùng làm chỉ số trong CSG để xác định địa chỉ của phân đoạn, còn *dpl* cho biết vị trí cấp phát trong phân đoạn đó.

Dưới đây là tập các chỉ thị được sửa đổi hoặc thêm mới của VIM:

Nạp và lưu

chỉ thị	ý nghĩa
ldcon intval	nạp giá trị số nguyên intval vào ngăn xếp
ldvar sn, dpl	sao chép giá trị số nguyên từ địa chỉ bộ nhớ (sn, dpl) vào định ngăn xếp
stvar sn, dpl	loại bỏ giá trị số nguyên ở định ngăn xếp và ghi nó vào địa chỉ bộ nhớ (sn, dpl)

Call và return

chỉ thị	ý nghĩa
call address	lưu địa chỉ trả về, và tiếp tục chạy tại địa chỉ address
return	lấy lại địa chỉ trả về và tiếp tục chạy tại địa chỉ này

Quản lý lưu trữ

chỉ thị	ý nghĩa
crseg sn, length	tạo ra một phân đoạn với số <i>sn</i> và độ dài dữ liệu <i>length</i> ; nếu mức của thủ tục được gọi là cao hơn 1 số so với nơi gọi nó thì phân đoạn mới được thêm vào nhóm hiện tại, nếu không một nhóm mới được tạo ra và nhóm cũ được lưu giữ; nhóm mới bao gồm các phân đoạn của nhóm cũ có mức thấp hơn mức của thủ tục được gọi, và phân đoạn của thủ tục này; các liên kết tĩnh và động của phân đoạn mới trỏ đến phân đoạn của thủ tục bao nó và nơi gọi nó.
dlseg	Xoá phân đoạn được tạo sau cùng; nếu mức của thủ tục được gọi cao hơn 1 số so với nơi gọi nó thì chỉ phân đoạn được tạo sau cùng sẽ được xoá khỏi nhóm hiện tại. Nếu không, nhóm cũ được lưu giữ.

Ta có các luật chuyển đổi mới và sửa lại như dưới đây. Chú ý là các chỉ số dưới cho biết địa chỉ, phân đoạn hay độ dài phải gắn với một tên hoặc số cụ thể nào đó.

```

< câu lệnh call > → call địa chỉ tên
< khaibáothuthuc > → < khối >
                                return
< khối > → < phần khaibáo >
                crseg sn_tên, length_tên
                < phần câu lệnh >
                dlseg
< phần khaibáo > → ( < khaibáohàng > | < khaibáobiển > )*
                                [ jump over
                                  < khaibáothuthuc >+
                                over:
                                ]
< khaibáohàng > → ε
< khaibáobiển > → ε
< toán hạng > →      ldvar sn_tên, dpl_tên
                        | ldcon inval_tên
                        | ldcon inval_số
                        | < biểu thức >
< phần trái > → stvar sn_tên, dpl_tên
< câu lệnh read > → ( rdint
                            stvar sn_tên, dpl_tên )+

```

Ví dụ 2.9: Câu lệnh gán sau

a := b + c

với a và b là các biến, c là hằng số sẽ được chuyển sang các chỉ thị VIM thành:

```

ldvar sn_b, dpl_b
ldcon intval_c
add
stvar sn_a, dpl_a

```

Câu lệnh call:

call p

được chuyển sang các chỉ thị VIM thành:

call address_p

Khai báo thủ tục:

proc p
begin

...

end;

được chuyển sang các chỉ thị VIM thành:

jump over
crseg sn_p, length_p

...

dlseg

return

over:

2. Cải tiến làm dễ cho việc sinh các chỉ thị nhảy và phân đoạn

Các chỉ thị nhảy và nhãn rất cần thiết cho việc dịch các câu lệnh if và while. Ta đã coi đơn giản rằng các nhãn này là duy nhất và bằng cách nào đó ta đã dịch được các chỉ thị này. Nay giờ ta sẽ cải tiến tiếp kiến trúc của VIM để có thể dịch được dễ dàng các chỉ thị nhảy.

Như vậy, ta đã coi địa chỉ là các tham số của các lệnh nhảy và chúng là địa chỉ của các ô trong bộ nhớ chương trình. Tuy nhiên, việc điền các tham số này (của các lệnh nhảy về phía trước) trong lúc dịch lại là một vấn đề khó, do các tham số đó phải để ngỏ cho đến khi biết được địa chỉ đích. Chú ý là địa chỉ nhảy quay lui thì tương đối dễ và có thể điền được ngay trong lúc sinh mã.

Bài toán này có thể được giải quyết đơn giản nếu ta tạo ra một chương trình hợp ngữ (assembly) mà mỗi chỉ thị làm đích của một lệnh nhảy đều có một nhãn. Bộ tạo hợp ngữ sẽ sinh mã máy trong hai giai đoạn. Giai đoạn đầu sẽ xây dựng một bảng địa chỉ trong đó mỗi nhãn ứng với một địa chỉ trong bộ nhớ chương trình. Bước thứ hai xây dựng chương trình mã máy với các tham số của các lệnh nhảy được đánh địa chỉ trong bộ nhớ chương trình. Chương trình dịch lúc này được gọi là duyệt hai lần.

Nếu đầu tiên ta không tạo ra mã hợp ngữ mà là mã máy ngay lập tức thì có hai phương pháp để giải quyết bài toán lệnh nhảy về phía trước: phương pháp thứ nhất là hoàn chỉnh lệnh nhảy ngay khi địa chỉ đích được biết. Điều này có nghĩa là ta phải quay lui để chỉnh sửa lại địa chỉ của một chỉ thị mà ta đã sinh trước đó. Việc đó cũng có thể thực hiện trong bước duyệt toàn bộ chương trình lần thứ hai và đòi hỏi ta phải nhớ được các địa chỉ của các lệnh nhảy. Chú ý là phương pháp này tương tự như các phương pháp mà các bộ dịch hợp ngữ (assembler) vẫn thường dùng.

Phương pháp thứ hai là dùng một bảng để lưu các địa chỉ đích của các chỉ thị nhảy. Bảng này được gọi là *bảng địa chỉ*. Tham số của một chỉ thị nhảy lúc này chỉ là một chỉ số trong bảng này. Khi một địa chỉ đích được biết thì nó sẽ được đặt vào bảng đó. Dĩ nhiên, trong phương pháp này, ta cần phải nhớ chỉ số trong bảng cho đến khi ta tìm được địa chỉ đích.

Vấn đề trên cũng xảy ra tương tự đối với các lệnh tạo phân đoạn. Độ dài của phân đoạn dữ liệu không được biết cho đến khi tất cả các khai báo của một thủ tục được hoàn thành. Ở đây lại có hai phương pháp: ta có thể hoàn chỉnh lệnh tạo phân đoạn ngay sau khi độ dài này được biết (bằng quay lui hay duyệt lại) hoặc ta có thể lưu nó trong *bảng độ dài*.

Ta chọn phương pháp thứ hai do có thể sinh mã "trên cùng chuyến bay" với bộ phân tích. Ngoài ra ta có thể chuyển về phương pháp thứ nhất bằng cách viết thêm một đoạn chương trình duyệt lần hai và hoàn chỉnh chương trình đích.

Tập các chỉ thị được điều chỉnh lần nữa và trở thành:

Nhảy không có điều kiện

chỉ thị	ý nghĩa
jump index	tiếp tục thực hiện chỉ thị tại địa chỉ trả bởi index trong bảng địa chỉ

Các lệnh nhảy có điều kiện

chỉ thị	ý nghĩa
jift index	loại bỏ giá trị chân lý (gọi là truthval) khỏi ngăn xếp; nếu giá trị của truthval là true thì tiếp tục thực hiện chỉ thị tại địa chỉ trả bởi index trong bảng địa chỉ, nếu không chỉ thị tiếp theo sẽ được thực hiện
jiff index	loại bỏ giá trị chân lý (gọi là truthval) khỏi ngăn xếp; nếu giá trị của truthval là false thì tiếp tục thực hiện chỉ thị tại địa chỉ trả bởi index trong bảng địa chỉ, nếu không chỉ thị tiếp theo sẽ được thực hiện

Call

<i>chi thi</i>	<i>ý nghĩa</i>
call index	lưu địa chỉ trả về, và tiếp tục chạy tại địa chỉ trả bởi index trong bảng địa chỉ

Quản lý lưu trữ

<i>chi thi</i>	<i>ý nghĩa</i>
crseg sn, index	tạo ra một phân đoạn với số sn và độ dài dữ liệu trả bởi index trong bảng độ dài; (phần phát biểu còn lại vẫn như cũ)

3. Bộ thông dịch VIM - version thực sự

Chương trình thông dịch của VIM bây giờ được sửa đổi như sau:

```
#include <stdio.h>
#include <conio.h>

#define max_prog      1000 /* độ dài bộ nhớ chương trình */
#define max_table     1000 /* độ dài tối đa các loại bảng */
#define max_data      100  /* độ dài bộ nhớ dữ liệu */
#define max_depth     15   /* độ sâu gọi lồng nhau tối đa */
#define max_stack     100  /* độ sâu tối đa của các ngăn xếp */

typedef enum {
    true = 1, false = 0
} boolean;

typedef enum {
    neg, abs_, add, sub, mul, dvi, mdl, eq, ne, lt, le, gt, ge,
    ldcon, ldvar, stvar, jump, jift, jiff, call, crseg, dlseg,
    return_, rdint, wrint, halt
} operation;

typedef struct {
    operation code;
    union {
        int value;           /* ldcon */
        struct {
            int sn1, dpl;    /* ldvar, stvar */
        } ul;
    } ul;
}
```

```

        int index1;           /* jump, jift, jiff, call */
        struct {
            int sn2, index2; /* crseg */
        } u2;
    } u;
} instruction;

instruction      prog[max_prog];
int              CSG[max_depth];
int              address_table[max_table];
int              length_table[max_table];
int              data[max_data];
int              stack[max_stack];
int              return_stack[max_stack];
int              sp, dsp, rsp, csn;

void read_code(void)
{
    char          fname[100];
    FILE         *fp;
    /* Đọc chương trình VIM từ file vào mảng prog */
    printf("Nhập tên chương trình VIM: "); gets(fname);
    if (NULL == (fp = fopen(fname, "rb"))) {
        printf("Không thể mở file %s", fname); exit(1);
    }
    /* Đọc toàn bộ file và ghi vào vùng nhớ chỉ bởi prog */
    fread(prog, filelength(fileno(fp)), 1, fp);
    fclose(fp);

    /* Đọc bảng địa chỉ từ file vào mảng address_table */
    printf("Nhập tên file bảng địa chỉ: "); gets(fname);
    if (NULL == (fp = fopen(fname, "rb"))) {
        printf("Không thể mở file %s", fname); exit(1);
    }
    fread(address_table, filelength(fileno(fp)), 1, fp);
    fclose(fp);

    /* Đọc bảng độ dài từ file vào mảng length_table */
    printf("Nhập tên file bảng độ dài: "); gets(fname);
    if (NULL == (fp = fopen(fname, "rb"))) {
        printf("Không thể mở file %s", fname); exit(1);
    }
    fread(length_table, filelength(fileno(fp)), 1, fp);
    fclose(fp);
}

/* Các thủ tục push, pop không trình bày lại ở đây */

```

```

void push_return(int x)
{
    return_stack[rsp] = x;
}

void pop_return(int *x)
{
    *x = return_stack[--rsp];
}

void read_int(int *x)
{
    printf("Nhập vào một số nguyên: "); scanf("%d", x);
}

void write_int(int x)
{
    printf("%d ", x);
}

void create_segment (int sn, int data_length)
{
    data[dsp] = sn;
    data[dsp] = data_length;
    data[dsp] = CSG[sn - 1];
    data[dsp] = CSG[csn];
    csn = sn;
    CSG[csn] = dsp;
    dsp += data_length;
}

void delete_segment (void)
{
    int current_segment, call_env, sn_caller, i, data_length,
        static_link, dynamic_link;

    current_segment = CSG[csn];
    data_length = data[current_segment - 3];
    static_link = data[current_segment - 2];
    dynamic_link = data[current_segment - 1];
    if (static_link == dynamic_link) {
        csn--; /* nơi gọi trong cùng nhóm phân đoạn */
    } else { /* nơi gọi khác nhóm phân đoạn */
        call_env = dynamic_link;
        /* lấy nhóm phân đoạn của nơi gọi */
    }
}

```

```

sn_caller = data[dynamic_link - 4];
for (i = sn_caller; i >= csn; i--) {
    /* lấy môi trường nơi gọi */
    CSG[i] = call_env;
    call_env = data[call_env - 2];
}
csn = sn_caller;
}
dsp -= data_length + 4;
}

void main(void)
{
    instruction      *instr;
    int              ic, next, left_operand, right_operand, operand,
                    result, length;
    boolean          truthval, stop;

    read_code();
    ic = 0;
    sp = 0;
    dsp = 0;
    rsp = 0;
    csn = 0;
    CSG[csn] = 0;
    stop = false;

    while (!stop) {
        next = ic + 1;
        instr = &prog[ic];

        switch (instr->code) {

            case neg:
            case abs:
                /* Không trình bày lại ở đây */
                break;

            case add:
            case sub:
            case mul:
            case dvi:
            case mdl:
                /* Không trình bày lại ở đây */
                break;

            case eq:
        }
    }
}

```

```

case ne:
case lt:
case le:
case gt:
case ge:
    /* Không trình bày lại ở đây */
    break;

case ldcon:
    push (instr->u.value);
    break;

case ldvar:
    push(data[CSG[instr->u.u1.sn1]+instr->u.u1.dpl]);
    break;

case stvar:
    pop (&data[CSG[instr->u.u1.sn1]+instr->u.u1.dpl]);
    break;

case jump:
    next = address_table[instr->u.index1];
    break;

case jift:
    pop (&operand);
    if (operand==1)
        next = address_table[instr->u.index1];
    break;

case jiff:
    pop (&operand);
    if (operand==0)
        next = address_table[instr->u.index1];
    break;

case call:
    push_return(next);
    next = address_table[instr->u.index1];
    break;

case crseg:
    length = length_table[instr->u.u2.index2];
    create_segment (instr->u.u2.sn2, length);
    break;

case dlseg:
    delete_segment();
    break;

```

```

case return_:
    pop_return(&next);
    break;

case rdint:
    /* Không trình bày lại ở đây */
    break;

case wrint:
    /* Không trình bày lại ở đây */
    break;

case halt:
    stop = true;
    break;
}
ic = next;
}

```

Chú ý là chương trình này vẫn chưa có bộ phát hiện lỗi. Lỗi vẫn chỉ là lỗi tràn bộ nhớ và ngăn xếp. Bạn hãy tự thêm các lời kiểm tra vào thủ tục *push* và *create_segment* để kiểm tra và báo những lỗi này.

Bài tập

1. Chương trình SLANG trong ví dụ 2.2 có một số chỗ sai. Bạn hãy căn cứ vào các luật từ tố và cú pháp để chỉ ra những chỗ sai đó.

2. Bạn hãy viết các chương trình cho SLANG để thực hiện những việc sau (chú ý: các chương trình con tuy cho phép gọi đệ quy nhưng chưa cho truyền tham số):

- Nhập dữ liệu cần thiết và tính diện tích hình vuông, chữ nhật, tròn.
- Giải phương trình bậc hai.
- Tính tất cả các số nguyên tố nhỏ hơn 1000.
- Nhập N và tính N!
- Bài toán dân gian: "Trăm trâu, trăm bò cỏ".
- Và mọi bài toán bạn nghĩ là có thể giải được bằng chương trình viết trong SLANG.

3. Bạn hãy thử chuyển đổi (bằng tay) toàn bộ chương trình viết trong ví dụ 2.1 sang chương trình VIM.

4. Bạn hãy thử chuyển đổi (bằng tay) toàn bộ chương trình viết trong ví dụ 2.2 và các chương trình trong bài tập 2 sang chương trình VIM.

5. Bạn hãy chạy (tưởng tượng) các chương trình VIM trong bài tập 3 và
4. Hãy cho biết ngăn xếp của máy tính ảo sẽ biến đổi ra sao.

6. Hãy thử mã hoá (bằng tay) các chương trình VIM trong bài tập 3 và 4 thành mã VIM thực sự và cho chạy trên máy tính mô phỏng này. Kiểm tra ngăn xếp (bằng cách dùng công cụ debug trong các chương trình dịch hoặc bạn tự thêm các lệnh in các kết quả trung gian vào chương trình thông dịch) có hoạt động đúng như bài tập 5 không.

7. Mở rộng định nghĩa của ngôn ngữ SLANG sao cho có thể khai báo được các hằng và biến với kiểu Boolean và thêm các toán tử logic dùng cho các toán hằng Boolean.

8. Mở rộng SLANG với các câu lệnh lặp repeat được định nghĩa như sau:

câu lệnh repeat → repeat_token phần câu lệnh

until token *quanhé* taeper_token

Chú ý: Câu lệnh này phải kết thúc bằng từ khoá taeper - từ viết ngược của repeat.

9. Bạn hãy tìm hiểu ưu /nhược điểm của việc kết thúc câu lệnh IF bằng từ khoá FI, WHILE bằng OD.

Bài tập lập trình

1. Thực hiện chương trình mô phỏng VIM ở trên.

2. Các chỉ thị của VIM có thể có độ dài ngắn khác nhau (từ 1 đến 6 byte). Để đơn giản, chương trình mô phỏng đã coi như các lệnh có chiều dài bằng nhau và đều là 6 byte (= **siziof** (instruction)), điều này có nghĩa nhiều lệnh phải ghi và đọc thừa. Bạn hãy cài tiến trình thông dịch VIM để làm việc được với các lệnh theo đúng độ dài của nó nhằm tiết kiệm bộ nhớ lưu giữ chương trình và file đích (và giống với thực tế hơn).

3. Sau khi thực hiện bài thực hành trên, hãy cấp phát động vùng nhớ làm chỗ chứa chương trình VIM thay cho mảng *prog*.

Bài đọc

Bao nhiêu chỉ thị thì đủ

Chương trình VIM mô phỏng một máy tính ảo ở trên có tất cả 22 chỉ thị. Có thể bạn giật mình tự hỏi, sao ít thế? và liệu có đủ làm được mọi việc trên đời như các máy tính bình thường hay không?

Xin trả lời: số lệnh này là quá đủ.

Người ta đã chứng minh rằng (trong môn Lý thuyết tính toán và độ phức tạp), máy tính muốn giải được mọi bài toán chỉ cần có 3 chỉ thị cơ bản sau:

$V = V + 1$

$V = V - 1$

if $V \neq 0$ GOTO L

Thế thì tại sao các bộ vi xử lý lại phải có hàng chục, thậm chí hàng trăm chỉ thị vậy? Ví dụ như dòng họ vi xử lý MC68000 có 56 lệnh, họ Z8000 có tới 110 chỉ thị. Còn họ vi xử lý rất quen thuộc với chúng ta 80x86 thì thế hệ đầu phải có trên 80 chỉ thị và đến các thế hệ mới nhất đã có trên 300 chỉ thị. Ngay bộ vi xử lý RISC có nguyên tắc thiết kế là chỉ có các chỉ thị rất hay dùng (nhằm giảm nhỏ tập lệnh) thì cũng có đến trên 200 chỉ thị.

Đó là do số chỉ thị càng nhiều thì người lập trình càng có nhiều lựa chọn tự do hơn và việc viết chương trình dịch cho nó càng dễ. Mặt khác, những chỉ thị phức tạp hơn tuy có thể viết bằng nhiều chỉ thị đơn giản nhưng nếu được xây dựng sẵn trong phần cứng thì sẽ được tối ưu hơn nhiều cả về kích thước mã lẫn tốc độ. Việc xây dựng sẵn cũng làm các công việc quản lý thanh ghi, bộ nhớ, cổng, ngắt... được thực hiện dễ dàng hơn. Nói cách khác, chúng đã được "đóng gói" sẵn. Điều này cũng giống như ta có thể xây nhà chỉ hoàn toàn từ vôi, đất, cát. Việc xây nhà sẽ nhanh và dễ hơn rất nhiều nếu ta dùng thêm các sản phẩm đã được chế biến sẵn từ các vật liệu đó như gạch, ngói, đá lát, tường đúc sẵn...

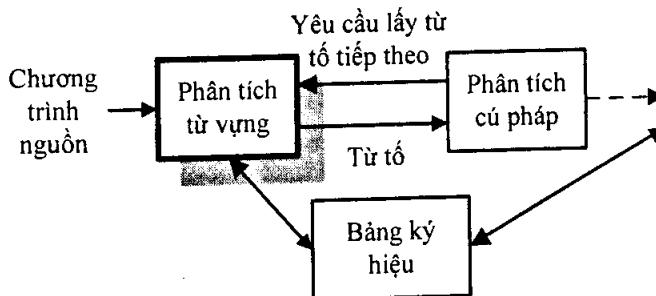
Chương 3

PHÂN TÍCH TỪ VỰNG

I. MỤC ĐÍCH, NHIỆM VỤ

Nhiệm vụ chính của phần phân tích từ vựng là đọc các ký tự vào từ văn bản chương trình nguồn và đưa ra lần lượt các từ tố (token) cùng một số thông tin thuộc tính (attribute).

Hình 3.1 là vị trí của bộ phân tích từ vựng trong một chương trình dịch. Mỗi khi nhận được yêu cầu lấy một từ tố tiếp theo từ bộ phân tích cú pháp, bộ phân tích từ vựng sẽ đọc các ký tự vào cho đến khi đưa ra được một từ tố.



Hình 3.1. Quan hệ giữa phân tích từ vựng và phân tích cú pháp.

II. ĐỒ THỊ DỊCH CHUYÊN

Căn cứ vào bảng các từ tố của SLANG cho trong chương trước, ta dựng được đồ thị chuyển cho từng thành phần rồi liên kết lại với nhau như hình 3.2.

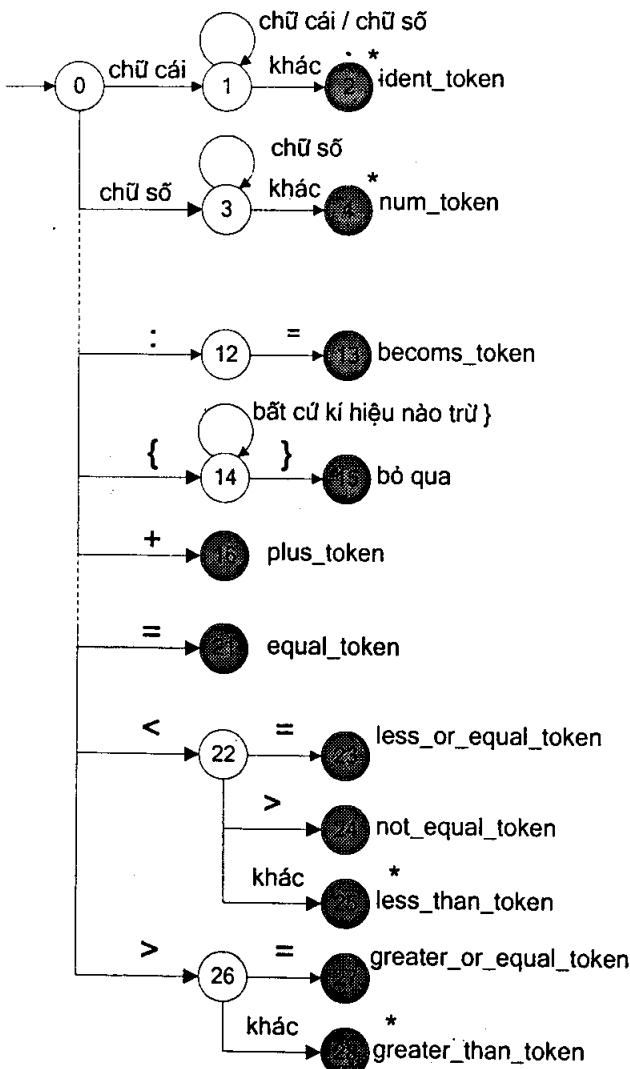
Các trạng thái kết thúc (được đánh dấu bằng các vòng tròn đậm và mầu sẫm) là các trạng thái có thể kết luận về từ vị vừa đọc vào có từ tố là gì. Nếu trạng thái kết thúc có đánh dấu * thì ở đó, chương trình đã đọc quá (thừa) một ký tự. Ký tự này cần trả lại cho lần nhận dạng từ tố sau. Ta có thể thấy là các trạng thái đánh dấu * đều có cạnh đi đến với nhãn khác.

Chú ý rằng, không có phần nhận dạng riêng cho các từ khoá mà sau khi nhận dạng ra từ khoá tên (ident_token), ta sẽ kiểm tra tiếp đó có phải là từ khoá không.

III. THỰC HIỆN

Có hai phương pháp viết chương trình phân tích từ vựng dựa theo đồ thị chuyển. Một phương pháp là lập bảng phân tích dựa theo đồ thị chuyển cho ôtômat hữu hạn đơn định (DFA) và viết chương trình mô phỏng ôtômat này.

Cách thứ hai mà ta trình bày chi tiết dưới đây là viết chương trình diễn giải dựa trên cấu trúc đồ thị chuyển.



Hình 3.2. Đồ thị chuyển tổng hợp.

Thủ tục chính trong phần này là *next_token()*. Thủ tục này mỗi khi được gọi sẽ đọc lần lượt mỗi lần một ký tự trong chương trình nguồn vào biến *ch* (bằng cách gọi thủ tục *next_character*) rồi phân tích cho đến khi tìm được một từ tố. Nó sẽ kết thúc và trả lại từ tố này trong biến *look_ahead*. Đồng thời nó còn dùng một số biến khác để trả về một số thuộc tính của từ tố đó (nếu có). Nếu từ tố này là số (*num_token*) thì nó sẽ đặt giá trị của số đó trong biến *num_value*. Nếu từ tố này là một từ khoá (keyword) hoặc là một tên (*ident_token*) thì nó sẽ đặt tên này trong xâu *ident_lexeme*.

Các từ tố được liệt kê và đánh số tự động nhờ kĩ thuật khai báo enum trong ngôn ngữ C. Do đó *ident_token* (từ tố tên) có giá trị là 0, *num_token* (từ tố số) có giá trị là 1... Việc số hoá này làm cho việc xử lý từ tố về sau nhanh và dễ hơn.

Để đoán nhận được từ tố, chương trình phải dùng một kĩ thuật gọi là nhìn trước một ký tự. Do đó, trong nhiều trường hợp, khi đọc quá một ký tự thì ký tự này không còn thuộc từ tố đang đoán nhận nữa mà thuộc từ tố sau và thủ tục *next_token()* sẽ dựng cờ *retract* để báo cho biết lần sau sẽ dùng ký tự đọc thừa này cho việc đoán nhận từ tố tiếp theo.

Chương trình phân tích từ tố chỉ là một phần trong chương trình dịch hoàn chỉnh. Do đó, chương trình nguồn sẽ còn được sửa đổi, thêm nhiều chức năng mới. Cách tổ chức chương trình nguồn tốt nhất trong C là chia thành nhiều file (mỗi chức năng nên có một file riêng) và quản lý bằng project (của Turbo C, Borland C, Visual C...) hoặc makefile (của MSC)... Nhưng ở đây, để tiện cho các bạn không quen C lắm, ta chỉ cần tổ chức thành hai file: file nguồn SLC.C (SLANG COMPILER) và file chứa khai báo SLC.H là đủ. Hai file này được đặt cùng thư mục và sẽ dịch trực tiếp file SLC.C.

SLANG.H sẽ chứa các định nghĩa về kiểu dữ liệu, khai báo của các thủ tục và hàm (khai báo giao thức). Tổ chức của SLC.H sẽ như sau:

Phần định nghĩa dữ liệu

Phần khai báo các hàm và thủ tục

SLC.C là file chứa chương trình nguồn. Tổ chức của file này như sau:

Phần khai báo về include

Phần khai báo biến và dữ liệu

Các chương trình con của phần phân tích từ vựng

Các chương trình con của phần phân tích cú pháp
Các chương trình con của phần phân tích ngữ nghĩa
Các chương trình con của phần sinh mã
Thân chương trình chính

Đầu các đoạn chương trình đều cho biết nó viết trên file nào. Chỉ những đoạn chương trình phải thêm hoặc phải sửa đổi mới được đề cập đến (những phần khác vẫn giữ nguyên). Bạn hãy tự xác định vị trí sửa đổi hoặc thêm mới và tiến hành những việc thích hợp. Chú ý rằng, việc phân chia vị trí như trên chỉ là tương đối do nhiều thủ tục và hàm khó chỉ ra chúng là của phần nào.

Sau đây là chương trình phân tích từ vựng, một số thủ tục chỉ có khung mà chưa có phần mã.

```
/** FILE SLC.H **/  
/* Phần khai báo của bộ phân tích từ vựng */  
#define max_ident_length 255  
  
typedef char string[max_ident_length + 1];  
  
typedef enum {  
    true = 1, false = 0  
} boolean;  
  
typedef enum { /* Liệt kê và đánh số tự động các từ tố */  
    ident_token, num_token, begin_token, end_token, int_token,  
    var_token, procedure_token, call_token, read_token, write_token,  
    if_token, then_token, else_token, fi_token, while_token, do_token,  
    od_token, negate_token, absolute_token, open_token, close_token,  
    list_token, period_token, separator_token, becomes_token,  
    plus_token, minus_token, times_token, over_token, modulo_token,  
    equal_token, not_equal_token, less_than_token,  
    less_or_equal_token, greater_than_token, greater_or_equal_token,  
    err_token  
} token;  


---

/** FILE SLC.C **/  
#include <process.h>  
#include <string.h>  
#include <stdio.h>  
#include <conio.h>  
#include <ctype.h>  
#include "slc.h"  
  
/* Phần khai báo biến và dữ liệu */
```

```

/* Bảng từ khoá sẽ đặt ở đây */
char ch; /* Ký tự đọc vào từ chương trình nguồn */
boolean retract; /* Cờ báo cho biết đã đọc quá */

token look_ahead; /* Từ tố của từ vị vừa phân tích */
string ident_lexeme; /* Xâu chứa từ vị tên */
int num_value; /* Giá trị của từ vị số */

/** Phân tích từ vựng ***/
void initialise_scanner(void)
{
    /* Đặt các lệnh khởi đầu ở đây */

    ch = ' ';
}

void next_character(void)
{
    /* Đọc từ chương trình nguồn vào một ký tự và đặt trong biến ch */
}

token look_up(string ident_lexeme)
{
    /* Trả lại từ tố của từ khoá đang nằm trong biến ident_lexeme */
}

/* Phân tích tên và kiểm tra xem có phải là từ khoá không. Trả về
   từ tố ident_token hoặc từ tố theo từ khoá, xâu tên được đặt
   trong xâu ident_lexeme */
void identifier(void)
{
    int index = 0;

    do {
        ident_lexeme[index] = ch;
        index++;
        next_character();
    } while (isalpha(ch) || isdigit(ch)); /* Lặp nếu ch là chữ, số */

    ident_lexeme[index] = 0; /* Thêm ký hiệu kết thúc xâu */
    look_ahead = look_up(ident_lexeme); /* Tìm và trả về từ tố nếu đó là từ khoá */
    if (look_ahead == err_token)
        look_ahead = ident_token; /* Nếu không phải từ khoá thì đó chỉ là một tên */
    retract = true;
}

/* Đoán nhận số. Hàm này cũng tính luôn giá trị của số đó và đặt
   vào biến num_value */

```

```

void number(void)
{
    num_value = 0;
    do {
        num_value = num_value * 10 + ch - '0';
        next_character();
    } while (isdigit(ch));
    look_ahead = num_token;
    retract = true;
}

/* Loại bỏ các chú thích */
void comment(void)
{
    do
        next_character();
    while (ch != '}');
}

/* Thủ tục chính của phân tích từ vựng: tìm từ tố tiếp theo */
void next_token(void)
{
    retract = false;
    /* Loại bỏ các ký hiệu trắng, cách, tab, xuống dòng, chú thích */
    while ((ch <= ' ' || ch == '{') && ch != EOF) {
        if (ch == '{') comment();
        next_character();
    }
    if (ch == EOF) return;

    switch (ch) {
        case '(':
            look_ahead = open_token;
            break;

        case ')':
            look_ahead = close_token;
            break;

        case ',':
            look_ahead = list_token;
            break;

        case '!':
            look_ahead = period_token;
            break;

        case ';':
            look_ahead = separator_token;
    }
}

```

```

break;

case '':
    next_character();
    if (ch == '=') look_ahead = becomes_token;
    break;

case '+':
    look_ahead = plus_token;
    break;

case '-':
    look_ahead = minus_token;
    break;

case '*':
    look_ahead = times_token;
    break;

case '/':
    look_ahead = over_token;
    break;

case '|':
    look_ahead = modulo_token;
    break;

case '=':
    look_ahead = equal_token;
    break;

case '<':
    next_character();
    if (ch == '>') look_ahead = not_equal_token;
    else if (ch == '=') look_ahead = less_or_equal_token;
    else {
        look_ahead = less_than_token;
        retract = true;
    }
    break;

case '>':
    next_character();
    if (ch == '=') look_ahead = greater_or_equal_token;
    else {
        look_ahead = greater_than_token;
        retract = true;
    }
}

```

```

    break;

default:
    if (isalpha(ch))
        identifier();
    else if (isdigit(ch))
        number();
}

if (!retract) next_character();
}

```

IV. HOÀN THIỆN CHƯƠNG TRÌNH PHÂN TÍCH TỪ VỰNG

Chương trình trên chưa chạy được ngay. Bạn cần phải hoàn thiện nó theo ý mình. Sau đây là một cách sửa.

Chương trình cần có thêm một bảng mẫu các từ khoá và các từ tố tương ứng với chúng. Bạn hãy thêm những dòng sau vào các vị trí thích hợp:

```

/** File SLC.H **/
typedef struct keyword {
    token erep;           /* Từ tố của từ khoá */
    char irep[8];         /* Mẫu từ khoá */
};

/** File SLC.C **/
/* Bảng các từ khoá và từ tố tương ứng. Có tất cả 17 từ khoá */
struct keyword keywtb[] =
    {{begin_token,"BEGIN"},{end_token,"END"},{int_token,"INT"},  

     {var_token,"VAR"},{procedure_token,"PROC"},{call_token,"CALL"},  

     {read_token,"READ"},{write_token,"WRITE"},{if_token,"IF"},  

     {then_token,"THEN"},{else_token,"ELSE"},{fi_token,"FI"},  

     {while_token,"WHILE"},{do_token,"DO"},{od_token,"OD"},  

     {negate_token,"NEG"},{absolute_token,"ABS"}};
}

```

Còn sau đây là các thủ tục và khai báo khác đặt trong SLC.C:

```

/** Phản khai báo biến và dữ liệu **/
FILE *fp;
...

/** Phản phân tích từ vựng **/
void initialise_scanner(void)
{

```

```

char filename[20];

printf("Nhập vào tên chương trình nguồn: "); gets(filename);

/* Mở file này để đọc */
if ((fp = fopen(filename, "rt"))== NULL) {
    printf("Không thể mở file này. \n"); exit(1);
}

ch = ' ';
}

void next_character(void)
{
    ch = fgetc(fp);           /* Đọc một ký tự vào và đặt vào biến ch */
    if (ch == EOF) fclose(fp); /* Nếu hết file thì đóng nó lại */
}

token look_up(string ident_lexeme)
{
    int i;
    for (i=0; i < 17; i++) /* Quét tất cả 17 từ khoá trong bảng */
        if (0 == strcmp(ident_lexeme, keywtb[i].irep))
            return keywtb[i].erep;
    return err_token;
}

```

V. KIỂM TRA THỬ NGHIỆM CHƯƠNG TRÌNH

Đến đây, phần phân tích từ vựng đã được hoàn thành. Nó là một khối trong nhiều khối của chương trình dịch chứ chưa phải là một chương trình hoàn chỉnh. Bạn không thể biết nó hoạt động như thế nào nếu chỉ có mình nó. Để kiểm tra xem nó có hoạt động tốt không, bạn cần phải thêm phần thân chương trình chính vào. Khi phát triển tiếp có thể xoá nó đi.

```

/* Phần thân chương trình để thử nghiệm bộ phân tích từ vựng.
   Khi kết nối với phần khác bạn có thể xoá phần này đi */

void main(void)
{
    /* Khai báo tên ứng với thứ tự các từ tố để in theo dõi */
    char *tokename[] = {"ident_token", "num_token", "begin_token",
                        "end_token", "int_token", "var_token", "procedure_token",
                        "call_token", "read_token", "write_token", "if_token",
                        "then_token", "else_token", "fi_token", "while_token",
                        "do_token", "od_token", "negate_token", "absolute_token",
                        "open_token", "close_token", "list_token", "period_token",
                        "separator_token", "becomes_token", "plus_token",

```

```

"minus_token", "times_token", "over_token", "modulo_token",
"equal_token", "not_equal_token", "less_than_token",
"less_or_equal_token", "greater_than_token", "greater_or_equal_token",
"err_token" };

initialise_scanner();

/* Chủ động phân tích từ tố cho toàn chương trình SLANG
và in ra tên mọi từ tố */
while (ch != EOF) {
    next_token();
    if (look_ahead == ident_token)
        printf("\n<%s, %s>", tokenname[look_ahead],
               ident_lexeme);
    else if (look_ahead == num_token)
        printf("\n<%s, %d>", tokenname[look_ahead],
               num_value);
    else
        printf("\n<%s, >", tokenname[look_ahead]);

    /* Chỉ làm việc đến dấu chấm hết chương trình và
    bỏ phần còn lại */
    if (look_ahead == period_token) break;
}
}

```

Bây giờ, bạn hãy viết các chương trình trong ngôn ngữ SLANG (như các ví dụ chương trước) và đưa vào cho chương trình trên phân tích.

Chương trình sẽ làm việc và in ra màn hình các thông báo dạng sau¹ (cho chương trình SLANG VIDU21.PRO trong ví dụ 2.1):

Nhập vào tên chương trình nguồn: VIDU21.PRO

```

<begin_token, >
<var_token, >
<int_token, >
<ident_token, number>
<list_token, >
<ident_token, sum>
<separator_token, >
<ident_token, sum>
<becomes_token, >
<num_token, 0>
<separator_token, >

```

¹ Để thu được các thông báo chạy vượt quá màn hình, cách đơn giản nhất là chuyển hướng hiện màn hình thành đưa ra file nhờ các lệnh của DOS. Ví dụ, để đưa tất cả các thông báo của SLC.EXE vào file T.TXT ta gọi chạy như sau ở dấu nhắc DOS: SLC > T.TXT.

```
<read_token, >
<open_token, >
<ident_token, number>
<close_token, >
<separator_token, >
<while_token, >
<ident_token, number>
<not_equal_token, >
<num_token, 0>
<do_token, >
<ident_token, sum>
<becomes_token, >
<ident_token, sum>
<plus_token, >
<ident_token, number>
<separator_token, >
<read_token, >
<open_token, >
<ident_token, number>
<close_token, >
<separator_token, >
<od_token, >
<separator_token, >
<write_token, >
<open_token, >
<ident_token, sum>
<close_token, >
<separator_token, >
<end_token, >
<period_token, >
```

Như vậy toàn bộ chương trình VIDU21.PRO phân tích được thành 41 từ tố. So sánh từng từ tố với chương trình nguồn, bạn có thể kiểm tra xem phần này có hoạt động tốt không hay phải tiến hành sửa chữa ở những chỗ thích hợp.

Chú ý rằng, chương trình trên còn chưa có phần phục hồi lỗi. Ta sẽ cải tiến và thêm phần này về sau.

Bài tập

1. Bạn hãy hoàn chỉnh đồ thị chuyển tổng hợp cho từ vựng của SLANG tại đầu chương này.

2. Bạn hãy thử sửa lại chương trình nguồn trong SLANG VIDU21.PRO như thêm bớt các từ, viết sai từ vựng và xem chương trình phân tích sẽ xử lý ra sao đối với những trường hợp này.

3. Thủ nghiệm chương trình VIDU22.PRO. Hãy cố gắng giải thích từ tố nào ứng với từ vị nào trong chương trình.

4. Giả sử bạn phải mở rộng SLANG cho các câu lệnh sau. Hãy sửa phần này để chương trình có thể phân tích từ vựng cho các câu lệnh mới.

- câu lệnh repeat.
- câu lệnh for.
- câu lệnh case.

Bài tập lập trình

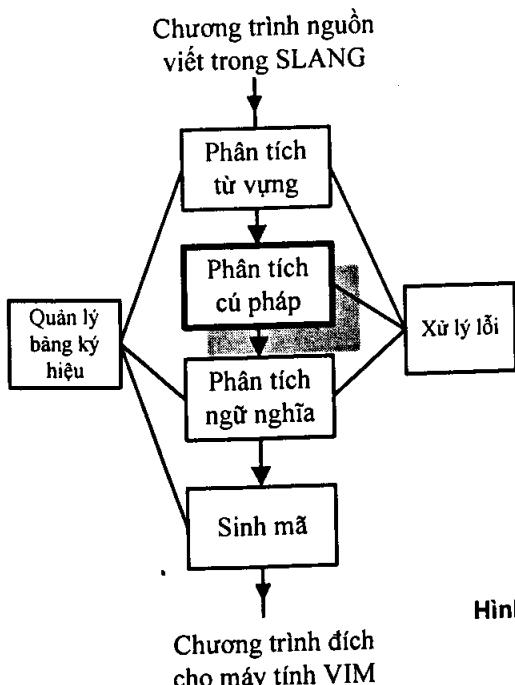
Tự hoàn thiện chương trình theo ý bạn.

Chương 4

PHÂN TÍCH CÚ PHÁP

I. MỤC ĐÍCH, NHIỆM VỤ

Phân tích cú pháp làm nhiệm vụ tách chương trình nguồn (mà bây giờ chứa toàn các từ tố - do phần phân tích từ vựng cung cấp) thành các phần theo văn phạm và biểu diễn cấu trúc này bằng một cây (gọi là cây phân tích) hoặc theo một cấu trúc nào đó tương đương với cây.



Hình 4.1. Vị trí của phân tích cú pháp
trong chương trình dịch.

II. PHƯƠNG PHÁP PHÂN TÍCH ĐỆ QUY ĐI XUỐNG

Trong phần này ta sẽ tóm lược về phân tích đệ quy đi xuống (recursive-descent parsing). *Tư tưởng cơ bản của phương pháp phân tích này là một ký*

hiệu không kết thúc của văn phạm được gắn với một thủ tục phân tích. Việc phân tích được bắt đầu bằng cách gọi thủ tục ứng với ký hiệu khởi đầu. Trong quá trình phân tích, mỗi một ký hiệu kết thúc bên về phải sản xuất sẽ khớp với ký hiệu nhìn trước, tức là các từ tố do bộ phân tích từ vựng trả về trong biến `look_ahead`, và mỗi một ký hiệu không kết thúc sẽ tạo ra một lời gọi thủ tục tương ứng. Các thủ tục phân tích do đó sẽ đi xuống theo cây phân tích khi chúng phân tích chương trình nguồn. Một cây phân tích được tạo ra thông qua chuỗi các sản xuất nhưng không rõ ràng (trong thực hành, cây phân tích rõ ràng có thể được tạo ra).

Nếu vé phải của một sản xuất lại bao gồm một số lựa chọn (ít nhất là 2) thì dựa vào ký hiệu nhìn trước ta cần phải xác định được là sẽ lấy lựa chọn nào. Nếu tất cả các lựa chọn đều bắt đầu với từ tố và các từ tố này lại khác nhau, ký hiệu nhìn trước lại khớp với một trong các từ tố này thì lựa chọn mà bắt đầu với từ tố đó sẽ được lấy.

Ví dụ các sản xuất sau có các lựa chọn bao gồm các từ tố đơn:

toán tử cộng → plus_token | minus_token .

toán tử nhân → times_token | over_token | modulo_token

toán tử một ngôi → negate_token | absolute_token

Còn sản xuất:

toán hạng → ident_token | num_token

| open_token biểu thức close_token

có lựa chọn đầu tiên và thứ hai bao gồm một từ tố đơn (nghĩa là `ident_token` và `num_token`) còn lựa chọn cuối (bắt đầu với `open_token`) cũng là một xâu được bắt đầu với một từ tố.

Việc lựa chọn sẽ không đơn giản nếu một số hoặc toàn bộ các lựa chọn lại bắt đầu với một ký hiệu không kết thúc như:

câu lệnh → câu lệnh_gán

| câu lệnh_if

| câu lệnh_while

| câu lệnh_call

| câu lệnh_read

| câu lệnh_write

Trong trường hợp này, ta cần biết xâu từ tố nào có thể suy dẫn được từ ký hiệu không kết thúc nằm bên về phải của sản xuất *câulệnh*. Việc tính này là nhờ hàm FIRST (X).

FIRST (*câulệnh_gán*) = { **ident_token** }

FIRST (*câulệnh_if*) = { **if_token** }

FIRST (*câulệnh_while*) = { **while_token** }

FIRST (*câulệnh_call*) = { **call_token** }

FIRST (*câulệnh_read*) = { **read_token** }

FIRST (*câulệnh_write*) = { **write_token** }

Các kết quả tính FIRST trên không giao nhau nên có thể chọn chính xác được về phải của *câulệnh*.

Trong trường hợp sản xuất có lựa chọn rỗng thì ta phải dùng hàm FOLLOW.

Ví dụ 4.1:

câulệnh → *câulệnh_gán*
| *câulệnh_if*
| *câulệnh_while*
| *câulệnh_call*
| *câulệnh_read*
| *câulệnh_write*
| *câulệnh_câm*

câulệnh_câm → ε

Ta có FIRST (*câulệnh_câm*) là rỗng. Để lấy lựa chọn *câulệnh_câm* trong quá trình phân tích, ta cần phải biết các từ tố có thể theo sau *câulệnh_câm*. Điều này có được là nhờ thủ tục FOLLOW và ta có kết quả như sau:

FOLLOW (*câulệnh_câm*) = { **end_token**, **separator_token**, **else_token**,
fi_token, **od_token** }

Bây giờ ta sẽ đưa ra các hàm EMPTY (rỗng) và DIRSET (set of director - tập chỉ dẫn). EMPTY(X) là một hàm boolean cho biết X có thể suy ra xâu rỗng hay không. DIRSET (X) định nghĩa một tập các ký hiệu chỉ hướng và được xác định như sau:

$$\text{DIRSET}(X) = \begin{cases} \text{FIRST}(X) \text{ nếu } \text{EMPTY}(X) \text{ sai} \\ \text{FIRST}(X) \cup \text{FOLLOW}(X) \text{ nếu ngược lại} \end{cases}$$

Ta có:

$$\text{DIRSET}(\text{câu lệnh gán}) = \{ \text{ident_token} \}$$

$$\text{DIRSET}(\text{câu lệnh if}) = \{ \text{if_token} \}$$

$$\text{DIRSET}(\text{câu lệnh while}) = \{ \text{while_token} \}$$

$$\text{DIRSET}(\text{câu lệnh call}) = \{ \text{call_token} \}$$

$$\text{DIRSET}(\text{câu lệnh read}) = \{ \text{read_token} \}$$

$$\text{DIRSET}(\text{câu lệnh write}) = \{ \text{write_token} \}$$

$$\begin{aligned} \text{DIRSET}(\text{câu lệnh câm}) = & \{ \text{end_token}, \text{separator_token}, \text{else_token}, \\ & \text{fi_token}, \text{od_token} \} \end{aligned}$$

Các tập trên không giao nhau nên cũng dễ dàng chọn được về phái thích hợp.

Như chương 2 đã đề cập, trong phần này ta dùng một cách biểu diễn các VPPNC gọi là VPPNC mở rộng. Trong cách biểu diễn thông thường thì có thể có nhiều luật có cùng vẽ trái. Trong VPPNC mở rộng thì chỉ có một luật mà thôi. Các vẽ phái của các sản xuất được biểu diễn theo kiểu biểu thức chính quy, do đó đôi khi người ta gọi VPPNC mở rộng là văn phạm vẽ phái chính quy.

Trong mô tả của SLANG, tất cả các luật đã được biểu diễn ở dạng VPPNC mở rộng này.

Xây dựng bộ phân tích

Bây giờ ta sẽ tìm hiểu cách tạo ra một bộ phân tích cú pháp đệ quy đi xuống cho một VPPNC mở rộng.

Ta đặt $\Pi(expr)$ biểu diễn phần văn bản chương trình nguồn của bộ phân tích cho biểu thức chính quy (vẽ phái) $expr$. Đôi với mỗi sản xuất dạng $A \rightarrow expr$ bộ phân tích sẽ có một thủ tục như sau:

void $A(\text{void})$

{

$\Pi(expr)$

}

Phần thân của thủ tục $\Pi(expr)$ được tạo ra bằng cách thay thế biểu thức chính quy $expr$ (và cứ vậy đệ quy cho mọi biểu thức con của $expr$) bằng các module phân tích tương ứng theo các luật cấu trúc cho trong bảng sau:

TT	Biểu thức chính quy	Module phân tích
1	a (ký hiệu kết thúc)	if (look_ahead == a) next_token();
2	A (ký hiệu không kết thúc)	$A()$; (nghĩa là gọi thủ tục ứng với A)
3	$E_1 E_2 \dots E_n$ ($n > 1$)	if (look_ahead \in DIRSET(E_1)) { $\Pi(E_1)$ } else if (look_ahead \in DIRSET(E_2)) { $\Pi(E_2)$ } else if (look_ahead \in DIRSET(E_n)) { $\Pi(E_n)$ }
*	$E_1 E_2 \dots E_n$	$\Pi(E_1) \Pi(E_2) \dots \Pi(E_n)$
5	E^*	while ((look_ahead \in FIRST(E)) { $\Pi(E)$ })
6	E	$\Pi(E)$
7	$[E]$	if ((look_ahead \in FIRST(E)) { $\Pi(E)$ })
8	E^+	$\Pi(E)$ while ((look_ahead \in FIRST(E)) { $\Pi(E)$ })
9	$E_1 (E_2 E_1)^*$	$\Pi(E_1)$ while ((look_ahead \in FIRST(E_2)) { $\Pi(E_2) \Pi(E_1)$ })

Ví dụ 4.2: Cho luật sau của SLANG:

$\text{ph\grave{a}nkhaib\'ao} \rightarrow (\text{khaib\'ao} \text{h\grave{a}ng} \mid \text{khaib\'ao} \text{bi\'en } k)^* \text{ khaib\'ao} \text{th\'utuc}^*$

Ta sẽ viết chương trình cho phần luật này dạng sau:

void *ph\grave{a}nkhaib\'ao* (**void**)

{

$\Pi ((\text{khaib\'ao} \text{h\grave{a}ng} \mid \text{khaib\'ao} \text{bi\'en } k)^* \text{ khaib\'ao} \text{th\'utuc}^*)$

}

Áp dụng luật 4 bằng trên ta có:

void *ph\grave{a}nkhaib\'ao* (**void**)

{

$\Pi ((\text{khaib\'ao} \text{h\grave{a}ng} \mid \text{khaib\'ao} \text{bi\'en } k)^*)$

$\Pi (\text{khaib\'ao} \text{th\'utuc}^*)$

}

Bây giờ áp dụng luật 5 ta có:

void *ph\grave{a}nkhaib\'ao* (**void**)

{

while (look_ahead == int_token || look_ahead == var_token)

{

$\Pi (\text{khaib\'ao} \text{h\grave{a}ng} \mid \text{khaib\'ao} \text{bi\'en } k)$

}

while (look_ahead == procedure_token) {

$\Pi (\text{khaib\'ao} \text{th\'utuc})$

}

}

Áp dụng luật 3 ta có:

void *ph\grave{a}nkhaib\'ao* (**void**)

```

while (look_ahead == int_token || look_ahead == var_token)
{
    if (look_ahead == int_token) {
        Π (khaibáohàngk)
    } else if (look_ahead == var_token) {
        Π (khaibáobién)
    }
}

while (look_ahead == procedure_token) {
    Π (khaibáothùtục)
}
}

```

Bây giờ áp dụng luật 2 ta có:

```

void phânkhaibáo (void)
{
    while (look_ahead == int_token || look_ahead == var_token)
    {
        if (look_ahead == int_token) {
            khaibáohàng ();
        } else if (look_ahead == var_token) {
            khaibáobién ();
        }
    }

    while (look_ahead == procedure_token) {
        khaibáothùtục ();
    }
}

```

Đây chính là nội dung của thủ tục declaration_part trong phần dưới đây.

III. BỘ PHÂN TÍCH CÚ PHÁP CHO SLANG

Phần sau đây là bộ phân tích cho SLANG. Phần này chưa có khôi phục lỗi:

/* *** SLC.H ***/

```
/* Khai báo các thủ tục trước khi dùng */
void block(void);
void declaration_part(void);
void constant_declarer(void);
void type_declarer(void);
void variable_declarer(void);
void procedure_declarer(void);
void statement_part(void);
void statement(void);
void assignment_statement(void);
void left_part(void);
void if_statement(void);
void while_statement(void);
void call_statement(void);
void read_statement(void);
void write_statement(void);
void expression(void);
void term(void);
void factor(void);
void operand(void);
void add_operator(void);
void multiply_operator(void);
void unary_operator(void);
void relation(void);
void relational_operator(void);
```

/* *** SLC.C ***/

/* *** Phần khai báo biến và dữ liệu nằm ở đây ***/

/* *** Phần phân tích từ vựng nằm ở đây ***/

```
/* *** Phần phân tích cú pháp ***/
void program_declarer(void)
{
    block();
    if(look_ahead == period_token) next_token();
}
```

```

void block(void)
{
    if(look_ahead == begin_token) next_token();
    declaration_part();
    statement_part();
    if(look_ahead == end_token) next_token();
}

void declaration_part(void)
{
    while (look_ahead == int_token || look_ahead == var_token) {
        if(look_ahead == int_token).
            constant_declaration();
        else if (look_ahead == var_token)
            variable_declaration();
    }

    while (look_ahead == procedure_token)
        procedure_declaration();
}

void constant_declaration(void)
{
    type_declarer();
    if(look_ahead == ident_token) next_token();
    if(look_ahead == equal_token) next_token();
    if(look_ahead == num_token) next_token();
    while (look_ahead == list_token) {
        if(look_ahead == list_token) next_token();
        if(look_ahead == ident_token) next_token();
        if(look_ahead == equal_token) next_token();
        if(look_ahead == num_token) next_token();
    }
    if(look_ahead == separator_token) next_token();
}

void type_declarer(void)
{
    if(look_ahead == int_token) next_token();
}

void variable_declaration(void)
{
    if(look_ahead == var_token) next_token();
    type_declarer();
}

```

```

if(look_ahead == ident_token) next_token();
while (look_ahead == list_token) {
    if(look_ahead == list_token) next_token();
    if(look_ahead == ident_token) next_token();
}
if(look_ahead == separator_token) next_token();
}

void procedure_declaration(void)
{
    if(look_ahead == procedure_token) next_token();
    if(look_ahead == ident_token) next_token();
    block();
    if(look_ahead == separator_token) next_token();
}

void statement_part(void)
{
    statement();
    while (look_ahead == separator_token) {
        if(look_ahead == separator_token) next_token();
        statement();
    }
}

void statement(void)
{
    if(look_ahead == ident_token) {
        assignment_statement();
    } else if(look_ahead == if_token) {
        if_statement();
    } else if(look_ahead == while_token) {
        while_statement();
    } else if(look_ahead == call_token) {
        call_statement();
    } else if(look_ahead == read_token) {
        read_statement();
    } else if(look_ahead == write_token) {
        write_statement();
    }
}

void assignment_statement(void)
{
    left_part();
    if(look_ahead == becomes_token) next_token();
}

```

```

        expression();
    }

void left_part(void)
{
    if(look_ahead == ident_token) next_token();
}

void if_statement(void)
{
    if(look_ahead == if_token) next_token();
    relation();
    if(look_ahead == then_token) next_token();
    statement_part();
    if(look_ahead == else_token) {
        if(look_ahead == else_token) next_token();
        statement_part();
    }
    if(look_ahead == fi_token) next_token();
}

void while_statement(void)
{
    if(look_ahead == while_token) next_token();
    relation();
    if(look_ahead == do_token) next_token();
    statement_part();
    if(look_ahead == od_token) next_token();
}

void call_statement(void)
{
    if(look_ahead == call_token) next_token();
    if(look_ahead == ident_token) next_token();
}

void read_statement(void)
{
    if(look_ahead == read_token) next_token();
    if(look_ahead == open_token) next_token();
    if(look_ahead == ident_token) next_token();
    while (look_ahead == list_token) {
        if(look_ahead == list_token) next_token();
        if(look_ahead == ident_token) next_token();
    }
    if(look_ahead == close_token) next_token();
}

```

```

}

void write_statement(void)
{
    if(look_ahead == write_token) next_token();
    if(look_ahead == open_token) next_token();
    expression();
    while (look_ahead == list_token) {
        if(look_ahead == list_token) next_token();
        expression();
    }
    if(look_ahead == close_token) next_token();
}

void expression(void)
{
    term();
    while(look_ahead == minus_token || look_ahead == plus_token) {
        add_operator();
        term();
    }
}

void term(void)
{
    factor();
    while (look_ahead == modulo_token || look_ahead == over_token
           || look_ahead == times_token) {
        multiply_operator();
        factor();
    }
}

void factor(void)
{
    if (look_ahead==absolute_token || look_ahead==negate_token) {
        unary_operator();
    }
    operand();
}

void operand(void)
{
    if(look_ahead == ident_token) {
        if(look_ahead == ident_token) next_token();
    } else if(look_ahead == num_token) {
}
}

```

```

        if(look_ahead == num_token) next_token();
    } else if(look_ahead == open_token) {
        if(look_ahead == open_token) next_token();
        expression();
        if(look_ahead == close_token) next_token();
    }
}

void add_operator(void)
{
    if(look_ahead == plus_token) {
        if(look_ahead == plus_token) next_token();
    } else if(look_ahead == minus_token) {
        if(look_ahead == minus_token) next_token();
    }
}

void multiply_operator(void)
{
    if(look_ahead == times_token) {
        if(look_ahead == times_token) next_token();
    } else if(look_ahead == over_token) {
        if(look_ahead == over_token) next_token();
    } else if(look_ahead == modulo_token) {
        if(look_ahead == modulo_token) next_token();
    }
}

void unary_operator(void)
{
    if(look_ahead == negate_token) {
        if(look_ahead == negate_token) next_token();
    } else if(look_ahead == absolute_token) {
        if(look_ahead == absolute_token) next_token();
    }
}

void relation(void)
{
    expression();
    relational_operator();
    expression();
}

void relational_operator(void)
{
}

```

```

        if(look_ahead == equal_token) {
            if(look_ahead == equal_token) next_token();
        } else if(look_ahead == not_equal_token) {
            if(look_ahead == not_equal_token) next_token();
        } else if(look_ahead == less_than_token) {
            if(look_ahead == less_than_token) next_token();
        } else if(look_ahead == less_or_equal_token) {
            if(look_ahead == less_or_equal_token) next_token();
        } else if(look_ahead == greater_than_token) {
            if(look_ahead == greater_than_token) next_token();
        } else if(look_ahead == greater_or_equal_token) {
            if(look_ahead == greater_or_equal_token) next_token();
        }
    }

**** Thân chương trình chính ****
void main(void)
{
    initialise_scanner();

    next_token(); /* Đọc trước một từ tố */
    program_declaration();
}

```

Chú ý là một số đoạn trong chương trình trên bị dư thừa và có thể viết lại tốt hơn. Ví dụ, trong thủ tục operand () có lệnh:

```

if(look_ahead == ident_token) {
    if(look_ahead == ident_token) next_token();
} else ...

```

Có thể sửa lại tốt hơn như sau:

```

if(look_ahead == ident_token) next_token();
else ...

```

Lý do của các lệnh dư thừa này là chúng được sinh ra bằng cách áp dụng máy móc các luật sinh trong bảng.

IV. THỦ NGHIỆM CHƯƠNG TRÌNH

Nếu bạn nhập chương trình trên, dịch tốt và cho chạy với file chương trình VIDU21.PRO thì chương trình này sẽ chạy một cách lảng lẽ và không thể hiện điều gì cả. Đó là do chúng còn thiếu các phần dịch nằm sau và chưa phải là một chương trình hoàn chỉnh.

Để kiểm tra xem phần này có tốt không và tìm hiểu hoạt động của nó, ta có thể thêm một số lệnh theo dõi. Do mỗi thủ tục trong phần phân tích tương

ứng với một ký hiệu không kết thúc và đều ứng với một luật sản xuất nào đó, cách tốt nhất là ta thêm các lệnh in vào đầu mỗi thủ tục đó để cho biết chương trình đã dùng các thủ tục nào, thứ tự ra sao (sau khi hoàn thành kiểm tra, bạn lại xoá chúng đi). Từ đó ta dựa vào chương trình nguồn trong ngôn ngữ SLANG kiểm tra xem bộ phân tích có hoạt động đúng hay không.

Để bạn tiện theo dõi và sửa chương trình nguồn, những chỗ mới thêm hoặc có sửa đổi đều được đánh dấu bằng một mũi tên trắng.

```
void program_declaration(void)
{
    ⇒ printf("program_declaration\n");
    block();
    if(look_ahead == period_token) next_token();
}

void block(void)
{
    ⇒ printf("block\n");
    if(look_ahead == begin_token) next_token();
    declaration_part();
    statement_part();
    if(look_ahead == end_token) next_token();
}

void declaration_part(void)
{
    ⇒ printf("declaration_part\n");
    ...
}
```

Thử nghiệm với VIDU21.PRO ta được kết quả hiện màn hình như sau (các con số do chúng tôi thêm vào để tiện trình bày):

Nhập vào tên chương trình nguồn: VIDU21.PRO

1. program_declaration
2. block
3. declaration_part
4. variable_declarer
5. type_declarer
6. statement_part
7. statement
8. assignment_statement
9. left_part
10. expression
11. term
12. factor
13. operand
14. statement

- 15. read_statement
- 16. statement
- 17. while_statement
- 18. relation
- 19. expression
- 20. term
- 21. factor
- 22. operand
- 23. relational_operator
- 24. expression
- 25. term
- 26. factor
- 27. operand
- 28. statement_part
- 29. statement
- 30. assignment_statement
- 31. left_part
- 32. expression
- 33. term
- 34. factor
- 35. operand
- 36. add_operator
- 37. term
- 38. factor
- 39. operand
- 40. statement
- 41. read_statement
- 42. statement
- 43. statement
- 44. write_statement
- 45. expression
- 46. term
- 47. factor
- 48. operand
- 49. statement

Từ các thông báo này ta có thể hiểu như sau. Dòng số 1 cho biết sản xuất đầu tiên được dùng là *khai báo chương trình* (program_declaration). Sản xuất này chỉ tạo nên một ký hiệu không kết thúc duy nhất là *khối* (block) với từ tố bắt đầu là BEGIN - ta nhận được thông tin khẳng định điều này trên dòng thứ 2. Sản xuất *khối* lại bao gồm *phản khai báo* (declaration_part - dòng 3) và *phản câu lệnh* (statement_part - dòng 6). Chương trình nguồn chỉ có duy nhất một khai báo về các biến:

VAR INT number, sum;

nên phần khai báo lại chỉ có duy nhất *khai báo biến* (variable_declaration - dòng 4), đến lượt mình *khai báo biến* lại chỉ có một kiểu khai báo (type_declarer - dòng 5).

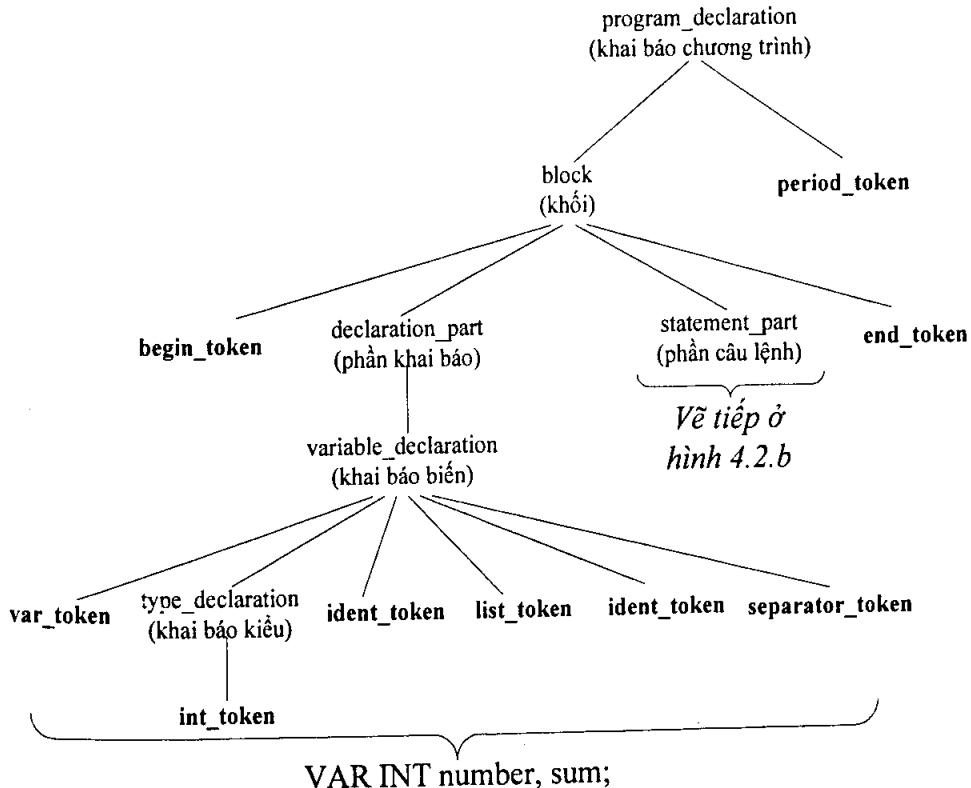
Trong phần câu lệnh, có tất cả là bốn lệnh. Lệnh đầu tiên:

sum := 0;

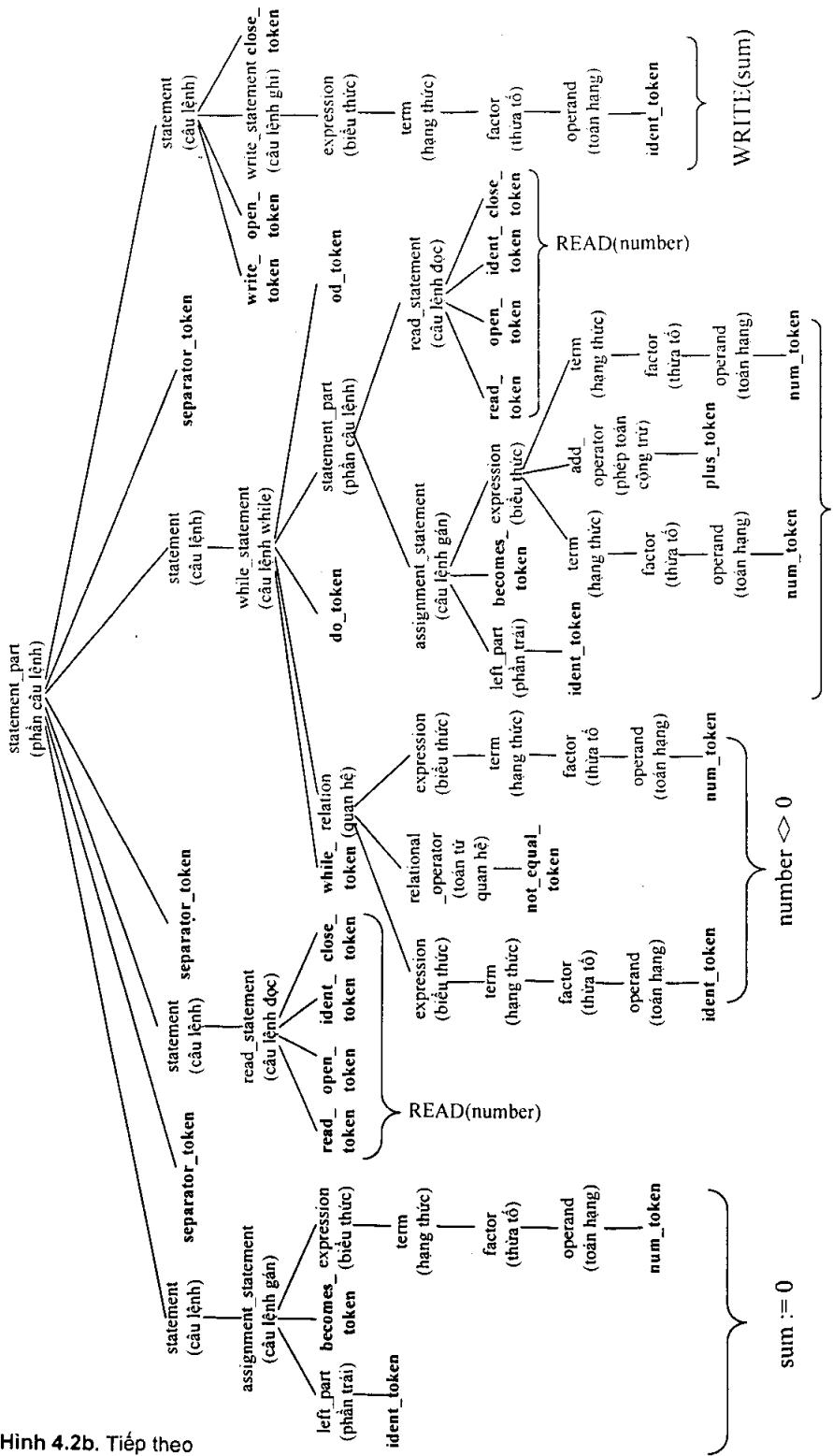
cho thông tin về sản xuất được dùng là lệnh gán (assignment_statement - dòng 8). Tương tự, các lệnh còn lại là read (read_statement - dòng 15), while (while_statement - dòng 17) và write (write_statement - dòng 44).

Cứ như vậy bạn có thể khẳng định được kết quả in ra phù hợp với chương trình nguồn SLANG và bộ phân tích làm việc tốt hay không.

Như vậy, chương trình nguồn VIDU21.PRO đã được phân tích thành 49 luật cú pháp. Toàn bộ các luật đó tạo thành một cây phân tích ẩn. Ta có thể dựa vào các luật này để dựng cây phân tích rõ như hình 4.2 (do hình lớn nên nó được chia thành hai phần *hình 4.2.a* và *hình 4.2.b*).



Hình 4.2.a. Cây phân tích và các dòng lệnh tương ứng.



Hình 4.2b. Tiếp theo

Nếu chương trình nguồn SLANG có lỗi cú pháp thì các luật in ra cho thấy quá trình phân tích diễn ra rất trực trắc, hỗn loạn và không định trước được. Tuy vậy hệ thống vẫn cố gắng hoạt động cho đến hết vì chưa có phần phát hiện, khôi phục lỗi.

Bài tập

1. Hãy thử viết sai cấu trúc của chương trình nguồn SLANG trên (như xóa bớt các dấu ;, thay := bằng =) và xem chương trình hoạt động ra sao đối với các trường hợp này.
2. Nghiệm chương trình trên với ví dụ 2.2 (VIDU22.PRO). Hãy tự giải thích các luật sản xuất nhận được và dựng cây phân tích từ các luật đó.

Bài tập lập trình

1. Tất cả những chỗ viết dư thừa trong chương trình trên và sửa lại cho tốt hơn.
2. Mở rộng chương trình trên cho các câu lệnh:
 - câu lệnh repeat
 - câu lệnh for
 - câu lệnh case
3. Viết chương trình tự động sinh ra chương trình nguồn cho bộ phân tích căn cứ vào các luật cú pháp cho trước.

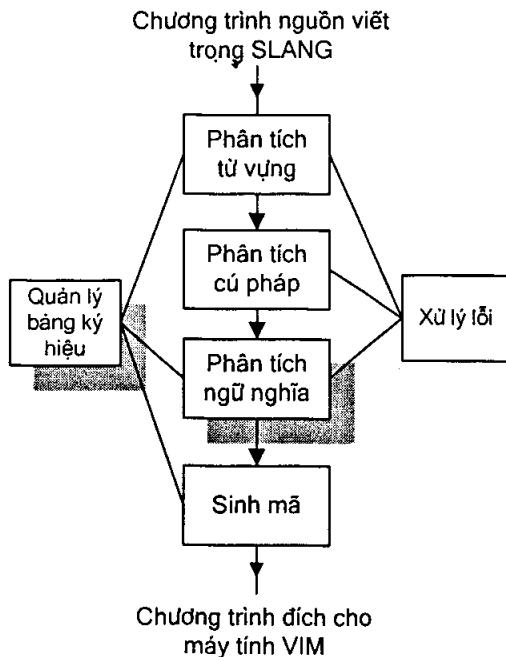
Chương 5

BẢNG KÝ HIỆU VÀ PHÂN TÍCH NGỮ NGHĨA

I. MỤC ĐÍCH, NHIỆM VỤ

Bảng ký hiệu

Ngôn ngữ SLANG đòi hỏi mọi việc sử dụng tên phải đúng theo khai báo của tên đó. Khai báo xác định phạm vi (khối), loại (hàng, biến hoặc thủ tục) và kiểu (số nguyên) của một tên. Để thống nhất giữa một bên là sử dụng và một bên là khai báo, chỉ được phép khai báo tên một lần trong cùng một khối. Mọi sử dụng tên phải đúng loại, kiểu và trong phạm vi khai báo.



Hình 5.1. Vị trí của phần quản lý bảng ký hiệu và phân tích ngữ nghĩa trong chương trình dịch.

Để dùng đúng, ta cần một bảng duy trì các bản ghi ứng với mỗi khai báo tên. Bảng này được gọi là bảng ký hiệu. Phần chương trình làm việc với bảng và bảng tạo thành chức năng quản lý bảng ký hiệu của chương trình dịch.

Mỗi khi cần xem xét một tên, chương trình dịch sẽ tìm trong bảng ký hiệu xem đã có tên đó chưa - nếu tên đó là mới thì thêm tên đó vào bảng ký hiệu. Nếu có sẵn thì phải nhanh chóng cho phép truy nhập vào thông tin liên quan đến khai báo của tên đó. Các thông tin bao gồm tên (là một chuỗi ký tự tạo nên), kiểu của nó (nghĩa là số nguyên, thực, xâu), dạng của nó (nghĩa là một biến hay một câu trúc), vị trí của nó trong bộ nhớ và các thuộc tính khác phụ thuộc vào ngôn ngữ lập trình. Các thông tin về tên được đưa vào bảng trong giai đoạn phân tích từ vựng và cú pháp.

Các thông tin có trong bảng ký hiệu được dùng ở mọi số quá trình dịch. Nó được dùng trong phân tích ngữ nghĩa, như kiểm tra xem việc dùng các tên này có khớp với khai báo của chúng hay không. Nó cũng được dùng trong giai đoạn sinh mã, ví dụ để chúng ta biết kích thước, loại bộ nhớ phải cấp phát cho một tên.

Cũng có một số nhu cầu dùng bảng ký hiệu theo cách khác như để phát hiện và khắc phục lỗi.

Phân tích ngữ nghĩa

Một chương trình dịch phải kiểm tra xem chương trình nguồn có theo các quy định về cú pháp và ngữ nghĩa của ngôn ngữ nguồn hay không. Việc kiểm tra này gọi là kiểm tra tĩnh (phân biệt với kiểm tra động trong quá trình chương trình đích chạy), đảm bảo các lỗi về kiểu sẽ được phát hiện và mô tả. Ví dụ, kiểm tra tĩnh có thể gồm:

1. *Kiểm tra kiểu.* Một chương trình đích phải báo lỗi nếu một toán tử được dùng cho các toán hạng không tương thích; ví dụ, khi ta cộng một xâu với một số nguyên.
2. *Kiểm tra dòng điều khiển.* Các câu lệnh làm thay đổi dòng điều khiển từ một câu trúc phải có một số vị trí để chuyển dòng điều khiển đến đó. Ví dụ, lệnh *break* trong C làm dòng điều khiển thoát khỏi vòng lặp *while*, *for*, hoặc *switch* gần nhất: sẽ có lỗi nếu như các vòng lặp đó lại không có.
3. *Kiểm tra tính nhất quán.* Có những hoàn cảnh mà trong đó một đối tượng được định nghĩa chỉ đúng một lần. Ví dụ, trong Pascal, một

định danh (tên) phải được khai báo là duy nhất, các nhãn trong *case* phải khác nhau, và các phần tử trong một kiểu vô hướng có thể không được lặp lại.

4. *Kiểm tra quan hệ tên.* Đôi khi, cùng một tên phải xuất hiện từ hai lần trở lên. Ví dụ, trong Assembly, một chương trình con có một tên mà chúng phải xuất hiện ở đầu và cuối của chương trình con này. Chương trình dịch sẽ kiểm tra tên đó phải được dùng ở cả hai nơi.

Như vậy, nếu chương trình viết trong ngôn ngữ nguồn hoàn toàn đúng về mặt ngữ nghĩa thì phần này không làm gì cả. Nếu có sai sót sẽ thông báo lỗi này.

Do định nghĩa hiện tại của SLANG chỉ có duy nhất một kiểu dữ liệu (kiểu số nguyên) nên chưa cần phải kiểm tra kiểu (tuy vậy bạn có thể dễ dàng thêm phần kiểm tra kiểu sau này). Phần phân tích ngữ nghĩa lúc này chỉ nhấn mạnh đến việc kiểm tra tính nhất quán mà thôi.

Để kiểm tra ngữ nghĩa, ta dùng cú pháp điều khiển - tức là kĩ thuật gắn các luật kiểm tra với các luật cú pháp, và việc kiểm tra này được thực hiện trong lúc phân tích, tức là phương pháp duyệt "quên lăng", hay cụ thể hơn là "trên cùng chuyến bay". Phương pháp này tuy bị hạn chế về lớp ngôn ngữ nhưng lại rất hiệu quả về thời gian thực hiện và bỏ bớt được các bước trung gian.

II. THIẾT KẾ BẢNG KÝ HIỆU VÀ PHÂN TÍCH NGỮ NGHĨA

Trước khi đưa ra một thiết kế của bảng ký hiệu, ta xem lại các luật về phạm vi và hiện diện (thấy được) trong SLANG. Một tên được buộc với một đối tượng trong một khai báo. Phạm vi của một ràng buộc là khối của thủ tục có khai báo đó. Một ràng buộc tồn tại trong một thủ tục thì cũng tồn tại trong một thủ tục con của thủ tục đó, trừ khi tên được định nghĩa lại trong khối của thủ tục con. Từ đó, luật hiện diện có thể được suy ra như sau:

Tại một điểm bất kì trong văn bản chương trình, chỉ các tên được khai báo trong khối hiện tại và các khối bao bọc là có thể truy nhập được.

Nếu một tên được khai báo trong nhiều khối, thì chỉ khai báo gần nhất đến điểm sử dụng là hiện diện.

Ví dụ 5.1: Xét chương trình dưới đây.

1. BEGIN {mức 1}
2. VAR INT p, w;
3. PROC d
4. BEGIN {mức 2}

```

5.      VAR INT a, r, w;
6.      PROC z
7.          BEGIN {mức 3}
8.              VAR INT e, p, w;
9.              ...
10.         END;
11.         ...
12.     END;
13. ...
14. END.

```

Chương trình này có 3 mức phạm vi:

1. Chương trình chính
2. Thân của thủ tục "d"
3. Thân của thủ tục "z"

các khai báo truy nhập được tại các dòng khác nhau trong chương trình được cho trong bảng dưới đây. Số trong cặp dấu ngoặc cho biết mức của một khai báo.

<i>dòng</i>	<i>các khai báo truy nhập được</i>
9	a(2), d(1), e(3), p(3), r(2), w(3), z(2)
11	a(2), d(1), p(1), r(2), w(2), z(2)
13	d(1), p(1), w(1)

Về kĩ thuật, bảng ký hiệu thường được thực hiện bằng bảng băm (hash table) hoặc cây nhị phân. Đối với các ngôn ngữ có cấu trúc khối (như SLANG) còn có lựa chọn giữa lập bảng riêng cho từng phạm vi hoặc một bảng đơn cho tất cả. Ta chọn lựa cách thực hiện là *cây nhị phân đơn toàn cục* (như ví dụ hình 5.2). Cây này được gọi là cây định danh. Một bản ghi của nút cây có các trường sau:

- Tên.
- Con trỏ đến một danh sách các bản ghi cho các định nghĩa (khai báo) khác nhau của tên đó. Chúng lập thành danh sách định nghĩa.
- Các con trỏ đến cây con trái và phải.

Mỗi tên đều có một danh sách định nghĩa, tức là một danh sách các bản ghi định nghĩa. Mỗi bản ghi định nghĩa biểu diễn một khai báo của tên đó. Một bản ghi định nghĩa bao gồm các trường như sau:

- Số phân đoạn.
- Vị trí dành chỗ (của một biến), giá trị (của một hằng số) hoặc chỉ số địa chỉ (của một thủ tục) trong bảng địa chỉ.
- Kiểu (số nguyên).
- Loại (hằng, biến, thủ tục).
- Con trỏ đến bản ghi định nghĩa tiếp theo.

Danh sách các bản ghi định nghĩa được tổ chức như một danh sách đầy xuống. Khi phân tích một khai báo của một tên, một bản ghi định nghĩa sẽ được tạo ra và được chèn vào mặt trước của danh sách này. Một bản ghi sẽ được loại khỏi danh sách khi ta rời khỏi mà trong đó có khai báo đó. Lược đồ này đảm bảo rằng, tại mọi lúc, chỉ các khai báo từ khôi hiện thời và các khôi bao quanh là thâm nhập được, và khai báo gần điểm tìm nhất sẽ được tìm thấy trước.

Khi ta rời khỏi, tất cả các bản ghi định nghĩa được tạo trong lúc phân tích khôi này phải được loại bỏ từ các danh sách định nghĩa. Điều này đòi hỏi thực hiện một tìm kiếm phức tạp trên cây định danh và các danh sách định nghĩa của nó. Để tối ưu quá trình tìm kiếm, ta nên có thêm một danh sách phạm vi dạng đầy xuống (như hình 5.3). Khi vào một khôi, một bản ghi phạm vi được tạo ra và chèn vào trước danh sách đó. Một bản ghi phạm vi bao gồm các trường sau:

- Một con trỏ đến một danh sách các bản ghi tên.
- Một con trỏ đến bản ghi phạm vi của phạm vi trước.

Các tên được khai báo trong phạm vi hiện tại, được tổ chức trong một *danh sách tên*, nghĩa là một danh sách các bản ghi tên. Một bản ghi tên bao gồm các trường sau đây:

- Tên.
- Con trỏ đến bản ghi tên tiếp theo trong danh sách.

Khi ta rời khỏi khôi, các tên trong danh sách tên của bản ghi cùng phạm vi sẽ được tìm trong cây định danh. Đôi với từng tên khai báo trong phạm vi hiện tại, bản ghi định nghĩa đầu tiên được bỏ khỏi danh sách các bản ghi định nghĩa. Cuối cùng, bản ghi phạm vi được loại khỏi danh sách phạm vi, cũng có nghĩa là bộ phân tích đã kết thúc phân tích khôi hiện tại và tiếp tục phân tích khôi bao ngoài nó.

Các khai báo sau đây dùng mô tả các bản ghi của cây định danh, danh sách định nghĩa, danh sách phạm vi và danh sách tên. Kiểu *e_type* có các ý nghĩa sau:

<i>no_type</i>	không có kiểu
<i>unknown_type</i>	không hiểu kiểu gì
<i>int_type</i>	số nguyên

Còn kiểu của *e_kind* như sau:

<i>unknown_kind</i>	không hiểu loại gì
<i>const_kind</i>	tên hằng số
<i>var_kind</i>	tên biến
<i>proc_kind</i>	tên thủ tục

Con trỏ *ident_tree* trỏ đến cây định danh và con trỏ *scope_list* trỏ đến danh sách phạm vi đầy xuống.

```
/** SLC.H **/
#define max_ident_length 255
typedef char string[max_ident_length + 1];

typedef enum {
    no_type, unknown_type, int_type
} e_type;

typedef enum {
    unknown_kind, const_kind, var_kind, proc_kind,
} e_kind;

typedef struct def_rec { /* Khai báo bản ghi định nghĩa */
    int                         sn, dpl_or_value_or_index;
    e_type                      type;
    e_kind                      kind;
    struct def_rec *next;
} *def_ptr;

typedef struct ident_rec { /* Khai báo bản ghi định danh */
    string                     ident;
    def_ptr                    def_list;
    struct ident_rec *left, *right;
} *ident_ptr;

typedef struct name_rec { /* Khai báo bản ghi tên */
    string                     ident;
    struct name_rec *next;
}
```

```

} *name_ptr;

typedef struct scope_rec {           /* Khai báo bản ghi phạm vi */
    name_ptr          name_list;
    struct scope_rec *prev_scope;
} *scope_ptr;

#define UNDECLARED NULL



---


/** SLC.C ***/
ident_ptr      ident_tree;
scope_ptr       scope_list;
int            scope_level;

```

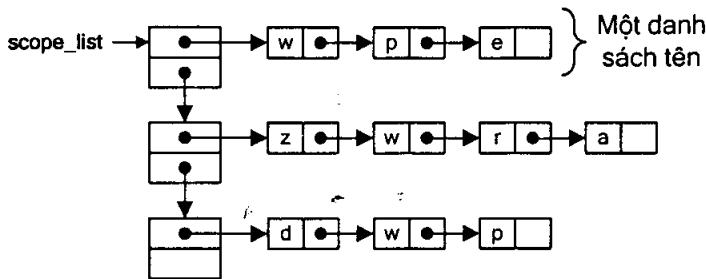
Ví dụ 5.2 :

Ta xem lại ví dụ 5.1. Hình 5.2 cho thấy cây định danh và các danh sách bản ghi định nghĩa của các tên "p", "w" và "z" khi bộ phân tích ở mức 3 (dòng 9, trong thân thủ tục "z"). Chú ý rằng, trường đầu tiên của bản ghi tên có chứa số phân đoạn. Trường thứ hai có nội dung là vị trí của các trường hợp "p" và "w", và chỉ số (trong bảng ký hiệu) trong trường hợp của "z". Trong ví dụ này, ta coi chỉ số là 1. Các trường còn lại chứa kiểu và loại của tên, và con trỏ đến trường thứ hai. Trong hình 5.3 ta sẽ thấy danh sách phạm vi tương ứng và các danh sách bản ghi tên.

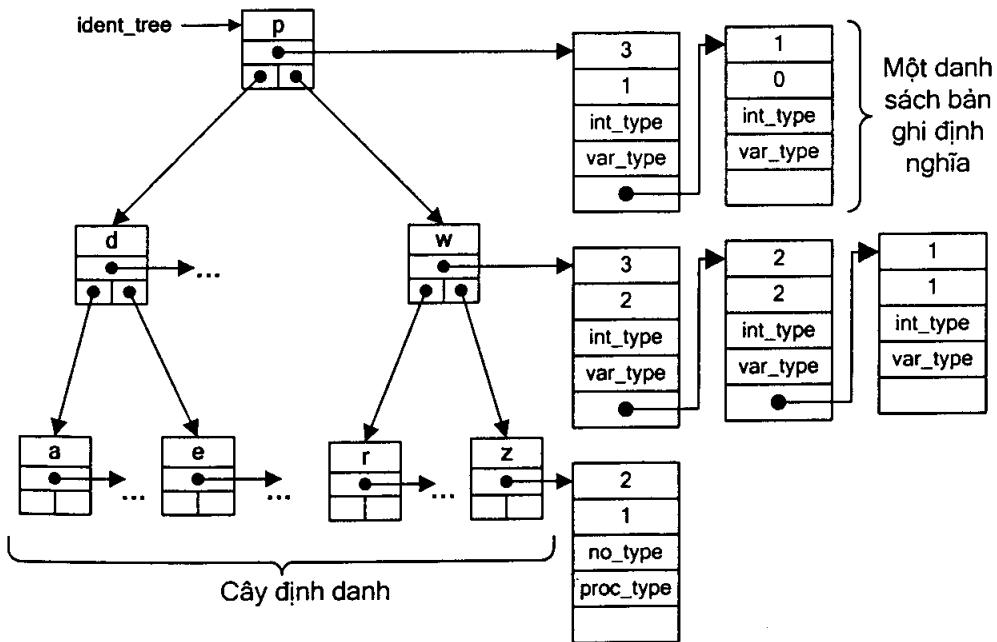
Trong các mắt xích, các thủ tục phân tích sẽ sử dụng các biến cục bộ sau để lưu thông tin từ bảng phân tích:

- Biến con trỏ *def* sẽ được dùng để giữ địa chỉ của bản ghi định nghĩa đang được xem xét hiện thời.
- Các biến *sn* và *dpl* sẽ được sử dụng theo hai cách: để giữ giá trị số phân đoạn hiện thời và vị trí tương đối, giữ các giá trị lấy từ các trường của bản ghi định nghĩa giám sát hiện thời trong bảng ký hiệu.
- Trong trường hợp một hằng số, biến *value* sẽ được dùng để giữ giá trị lấy ra này.
- Trong trường hợp một thủ tục, biến *address* sẽ được dùng để giữ chỉ số lấy ra của địa chỉ trong bảng ký hiệu.
- Trong tất cả các trường hợp trên, biến *kind* giữ thông tin về loại lấy được.

Kiểu không được lấy ra từ bảng ký hiệu vì trong trường hợp của biến hoặc hằng số, chỉ có một kiểu là kiểu số nguyên.



Hình 5.2. Cây định danh và các danh sách bản ghi định nghĩa.



Hình 5.3. Danh sách phạm vi và các danh sách bản ghi tên.

Các thủ tục và hàm làm việc với bảng ký hiệu

Khi bắt đầu quá trình phân tích, bảng ký hiệu sẽ được khởi đầu với cây định danh rỗng (nghĩa là biến *ident_tree* sẽ được đặt thành con trỏ null), và danh sách phạm vi đầy xuống sẽ được khởi đầu là một danh sách rỗng (nghĩa là biến *scope_list* sẽ được đặt thành con trỏ null). Độ sâu lồng nhau cũng được khởi đầu, nghĩa là mức phạm vi *scope_level* sẽ được đặt về 0. Bây giờ ta sẽ tìm hiểu một số thủ tục hoạt động với bảng ký hiệu.

Khi vào một khối, ta đã vào một phạm vi mới. Do đó, một bản ghi phạm vi mới sẽ phải được tạo ra và đẩy vào trong danh sách phạm vi. Việc này được thực hiện bằng thủ tục:

```
void enter_scope(void);
```

thủ tục này cũng sẽ tăng mức phạm vi lên.

Khi ta rời khỏi khối, phạm vi hiện tại được thoát ra. Tất cả các tên định nghĩa trong phạm vi hiện tại sẽ trở thành mảng giá trị. Do đó, các bản ghi định nghĩa của chúng phải được loại bỏ khỏi bảng ký hiệu, và bản ghi phạm vi hiện tại phải được loại khỏi đỉnh của danh sách đầy xuống. Các hành động này được thực hiện nhờ thủ tục:

```
void exit_scope(void);
```

thủ tục này cũng sẽ giảm mức phạm vi đi.

Hàm:

```
int current_scope(void);
```

trả lại mức phạm vi của một khối hiện đang được phân tích.

Bản ghi định nghĩa phải được tạo cho mỗi khai báo của một tên và cả tên lẫn bản ghi định nghĩa đó phải được thêm vào bảng ký hiệu. Hơn nữa, vì lý do tối ưu, tên phải được thêm vào danh sách tên trên đỉnh của danh sách phạm vi đầy xuống. Để lưu khai báo của các tên, hàm và thủ tục sau được định nghĩa:

```
def_ptr create_definition (int sn, int dpl_or_value_or_index,  
                           e_type type, e_kind kind);
```

```
void insert_definition(string ident, def_ptr def);
```

Khi gọi hàm *create_definition* sẽ tạo ra kết quả là một bản ghi định nghĩa mới. Các trường của bản ghi này được khởi đầu với các tham số *sn*, *dpl_or_value_or_index*, *type*, *kind*. Hàm trả lại một con trỏ đến bản ghi được tạo ra.

Ví dụ 5.3: Lệnh gán

```
def = create_definition (sn, dpl, int_type, var_kind);
```

tạo ra một bản ghi định nghĩa cho một biến có kiểu số nguyên với số phân đoạn là giá trị của *sn* và vị trí chứa giá trị *dpl*. Tương tự, phép gán:

```
def = create_definition (sn, num_value, int_type, const_kind);
```

tạo ra một bản ghi định nghĩa cho một hằng với kiểu số nguyên, số phân đoạn là giá trị của *sn* và giá trị hiện tại là *num_value* (do phần phân tích từ vựng đưa ra).

Thủ tục *insert_definition* được gọi sẽ tạo ra các kết quả sau:

1. Một tên, ứng với tham số *ident*, được thêm vào cây định danh nếu nó còn chưa có trong đó.
2. Bản ghi định nghĩa được trả bằng *def*, được thêm vào danh sách các bản ghi định nghĩa cho tên đó.
3. Tên được thêm vào danh sách tên, trên đỉnh của danh sách phạm vi đầy xuống. Nếu danh sách tên có sẵn một bản ghi cho tên đó thì thông báo lỗi "Tên được định nghĩa hai lần" sẽ được in ra.

Ví dụ 5.4:

Lời gọi

```
insert_definition(ident_lexeme, def);
```

thêm tên trong bộ đệm *ident_lexeme* vào cây định danh và vào danh sách tên trên đỉnh của danh sách phạm vi. Hơn nữa, bản ghi được trả bởi *def* được chèn vào mặt trước của danh sách các bản ghi định nghĩa của tên đang xét.

Để giữ các khai báo sử dụng của các tên, các hàm sau được định nghĩa:

```
def_ptr find_definition(string ident);
int get_sn(def_ptr def);
int get_dpl(def_ptr def);
int get_value(def_ptr def);
int get_address(def_ptr def);
e_type get_type(def_ptr def);
e_kind get_kind(def_ptr def);
```

Hàm *find_definition* trả lại một con trỏ trỏ đến bản ghi định nghĩa của tên *ident*. Nếu không tìm được bản ghi định nghĩa nào cho tên này thì in ra thông báo lỗi "Tên chưa được định nghĩa". Nếu tên không tồn tại trong toàn bộ cây định danh thì thông báo lỗi "Tên không có trong cây định danh". Trong cả hai trường hợp trên, giá trị UNDECLARED (không khai báo) được trả lại và trả đến bản ghi ngầm định. Hàm *get_sn*, *get_dpl* (cho một biến), *get_value* (cho một hàng), *get_address* (cho một thủ tục), *get_type* và *get_kind* lấy các thông tin cần thiết từ các trường của bản ghi định nghĩa. Nếu không có bản ghi nào tồn tại thì giá trị ngầm định (0 cho *sn* và *dpl*, *value* và *address*, *unknown_type* cho *type*, và *unknown_kind* cho *kind*) sẽ được trả lại. Chú ý là hàm *get_address* trả lại chỉ số trong bảng địa chỉ tại địa chỉ của một thủ tục được lưu và không phải bản thân địa chỉ đó.

Ví dụ 5.5

Lệnh:

```
def = find_definition(ident_lexeme);
kind = get_kind(def);
if (kind != var_kind)
    report_error("Tên không phải là một biến");
```

trả lại loại (biến, hằng hoặc thủ tục) của tên trong bộ đệm *ident_lexeme* và kiểm tra xem tên này có phải là một biến hay không.

Phân tích ngữ nghĩa

Trong phần này ta sẽ cải tiến bộ phân tích trong chương trước với việc thêm các hành động để kiểm tra các ràng buộc về phạm vi và kiểu, loại. Đây là những nhiệm vụ quan trọng của phần phân tích ngữ nghĩa.

Mỗi khôi biểu diễn một phạm vi. Do vậy, trong lúc phân tích, khi vào trong một khôi, một phạm vi mới phải được mở ra, và tại điểm ra của khôi, phạm vi phải được kết thúc. Điều này được thực hiện bằng các thủ tục *enter_scope* và *exit_scope* tương ứng. Thủ tục *program_declaration* do vậy trở thành:

```
void program_declaration(void)
{
    ⇒ enter_scope();
    block();
    ⇒ exit_scope();
    if (look_ahead == period_token) next_token();
}
```

Tại phần đầu của khôi khai báo, giá trị của vị trí *dpl* được khởi đầu là 0, và giá trị này được tăng cho mỗi biến con trong phạm vi - ứng với thủ tục *variable_declaration*. Do vậy, thủ tục *variable_declaration* có *dpl* trong cả tham số vào và ra. Chú ý rằng, khôi của mọi khai báo thủ tục có giá trị *dpl* cho bản thân nó. Do đó, biến được khai báo cục bộ trong thủ tục *declaration_part* và trở thành:

```
void declaration_part(void)
{
    ⇒ int dpl;
    ⇒ dpl = 0;
    while (look_ahead == int_token || look_ahead == var_token) {
        if (look_ahead == int_token)
            constant_declaration();
```

```

    else if (look_ahead == var_token)
        variable_declaration(&dpl);
    }

    while (look_ahead == procedure_token)
        procedure_declaration();
}

```

Đối với mỗi khai báo của một tên, thông tin khai báo của nó phải được thêm vào bảng ký hiệu nên khi sử dụng tên này, các thông tin cần thiết có thể được lấy ra từ bảng ký hiệu. Các câu lệnh:

```

def = create_definition (sn, num_value, int_type, const_kind);
insert_definition (ident_lexeme, def);

```

tạo ra một bản ghi định nghĩa mới cho một hằng số và thêm cả tên và bản ghi định nghĩa của nó vào bảng ký hiệu. Đối với một biến, các câu lệnh là:

```

def = create_definition (sn, dpl, int_type, var_kind);
insert_definition (ident_lexeme, def);

```

và đối với một thủ tục là:

```

def = create_definition (sn, proc_index, no_type, proc_kind);
insert_definition (ident_lexeme, def);

```

Kiểu *no_type* là do thủ tục không ứng với một kiểu nào cả. Điều này giả sử là biến *proc_index* giữ giá trị của chỉ số trong bảng địa chỉ của thủ tục này. Vai trò của bảng địa chỉ cho các thủ tục sẽ được giải thích trong phần sau.

Dưới đây là 3 thủ tục đã được sửa:

```

void constant_declaration(void)
{
    ⇒     int          sn;
    ⇒     def_ptr      def;

    ⇒     sn = current_scope();
    type_declarer();
    if (look_ahead == ident_token) next_token();
    if (look_ahead == equal_token) next_token();
    if (look_ahead == num_token) next_token();
    ⇒     def = create_definition (sn, num_value, int_type, const_kind);
    ⇒     insert_definition (ident_lexeme, def);
    while (look_ahead == list_token) {
        if (look_ahead == list_token) next_token();
        if (look_ahead == ident_token) next_token();
        if (look_ahead == equal_token) next_token();
        if (look_ahead == num_token) next_token();
    }
    ⇒     def = create_definition (sn, num_value,

```

```

                                int_type, const_kind);
⇒      insert_definition (ident_lexeme, def);
}
if (look_ahead == separator_token) next_token();
}

⇒ void variable_declaration(int *dpl)
{
⇒      int          sn;
⇒      def_ptr  def;

⇒      sn = current_scope();
if (look_ahead == var_token) next_token();
type_declarer();
if (look_ahead == ident_token) next_token();
⇒      def = create_definition (sn, *dpl, int_type, var_kind);
⇒      insert_definition (ident_lexeme, def);
⇒      (*dpl)++;
while (look_ahead == list_token) {
    if (look_ahead == list_token) next_token();
    if (look_ahead == ident_token) next_token();
⇒      def = create_definition (sn, *dpl, int_type, var_kind);
⇒      insert_definition (ident_lexeme, def);
⇒      (*dpl)++;
}
if (look_ahead == separator_token) next_token();
}

void procedure_declaration(void)
{
⇒      int          sn, address;
⇒      def_ptr  def;

⇒      sn = current_scope();
if (look_ahead == procedure_token) next_token();
/* address = sẽ được sửa sau */
⇒      def = create_definition (sn, address, no_type, proc_kind);
⇒      insert_definition (ident_lexeme, def);
if (look_ahead == ident_token) next_token();
⇒      enter_scope();
block();
⇒      exit_scope();
if (look_ahead == separator_token) next_token();
}

```

Trong trường hợp câu lệnh gán vé trái phải là một biến, nếu không một thông báo lỗi sẽ được in ra. Điều này cũng tương tự đối với câu lệnh read. Các câu lệnh sau:

```
def = find_definition (ident_lexeme);
kind = get_kind(def);
if (kind != var_kind) report_error("Tên không là một biến");
```

lấy kiểu của tên này trong bản ghi định nghĩa. Nếu tên không được khai báo là một biến thì thông báo lỗi "Tên không là một biến" sẽ được in ra.

Chú ý là việc kiểm tra kiểu giữa vé trái và vé phải trong phép gán là không cần thiết trong ngôn ngữ SLANG do nó chỉ có mỗi một kiểu dữ liệu.

Thủ tục *left_part* và *read_statement* trở thành:

```
void left_part(void)
{
⇒    def_ptr  def;
⇒    e_kind   kind;

⇒    def = find_definition (ident_lexeme);
⇒    kind = get_kind(def);
⇒    if (kind != var_kind) report_error("Tên không là một biến");
    if (look_ahead == ident_token) next_token();
}

void read_statement(void)
{
⇒    def_ptr  def;
⇒    e_kind   kind;

    if (look_ahead == read_token) next_token();
    if (look_ahead == open_token) next_token();
⇒    def = find_definition (ident_lexeme);
⇒    kind = get_kind(def);
⇒    if (kind != var_kind) report_error("Tên không là một biến");
    if (look_ahead == ident_token) next_token();
    while (look_ahead == list_token) {
        if (look_ahead == list_token) next_token();
        def = find_definition (ident_lexeme);
        kind = get_kind(def);
    }
}
```

```

⇒ if (kind != var_kind)
    report_error("Tên không là một biến");
    if (look_ahead == ident_token) next_token();
}
if(look_ahead == close_token) next_token();
}

```

Tương tự như vậy, tên trong một câu lệnh call phải là tên của một thủ tục.

```

void call_statement(void)
{
⇒ def_ptr def;
⇒ e_kind kind;

if (look_ahead == call_token) next_token();
⇒ def = find_definition (ident_lexeme);
⇒ kind = get_kind(def);
⇒ if (kind != proc_kind)
    report_error("Tên không là một thủ tục");
    if (look_ahead == ident_token) next_token();
}

```

Cuối cùng, một tên làm toán hạng trong một biểu thức phải là một biến hoặc một hằng số.

```

void operand(void)
{
⇒ def_ptr def;
⇒ e_kind kind;

if (look_ahead == ident_token) {
    def = find_definition (ident_lexeme);
    kind = get_kind(def);
    if (kind==proc_kind || kind==unknown_kind)
        report_error("Tên phải là biến hoặc hằng số");
    if (look_ahead == ident_token) next_token();
}
else if (look_ahead == num_token) {
    if (look_ahead == num_token) next_token();
} else if (look_ahead == open_token) {
    if (look_ahead == open_token) next_token();
    expression();
    if (look_ahead == close_token) next_token();
}
}

```

III. HOÀN THIỆN VÀ THỬ NGHIỆM CHƯƠNG TRÌNH

1. Hoàn thiện

Ta hoàn thiện một số thủ tục còn lại như sau:

```
/** SLC.C ***/  
  
/* Phần bảng ký hiệu và phân tích ngữ nghĩa */  
void initialise_symboltable(void)  
{  
    scope_list = NULL;  
    ident_tree = NULL;  
    scope_level = 0;  
}  
  
void enter_scope(void)  
{  
    scope_ptr      scp;  
    /* Thủ tục này tạo ra một bản ghi phạm vi mới và chèn vào  
       đầu danh sách phạm vi */  
    scp = (scope_ptr) malloc(sizeof(struct scope_rec));  
    scp->name_list = NULL;  
    scp->prev_scope = scope_list;  
    scope_list = scp;  
    scope_level++;  
}  
  
void exit_scope(void)  
{  
    scope_ptr      scp;  
    name_ptr       np1, np2;  
    ident_ptr      ip;  
    def_ptr        dp;  
    int            k;  
  
    /* Phần này tìm và xóa tất cả các bản ghi trong danh sách tên  
       ứng với phạm vi trên cùng */  
    for (np1 = scope_list->name_list; np1 != NULL; np1 = np2) {  
        np2 = np1->next;  
        /* Phần này tìm và xóa bản ghi đầu tiên trong danh sách  
           bản ghi định nghĩa ứng với tên của bản ghi tên - để  
           làm điều này đòi hỏi phải duyệt cây định danh */  
        for (ip=ident_tree; ip!=NULL;) {  
            k = strcmp(ip->ident, np1->ident);  
            if (k == 0) {  
                dp = ip->def_list;  
                ip->def_list = dp->next;  
                free(dp);  
                break;  
            }  
        }  
    }  
}
```

```

        }
        if (k > 0) ip = ip->left; else ip = ip->right;
    }

    free(np1);
}

/* Xóa bản ghi trên cùng của danh sách phạm vi */
scp = scope_list;
scope_list = scope_list->prev_scope;
free(scp);
scope_level--;
}

int current_scope(void)
{
    return scope_level;
}

def_ptr create_definition (int sn, int dpl_or_value_or_index, e_type type, e_kind kind)
{
    def_ptr dp;

    dp = (def_ptr) malloc(sizeof(struct def_rec));
    dp->sn = sn;
    dp->dpl_or_value_or_index = dpl_or_value_or_index;
    dp->type = type;
    dp->kind = kind;
    dp->next = NULL;

    return dp;
}

void insert_definition (string ident, def_ptr def)
{
    ident_ptr ip1, ip2;
    name_ptr np;
    int k;

    /* Tìm bản ghi ứng với tên trong cây định danh */
    ip2 = ident_tree;
    while (ip2 != NULL) {
        ip1 = ip2;
        k = strcmp(ip1->ident, ident);
        if (k == 0) break;
        if (k > 0) ip2 = ip1->left; else ip2 = ip1->right;
    }

    /* Nếu tìm thấy thì thêm bản ghi định nghĩa trả bởi def vào
       đầu danh sách định nghĩa */
    if (k==0 && ident_tree==NULL) {

```

```

    def->next = ip1->def_list;
    ip1->def_list = def;
}
else {
    /* Nếu không thấy thì tạo nút mới cho cây định danh */
    ip2 = (ident_ptr) malloc(sizeof(struct ident_rec));
    strcpy(ip2->ident, ident);
    ip2->def_list = def;
    ip2->left = ip2->right = NULL;

    if (ident_tree == NULL) ident_tree = ip2; /* Trường hợp
                                               nút đầu tiên của cây */
    else
        if (k > 0) ip1->left = ip2; else ip1->right = ip2;
}

/* Kiểm tra xem tên đã cho có sẵn trong danh sách tên của
   mức phạm vi hiện tại hay không (kiểm tra tính duy nhất) */
for (np = scope_list->name_list; np != NULL; np = np->next)
    if (strcmp(ident, np->ident)==0) {
        report_error("Tên được định nghĩa hai lần");
        return;
    }

/* Nếu chưa thì thêm vào đầu danh sách tên */
np = (name_ptr) malloc(sizeof(struct name_rec));
strcpy(np->ident, ident);
np->next = scope_list->name_list;
scope_list->name_list = np;
}

def_ptr find_definition (string ident)
{
    ident_ptr ip1, ip2;
    int      k;

    /* Duyệt cây định danh */
    ip2 = ident_tree;
    while (ip2 != NULL) {
        ip1 = ip2;
        k = strcmp(ip1->ident, ident);
        if (k == 0) break;
        if (k > 0) ip2 = ip1->left; else ip2 = ip1->right;
    }

    if (k == 0) {
        if (ip1->def_list != NULL)
            return ip1->def_list;
        report_error("Tên chưa được định nghĩa");
    }
}

```

```

    else
        report_error("Tên không có trong cây định danh");

    return UNDECLARED;
}

int get_sn (def_ptr def)
{
    return def->sn;
}

int get_dpl (def_ptr def)
{
    return def->dpl_or_value_or_index;
}

int get_value (def_ptr def)
{
    return def->dpl_or_value_or_index;
}

int get_address (def_ptr def)
{
    return def->dpl_or_value_or_index;
}

e_type get_type(def_ptr def)
{
    return def->type;
}

e_kind get_kind(def_ptr def)
{
    return def->kind;
}

void report_error(char *errmsg)
{
    printf("%s\n", errmsg);
}

/*** Thủ chương trình chính ***/
void main(void)
{
    initialise_scanner();
    initialise_symboltable();
    next_token();
    program_declaration();
}

```

2. Thủ nghiệm chương trình

Phương pháp đơn giản nhất để thử nghiệm chương trình này là cho nó dịch các chương trình nguồn SLANG viết đúng và viết sai về mặt ngữ nghĩa.

Chương trình trên nếu hoạt động tốt thì khi dịch các chương trình nguồn SLANG viết đúng sẽ không đưa ra một thông báo gì. Bạn hãy thử với các ví dụ 2.1 và 2.2 (VIDU21.PRO và VIDU22.PRO) cũng như tất cả các chương trình khác do bạn viết trong SLANG và nếu viết đúng đều phải không có thông báo gì.

Còn nếu gặp chương trình viết sai thì tùy theo lỗi sai, nó sẽ đưa ra các thông báo thích hợp. Các ví dụ dưới đây có nhiều chỗ có lỗi ngữ nghĩa. Các chỗ gây lỗi được in đậm để bạn dễ nhận ra.

Ví dụ 5.6: Nếu bạn cho dịch chương trình sau:

```
begin  
var int sum, number, sum;  
    number := 10;  
end.
```

sẽ phải có thông báo đưa ra như sau:

Tên được định nghĩa hai lần

Ví dụ 5.7: Nếu bạn cho dịch chương trình sau:

```
begin  
var int sum, number;  
    proc abc  
        begin  
            var int k;  
            sum := number + 10*k;  
        end;  
    end.
```

lại là một chương trình viết đúng và không có thông báo gì.

Ví dụ 5.8: Nếu bạn cho dịch chương trình sau:

```
begin  
var int sum, number;  
    proc abc
```

```
begin
    var int k;
        sum := abc + 10*k;
    end;
end.
```

sẽ phải có thông báo đưa ra như sau:

Tên phải là một biến hoặc một hằng số

Ví dụ 5.9: Nếu bạn cho dịch chương trình sau:

```
begin
    var int sum, number;
        proc abc
            begin
                var int k;
                    sum := number + 10*k;
            end;
            sum := number + 10*k;
        end.
```

sẽ phải có các thông báo đưa ra như sau:

Tên chưa được định nghĩa

Bài tập

1. Cho chương trình viết trong SLANG như sau:

1) BEGIN VAR INT i;

2) PROC p

3) BEGIN VAR INT i;

4) ...

5) END;

6) PROC q

7) BEGIN

8) END;

9) ...

10) END.

Hãy vẽ bảng ký hiệu khi phân tích xong các dòng 3, 7 và 8.

2. Ta đã thấy (trong chương trình SLC trên) để làm việc với bảng ký hiệu, các thủ tục /hàm quản lý bảng ký hiệu được thêm vào các thủ tục của phần phân tích cú pháp. Ta cũng thấy trong chương 3, khi viết khối phân tích từ vựng thì chưa hề có bảng ký hiệu. Bạn hãy suy nghĩ và cho biết các nhận định sau có đúng không? Lý giải?

- Phần phân tích cú pháp làm việc trực tiếp với bảng ký hiệu. Nói cách khác, trong trường hợp này phải có đường nối giữa hai khối đó với nhau.
- Phần phân tích không làm việc với bảng ký hiệu. Nói cách khác, trong trường hợp này phải xoá đường nối giữa hai khối đó.

Bài tập lập trình

Giả sử bạn đã mở rộng ngôn ngữ SLANG cho các lệnh repeat, for và case. Hãy sửa chương trình để kiểm tra ngữ nghĩa cho những mở rộng này.

Chương 6

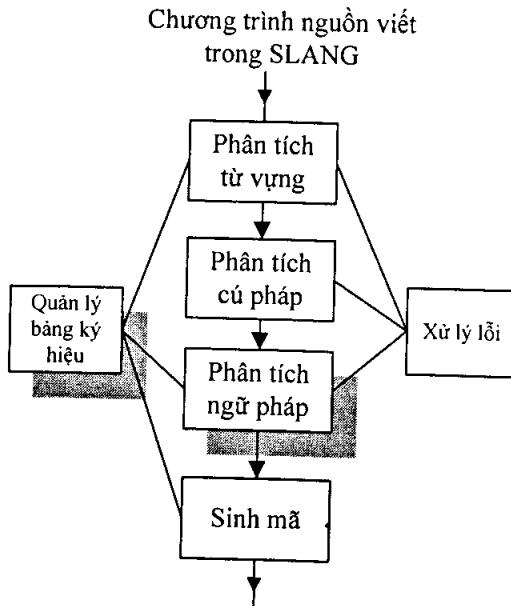
SINH MÃ

I. MỤC ĐÍCH, NHIỆM VỤ

Sinh mã là giai đoạn cuối của quá trình dịch. Nhiệm vụ của khối này là sinh ra mã đối tượng cho một máy tính đích cụ thể nào đó.

Trong phần thực hành ta sẽ sinh mã cho máy tính ảo VIM (đã được trình bày trong chương 2).

Cũng như chương trước, để sinh mã ta phải gắn các luật sinh mã với các luật cú pháp (tức là cú pháp điều khiển) và dùng phương pháp duyệt "trên cùng chuyên bay" để thực hiện việc sinh mã ngay trong quá trình phân tích.



Chương trình đích cho máy tính
VIM

Hình 6.1. Vị trí của bộ sinh
mã trong chương trình
dịch thực hành.

II. SINH MÃ CHO VIM

Trong phần này chúng ta cần phải có nhiều thủ tục khác nhau để sinh mã VIM. Các thủ tục này có số lượng và loại tham số khác nhau.

Khi bắt đầu và kết thúc sinh mã phải gọi hai thủ tục:

```
void initialise_vimcode(void);
void finalise_vimcode(void);
```

Thủ tục `initialise_vimcode` khởi đầu việc sinh các chỉ thị lệnh VIM. Nó mở một file để ghi các chỉ thị lệnh ra (file ra). Còn thủ tục `finalise_vimcode` đầu tiên tạo ra lệnh `halt`, sau đó ghi bảng địa chỉ và bảng độ dài vào các file khác (nguyên nhân có các bảng này đã được giải thích ở chương 2), và cuối cùng thì đóng file ra lại. Thủ tục `program_declaration` bây giờ trở thành:

```
void program_declaration(void)
{
    ⇒ initialise_vimcode();
    enter_scope();
    block();
    exit_scope();
    ⇒ finalise_vimcode();
    if(look_ahead == period_token) next_token();
}
```

Khi cho chạy một khối của thủ tục, cần phải tạo một phân đoạn mới (segment) cho khối này. Các thủ tục dùng để sinh các chỉ thị `crseg sn, index` và `dlseg` là

```
void emit_crseg(int level, int index);
void emit(operation code);
```

Thủ tục `emit_crseg` chỉ sinh mã cho một chỉ thị lệnh cụ thể, còn thủ tục `emit` được dùng để sinh các chỉ thị lệnh VIM không có tham số. Miền của kiểu `operation` là tập chỉ thị của máy VIM.

Hàm `current_scope` cho mức phạm vi hiện tại. Đó là mức dùng trong lời gọi `emit_crseg`. Thủ tục này lấy tham số thứ hai từ bảng độ dài. Một vị trí trống trong bảng độ dài có thể nhận được từ thủ tục

```
int get_index(void);
```

Bước tiếp theo là lưu độ dài của phân đoạn tại vị trí này. Điều đó được thực hiện bằng thủ tục

```
void enter_length(int index, int length);
```

Độ dài của phân đoạn được tạo ra bằng thủ tục *declaration_part*, do vậy nó có tham số *length*. Thủ tục *block* trở thành:

```
void block(void)
{
    int      level, length_index, length;
    if(look_ahead == begin_token) next_token();
    declaration_part(&length);
    level = current_scope();
    length_index = get_index();
    enter_length(length_index, length);
    emit_crseg(level, length_index);
    statement_part();
    emit(dlseg);
    if(look_ahead == end_token) next_token();
}
```

Trong thủ tục *declaration_part*, ta phải thực hiện một lệnh nhảy qua các mã của các thủ tục cục bộ trong phần khai báo này, do đó các khối con của nó chỉ được thực hiện khi các thủ tục con này được gọi. Ta sẽ bỏ qua lệnh nhảy này nếu không có khai báo của các thủ tục cục bộ.

Các lệnh VIM gây ảnh hưởng đến thanh đếm chỉ thị lệnh (biến ic - bao gồm các lệnh jump, jift, jiff, call) được tạo ra bằng thủ tục sinh mã:

```
void emit_jump(operation code, int index);
```

Thủ tục này sẽ lấy tham số thứ hai từ bảng địa chỉ. Một vị trí còn trống trong bảng địa chỉ được trả về nhờ hàm:

```
int get_label(void);
```

Lệnh gán

```
over_procs = get_label();
```

sẽ gán chỉ số của vị trí trống trong bảng địa chỉ cho over_procs. Lời gọi

```
emit_jump(jump, over_procs);
```

sẽ dùng vị trí này. Địa chỉ nhảy được biết tại điểm cuối của *declaration_part*. Tại điểm này, giá trị hiện tại của con trỏ lệnh ic phải được lưu trong bảng địa chỉ tại vị trí over_procs. Thực hiện điều này bằng thủ tục

```
void emit_label(int index);
```

Thủ tục *declaration_part* trở thành:

```
⇒ void declaration_part(int *length)
{
    ⇒     int      dpl, over_procs;

    dpl = 0;
    while (look_ahead == int_token || look_ahead == var_token) {
        if (look_ahead == int_token) {
            constant_declaration();
        } else if (look_ahead == var_token) {
            variable_declaration(&dpl);
        }
    }

    ⇒     *length = dpl;
    ⇒     if (look_ahead == procedure_token) {
    ⇒         over_procs = get_label();
    ⇒         emit_jump(jump, over_procs); /* Sinh chỉ thị nhảy jump over_procs */
    ⇒         procedure_declaration();

        while (look_ahead == procedure_token)
            procedure_declaration();

        ⇒         emit_label(over_procs); /* Đánh nhãn cho over_procs */
    }
}
```

Chú ý là giá trị cuối của *dpl* được trả về làm độ dài của phân đoạn dữ liệu của thủ tục.

Các thủ tục *constant_declaration* và *variable_declaration* không sinh ra một mã nào, nhưng lại lưu thông tin vào bảng ký hiệu. Trong trường hợp của hằng số, ta lưu giá trị của nó, và trong trường hợp của một biến là số phân đoạn và vị trí của nó. Khi sinh mã, các thông tin này được lấy từ bảng ký hiệu và sử dụng để sinh mã.

Thủ tục *procedure_declaration* cần một chỉ số trong bảng địa chỉ cho địa chỉ của nó. Giá trị của chỉ số này phải được lưu trong bản ghi định nghĩa của thủ tục (trong bảng ký hiệu), nên câu lệnh call có thể tìm địa chỉ trong thân thủ tục. Một câu lệnh return phải được sinh ra sau khi thủ tục *block* dịch xong. Thủ tục *procedure_declaration* trở thành:

```
void procedure_declaration(void)
{
    int      sn, address;
```

```

def_ptr def;

sn = current_scope();
if (look_ahead == procedure_token) next_token();
⇒ address = get_label();
⇒ emit_label(address);
def = create_definition (sn, address, no_type, proc_kind);
insert_definition (ident_lexeme, def);
if (look_ahead == ident_token) next_token();
enter_scope();
block();
exit_scope();
⇒ emit(return_);
if(look_ahead == separator_token) next_token();
}

```

Các thủ tục statement_part và statement cũng không sinh ra mã nào cả.

Đối với một câu lệnh gán, trước tiên ta phải đánh giá giá trị của biểu thức về phải, và sau đó mới gán giá trị này cho biến ở về trái. Đối với phép gán, ta cần chỉ thị stvar, với các tham số là số phân đoạn và vị trí của biến về trái. Thủ tục *left_part* cần cấp các giá trị đó và do vậy phải có thêm tham số.

Các chỉ thị VIM ldvar sn, dpl và stvar sn, dpl được sinh nhờ các thủ tục sau:

```

void emit_ldvar (int sn, int dpl);
void emit_stvar (int sn, int dpl);

```

Các thủ tục assignment_statement và left_part trở thành:

```

void assignment_statement(void)
{
    ⇒ int     sn, dpl;

    ⇒ left_part(&sn, &dpl);
    if(look_ahead == becomes_token) next_token();
    expression();
    ⇒ emit_stvar(sn, dpl);
}

⇒ void left_part(int *sn, int *dpl)
{
    def_ptr def;
    e_kind kind;

    def = find_definition (ident_lexeme);
    kind = get_kind(def);
    if (kind != var_kind) report_error("Tên không là một biến");
    ⇒ *sn = get_sn(def);
}

```

```
⇒ *dpl = get_dpl(def);
    if(look_ahead == ident_token) next_token();
}
```

Việc dịch câu lệnh điều kiện và lặp đòi hỏi phải có chỉ thị lệnh nhảy và bảng địa chỉ. Các thủ tục if_statement và while_statement được sửa như sau:

```
void if_statement(void)
{
    ⇒ int      over_then_part, over_else_part;

    if(look_ahead == if_token) next_token();
    relation();
    if(look_ahead == then_token) next_token();
    ⇒ over_then_part = get_label();
    ⇒ emit_jump(jiff, over_then_part);
    statement_part();
    if (look_ahead == else_token) {
        ⇒ over_else_part = get_label();
        ⇒ emit_jump(jump, over_else_part);
        ⇒ emit_label(over_then_part);
        if (look_ahead == else_token) next_token();
        statement_part();
        ⇒ emit_label(over_else_part);
    } else
    ⇒ emit_label(over_then_part);

    if(look_ahead == fi_token) next_token();
}
```

```
void while_statement(void)
{
    ⇒ int      begin_while_stat, end_while_stat;

    if(look_ahead == while_token) next_token();
    ⇒ begin_while_stat = get_label();
    ⇒ emit_label(begin_while_stat);
    relation();
    ⇒ end_while_stat = get_label();
    ⇒ emit_jump(jiff, end_while_stat);
    if(look_ahead == do_token) next_token();
    statement_part();
    if(look_ahead == od_token) next_token();
    ⇒ emit_jump(jump, begin_while_stat ;
    ⇒ emit_label(end_while_stat);
```

Câu lệnh call cần chỉ số của địa chỉ thủ tục trong bảng địa chỉ. Thủ tục lưu chỉ số này trong bản ghi định nghĩa trong bảng ký hiệu. Do đó thủ tục phải lấy lại chỉ số từ bản ghi định nghĩa và trở thành:

```
void call_statement(void)
{
    def_ptr def;
    e_kind kind;
    ⇒ int address;

    if(look_ahead == call_token) next_token();
    def = find_definition(ident_lexeme);
    kind = get_kind(def);
    if (kind != proc_kind)
        report_error("Tên không là một thủ tục");
    ⇒ address = get_address(def);
    ⇒ emit_jump(call, address);
    if(look_ahead == ident_token) next_token();
}
```

Câu lệnh read đọc các con số từ đầu vào và gán chúng cho các tên trong danh sách, được thực hiện bằng chuỗi chỉ thị rdint và stvar. Câu lệnh write đưa ra các giá trị của chuỗi các biểu thức, được thực hiện thông qua chuỗi các chỉ thị wrint. Giá trị của từng biểu thức được đặt trong ngăn xếp và tiếp theo các lệnh wrint sẽ đưa chúng ra màn hình.

Các thủ tục khác liên quan được sửa như sau:

```
void read_statement(void)
{
    def_ptr def;
    e_kind kind;
    ⇒ int sn, dpl;

    if(look_ahead == read_token) next_token();
    if(look_ahead == open_token) next_token();
    def = find_definition(ident_lexeme);
    kind = get_kind(def);
    if (kind != var_kind) report_error("Tên không là một biến");
    ⇒ sn = get_sn(def);
    ⇒ dpl = get_dpl(def);
    ⇒ emit(rdint);
    ⇒ emit_stvar(sn, dpl);
    if(look_ahead == ident_token) next_token();
    while (look_ahead == list_token) {
        if(look_ahead == list_token) next_token();
        def = find_definition(ident_lexeme);
        kind = get_kind(def);
```

```

    if (kind != var_kind)
        report_error("Tên không là một biến");
⇒     sn = get_sn(def);
⇒     dpl = get_dpl(def);
⇒     emit(rdint);
⇒     emit_stvar(sn, dpl);
⇒     if(look_ahead == ident_token) next_token();
}
if(look_ahead == close_token) next_token();
}

void write_statement(void)
{
    if(look_ahead == write_token) next_token();
    if(look_ahead == open_token) next_token();
    expression();
⇒     emit(wrint);
    while (look_ahead == list_token) {
        if(look_ahead == list_token) next_token();
        expression();
⇒     emit(wrint);
    }
    if(look_ahead == close_token) next_token();
}

```

Các thủ tục expression, term, factor và relation cần phải biết được kiểu của phép toán sẽ dùng. Do vậy ta phải mở rộng các thủ tục add_operator, multiply_operator, unary_operator và relational_operator để thêm các tham số cho biết toán tử nào được dùng. Do vậy, kiểu của các tham số được khai báo như sau:

```

typedef enum { plus, minus } add_op;
typedef enum { times, over, modulo } mul_op;
typedef enum { absolute, negate } unary_op;
typedef enum { equal, not_equal, less_than, less_or_equal,
                  greater_than, greater_or_equal } rel_op;

```

Các thủ tục liên quan được sửa như sau:

```

void expression(void)
{
⇒     add_op op;
    term();
    while(look_ahead == minus_token || look_ahead == plus_token){
        add_operator(&op);
        term();
⇒      if (op == plus) emit(add);
⇒      else if (op == minus) emit(sub);
}

```

```

        }

void term(void)
{
    ⇒      mul_op op;

    .factor();
    while (look_ahead == modulo_token || look_ahead == over_token
           || look_ahead == times_token) {
        ⇒      multiply_operator(&op);
        factor();
        ⇒      if (op == times) emit(mul);
        ⇒      else if (op == over) emit(dvi);
        ⇒      else if (op == modulo) emit(mdl);
    }
}

void factor(void)
{
    ⇒      unary_op      op;

    if (look_ahead==absolute_token || look_ahead==negate_token) {
        ⇒      unary_operator(&op);
        ⇒      operand();
        ⇒      if (op == negate) emit(neg);
        ⇒      else if (op == absolute) emit(abs_);
    } else /* Nếu không có phép toán một ngôi */
        ⇒      operand();
}

⇒void add_operator(add_op *op)
{
    if(look_ahead == plus_token) {
        ⇒      if(look_ahead==plus_token) {*op = plus; next_token(); }
    } else if(look_ahead == minus_token) {
        ⇒      if(look_ahead==minus_token){*op = minus; next_token(); }
    }
}

⇒void multiply_operator(mul_op *op)
{
    if(look_ahead == times_token) {
        ⇒      if (look_ahead==times_token){*op = times; next_token();}
    } else if(look_ahead == over_token) {
        ⇒      if (look_ahead==over_token) { *op = over; next_token();}
    } else if(look_ahead == modulo_token) {
        ⇒      if (look_ahead==modulo_token){*op=modulo; next_token();}
    }
}

```

```

}

⇒void unary_operator(unary_op *op)
{
    if(look_ahead == negate_token) {
        if(look_ahead==negate_token){*op=negate; next_token();}
    } else if(look_ahead == absolute_token) {
        if ( look_ahead == absolute_token )
            { *op = absolute; next_token(); }
    }
}

void relation(void)
{
    rel_op op;

    expression();
    ⇒ relational_operator(&op);
    expression();
    ⇒ if (op == equal) emit(eq);
    ⇒ else if (op == not_equal) emit(ne);
    ⇒ else if (op == less_than) emit(lt);
    ⇒ else if (op == less_or_equal) emit(le);
    ⇒ else if (op == greater_than) emit(gt);
    ⇒ else if (op == greater_or_equal) emit(ge);
}

⇒void relational_operator(rel_op *op)
{
    if(look_ahead == equal_token) {
        if(look_ahead==equal_token) {*op=equal; next_token();}
    } else if(look_ahead == not_equal_token) {
        if ( look_ahead == not_equal_token )
            { *op = not_equal; next_token(); }
    } else if(look_ahead == less_than_token) {
        if(look_ahead == less_than_token)
            { *op = less_than; next_token(); }
    } else if(look_ahead == less_or_equal_token) {
        if(look_ahead == less_or_equal_token)
            { *op = less_or_equal; next_token(); }
    } else if(look_ahead == greater_than_token) {
        if(look_ahead == greater_than_token)
            { *op = greater_than; next_token(); }
    } else if(look_ahead == greater_or_equal_token) {
        if(look_ahead == greater_or_equal_token)
            { *op = greater_or_equal; next_token(); }
    }
}

```

Một toán hạng có thể là một biến, hằng số hoặc một biểu thức trong cặp ngoặc. Để sinh chỉ thị *ldcon value* ta đưa ra thủ tục sinh một loại mã như sau:

```
void emit_ldcon(int value);
```

Trong trường hợp một hằng số, giá trị của nó được lấy từ bản ghi định nghĩa trong bảng ký hiệu, và trong trường hợp một số, dùng giá trị hiện tại của *num_value*. Cuối cùng, thủ tục *operand* được sửa là:

```
void operand(void)
{
    def_ptr def;
    e_kind kind;
    ⇒ int sn, dpl, value;

    if(look_ahead == ident_token) {
        def = find_definition(ident_lexeme);
        kind = get_kind(def);
        ⇒ if (kind == var_kind) {
        ⇒     sn = get_sn(def);
        ⇒     dpl = get_dpl(def);
        ⇒     emit_ldvar(sn, dpl);
        ⇒ } else if (kind == const_kind) {
        ⇒     value = get_value(def);
        ⇒     emit_ldcon(value);
        ⇒ } else if (kind == proc_kind || kind == unknown_kind)
            report_error("Tên phải là biến hoặc hằng số");
        ⇒ if(look_ahead == ident_token) next_token();
    }
    else if(look_ahead == num_token) {
        if(look_ahead == num_token) next_token();
        ⇒ emit_ldcon(num_value);
    } else if(look_ahead == open_token) {
        if(look_ahead == open_token) next_token();
        expression();
        if(look_ahead == close_token) next_token();
    }
}
```

III. HOÀN CHỈNH VÀ THỬ NGHIỆM

1. Hoàn chỉnh

Bây giờ, khi đã hiểu cách hoạt động của phần sinh mã, ta hoàn thiện các thủ tục cụ thể như sau:

Thêm vào phần khai báo của chương trình những dòng sau:

```
/** SLCH ***/  
typedef enum {  
    neg, abs_, add, sub, mul, dvi, mdl, eq, ne, lt, le, gt, ge,  
    ldcon, ldvar, stvar, jump, jift, jiff, call, crseg, dlseg,  
    return_, rdint, wrint, halt  
} operation;  
  
typedef enum { plus, minus } add_op;  
typedef enum { times, over, modulo } mul_op;  
typedef enum { absolute, negate } unary_op;  
typedef enum { equal, not_equal, less_than, less_or_equal,  
             greater_than, greater_or_equal } rel_op;  
  
/* Khai báo các thủ tục trước khi sử dụng */  
void block(void);  
void declaration_part(int *length);  
void constant_declaration(void);  
void type_declarer(void);  
void variable_declarer(int *dpl);  
void procedure_declarer(void);  
void statement_part(void);  
void statement(void);  
void assignment_statement(void);  
void left_part(int *sn, int *dpl);  
void if_statement(void);  
void while_statement(void);  
void call_statement(void);  
void read_statement(void);  
void write_statement(void);  
void expression(void);  
void term(void);  
void factor(void);  
void operand(void);  
void add_operator(add_op *op);  
void multiply_operator(mul_op *op);  
void unary_operator(unary_op *op);  
void relation(void);  
void relational_operator(rel_op *op);  
  
void enter_scope(void);
```

```

void exit_scope(void);
def_ptr create_definition (int sn, int dpl_or_value_or_index, e_type type, e_kind kind);
void insert_definition (string ident, def_ptr def);
def_ptr find_definition (string ident);
int get_sn (def_ptr def);
int get_dpl (def_ptr def);
int get_value (def_ptr def);
int get_address (def_ptr def);
e_type get_type(def_ptr def);
e_kind get_kind(def_ptr def);
void report_error(char *errmsg);
int current_scope(void);

void initialise_vimcode(void);
void finalise_vimcode(void);
void emit_crseg(int level, int index);
void emit(operation code);
void enter_length(int index, int length);
void emit_jump(operation code, int index);
int get_index(void);
int get_label(void);
void emit_label(int index);
void emit_ldvar(int sn, int dpl);
void emit_stvar(int sn, int dpl);
void emit_ldcon(int value);

```

```

*** SLC.C ***
*** Phần khai báo biến và dữ liệu ***
int address_table[max_table];
int length_table[max_table];
int len_index, add_index, ic;
FILE *cfp, *afp, *lfp;

struct {
    operation code;
    union {
        int value; /* Idcon */
        struct {
            int sn1, dpl; /* ldvar, stvar */
            } u1;
        int index1; /* juip, jift, jiff, call */
        struct {
            int sn2, index2; /* creg */
            } u2;
        } u;
    } instruction;

/* Phần sinh mã */
void initialise_vimcode(void)
{

```

```

char      fname[100];

/* Nhập tên file sẽ chứa chương trình VIM */
printf("Ten file ra cua VIM: "); gets(fname);
if (NULL==(cfp = fopen(fname, "wb"))) {
    printf("Không thể mở file %s", fname); exit(1);
}
/* File sẽ chứa bảng địa chỉ */
printf("Tên file bảng địa chỉ: "); gets(fname);
if (NULL == (afp = fopen(fname, "wb"))) {
    printf("Không thể mở file %s", fname); exit(1);
}
/* File sẽ chứa bảng độ dài */
printf("Tên file bảng độ dài: "); gets(fname);
if (NULL==(lfp = fopen(fname, "wb"))) {
    printf("Không thể mở file %s", fname); exit(1);
}
len_index=0; add_index = 0; ic = 0;;
}

void finalise_vimcode(void)
{
    emit(halt);
    fwrite(&address_table, add_index*sizeof(int), 1, afp);
    fwrite(&length_table, len_index* sizeof(int), 1, lfp);
    fclose(cfp); fclose(afp); fclose(lfp);
}

void emit_crseg(int level, int index)
{
    ic++;
    instruction.code = crseg; instruction.u.u2.sn2 = level;
    instruction.u.u2.index2 = index;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
}

void emit(operation code)
{
    ic++;
    instruction.code = code;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
}

void emit_jump(operation code, int index)
{
    ic++;
    instruction.code = code; instruction.u.index1 = index;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
}

```

```

void enter_length(int index, int length)
{
    length_table[index] = length;
}

int get_index(void)
{
    len_index++;
    return (len_index-1);
}

int get_label(void)
{
    add_index++;
    return (add_index-1);
}

void emit_label(int index)
{
    address_table[index] = ic;
}

void emit_ldvar(int sn, int dpl)
{
    ic++;
    instruction.code = ldvar; instruction.u.ul.sn1 = sn;
    instruction.u.ul.dpl = dpl;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
}

void emit_stvar(int sn, int dpl)
{
    ic++;
    instruction.code = stvar; instruction.u.ul.sn1 = sn;
    instruction.u.ul.dpl = dpl;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
}

void emit_ldcon(int value)
{
    ic++;
    instruction.code = ldcon; instruction.u.value = value;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
}

```

2. Kiểm thử chương trình

Chương trình dịch ở trên đã là một chương trình làm việc được rồi. Bây giờ bạn hãy thử nó bằng cách viết các chương trình nguồn trong ngôn ngữ SLANG từ đơn giản đến phức tạp, dịch nó bằng chương trình trên (chú ý là phải nhập những 4 tên file khác nhau) và mang kết quả sang chạy thử bằng trình mô phỏng máy tính ảo VIM.

Nếu chương trình dịch và VIM hoạt động tốt, nó phải thực hiện được chương trình SLANG do bạn viết (bạn hãy theo dõi kết quả bằng lệnh WRITE).

a. Kiểm thử phần sinh mã

Sau đây là một cách trực quan nhằm kiểm tra chương trình có sinh mã đúng hay không. Ý tưởng chính của cách kiểm tra này là mỗi khi ghi một chỉ thị lệnh ra file bằng câu lệnh fwrite(&instruction,sizeof(instruction),1,cfp) thì ta cũng in luôn nội dung của lệnh đó ra màn hình. Việc in nội dung chỉ thị nhờ gọi thủ tục instruction_print. Phần sinh mã sẽ được sửa như sau:

```
void instruction_print(void);

void emit_crseg(int level, int index)
{
    ic++;
    instruction.code = crseg; instruction.u.u2.sn2 = level;
    instruction.u.u2.index2 = index;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
    ⇒ instruction_print();
}

void emit(operation code)
{
    ic++;
    instruction.code = code;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
    ⇒ instruction_print();
}

void emit_jump(operation code, int index)
{
    ic++;
    instruction.code = code; instruction.u.index1 = index;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
    ⇒ instruction_print();
}

void emit_ldvar(int sn, int dpl)
```

```

{
    ic++;
    instruction.code = ldvar; instruction.u.u1.sn1 = sn;
    instruction.u.u1.dpl = dpl;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
⇒     instruction_print();
}

void emit_stvar(int sn, int dpl)
{
    ic++;
    instruction.code = stvar; instruction.u.u1.sn1 = sn;
    instruction.u.u1.dpl = dpl;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
⇒     instruction_print();
}

void emit_ldcon(int value)
{
    ic++;
    instruction.code = ldcon; instruction.u.value = value;
    fwrite(&instruction, sizeof(instruction), 1, cfp);
⇒     instruction_print();
}

⇒
void instruction_print(void)
{
    static char *op_name[] = {"neg", "abs_ ", "add", "sub", "mul",
                             "dvi", "mdl", "eq", "ne", "lt", "le", "gt", "ge",
                             "ldcon", "ldvar", "stvar", "jump", "jift",
                             "jiff", "call", "crseg", "dlseg",      "return_",
                             "rdint", "wrint", "halt" };

    printf("  %-10s", op_name[instruction.code]);

    switch(instruction.code) {
        case ldcon:
            printf("%d", instruction.u.value);
            break;
        case ldvar:
        case stvar:
            printf("%d, %d", instruction.u.u1.sn1,
                   instruction.u.u1.dpl);
            break;
        case jump:
        case jift:
        case jiff:
        case call:
            printf("%d", instruction.u.index1);
    }
}

```

```

        break;
    case crseg:
        printf("%d, %d", instruction.u.u2.sn2,
               instruction.u.u2.index2);
        break;
    }
    printf("\n");
}

```

Bạn hãy gọi chương trình dịch trên với ví dụ 2.1 và nhập các tên file đích khác, kết quả hiện màn hình sẽ phải như sau (các con số do chúng tôi thêm vào để tiện trình bày):

```

Nhap vao ten chuong trinh nguon: VIDU21.PRO
Nhap ten file ra cua VIM : VIDU21.VIM
Nhap ten file bang dia chi: VIDU21.ADS
Nhap ten file bang do dai : VIDU21.LEN
1.      crseg   1, 0
2.      ldcon   0
3.      stvar   1, 1
4.      rdint
5.      stvar   1, 0
6.      ldvar   1, 0
7.      ldcon   0
8.      ne
9.      jiff   1
10.     ldvar  1, 1
11.     ldvar  1, 0
12.     add
13.     stvar  1, 1
14.     rdint
15.     stvar  1, 0
16.     jump
17.     ldvar  1, 1
18.     wrint
19.     dlseg
20.     halt

```

Lệnh thứ nhất cho biết máy tính ảo VIM ban đầu phải cấp phát một phân đoạn chính trong bộ nhớ, với số phân đoạn là 1 và độ dài vùng nhớ này chứa ở ô 0 trong bảng độ dài (length_table). Lệnh thứ 2 và 3 cho biết, máy VIM sẽ nạp hằng số 0 lên ngăn xếp rồi cát vào trong bộ nhớ tại phân đoạn 1 (phân đoạn chính) và chuyển dịch 1. Đến đây, máy VIM đã thực hiện xong lệnh sum:= 0 của VIDU21.PRO. Lệnh 4 và 5 sẽ đọc một số nguyên từ bàn phím vào và lưu nó tại phân đoạn 1 và chuyển dịch 0. Đó chính là mã của câu lệnh read (number) của chương trình SLANG. Các lệnh 6, 7, 8 bao gồm

việc nạp biến tại phân đoạn 1 chuyển dịch 0 (biến number), rồi nạp hằng số 0 lên ngăn xếp và so sánh khác (ne) với nhau. Căn cứ vào kết quả so sánh đặt trên ngăn xếp, lệnh nhảy có điều kiện ở dòng 9 sẽ quyết định chương trình thoát khỏi vòng lặp while number \neq 0 hay chưa (dòng 6, 7, 8 là mã của câu lệnh while)...

Cứ thế, bạn có thể kiểm tra tính đúng đắn khi sinh mã và biết sửa đúng chỗ cần thiết nếu có trực trặc xảy ra.

b. Kiểm thử máy ảo VIM

Bây giờ, bạn hãy chạy máy ảo VIM, nhập vào 3 tên file vừa được tạo ra từ chương trình dịch SLC. Nếu chương trình VIM hoạt động bình thường, nó phải cho phép bạn nhập vào một số số nguyên cho đến khi bạn nhập vào số không rồi in ra kết quả tổng các số nguyên đó. Ví dụ sau là một lần hoạt động của chương trình:

Nhap ten chuong trinh VIM: VIDU21.VIM

Nhap ten bang dia chi: VIDU21.ADS

Nhap ten bang do dai: VIDU21.LEN

Nhap vao mot so nguyen: 12

Nhap vao mot so nguyen: 8

Nhap vao mot so nguyen: 7

Nhap vao mot so nguyen: 25

Nhap vao mot so nguyen: 0

52

Bài tập lập trình

1. Giả sử bạn đã mở rộng ngôn ngữ SLANG cho các lệnh repeat, for và case. Hãy sửa chương trình để sinh mã cho những phần thêm này.

2. Cải tiến chương trình dịch và chương trình mô phỏng máy tính ảo sao cho chỉ cần dùng đúng một file đích (bạn cần thêm ít nhất một lần duyệt nữa để gộp các dữ liệu đích thành một file đích).

3. Trong chương 2, có một bài tập lập trình với yêu cầu đọc file và xử lý các chỉ thị lệnh theo đúng độ dài thực tế của nó. Bạn hãy sửa phần sinh mã ở trên sao cho tạo được các file đích thích hợp với cải tiến này.

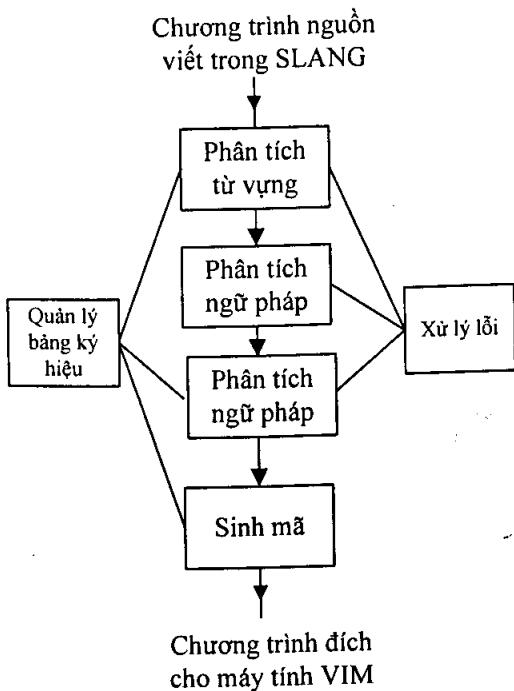
4. Hãy gộp chương trình trên và chương trình mô phỏng máy tính ảo VIM làm thành một chương trình dịch - chạy duy nhất (cái lợi của môi trường mới này là tổng thời gian dịch và chạy ngắn hơn nhiều, bạn không cần phải nhập các tên file đích...).

Chương 7

XỬ LÝ LỖI

I. MỤC ĐÍCH, NHIỆM VỤ

Nếu một chương trình dịch chỉ phải dịch các chương trình nguồn viết đúng thì thiết kế và hoạt động của nó sẽ rất đơn giản. Nhưng những người lập trình lại rất thường xuyên tạo ra những chương trình viết sai. Một chương trình dịch tốt phải phát hiện, định vị, phân loại được tất cả các lỗi để giúp đỡ người viết. Do vậy, việc thiết kế và thực hiện một chương trình dịch có phần xử lý lỗi tốt trở thành một thách thức lớn.



Hình 7.1. Vị trí của
phần xử lý lỗi trong
chương trình dịch
thực hành.

Phần lớn các ngôn ngữ lập trình không cho biết một chương trình dịch phải xử lý lỗi như thế nào. Người thiết kế chương trình dịch phải tự giải

quyết vấn đề này. Ta phải thiết kế các chương trình dịch xử lý lỗi một cách đúng đắn ngay từ khi bắt đầu hoạt động.

Về lý thuyết, lỗi có thể xuất hiện và phải xử lý trong mọi giai đoạn của một chương trình dịch. Nhưng ở đây, ta chỉ xét những nơi xử lý lỗi thông thường, do đó ta bỏ phần xử lý lỗi liên quan đến phần sinh mã.

Cũng như các phần khác, ta sẽ giới hạn mức độ phức tạp của phần xử lý lỗi, sao cho đủ phức tạp để ta có thể hiểu được một bộ xử lý lỗi thực sự phải như thế nào, nhưng cũng vừa đủ đơn giản để có thể thực hiện được.

II. XỬ LÝ LỖI TRONG PHẦN PHÂN TÍCH TỪ VỰNG

Chỉ có rất ít lỗi được phát hiện trong lúc phân tích từ vựng, vì bộ phân tích từ vựng chỉ quan sát chương trình nguồn một cách rất cục bộ. Các lỗi mà bộ phân tích từ vựng phát hiện được là các lỗi về từ vị không đúng. Khi gặp lỗi, cách xử lý đơn giản nhất là hệ thống sẽ ngừng hoạt động và báo lỗi cho người sử dụng. Cách xử lý tốt hơn và hiệu quả hơn là bộ phân tích từ vựng sẽ ghi lại các lỗi này và cố gắng khắc phục chúng (khôi phục sau khi gặp lỗi) để hệ thống có thể tiếp tục làm việc, nhằm phát hiện đồng thời thêm nhiều lỗi khác.

Tại mức phân tích từ vựng, lỗi được phát hiện rất dễ dàng, thế nhưng khôi phục lỗi vẫn rất khó do thông tin không dư thừa nhiều. Trong các chương trước, ta đã khai báo thêm một từ tố đặc biệt là từ tố lỗi - *err_token*, từ tố đó rất có ích trong phần này. Ta sẽ mở rộng bộ phân tích từ vựng tại hai nơi để phát hiện và khôi phục lỗi.

Mở rộng thứ nhất là mỗi khi đọc được một ký hiệu vào không mong muốn thì chương trình tạo ra một từ tố *err_token* và đọc một ký hiệu vào tiếp theo. Mở rộng thứ hai, nếu ta đã đọc được ký tự vào ":" (với ý nghĩa là bắt đầu của từ tố phép gán becomes _token), và nếu ta đọc được một ký tự nào đó khác với "=" thì *err_token* cũng được tạo ra.

Thủ tục *next_token* được mở rộng như sau:

```
void next_token(void)
{
    /* Những phần khác không trình bày ở đây */

    switch (ch) {
        /* Những ký hiệu đơn khác không trình bày ở đây */
        case ':':
            next_character();
            if (ch == '=') look_ahead = becomes_token;
    }
}
```

```

⇒     else {
⇒         look_ahead = err_token;
⇒         retract = true;
⇒     }
⇒     break;

case '<':
    next_character();
    if (ch == '>') look_ahead = not_equal_token;
    else if (ch == '=') look_ahead = less_or_equal_token;
    else {
        look_ahead = less_than_token;
        retract = true;
    }
    break;

case '>':
    next_character();
    if (ch == '=') look_ahead = greater_or_equal_token;
    else {
        look_ahead = greater_than_token;
        retract = true;
    }
    break;

default:
    if (isalpha(ch))
        identifier();
    else if (isdigit(ch))
        number();
    ⇒     else
    ⇒     look_ahead = err_token;
    }
    if (!retract) next_character();
}

```

Ta đã biết bộ đệm để chứa tên (ident_lexeme) chỉ có giới hạn nhất định và đã được định nghĩa trong hằng số max_ident_length. Tuy độ lớn này (255 ký tự) là đủ cho đa số trường hợp dùng bình thường và kể cả gấp lỗi, thì trong một số trường hợp thật đặc biệt bạn có thể gặp những cái tên dài quá giới hạn này. Cách giải quyết là coi những cái tên đó là lỗi (lỗi vượt quá giới hạn chương trình dịch). Chương trình vẫn tiếp tục đọc phần vượt quá này cho đến hết tên đó nhưng không lưu thêm vào bộ đệm này nữa. Bạn hãy tự sửa chương trình để làm nhiệm vụ này như một bài tập.

III. XỬ LÝ LỖI TRONG PHẦN PHÂN TÍCH CÚ PHÁP

1. Phát hiện lỗi

Giai đoạn phân tích cú pháp phát hiện và khắc phục được khá nhiều lỗi. Một trong các lý do là nhiều loại lỗi khác nhau đồng thời cũng là lỗi cú pháp. Một lý do khác là nhờ sự chính xác của các phương pháp phân tích. Việc phát hiện chính xác lỗi trong thời gian dịch là một nhiệm vụ khó khăn cần nhiều nỗ lực. Đáp ứng các yêu cầu trên là một thách thức lớn. May mắn là các lỗi thông thường cũng là các lỗi đơn giản và một cơ chế bắt lỗi không phức tạp lắm cũng thường đủ dùng.

Trong lúc phân tích, mỗi một ký hiệu kết thúc ở về phải một sản xuất phải khớp với ký hiệu nhìn trước. Nếu điều kiện này bị xâm phạm thì đó là một lỗi cú pháp. Nếu cần phải chọn giữa một số lựa chọn, và ký hiệu nhìn trước lại không trùng với một ký hiệu đầu tiên của một lựa chọn nào thì không thể thực hiện được việc chọn, đó lại là một lỗi nữa.

Trong chương 4 ta đã đưa ra bảng luật để viết chương trình phân tích cú pháp. Trong phần này ta sẽ đưa ra một bảng mới mở rộng nhằm phát hiện lỗi nhưng chưa có phần khôi phục lỗi. Chú ý là luật 1 và 3 xác định các lỗi cú pháp.

TT	<i>Biểu thức chính quy</i>	<i>Module phân tích</i>
10	α (ký hiệu kết thúc)	<code>if (look_ahead == α) next_token(); else report_error("Thiếu từ tố α");</code>
11	A (ký hiệu không kết thúc)	$A();$ (nghĩa là gọi thủ tục ứng với A)
12	$E_1 E_2 \dots E_n \ (n > 1)$	<code>if (look_ahead ∈ DIRSET(E_1)) { $\Pi(E_1)$ } else if (look_ahead ∈ DIRSET(E_2)) { $\Pi(E_2)$ } else if (look_ahead ∈ DIRSET(E_n)) { $\Pi(E_n)$ } else report_error("Lựa chọn không đúng");</code>
13	$E_1 E_2 \dots E_n$	$\Pi(E_1) \Pi(E_2) \dots \Pi(E_n)$
14	E^*	<code>while ((look_ahead ∈ FIRST(E)) { $\Pi(E)$ }</code>
15	E	$\Pi(E)$
16	$[E]$	<code>if ((look_ahead ∈ FIRST(E)) { $\Pi(E)$ }</code>

17	E^+	$\Pi(E)$ While ((look_ahead \in FIRST(E)) { $\Pi(E)$ })
18	$E_1 (E_2 E_1)^*$	$\Pi(E_1)$ while ((look_ahead \in FIRST(E_2)) { $\Pi(E_2) \Pi(E_1)$ })

2. Khôi phục lỗi

Bây giờ ta tiếp tục xem xét mở rộng chương trình để khôi phục lỗi theo chiến lược “khôi phục cụm từ”. Nếu một lỗi được xác định, thì một từ tố thiếu sẽ được chèn thêm vào; hoặc một từ tố không đúng sẽ được xoá đi. Nhưng làm sao ta biết là một ký hiệu bị thiếu hay không đúng?

Khi phân tích đầu vào, bộ phân tích sẽ tiến hành dựa theo về phái của các sản xuất. Tại một thời điểm bất kì, bộ phân tích đang làm việc với một sản xuất nào đó, và có một số sản xuất còn đang làm việc dở (chúng là cha, ông của sản xuất hiện thời). Để khôi phục lỗi, bộ phân tích cần ba tập từ tố gọi là các tập từ tố đồng bộ:

- **Tập đồng bộ tức thời** (ký hiệu là immediate _markers): Tập các từ tố sẽ được triển khai tiếp.
- **Tập đồng bộ cục bộ** (ký hiệu là local _markers): Tập các từ tố sẽ được triển khai trong phần còn lại của sản xuất đang dùng phân tích (trừ các tập đồng bộ trung gian).
- **Tập đồng bộ toàn cục** (ký hiệu là global _markers): Tập các từ tố mà sẽ được triển khai trong phần còn lại của tất cả các sản xuất đang làm việc dở (trừ các tập đồng bộ cục bộ mà các tập đồng bộ trong sản xuất ta đang làm việc với nó).

Ta minh họa các tập đồng bộ trong ví dụ sau.

Ví dụ 7.1: Xét chương trình sau. Đúng trên quan điểm khôi phục lỗi, ta thấy câu lệnh if theo ý muốn của người lập trình là đặt trong câu lệnh while và câu lệnh này lại đặt trong khối của chương trình.

```

BEGIN VAR INT a, b, c;
    a := 1; b := 2;
    WHILE a < b DO
        IF a < b THEN a := b + 2 ELSE b := a - 2 FI; c := a
    OD
END.

```

Giả sử là bộ phân tích đang phân tích biểu thức quan hệ của câu lệnh if, nghĩa là từ tố *if_token* đã được nhận ra và quan hệ của nó đang được phân tích. Tại thời điểm này, các thủ tục cú pháp đã được gọi đệ quy và còn chưa kết thúc (đang làm việc đó) là: *program_declaration*, *block*, *statement_part*, *statement*, *while_statement*, *statement_part*, *statement* và thủ tục trên cùng đang làm việc là *if_statement*. Trong bảng dưới đây, ta thấy các thủ tục cú pháp và các tập đồng bộ tương ứng. Chú ý là các sản xuất được liệt kê theo thứ tự ngược với thứ tự gọi.

Tập đồng bộ tức thời

relation: DIRSET(*relation*)

Tập đồng bộ cục bộ

if_statement: {*then_token*} \cup DIRSET(*statement_part*) \cup
DIRSET([*else_token statement_part*]) \cup {*fi_token*}

Tập đồng bộ toàn cục

statement: ϵ

statement_part: DIRSET((*separator_token statement*)*)

while_statement: {*od_token*}

statement: ϵ

statement_part: DIRSET((*separator_token statement*)*)

block: {*end_token*}

program_declaration: {*period_token*}

Thực hiện các phép tính trên với SLANG ta có:

Tập đồng bộ tức thời: {*ident_token*, *num_token*, *open_token*, *absolute_token*,
negate_token}

Tập đồng bộ cục bộ: {*then_token*, *else_token*, *fi_token*, *ident_token*,
if_token, *while_token*, *call_token*, *read_token*,
write_token}

Tập đồng bộ toàn cục: {*separator_token*, *od_token*, *end_token*, *period_token*}

Thủ tục *if_statement* bây giờ sẽ gọi thủ tục *relation*. Các tập đồng bộ tức thời và cục bộ được thực hiện từ vé phải của sản xuất *relation*, và tập đồng bộ toàn cục (tập mới) là sự kết hợp của các tập đồng bộ cũ và toàn

cục. Các từ tố trong phần còn lại của về phải sản xuất *relation* do vậy được thêm vào tập tổng. Tổng quát, tập tổng của các tập đồng bộ sẽ tăng trưởng mỗi lần ta gặp một ký hiệu không kết thúc và bắt đầu xem xét về phải của sản xuất đó.

Khi phân tích xong biểu thức quan hệ, bộ phân tích trở về thủ tục *if_statement*. Từ tố *then _token* sẽ phải là từ tố được phân tích tiếp theo. Ngay trước khi phân tích từ tố *then _token*, ba tập đồng bộ là:

Tập đồng bộ tức thời: { *then_token* }

Tập đồng bộ cục bộ: { *else_token*, *fi_token*, *ident_token*, *if_token*,
while_token, *call_token*, *read_token*, *write_token* }

Tập đồng bộ toàn cục: { *separator_token*, *od_token*, *end_token*, *period_token* }

Sau khi phân tích xong từ tố *then _token*, thì các từ tố trong DIRSET (*statement_part*) trở thành tập đồng bộ tức thời, và các từ tố đó được xoá khỏi tập đồng bộ cục bộ. Tập đồng bộ toàn cục vẫn được giữ như cũ. Do đó, từ tố *then _token* được loại khỏi tập đồng bộ tổng. Mỗi khi một phần của một sản xuất được phân tích xong, tập từ tố đầu tiên của nó phải được loại bỏ khỏi tập đồng bộ tổng. Do vậy, tập tổng co dần mỗi khi một phần của một về phải được phân tích.

Chiến lược khôi phục lỗi như sau: nếu ký hiệu nhìn trước không được mong đợi, nhưng lại có trong các tập đồng bộ hiện thời, thì bộ phân tích coi là một từ tố bị mất, nó chèn thêm từ tố này vào và tiếp tục phân tích. Nếu như từ tố này không có trong các tập đồng bộ, thì bộ phân tích coi ký hiệu nhìn trước này là thừa và xoá nó đi, sau đó lại tiếp tục phân tích. Ta sẽ minh họa bằng hai ví dụ dưới đây.

Ví dụ 7.2: Giả sử từ tố *then _token* bị mất trong ví dụ trên, nghĩa là câu lệnh phân tích chỉ là:

IF a < b a := b+2 ELSE b := a-2 FI; c := a

Ký hiệu nhìn trước (nghĩa là *ident_token* ứng với về trái của câu lệnh gán thứ nhất) có trong các tập đồng bộ cục bộ. Do vậy, bộ phân tích coi là từ tố *then _token* bị mất, nó sẽ chèn thêm từ tố này vào và in ra một thông báo: “Lỗi: chèn thêm *then _token*” và làm việc tiếp tục.

Ví dụ 7.3: Trong ví dụ này, giả sử từ tố *then _token* được thay bằng *_token*, nghĩa là câu lệnh trở thành:

IF a < b DO a := b+2 ELSE b := a-2 FI; c := a

Ký hiệu nhìn trước lúc này là từ tố do `_token` sẽ không có trong các tập đồng bộ. Do đó, bộ phân tích coi từ tố do `_token` là dư thừa nên sẽ xoá nó đi, và in ra thông báo lỗi “Lỗi: xoá do `_token`”. Tiếp theo, bộ phân tích kết luận là từ tố `then _token` bị thiếu và xử lý giống như ví dụ trên.

Để xoá các từ tố dư thừa, ta đưa ra một thủ tục mới `delete_until` (xoá cho đến) với một tham số `markers` có kiểu là `token_set`, tham số này chính là sự kết hợp của các tập đồng bộ tức thời, địa phương và toàn cục. Các từ tố bị xoá khỏi đầu vào cho đến khi ký hiệu nhìn trước có ở một trong các tập đồng bộ đó.

Các luật mới để xây dựng bộ phân tích có phục hồi lỗi được cho trong bảng dưới đây. Chú ý rằng, trong bảng này tập đồng bộ tức thời chính là biểu diễn của các tập FIRST, FOLLOW và DIRSET. Ví dụ, tập đồng bộ tức thời trong trường hợp E^* là $\text{FIRST}(E) \cup \text{FOLLOW}(E)$.

<i>TT</i>	<i>Biểu thức chính quy</i>	<i>Module phân tích</i>
1	a (ký hiệu kết thúc)	<code>delete_until ([a] ∪ local_markers ∪ global_markers);</code> <code>if (look_ahead == a) next_token();</code> <code>else report_error ("Thiếu từ tố a");</code>
2	A (ký hiệu không kết thúc)	$A();$ (nghĩa là gọi thủ tục ứng với A)
3	$E_1 E_2 \dots E_n \ (n > 1)$	<code>delete_until (FIRST(E_1) ∪ FIRST(E_2) ∪ ... ∪ FIRST(E_n) ∪ global_markers);</code> <code>if (look_ahead ∈ DIRSET(E_1)) {</code> <code> Π (E_1)</code> <code>} else if (look_ahead ∈ DIRSET(E_2)) {</code> <code> Π (E_2)</code> <code>....</code> <code>} else if (look_ahead ∈ DIRSET(E_n)) {</code> <code> Π (E_n)</code> <code>} else report_error ("Lựa chọn không đúng");</code>
4	$E_1 E_2 \dots E_n$	$\Pi (E_1) \Pi (E_2) \dots \Pi (E_n)$
5	E^*	<code>delete_until (FIRST(E) ∪ FOLLOW(E) ∪ local_markers ∪ global_markers);</code> <code>while ((look_ahead ∈ FIRST(E)) { Π (E) }</code>
6	E	$\Pi (E)$
7	$[E]$	<code>delete_until (FIRST(E) ∪ FOLLOW(E) ∪ local_markers ∪ global_markers);</code>

		if ((look_ahead ∈ FIRST(E)) { $\Pi(E)$ }
8	E^+	$\Pi(E)$ while ((look_ahead ∈ FIRST(E)) { $\Pi(E)$ })
9	$E_1 (E_2 E_1)^*$	$\Pi(E_1)$ delete_until (FIRST(E_1) ∪ FIRST(E_2) ∪ FOLLOW(E_1) ∪ local_markers ∪ global_markers); while ((look_ahead ∈ FIRST(E_2)) { $\Pi(E_2) \Pi(E_1)$ delete_until (FIRST(E_1) ∪ FIRST(E_2) ∪ FOLLOW(E_1) ∪ local_markers ∪ global_markers); }

Việc khôi phục lỗi trong trường hợp của $\Pi(a)$ đòi hỏi phải xoá các từ tố dư thừa và tiếp theo nếu từ tố a không phải là ký hiệu nhìn trước thì chèn thêm từ tố này vào. Trong trường hợp của $\Pi(E_1 | E_2 \dots | E_n)$, việc khôi phục lỗi bao gồm xoá từ tố dư thừa và sau đó tìm một lựa chọn. Nếu không tìm được một lựa chọn nào thì bộ phân tích coi như có một lựa chọn được chèn vào và được phân tích. Khôi phục lỗi trong các trường hợp E^* , $[E]$, và E^+ đòi hỏi xoá các từ tố thừa, và xem xét tiếp biểu thức con.

Để minh họa lược đồ phát hiện và khôi phục lỗi, ta mở rộng các thủ tục phân tích *statement*, *if_statement* và *relation* như sau:

```

⇒ void statement(token_set global_markers)
{
⇒     token_set immediate_markers, local_markers, all_markers,
          local_global_markers;

⇒     immediate_markers = create_set(ident_token, if_token,
                                      while_token, call_token,
                                      read_token, write_token, -1);

⇒     local_markers = create_set(-1);
⇒     all_markers = joint_sets(immediate_markers, local_markers,
                                global_markers, NULL);

⇒     delete_until(all_marker);
⇒     local_and_global_markers = joint_sets(local_markers,
                                              global_markers, NULL);

⇒     if(look_ahead == ident_token) {
⇒         assignment_statement(local_and_global_markers);
⇒     } else if(look_ahead == if_token) {
⇒         if_statement(local_and_global_markers);
⇒     } else if(look_ahead == while_token) {

```

```

⇒           while_statement(local_and_global_markers);
⇒ } else if(look_ahead == call_token) {
⇒         call_statement(local_and_global_markers);
⇒ } else if(look_ahead == read_token) {
⇒         read_statement(local_and_global_markers);
⇒ } else if(look_ahead == write_token) {
⇒         write_statement(local_and_global_markers);
⇒ } else report_error("Lựa chọn được chèn vào");
}

⇒void if_statement(token_set global_markers)
{
    int over_then_part, over_else_part;
⇒ token_set immediate_markers, local_markers, all_markers,
    local_global_markers;

⇒ immediate_markers = create_set(if_token, -1);
⇒ local_markers = create_set(ident_token, num_token, open_token,
                           absolute_token, negate_token,
                           then_token, else_token,
                           fi_token, if_token, while_token,
                           call_token, read_token,
                           write_token, -1);
⇒ all_markers = joint_sets(immediate_markers, local_markers,
                           global_markers, NULL);
⇒ delete_until(all_marker);

    if(look_ahead == if_token) next_token();
    ⇒ else report_error("Chèn thêm if_token");
    ⇒ local_markers = create_set(      then_token, else_token,
                                   fi_token, ident_token, if_token,
                                   while_token, call_token,
                                   read_token, write_token, -1);
⇒ local_and_global_markers = joint_sets(local_markers,
                                         global_markers, NULL);

⇒ relation(local_and_global_markers);
⇒ immediate_markers = create_set(then_token, -1);
⇒ local_markers = create_set(else_token, fi_token, ident_token,
                           if_token, while_token, call_token,
                           read_token, write_token, -1);
⇒ all_markers = joint_sets(immediate_markers, local_markers,
                           global_markers, NULL);
⇒ delete_until(all_markers);

    if(look_ahead == then_token) next_token();
    over_then_part = get_label();
    emit_jump(jiff, over_then_part);
    ⇒ local_markers = create_set(else_token, fi_token, -1);
}

```

```

⇒ local_and_global_markers = joint_sets(local_markers,
                                         global_markers, NULL);

⇒ statement_part(local_and_global_markers);
⇒ immediate_markers = create_set(else_token, fi_token, -1);
⇒ local_markers = create_set(fi_token, -1);
⇒ all_markers = joint_sets(immediate_markers, local_markers,
                           global_markers, NULL);
⇒ delete_until(all_markers);

⇒ if (look_ahead == else_token) {
    over_else_part = get_label();
    emit_jump(jump, over_else_part);
    emit_label(over_then_part);

⇒ immediate_markers = create_set(else_token, -1);
⇒ local_markers = create_set(fi_token, ident_token,
                           if_token, while_token, call_token,
                           read_token, write_token, -1);
⇒ all_markers = joint_sets(immediate_markers,
                           local_markers, global_markers, NULL);
⇒ delete_until(all_markers);

⇒ if (look_ahead == else_token) next_token();
⇒ else report_error("Chèn thêm else_token");

⇒ local_markers = create_set(fi_token, -1);
⇒ local_and_global_markers = joint_sets(local_markers,
                                         global_markers, NULL);

⇒ statement_part(local_and_global_markers);
⇒ emit_label(over_else_part);
} else
⇒     emit_label(over_then_part);

⇒ immediate_markers = create_set(fi_token, -1);
⇒ local_markers = create_set(-1);
⇒ all_markers = joint_sets(immediate_markers, local_markers,
                           global_markers, NULL);
⇒ delete_until(all_markers);

⇒ if(look_ahead == fi_token) next_token();
⇒ else report_error("Chèn thêm fi_token");
}

void relation(token_set global_markers)
{

```

```

    rel_op op;
⇒      token_set      local_markers, local_global_markers;

⇒      local_markers = create_set(equal_token, not_equal_token,
                                less_than_token, less_or_equal_token,
                                greater_than_token,
                                greater_or_equal_token, ident_token,
                                num_token, open_token, absolute_token,
                                negate_token, -1);
⇒      local_and_global_markers = joint_sets(local_markers,
                                              global_markers, NULL);

      expression(local_and_global_markers);

⇒      local_markers = create_set(ident_token, num_token, open_token,
                                absolute_token, negate_token, -1);
⇒      local_and_global_markers = joint_sets(local_markers,
                                              global_markers, NULL);

      relational_operator(&op, local_and_global_markers);
⇒      local_markers = create_set(-1);
⇒      local_and_global_markers = joint_sets(local_markers,
                                              global_markers, NULL);

      expression(local_and_global_markers);
      if (op == equal) emit(eq);
      else if (op == not_equal) emit(ne);
      else if (op == less_than) emit(lt);
      else if (op == less_or_equal) emit(le);
      else if (op == greater_than) emit(gt);
      else if (op == greater_or_equal) emit(ge);
}

```

Chú ý: Có hai hàm *create_set()* và *join_sets()* không trình bày ở đây. Hàm *create_set()* tạo một tập từ tố từ các từ tố cho trong tham số, kết thúc bằng -1. Hàm *join_sets()* nhập các tập từ tố kết thúc bằng con trỏ NULL thành một tập từ tố mới. Bạn hãy tự mình thiết kế và thực hiện các hàm này (có thể sửa lại cách truyền tham số sao cho dễ thực hiện nhất). Bạn hãy tự kiểm tra và sửa các thủ tục khác như một bài tập.

Chú ý rằng, do ta áp dụng một cách chân phương các luật trong bảng vào mở rộng các thủ tục phân tích nên một số cấu trúc bị thừa. Ví dụ như thủ tục *if_statement* có nhiều lệnh không tối ưu.

Phương pháp tìm và phục hồi lỗi đơn giản như trên làm việc rất tốt trong nhiều trường hợp. Tuy nhiên, cũng giống như mọi phương pháp khác, trong một số trường hợp nó làm việc không được tốt lắm. Ta minh họa điều này qua một ví dụ sau:

Ví dụ 7.4: Xem xét câu lệnh điều kiện sau:

IF a < b < c THEN a := b+2 ELSE b := a-2 FI; c := a

câu lệnh này lại được đặt trong vòng lặp while như ví dụ 7.1 và câu lệnh while lại đặt trong thân của chương trình. Khi bộ phân tích tìm thấy dấu nhỏ hơn “<” thứ hai thì nó biết rằng đó là một lỗi và thông báo “Lỗi: Xoá <”. Khi đến từ tố tiếp theo - từ tố tên ident _token (của biến c) thì có trong tập đồng bộ cục bộ. Từ tố này có thể là phần đầu của phần câu lệnh trong câu lệnh if. Điều này dĩ nhiên là không đúng và thế là bộ phân tích sẽ in ra các thông báo không tốt. Đầu tiên, nó đưa ra thông báo “Lỗi: chèn thêm then _token”. Bộ phân tích coi rằng nó tiếp tục phân tích các lệnh của phần câu lệnh của lệnh if. Nó gặp tên c và coi đó là về trái của một lệnh gán và thế là khi gặp từ khoá “then” nó sẽ coi là một lỗi nữa và in ra “Lỗi: Xoá từ tố then _token” và “Lỗi: chèn becomes _token”. Bạn có thể tự tìm hiểu tiếp điều gì sẽ xảy ra khi chương trình phân tích tiếp đến từ khoá ELSE.

Phương pháp giải quyết vấn đề này là dùng một cái sàng, để ngăn lại những từ khoá dĩ nhiên trong thành phần của các tập đồng bộ. Một ứng cử viên để sàng ngăn lại là ident _token. Chú ý là ví dụ cuối của chúng ta làm việc tốt nếu như ident _token không có trong các tập đồng bộ. Phần này cũng được ra như một bài tập.

IV. XỬ LÝ LỖI TRONG PHẦN PHÂN TÍCH NGỮ NGHĨA

Trong chương 5 ta đã học về phân tích ngữ nghĩa. Thực chất khói này làm nhiệm vụ phát hiện và xử lý lỗi ngữ nghĩa. Thông thường, nó chỉ cần gọi phần xử lý lỗi ở đây để giao tiếp với người dùng như thông báo loại lỗi, vị trí lỗi...

Trong giáo trình của ta, giao diện với người dùng được giữ đơn giản nhất chính là thủ tục report _error mà nhiệm vụ chỉ là thông báo lỗi. Do đó, ta không cần phải để cập thêm về phần này nữa.

Bài tập lập trình

1. Hoàn chỉnh mọi phần xử lý lỗi.
2. Bạn hãy tự kiểm tra và khẳng định các phần báo lỗi hoạt động tốt.

Chương 8

MỞ RỘNG NGÔN NGỮ NGUỒN VÀ MÁY TÍNH ẢO

I. MỤC ĐÍCH

Trong chương này, ta sẽ thêm mảng và tham số thủ tục cho SLANG. Khác với các ngôn ngữ như Pascal, C, mảng trong SLANG (tổng quát hơn) là mảng *động*, với nghĩa các kích thước của nó (được xác định bằng các cặp giới hạn trên, dưới - còn gọi là các cặp buộc) là các biểu thức của các biến hay tham số thủ tục. Các kích thước của một mảng sẽ được tính khi khai báo của mảng được cho chạy, nghĩa là sau khi vào thủ tục có khai báo mảng đó.

Các thủ tục lại có thể truyền tham số ở dạng biến hoặc giá trị (gọi tắt là tham biến và tham trị) như nhiều ngôn ngữ lập trình thông thường cho phép.

Trong các phần sau ta sẽ định nghĩa lại các ký hiệu, luật cú pháp của SLANG. Việc mở rộng ngôn ngữ nguồn đòi hỏi phải sửa đổi lại cách tổ chức bộ nhớ, tập chỉ thị lệnh và trình thông dịch của máy tính ảo VIM (về nguyên tắc ta có thể sinh mã cho máy tính VIM nguyên bản trong chương 2 để nó thực hiện được những mở rộng SLANG, nhưng việc sửa đổi sẽ làm việc sinh mã dễ dàng rất nhiều).

Ta cũng sẽ xem xét xác định lại nhiệm vụ của các thủ tục và hàm khác nhau trong chương trình dịch SLC nhằm thực hiện được những cải tiến mới. Việc sửa đổi chi tiết từng thủ tục /hàm sẽ được giao cho bạn đọc như một bài tập.

II. MỞ RỘNG NGÔN NGỮ NGUỒN SLANG (bước 2)

1. Mở rộng từ tố

Các cấu trúc của SLANG liên quan đến mảng và tham số của thủ tục cần phải được định nghĩa lại. Các từ tố cần cho tham số đã có sẵn. Đối với mảng, ta cần thêm ba ký hiệu mới nữa, đó là ngoặc vuông trái và phải (*left_bracket_token*, *right_bracket_token*) và cho - đến (*up_to_token*) như bảng sau:

Bảng các ký hiệu đặc biệt (cần cho mảng)

<i>left_bracket_token</i>	[
<i>right_bracket_token</i>]
<i>up_to_token</i>	..

2. Mở rộng các luật cú pháp

Bây giờ, ta sẽ xem xét các cấu trúc ngôn ngữ liên quan đến mảng và tham số thủ tục. Có một số luật sản xuất như vậy. Cũng như chương 2, đối với mỗi luật sản xuất ta sẽ nêu các điều kiện ràng buộc và ngữ nghĩa. Chú ý là các phần không liên quan sẽ không cần phải đề cập lại ở đây.

Các khai báo biến (*variable_declaration*, *type_declarer*, *bound_part*, *bound_pair*, *lower_bound*, *upper_bound*)

khaibáobiến → **var_token** *khaibáokiểu*

ident_token [*phânbuộc*]

(*list_token* **ident_token** [*phânbuộc*])*

separator_token

khaibáokiểu → **int_token**

phânbuộc → **left_bracket_token**

căpbuộc (*list_token* *căpbuộc*)*

right_bracket_token

căpbuộc → *buộcdưới up_to_token buộctrên*

buộcdưới → *biéuthúc*

buộctrên → *biéuthúc*

Ràng buộc: Mỗi tên trong khai báo biến phải ứng với một biến. Nó là một biến đơn giản nếu *phần buộc* của nó rỗng, là một biến mảng nếu ngược lại. Phạm vi của một tên là trong khối có khai báo của nó. Một tên không được định nghĩa quá một lần trong cùng một khối.

Không giống một biểu thức thông thường, một biểu thức buộc (nghĩa là một biểu thức dùng làm chỉ số dưới hoặc trên) có ràng buộc chặt chẽ: mọi tên dùng trong biểu thức phải là một biến hoặc một hằng số đã được khai báo ở quanh khối này.

Ngữ nghĩa: Đối với mỗi tên trong danh sách, một vị trí mới (gọi là *loc*) được tạo ra để lưu nó và *loc* được buộc với tên này.

Đối với mỗi tên trong khai báo biến mảng, phải thực hiện ba bước sau:

1. *n* là số các cặp trong phần buộc. Đối với mỗi cặp buộc, tính biểu thức buộc dưới và biểu thức buộc trên. Giả sử các giá trị của các cặp này là $(lo_1, up_1), (lo_2, up_2), \dots, (lo_n, up_n)$.
2. Kiểm tra quan hệ $lo_i \leq up_i$ đối với mỗi cặp (lo_i, up_i) ($1 \leq i \leq n$).
3. Cấp phát bộ nhớ một lượng $m = (up_1 - lo_1 + 1) * (up_2 - lo_2 + 1) * \dots * (up_n - lo_n + 1)$ bắt đầu tại địa chỉ *loc*. *loc* và *m* được buộc vào tên.

Đối với mỗi cặp buộc, giá trị buộc dưới phải nhỏ hơn hoặc bằng giá trị buộc trên.

Các khai báo thủ tục (procedure_declaration, parameter_part, val_parameter_part, ref_parameter_part)

khaibáothutục → procedure_token ident_token open_token
phầnthamsó (separator_token phầnthamsó)*
close_token khối separator_token
phầnthamsó → phầnthamtri | phầnthambién
phầnthamtri → khaibáokiểu ident_token (list_token ident_token)*
phầnthambién → var_token
khaibáokiểu ident_token (list_token ident_token)*

Ràng buộc: Mỗi tên trong khai báo thủ tục (ta sẽ gọi gọn là *tên thủ tục*) phải ứng với một thủ tục. Phạm vi của một tên là khối có khai báo có nó trong đó. Một tên không được định nghĩa quá một lần trong cùng một khối.

Mỗi tên trong phần tham trị (ký hiệu là *val_parameter_part*) ký hiệu cho một tham số là giá trị. Mỗi tên trong phần tham biến (ký hiệu là

ref_parameter_part) ký hiệu cho một tham số truy cập qua địa chỉ. Một tham trị biểu diễn một giá trị (giống như hằng số), với nghĩa là các tham số này có thể dùng nhưng không gán được. Một tham biến biểu diễn một biến đơn giản hoặc một biến chỉ số, có thể dùng giá trị của nó và gán cho nó được.

Một tên không được định nghĩa quá một lần trong phần khai báo tham số và trong khối của cùng thủ tục.

Ngữ nghĩa: Một khối được buộc vào một tên thủ tục.

Các câu lệnh gán (*assignment_statement*, *left_part*, *index_part*, *index*)

câu lệnh gán → *phản trái becomes_token* *biểu thức*

phản trái → *ident_token* [*phản chỉ số*]

phản chỉ số → *left_bracket_token* *chỉ số* (*list_token* *chỉ số*)*

right_bracket_token

chỉ số → *biểu thức*

Ràng buộc: Câu lệnh gán phải xuất hiện trong phạm vi hoạt động của tên trong về trái và tên này phải là một biến.

Nếu phần chỉ số của tên này không rỗng thì tên phải là một biến mảng, nếu không thì tên phải là một biến đơn hoặc một tham biến. Số lượng chỉ số trong phần chỉ số của một biến chỉ số (một mảng và chỉ số của nó tạo thành cái gọi là biến chỉ số) phải bằng số cặp buộc trong phần khai báo của biến mảng tương ứng.

Ngữ nghĩa: Phải thực hiện hai bước sau:

1. Xác định đích của câu lệnh gán. Trong trường hợp tên là một biến đơn, đích là vị trí buộc với tên. Đối với tên là một biến mảng, các biểu thức trong phần chỉ số được tính. Các giá trị của chúng cùng với tên xác định vị trí của biến chỉ số. Đối với tên là tham biến, đích là vị trí của tham số.
2. Biểu thức được tính và giá trị được ghi vào vị trí đích xác định trong bước 1.

Các câu lệnh call (*call_statement*, *argument*)

câu lệnh call → *call_token* *ident_token*

[*open_token* *thông số* (*list_token* *thông số*)*
close_token]

thôngsố → **ident_token** [*phânchisó*] | *bieu thuc*

Ràng buộc: Câu lệnh call phải xuất hiện trong phạm vi hoạt động của tên và tên này phải là tên của một thủ tục. Lời gọi cũng phải xuất hiện sau khi đã khai báo tên và thủ tục đó.

Câu lệnh call có thể chứa một danh sách các tham số và có thể không (danh sách rỗng). Khai báo thủ tục và thủ tục trong lời gọi phải có cùng số tham số. Trong trường hợp tham trị, tham số trong lời gọi phải là một biểu thức (có thể là một con số, hằng, biến đơn, biến chỉ số). Trong trường hợp tham biến, tham số trong lời gọi phải là một biến đơn, biến chỉ số hoặc một tham biến khác từ thủ tục bao quanh nó.

Câu lệnh call chỉ được xuất hiện trong phạm vi của mọi tên trong danh sách tham số. Số chỉ số trong phần chỉ số của một biến chỉ số phải bằng số cặp buộc trong khai báo biến mảng tương ứng.

Ngữ nghĩa: Mọi tham trị biểu diễn một giá trị. Giá trị của một tham số biểu thức được buộc vào tham số. Giá trị này được xác định khi thủ tục được gọi.

Mọi tham biến biểu diễn cho một biến đơn hoặc một biến chỉ số. Vị trí buộc trong khai báo sẽ được buộc với tham số. Vị trí này được xác định khi thủ tục được gọi. Nếu tham số là một biến chỉ số, biểu thức của nó trong phần chỉ số được tính khi thủ tục được gọi.

Khối bao quanh tên được hoạt động trong môi trường mà tên khai báo. Khi chạy xong một câu lệnh call, thì phần câu lệnh (mà câu lệnh call vừa chạy là một thành phần) được cho chạy tiếp tại điểm bị ngắt.

Các câu lệnh read (read_statement)

câu lệnh read → **read_token open_token**
ident_token [*phânchisó*]
(list_token ident_token [*phânchisó*])
close_token

Ràng buộc: Câu lệnh read phải xuất hiện trong phạm vi hoạt động của mọi tên trong danh sách và mọi tên đều phải là biến.

Nếu phần chỉ số của một tên không đóng thì tên phải là một biến mảng, nếu không tên phải là một biến đơn hoặc tham biến. Số chỉ số trong phần chỉ số của một biến chỉ số phải bằng số cặp buộc trong khai báo của biến mảng đó.

Ngữ nghĩa: Đối với mỗi tên trong danh sách và theo thứ tự xuất hiện, hai bước sau được thực hiện:

1. Xác định vị trí đích. Trong trường hợp một tên là một biến đơn, đích là vị trí buộc với tên. Đối với một tên là một biến mảng, các biểu thức trong phần chỉ số được tính. Các giá trị của chúng cùng với tên xác định vị trí của biến chỉ số. Đối với một tên là tham biến, đích là vị trí chỉ bởi tham số.
2. Số đầu vào đầu tiên được đọc và ghi vào vị trí xác định trong bước 1, sau đó bị loại ra khỏi đầu vào.

Các biểu thức (expression, term, factor, operand, add_operator, multiply_operator, unary_operator)

biểuthức → *hạngthức* (*phéptoáncongtrù* *hạngthức*)*

hạngthức → *thìratô* (*phéptoánnhâncchia* *thìratô*)*

thìratô → [*phéptoánmộtngôi*] *toánhạng*

toánhạng → **ident_token** [*phânchisô*]

| **num_token**

| **open_token** *biểuthức* **close_token**

phéptoáncongtrù → **plus_token** | **minus_token**

phéptoánnhâncchia → **times_token** | **over_token** | **modulo_token**

phéptoánmộtngôi → **negate_token** | **absolute_token**

Ràng buộc: Biểu thức phải xuất hiện trong phạm vi hoạt động của mọi tên có trong toán hạng của biểu thức. Nếu phần chỉ số của một tên không rỗng thì tên phải là một biến mảng, nếu không nó phải là biến đơn, hằng số hoặc tham số. Số các chỉ số trong phần chỉ số của một biến chỉ số phải bằng số cặp buộc trong phần khai báo của biến mảng đó.

Ngữ nghĩa: Một biểu thức là một luật tính một giá trị có kiểu số nguyên. Giá trị thu được bằng cách áp dụng các phép toán số học với các toán hạng. Trong trường hợp một số hoặc tên là một hằng thì giá trị của toán hạng chính là giá trị thực. Trong trường hợp tên là một biến đơn thì đó là giá trị hiện tại tại vị trí của tên này. Với tên là một biến mảng, biểu thức trong phần chỉ số được tính. Giá trị của chúng cùng với tên sẽ xác định vị trí của biến chỉ số. Giá trị hiện tại tại vị trí này chính là giá trị thực của toán hạng. Đối với một tên là tham số, giá trị hiện tại của nó sẽ được thay cho tham số này.

Các toán tử số học cộng, trừ, nhân, chia, modulo, đảo dấu và tuyệt đối (plus, minus, times, over, modulo, negate, absolute) có các nghĩa thông thường. Chúng có mức độ ưu tiên như sau:

- Ưu tiên cao nhất: đảo dấu, tuyệt đối
- Ưu tiên thứ nhì: nhân, chia, modulo
- Ưu tiên thấp nhất: cộng, trừ

Một biểu thức được đặt trong cặp ngoặc đơn sẽ được tính trước và giá trị thu được sẽ dùng để tính tiếp.

III. MỞ RỘNG VIM CHO MẢNG

Bây giờ, ta sẽ xem xét mở rộng kiến trúc, tập chỉ thị lệnh và trình thông dịch của máy ảo VIM sao cho có thể dễ dàng khai báo và dùng được các mảng. Các tham số của thủ tục sẽ được xem xét ở phần sau.

1. Kiến trúc

Trong phần này, ta sẽ nghiên cứu cách dùng địa chỉ làm chỉ số và cách đặt các mảng động trong phân đoạn của bộ nhớ dữ liệu.

Địa chỉ của các biến chỉ số

Ta xét một mảng n chiều có dạng $A[lo_1..up_1, lo_2..up_2, \dots, lo_n..up_n]$. Các phân đoạn dữ liệu của VIM (và các bộ nhớ của máy tính nói chung) có cấu trúc tuyến tính. Do đó ta sẽ phải tuyến tính hóa mảng A n chiều này và ánh xạ nó vào một mảng một chiều M $[offset .. offset + (up_1 - lo_1 + 1) * (up_2 - lo_2 + 1) * \dots * (up_n - lo_n + 1) - 1]$. Việc thực hiện tuyến tính có thể theo nhiều cách, ví dụ như tuyến tính theo chuỗi các hàng hoặc chuỗi các cột. Ta sẽ ánh xạ các mảng theo độ rộng của hàng vào bộ nhớ tuyến tính, nghĩa là ta sẽ áp dụng các phương pháp làm chỉ số cuối thay đổi nhanh nhất. Đối với mảng ba chiều điều này có nghĩa là:

$p = offset;$

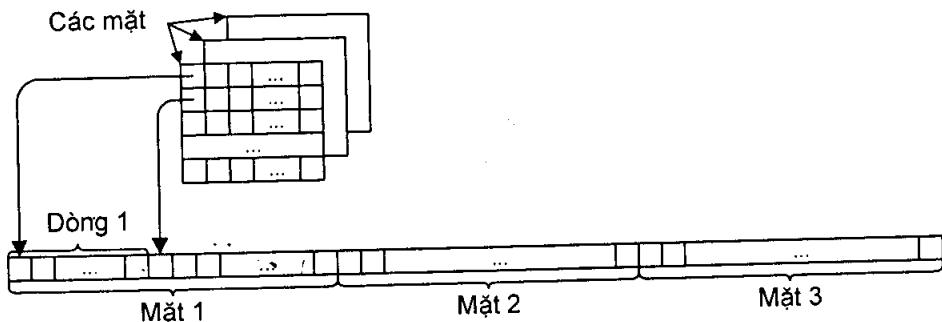
for ($i = lo_1; i \leq up_1; i++$)

for ($j = lo_2; j \leq up_2; j++$)

for ($k = lo_3; k \leq up_3; k++$)

 { $M[p] = A[i][j][k]; p++;$ }

Chỉ số i chỉ các mặt, chỉ số j chỉ các hàng, và chỉ số k là các phần tử trong hàng.



Hình 8.1. Mảng ba chiều và tuyến tính của nó.

Chỉ số trong M ứng với biến chỉ số

$$A[i, j, k] \quad (lo_1 \leq i \leq up_1, lo_2 \leq j \leq up_2, lo_3 \leq k \leq up_3)$$

là

$$\begin{aligned} p = offset &+ (i - lo_1) * (up_2 - lo_2 + 1) * (up_3 - lo_3 + 1) \\ &+ (j - lo_2) * (up_3 - lo_3 + 1) \\ &+ (k - lo_3) \end{aligned}$$

Nói cách khác, ta bắt đầu đi qua các mặt ($i - lo_1$), sau đó qua các dòng ($j - lo_2$), và cuối cùng qua các phần tử ($k - lo_3$) trước khi ta đến được biến chỉ số $A[i, j, k]$.

Để tính một chỉ số bất kì, ta cần tính giá trị của $(up_2 - lo_2 + 1) * (up_3 - lo_3 + 1)$ và $(up_3 - lo_3 + 1)$. Ta sẽ viết lại công thức trên thành:

$$\begin{aligned} p = offset &+ (i - lo_1) * d_2 * d_3 \\ &+ (j - lo_2) * d_3 \\ &+ (k - lo_3) \end{aligned}$$

với $d_2 = (up_2 - lo_2 + 1)$ và $d_3 = (up_3 - lo_3 + 1)$. Chúng ta cũng có thể viết lại công thức thành dạng:

$$\begin{aligned} p = offset &+ (i - lo_1) * s_1 \\ &+ (j - lo_2) * s_2 \\ &+ (k - lo_3) * s_3 \end{aligned}$$

với $s_1 = d_2 * d_3$, $s_2 = d_3$ và $s_3 = 1$.

Quay trở lại với mảng n chiều $A[lo_1..up_1, lo_2..up_2, \dots, lo_n..up_n]$, chỉ số trong M của biến chỉ số $A[ix_1, ix_2, \dots, ix_n]$ là

$$\begin{aligned} p = & offset + (ix_1 - lo_1) * s_1 \\ & + (ix_2 - lo_2) * s_2 \\ & \dots \\ & + (ix_n - lo_n) * s_n \end{aligned}$$

với $d_k = (up_k - lo_k + 1)$ ($1 \leq k \leq n$) và $s_{k-1} = d_k * \dots * d_n$ ($1 \leq k \leq n$), $s_n = 1$.

Quá trình tính s_k sẽ như sau:

$$\begin{aligned} s_n &= 1 \\ s_{k-1} &= d_k * s_k \quad (1 \leq k \leq n), \end{aligned}$$

Giá trị của s_0 không đóng vai trò gì trong việc tính chỉ số trong M nhưng nó chính là kích thước của toàn mảng, và do vậy rất quan trọng cho các mục đích cấp phát khoảng trống.

Các giá trị của lo_i và s_i cung cấp các thông tin cần cho việc kiểm tra buộc như dưới đây. Đối với mỗi k ($1 \leq k \leq n$) phải tuân theo:

$$lo_k \leq ix_k \leq up_k$$

có thể viết lại thành

Hình 8.2. Bộ mô tả của một mảng.

$$0 \leq (ix_k - lo_k) * s_k \leq (up_k - lo_k) * s_k$$

Với

$$(up_k - lo_k) * s_k = (up_k - lo_k + 1) * s_k - s_k = d_k * s_k - s_k = s_{k-1} - s_k$$

ta viết lại thành

$$0 \leq (ix_k - lo_k) * s_k \leq s_{k-1} - s_k$$

Ở trên, ta đã thấy các thông tin cho việc cấp phát bộ nhớ, tính địa chỉ và kiểm tra ràng buộc với giá trị của n (số chiều), offset và s_i ($0 \leq i \leq n - 1$) và lo_i ($1 \leq i \leq n$). Các thông tin này được đặt cạnh nhau trong một mảng gọi là *bộ mô tả* như hình 8.2. Bộ mô tả này sẽ được lưu trong phân đoạn dữ liệu của khối mà mảng được khai báo trong đó.

Quản lý bộ nhớ của mảng

Do mọi biến đơn có kích thước cố định, khoảng trống cần cho các biến đơn trong một khối được biết trong thời gian dịch. Kích thước của một biến

n
$offset$
s_0
lo_1
s_1
lo_2
s_2
...
s_{n-1}
lo_n

mảng dao động nên nói chung lại không được biết trong thời gian này nếu các ràng buộc là các biểu thức chứa các biến với giá trị chỉ được biết trong thời gian chạy. Trong định nghĩa của các buộc (của mảng) nói rằng các toán tử của các biểu thức buộc có thể là biến đơn, biến chỉ số, hằng số và cần được định nghĩa trong một khái bao ngoài cùng với các thông số. Điều này có nghĩa là các toán tử đó phải có một giá trị tại thời điểm khái được kích hoạt. Do kích thước của mảng không được biết trong thời gian dịch, các offset của chúng cũng không được xác định lúc này. Kích thước của bộ mô tả lại được biết trong lúc dịch. Độ lớn của bộ mô tả của một mảng có n chiều là $2 * n + 2$, tức là một vị trí cho n , 1 cho offset, n vị trí cho các ràng buộc dưới và n vị trí cho các giá trị s .

Chúng ta đã biết một phân đoạn dữ liệu có một phần tĩnh và một phần động. Phần tĩnh có chứa các ô cho các biến đơn và các ô được lưu trong bộ mô tả của các biến mảng. Các biến đơn và bộ mô tả của mảng được gán một chuyển dịch nhằm dùng được trong phân khai báo. Phần động có chứa các ô cho bản thân các mảng đó.

Việc tính toán *offset* và độ dài của các mảng và mở rộng của phân đoạn dữ liệu với phân động được xử lý như sau: Lúc đầu, độ dài hiện tại của phân đoạn dữ liệu là bằng độ dài của phần tĩnh. *offset* của một mảng bằng độ dài hiện tại của phân đoạn dữ liệu đó. Kích thước của mảng được tính và thêm vào độ dài hiện tại của phân đoạn này. Độ dài mới sẽ là *offset* cho mảng tiếp theo trong phân khai báo.

2. Tập chỉ thị

Trong phần này, ta sẽ thêm các chỉ thị mới để làm việc với các biến chỉ số và thực hiện các khai báo mảng động.

Chỉ thị *ldvar* sao chép giá trị của một biến động từ phân đoạn với số *sn* và vị trí *dpl* trong phân đoạn vào định ngăn xếp. Chỉ thị *stvar* loại giá trị của một biến đơn từ định của ngăn xếp và lưu nó trong phân đoạn có số *sn* và tại vị trí *dpl* trong phân đoạn. Đối với các biến chỉ số ta cần các lệnh nạp và lưu khác. Chúng sẽ được gọi là *idxvar* và *stxvar*. Cả hai chỉ thị này hoạt động với bộ địa chỉ (*sn*, *dpl*) chỉ địa chỉ của bộ mô tả trong phân đoạn dữ liệu. Các chỉ số của các biến chỉ số được đặt trong ngăn xếp. Cả hai chỉ thị kiểm tra từng chỉ số có nằm trong ràng buộc thấp và cao của nó hay không. Nếu không thì sẽ có một lỗi được thông báo và chương trình dừng.

Để tính các trường của bộ mô tả và tính lại độ dài phân đoạn, ta đưa ra các chỉ thị *descr* *sn*, *dpl*, *n* với (*sn*, *dpl*) là địa chỉ của bộ mô tả và *n* là chiều của mảng. Độ dài của phân đoạn dữ liệu được lưu trong phần quản lý của

phân đoạn. Giá trị khởi đầu (độ dài của phần tĩnh) được đặt trong phân đoạn dữ liệu nhờ các chỉ thị `crseg sn`, index với `sn` là số phân đoạn và index là vị trí của độ dài trong bảng địa chỉ. Chỉ thị `descr sn, dpl, n` điền vào bộ mô tả, tính lại độ dài phân đoạn và đặt giá trị mới trong phân đoạn dữ liệu. Các chỉ thị được cho như bảng dưới:

Nạp và lưu

chỉ thị lệnh	ý nghĩa
<code>ldcon intval</code>	nạp giá trị (số nguyên) <code>intval</code> vào ngăn xếp
<code>ldvar sn, dpl</code>	sao chép giá trị (số nguyên) từ địa chỉ bộ nhớ (<code>sn, dpl</code>) vào định ngăn xếp
<code>stvar sn, dpl</code>	loại bỏ giá trị (số nguyên) ở định ngăn xếp và ghi nó vào địa chỉ bộ nhớ (<code>sn, dpl</code>)
<code>lidxvar sn, dpl</code>	tính địa chỉ của một biến chỉ số; bộ mô tả của một biến mảng có địa chỉ bộ nhớ (<code>sn, dpl</code>) và chỉ số được lấy (và loại khỏi ngăn xếp); sao chép giá trị (số nguyên) từ địa chỉ bộ nhớ của biến chỉ số vào định ngăn xếp
<code>stxvar sn, dpl</code>	tính địa chỉ của một biến chỉ số; bộ mô tả của một biến mảng có địa chỉ bộ nhớ (<code>sn, dpl</code>) và chỉ số được lấy (và loại khỏi ngăn xếp); loại bỏ giá trị (số nguyên) ở định ngăn xếp và ghi nó vào địa chỉ bộ nhớ của biến chỉ số vào định ngăn xếp

Xử lý bộ mô tả

chỉ thị lệnh	ý nghĩa
<code>descr sn, dpl, n</code>	tính các trường của bộ mô tả của một biến mảng và tính lại độ dài phân đoạn; bộ mô tả của mảng có địa chỉ bộ nhớ (<code>sn, dpl</code>), chiều là <code>n</code> và buộc của mảng được lấy (và loại bỏ) từ ngăn xếp.

Ví dụ 8.1: Xem xét câu lệnh gán:

$$a[2*p, q] := b[p+3, 7]$$

với p và q là các biến đơn. Địa chỉ của biến đơn x được ký hiệu bằng cặp sn_x, dpl_x và địa chỉ của bộ mô tả của một mảng m bằng sn_m, dpl_m . Các chỉ thị lệnh của VIM tương ứng với câu lệnh gán đó là:

```

ldvar   snp, dplp
ldcon   3
add
ldcon   7
ldxvar  snb, dplb
ldcon   2
ldxvar  snp, dplp
mul
ldvar   snq, dplq
stxvar  sna, dpla

```

Ví dụ 8.2: Xét thủ tục:

```

Proc pr
begin
    var int a, b[1..10], c, d[1..q, 2*q..q+4];
    ...
end;

```

Các biến đơn p và q được khai báo (và gán giá trị trước khi pr được gọi) trong một khối bao quanh. Việc chuyển thành chỉ thị VIM như ở dưới đây. Chú ý là sn_{pr}, sn_b và sn_d có cùng số phân đoạn.

```

crseg   snpr, length_indexpr
ldcon   1
ldcon   10
descr   snb, dplb, 1
ldcon   1
ldvar   snq, dplq
ldcon   2
ldvar   snp, dplp
mul
ldvar   snq, dplq
ldcon   4
add
descr   snd, dpld, 2

```

```
...
dlseg
return
```

3. Trình thông dịch

Do có thêm một số chỉ thị mới, trình thông dịch VIM bắt buộc phải sửa đổi. Version mới của trình thông dịch như dưới đây.

```
#include <stdio.h>
#include <conio.h>

#define max_prog      1000 /* độ dài bộ nhớ chương trình */
#define max_table     1000 /* độ dài tối đa các loại bảng */
#define max_data100   1000 /* độ dài bộ nhớ dữ liệu */
#define max_depth     15   /* độ sâu gọi lồng nhau tối đa */
#define max_stack     100  /* độ sâu tối đa của các ngăn xếp */

typedef enum {
    true = 1, false = 0
} boolean;

typedef enum {
    neg, abs_, add, sub, mul, dvi, mdl, eq, ne, lt, le, gt, gc,
    ⇒ ldcon, ldvar, stvar, ldxvar, stxvar, jump, jift, jiff, call,
    ⇒ descr, crseg, dlseg, return_, rdint, wrint, halt
} operation;

typedef struct {
    operation code;
    union {
        int value;           /* ldcon */
        struct {
            int sn1, dpl;   /* ldvar,stvar,ldxvar,stxvar */
            } u1;
        int index1;         /* jump, jift, jiff, call */
        struct {
            int sn2, dpl2, n; /* descr */
            } u2;
        struct {
            int sn3, index3; /* crseg */
            } u3;
        } u;
    } instruction;

instruction prog[max_prog];
```

```

int          CSG[max_depth];
int          address_table[max_table];
int          length_table[max_table];
int          data[max_data];
int          stack[max_stack];
int          return_stack[max_stack];
⇒int         dsp;

void read_code(void)
{
    /* Bạn hãy tự mình thiết kế lại phần này */
}

/* Các thủ tục push, pop, push_return, pop_return, read_int   write_int, create_segment,
delete_segment không trình bày lại ở đây */

/* Thêm mới */
void descriptor(int sn, int dpl, int n)
{
    int      length, s, k, up, lo;

    s = 1;
    data[CSG[sn]+dpl] = n;
    for (k = dpl+n*2+1; k >= dpl+3; k -= 2) {
        pop(&up);
        pop(&lo);
        if (up < lo)
            report_error("Buộc trên nhỏ hơn buộc dưới");
        data[CSG[sn]+k] = lo;
        s *= up - lo + 1;
        data[CSG[sn]+k-1] = s;
    }
    length = data[CSG[sn]-3];
    data[CSG[sn]+dpl+1] = length;
    length += s;
    data[CSG[sn]-3] = length;
    dsp += s;
}

void index_dpl(int sn, int dpl)
{
    int      ix, k, lo, n, s, sl, offset, ∵dpl;

    n = data[CSG[sn]+dpl];
    offset = data[CSG[sn]+dpl+1];
    k = dpl+n*2;
    pop(&ix);
}

```

```

lo = data[CSG[sn]+k+1];
xdpl = ix-lo;
s1 = data[CSG[sn]+k];
if (ix-lo < 0 || ix-lo > s1-1)
    report_error("Chỉ số của mảng nằm ngoài các ràng buộc");
while (k >= dpl+4) {
    pop(&ix);
    lo = data[CSG[sn]+k-1];
    s = data[CSG[sn]+k];
    xdpl += (ix-lo)*s;
    s1 = data[CSG[sn]+k-2];
    if ((ix-lo)*s < 0 || (ix-lo)*s > s1-s)
        report_error("Chỉ số của mảng nằm ngoài các ràng buộc");
    k -= 2;
}
xdpl += offset;
return xdpl;
}

void main(void)
{
    instruction      *instr;
⇒    int          ic, next, sp, rsp, csn, xdpl, left_operand,
           right_operand, operand, result, length;
    boolean truthval, stop;

    read_code();
    ic = 0;
    sp = 0;
    dsp = 0;
    rsp = 0;
    csn = 0;
    CSG[csn] = 0;
    stop = false;

    while (!stop) {
        next = ic + 1;
        instr = &prog[ic];

        switch (instr->code) {

            case neg:
            case abs_:
                /* Không trình bày lại ở đây */
                break;

            case add:
            case sub:
            case mul:
            case dvi:
        }
    }
}

```

```

case mdl:
    /* Không trình bày lại ở đây */
    break;

case eq:
case ne:
case lt:
case le:
case gt:
case ge:
    /* Không trình bày lại ở đây */
    break;

case ldcon:
    push (instr->u.value);
    break;

case ldvar:
    push(data[CSG[instr->u.u1.sn1]+instr->u.u1.dpl]);
    break;

case stvar:
    pop(&data[CSG[instr->u.u1.sn1]+instr->u.u1.dpl]);
    break;

⇒ case ldxvar:
    xdpl = index_dpl(instr->u.u1.sn1,instr->u.u1.dpl);
    push(data[CSG[instr->u.u1.sn1]+xdpl]);
    break;

⇒ case stxvar:
    xdpl = index_dpl(instr->u.u1.sn1,instr->u.u1.dpl);
    pop(&(data[CSG[instr->u.u1.sn1]+xdpl]));
    break;

case jump:
case jift:
case jiff:
case call:
    /* Không trình bày lại ở đây */
    break;

⇒ case descr:
    descriptor (instr->u.u2.sn2, instr->u.u2.dp12,
                instr->u.u2.n);
    break;

```

```

case crseg:
case dlseg:
case return_:
    /* Không trình bày lại ở đây */
    break;

case rdint:
case wrint:
    /* Không trình bày lại ở đây */
    break;

case halt:
    stop = true;
    break;
}
ic = next;
}

```

4. Chuyển đổi sang mã VIM cho mảng

Khai báo

Trước khi đề cập đến dịch các khai báo biến, ta sẽ xét việc chuyển đổi cho khai báo thủ tục, khối và phần khai báo. Các thủ tục là:

<i>khaibáothủtục</i> →	procedure_token ident_token
	[open_token
	<i>phànthsamsó</i> (separator_token <i>phànthsamsó</i>)*
	close_token]
	<i>khối</i> separator_token

khối → **begin_token** *phânkhaibáo* *phâncâulệnh* **end_token**

phânkhaibáo → (*khaibáohàng* | *khaibáobiến* *k*)* *khaibáothủtục* *

Ta đã biết cách chuyển các luật trên về ký pháp hậu tố như sau:

< *khaibáothủtục.*> → < *khối* >

return

< *khối* > → < *phânkhaibáo* >

crseg *sn_{lên}*, *length_{lên}*

< *phâncâulệnh* >

dlseg

<phản khaibáo> → (*<khaibáohàng>* | *<khaibáobiển>*)*

[jump over
<khaibáothùtục>⁺

over:
]

Các sản xuất sau dùng để khai báo biến:

khaibáobiển → **var_token** *khaibáokiểu*
ident_token [*phảnbuộc*]
(*list_token ident_token* [*phảnbuộc*])^{*}
separator_token

khaibáokiểu → **int_token**

phảnbuộc → **left_bracket_token**
căpbuộc (*list_token căpbuộc*)^{*}
left_bracket_token

căpbuộc → *buộcđuới up_to_token* *buộctrên*

buộcđuới → *biểuthúc*

buộctrên → *biểuthúc*

Ta sẽ ký hiệu số phân đoạn và dịch chuyển của một biến *tên* là *sn_{tên}* và *dpl_{tên}*. Việc chuyển đổi như sau:

<i><khaibáobiển></i> →	ϵ	{ Cho biến đơn }
<i><phảnbuộc></i>		{ Cho biến chỉ số }
	descr <i>sn_{tên}, dpl_{tên}, n</i>	
<i><phảnbuộc></i> → <i><căpbuộc></i> ⁺		
<i><căpbuộc></i> → <i><buộcđuới></i> <i><buộctrên></i>		
<i><buộcđuới></i> → <i><biểuthúc></i>		
<i><buộctrên></i> → <i><biểuthúc></i>		

Chú ý là chuyển đổi khai báo của một biến là rỗng, nghĩa là không sinh mã VIM cho một biến đơn. Điều này cũng tương tự cho khai báo hàng số.

Biến

Các luật liên quan đến biến:

toánhạng → **ident_token** [*phảnchisố*]

```

| num_token
| open_token biéuthúc close_token

phântrái → ident_token [phânchisó]
câulệnh_read →      read_token open_token
                      ident_token [phânchisó]
                      ( list_token ident_token [phânchisó])*
                      close_token

```

Phân chỉ số của các biến chỉ số được định nghĩa trong các thủ tục:

```

phânchisó → left_bracket_token chisó ( list_token chisó )*
                      right_bracket_token
chisó → biéuthúc

```

Cách chuyển chúng sang SLANG như sau:

```

<toánhạng> →      ldvar      sn_tên, dpl_tên { Cho biến đơn }
                     | <phânchisó>          { Cho biến chỉ số }
                     | ldxvar      sn_tên, dpl_tên
                     | ldcon intval_tên     { Cho hằng }
                     | ldcon intval_sô      { Cho con số }
                     | <biéuthúc>          { Cho biểu thức }

<phântrái> →      stvar      sn_tên, dpl_tên { Cho biến đơn }
                     | <phânchisó>          { Cho biến chỉ số }
                     | stxvar      sn_tên, dpl_tên

<câulệnh_read> → ( rdint
                     ( stvar sn_tên, dpl_tên { Cho biến đơn }
                     | <phânchisó>          { Cho biến chỉ số }
                     | stxvar      sn_tên, dpl_tên )
                     )+
<phânchisó> → <chisó>+
<chisó> → <biéuthúc>

```

IV. MỞ RỘNG VIM CHO THAM SỐ

1. Kiến trúc

Trong phần trước của chương này, biến và các tham số đã được định nghĩa. Tham số ứng với tham trị phải là một biểu thức. Các toán tử của nó

có thể là các biến đơn, số và tham số. Các tham số là tham biến phải là một biến đơn hoặc biến chỉ số, hoặc là một tham số từ một thủ tục bao quanh. Chú ý là nơi gọi và thủ tục bị gọi có thể ở trong những phân đoạn khác nhau, nên điều quan trọng đối với kẻ bị gọi là tính được các tham số. Việc tính này lại là nhiệm vụ của phần gọi. Nhiệm vụ của thủ tục được gọi là lưu các tham số trong phân đoạn dữ liệu của nó, sau khi tạo ra phân đoạn.

Trong VIM, việc đánh giá một biểu thức được tổ chức sao cho kết quả được để lại trong ngăn xếp. Điều này có nghĩa là nơi gọi sẽ lưu các giá trị tham số trong ngăn xếp nên thủ tục được gọi có thể tìm thấy chúng. Ta sẽ thực hiện tương tự ứng với các tham biến. Nơi gọi sẽ lưu các tham số trong ngăn xếp và thủ tục được gọi sẽ tìm thấy chúng. Do vậy, ngăn xếp được dùng như là một kho chứa tức thời để truyền các tham số từ nơi gọi sang thủ tục được gọi. Trong trường hợp một tham trị, nơi gọi chỉ đơn giản đặt giá trị tính được vào ngăn xếp. Thủ tục được gọi sẽ lấy giá trị này khỏi ngăn xếp và lưu nó trong phân đoạn dữ liệu. Trong trường hợp tham biến, tham số phải ứng với vị trí của một biến đơn hoặc chỉ số. Địa chỉ của biến này được nơi gọi xác định và đẩy vào ngăn xếp. Sau đó nó được thủ tục bị gọi lấy ra và lưu trong phân đoạn dữ liệu. Thủ tục được gọi có thể đọc và ghi vào vị trí của biến đó thông qua địa chỉ này.

Cho đến nay, ta đã sử dụng cặp (sn, dpl) làm địa chỉ. Do nơi gọi và thủ tục được gọi có thể ở trong những nhóm phân đoạn khác nhau, với cùng số phân đoạn sn có thể biểu diễn những phân đoạn khác nhau trong cả hai nhóm. Do vậy cần phải tính chỉ số trong bộ nhớ dữ liệu ứng với cặp (sn, dpl) và truyền địa chỉ tuyệt đối thay cho cặp (sn, dpl) từ nơi gọi đến nơi được gọi. Tham số cho một tham biến sẽ là giá trị địa chỉ CSG [sn]+dpl.

Chúng ta coi các tham số và biến được lưu trong phân đoạn dữ liệu theo thứ tự sau:

1. Các tham số theo thứ tự xuất hiện trong danh sách tham số.
2. Các biến đơn và các bộ mô tả của các biến mảng theo thứ tự xuất hiện trong phần khai báo.
3. Các mảng theo thứ tự xuất hiện trong phần khai báo.

Như vậy, tham số được truyền như sau. Nơi gọi tính các tham số theo thứ tự xuất hiện trong danh sách tham số.

- Trong trường hợp một tham trị, giá trị được tính và đẩy vào ngăn xếp.

- Trong trường hợp một tham biến, địa chỉ tuyệt đối CSG [sn]+ dpl được tính và đẩy vào ngăn xếp.

Các thủ tục được gọi phải lấy các tham số khỏi ngăn xếp và lưu chúng vào phân đoạn dữ liệu của nó. Tham số đầu có dpl thấp nhất, và tham số cuối có dpl cao nhất. Các tham số được lấy khỏi ngăn xếp theo thứ tự ngược, tức là phần tham số của phân đoạn dữ liệu được điền từ chuyên dịch cao nhất đến thấp nhất.

2. Tập chỉ thị

Ta cần các chỉ thị mới để đánh địa chỉ các tham số. Chỉ thị *varaddr sn, dpl* tính địa chỉ CSG [sn] + dpl và đẩy vào ngăn xếp. Chỉ thị *xvaraddr sn, dpl* đọc chỉ số từ ngăn xếp, lấy địa chỉ (sn, dpl) của bộ mô tả, tính độ dịch chuyên xdpl của biến chỉ số và sau đó là địa chỉ tuyệt đối CSG [sn] + xdpl, và đẩy kết quả vào ngăn xếp. Để truy cập vào các tham biến, ta cần các *địa chỉ không trực tiếp*. Chỉ thị *ldind* sẽ lấy địa chỉ tuyệt đối trên đỉnh ngăn xếp. Địa chỉ này được loại khỏi ngăn xếp, và giá trị tại địa chỉ này được đẩy vào ngăn xếp. Chỉ thị *stind* cần địa chỉ tuyệt đối và một giá trị trên đỉnh của ngăn xếp. Địa chỉ và giá trị (theo thứ tự này) được loại khỏi ngăn xếp và giá trị được lưu vào địa chỉ đó.

Các chỉ thị lệnh nhằm tính các địa chỉ tuyệt đối và nạp / lưu các giá trị cho trong bảng dưới đây.

Tính địa chỉ tuyệt đối

chỉ thị lệnh	ý nghĩa
varaddr sn, dpl	đẩy địa chỉ CSG [sn]+ dpl vào ngăn xếp
xvaraddr sn, dpl	đẩy địa chỉ CSG [sn]+ xdpl vào ngăn xếp, với xdpl là độ dịch chuyên của một biến chỉ số; bộ mô tả của một mảng có địa chỉ bộ nhớ (sn, dpl) và chỉ số được lấy (và loại bỏ) từ ngăn xếp.

Tải và lưu

chỉ thị lệnh	ý nghĩa
Ldcon intval	nạp giá trị (số nguyên) intval vào ngăn xếp
Ldvar sn, dpl	sao chép giá trị (số nguyên) từ địa chỉ bộ nhớ (sn, dpl) vào đỉnh ngăn xếp
Stvar sn, dpl	loại bỏ giá trị (số nguyên) ở đỉnh ngăn xếp và ghi nó vào địa chỉ bộ nhớ (sn, dpl)

Idxvar sn, dpl	tính địa chỉ của một biến chỉ số; bộ mô tả của một biến mảng có địa chỉ bộ nhớ (sn, dpl) và chỉ số được lấy (và loại khỏi ngăn xếp); sao chép giá trị (số nguyên) từ địa chỉ bộ nhớ của biến chỉ số vào định ngăn xếp
stxvar sn, dpl	tính địa chỉ của một biến chỉ số; bộ mô tả của một biến mảng có địa chỉ bộ nhớ (sn, dpl) và chỉ số được lấy (và loại khỏi ngăn xếp); loại bỏ giá trị (số nguyên) ở định ngăn xếp và ghi nó vào địa chỉ bộ nhớ của biến chỉ số vào định ngăn xếp
Ldind	loại bỏ địa chỉ tuyệt đối trên định ngăn xếp và sao chép giá trị (số nguyên) tại địa chỉ này vào định ngăn xếp
stind	loại bỏ địa chỉ tuyệt đối và giá trị (số nguyên) trên định ngăn xếp (theo thứ tự) và ghi giá trị vào địa chỉ này.

Ví dụ 8.3: Xét một thủ tục được khai báo như sau:

```

proc pr (int a, b; var int c, d, e)
begin
    ...
    c := a;
    d := b + c * e;
    ...
end;

```

và lệnh gọi:

```
call pr (3, p+q, x[i], y, z)
```

với i, p, q và y là các biến đơn, x[i] là một biến chỉ số và z là một tham biến từ thủ tục quanh nó.

Các chỉ thị VIM tương ứng với khai báo thủ tục trên như sau:

```

crseg   snpr, length_indexpr
stvar   sne, dple
stvar   snd, dpld
stvar   snc, dplc
stvar   snb, dplb
stvar   sna, dpla
...

```

ldvar	sn_a, dpl_a
ldvar	sn_c, dpl_c
stind	
ldvar	sn_b, dpl_b
ldvar	sn_e, dpl_e
ldind	
ldvar	sn_e, dpl_e
ldind	
mul	
add	
ldvar	sn_d, dpl_d
stind	
...	
dlseg	
return	

Chú ý là sn_{pr} , sn_a , sn_b , sn_c , sn_d và sn_e biểu diễn cùng số phân đoạn.

Các chỉ thị sau ứng với lời gọi:

ldcon	3
ldvar	sn_p, dpl_p
ldvar	sn_q, dpl_q
add	
ldvar	sn_i, dpl_i
xvaraddr	sn_x, dpl_x
varadd	sn_y, dpl_y
ldvar	sn_z, dpl_z
call	address_index _{pr}

Chú ý là chỉ thị ldvar sn_z, dpl_z nạp địa chỉ tuyệt đối của tham số tương ứng với tham biến z mà không phải là giá trị.

3. Trình thông dịch

Do có thêm một số chỉ thị mới, trình thông dịch VIM bắt buộc phải sửa đổi lần nữa. Version mới của trình thông dịch như dưới đây.

```

#include <stdio.h>
#include <conio.h>

#define max_prog 1000 /* độ dài bộ nhớ chương trình */
#define max_table 1000 /* độ dài tối đa các loại bảng */
#define max_data 100 /* độ dài bộ nhớ dữ liệu */
#define max_depth 15 /* độ sâu gọi lồng nhau tối đa */
#define max_stack 100 /* độ sâu tối đa của các ngăn xếp */

typedef enum {
    true = 1, false = 0
} boolean;

typedef enum {
    neg, abs_, add, sub, mul, dvi, mdl, eq, ne, lt, le, gt, ge,
    ldcon, ldvar, stvar, ldxvar, stxvar, varaddr, xvaraddr, jump,
    jift, jiff, call, descr, crseg, dlseg, return_, rdint, wrint,
    halt
} operation;

typedef struct instruction {
    operation code;
    union {
        int value; /* ldcon */
        struct {
            int sn1, dpl; /* ldvar, stvar, ldxvar, stxvar
                           varaddr, xvaraddr */
            } u1;
        int index1; /* jump, jift, jiff, call */
        struct {
            int sn2, dpl2, n; /* descr */
            } u2;
        struct {
            int sn3, index3; /* crseg */
            } u3;
    } u;
} instruction;

instruction prog[max_prog];
int CSG[max_depth];
int address_table[max_table];
int length_table[max_table];
int data[max_data];
int stack[max_stack];
int return_stack[max_stack];

```

/* Các thủ tục read_code, push, pop, push_return, pop_return, read_int, write_int, create_segment, delete_segment, descriptor, index_dpl không trình bày lại ở đây */

```

void main(void)
{
    instruction      *instr;
    int             ic, next, sp, rsp, csn, xdpl, i, left_operand,
                    right_operand, operand, result, length, address;
    boolean truthval, stop;

    read_code();
    ic = 0;
    sp = 0;
    dsp = 0;
    rsp = 0;
    csn = 0;
    CSG[csn] = 0;
    stop = false;

    while (!stop) {
        next = ic + 1;
        instr = &prog[ic];

        switch (instr->code) {

            case neg:
            case abs_:
                /* Không trình bày lại ở đây */
                break;

            case add:
            case sub:
            case mul:
            case dvi:
            case mdl:
                /* Không trình bày lại ở đây */
                break;

            case eq:
            case ne:
            case lt:
            case le:
            case gt:
            case ge:
                /* Không trình bày lại ở đây */
                break;

            case ldcon:
                push (instr->u.value);
                break;

            case ldvar:
        }
    }
}

```

```

        push(data[CSG[instr->u.u1.sn1]+instr->u.u1.dpl]);
        break;

case stvar:
    pop(&data[CSG[instr->u.u1.sn1]+instr->u.u1.dpl]);
    break;

case ldxvar:
    xdpl = index_dpl(instr->u.u1.sn1,instr->u.u1.dpl);
    push(data[CSG[instr->u.u1.sn1] + xdpl]);
    break;

case stxvar:
    xdpl = index_dpl(instr->u.u1.sn1,instr->u.u1.dpl);
    pop(&(data[CSG[instr->u.u1.sn1]+xdpl]));
    break;

⇒ case varaddr:
    push(CSG[instr->u.u1.sn1] + instr->u.u1.dpl);
    break;

⇒ case xvaraddr:
    xdpl = index_dpl(instr->u.u1.sn1,instr->u.u1.dpl);
    push( [CSG[instr->u.u1.sn1] + xdpl]);
    break;

⇒ case ldind:
    pop(&address);
    push(data[address]);
    break;

⇒ case stind:
    pop(&address);
    pop(&(data[address]));
    break;

case jump:
case jift:
case jiff:
case call:
    /* Không trình bày lại ở đây */
    break;

case descr:
    descriptor (instr->u.u2.sn2, instr->u.u2.dpl2,
                instr->u.u2.n);
    break;

case crseg:

```

```

case dlseg:
case return_:
    /* Không trình bày lại ở đây */
    break;

case rdint:
case wrint:
    /* Không trình bày lại ở đây */
    break;

case halt:
    stop = true;
    break;
}
ic = next;
}
}

```

4. Chuyển đổi sang mã VIM cho tham số

Ta có các luật chuyển đổi cho tham số như dưới đây.

Thủ tục

< khaibáothùtục > → crseg sn_{tên}, length_index_{tên}

[stvar sn_{tên}, nr_of_pars_{tên} - 1

stvar sn_{tên}, nr_of_pars_{tên} - 2

...

stvar sn_{tên}, 1

stvar sn_{tên}, 0

]

< khói >

dlseg

return

< khói > → *< phânkhaibáo >* *< phâncâulệnh >*

< câulệnh_call > → [*< thamsô >* ⁺]

call address_index_{tên}

< thamsô > → varaddr sn_{tên}, dpl_{tên}

| ldvar sn_{tên}, dpl_{tên}

| <phânchisô >

xvaraddr sn_{tên}, dpl_{tên}

| <biểu thức >

Biến

< toán hạng > →	ldvar	sn _{tên} , dpl _{tên}	{ biến đơn }
	ldvar	sn _{tên} , dpl _{tên}	{ tham trị }
	ldvar	sn _{tên} , dpl _{tên}	{ tham biến }
	ldind		
	<phânchisô >		{ biến chỉ số }
	ldxvar	sn _{tên} , dpl _{tên}	
	ldcon intval _{tên}		{ hằng }
	ldcon intval _{số}		{ số }
	<biểu thức >		{ (biểu thức) }

< phần trái > →	stvar	sn _{tên} , dpl _{tên}	{ biến đơn }
	ldvar	sn _{tên} , dpl _{tên}	{ tham biến }
	stind		
	<phânchisô >		{ biến chỉ số }
	stxvar	sn _{tên} , dpl _{tên}	

< câu lệnh _read > → (rdint	(stvar	sn _{tên} , dpl _{tên}	
		ldvar	sn _{tên} , dpl _{tên}	{ tham biến }
		stind		
		<phânchisô >		
		stxvar	sn _{tên} , dpl _{tên}	
) ⁺			

< phânchisô > → < chisô >⁺

< chisô > → < biểu thức >

V. HƯỚNG DẪN MỞ RỘNG CHƯƠNG TRÌNH DỊCH

1. Mở rộng bảng ký hiệu

Tổ chức của cây tên và danh sách phạm vi đầy xuống vẫn được giữ nguyên. Tất cả các tên (không chỉ tên của các hằng và biến mà còn tên của mảng và tham số) được lưu trong cây tên và các danh sách tên của danh sách phạm vi đầy xuống. Ta cần lưu thêm các thông tin cần cho các biến mảng và tham số. Để khai báo một biến mảng, ta cần phải lưu số chiều của nó và đối với một khai báo thủ tục, ta cần lưu số tham số của nó, và đối với

mỗi tham số ta cần biết đó là tham trị hay tham biến. Đối với mỗi tên, có một danh sách các bản ghi định nghĩa. Mỗi một bản ghi định nghĩa biểu diễn một định nghĩa của tên đó. Bản ghi định nghĩa chứa các trường sau:

- Số phân đoạn.
- Độ dịch chuyển (của một biến đơn hoặc bộ mô tả của một biến mảng), giá trị (của một hằng số), hoặc chỉ số của địa chỉ (của một thủ tục) trong bảng địa chỉ.
- Kiểu.
- Loại.
- Chiều (của một biến mảng) hoặc số tham số (của một thủ tục).
- Một con trỏ đến một danh sách các bản ghi cho các tham số của thủ tục.
- Một con trỏ đến bản ghi định nghĩa tiếp theo.

Đối với mỗi tham số có một *bản ghi tham số*. Các bản ghi tham số có các trường:

- Phương pháp truy cập, nghĩa là tham trị hay tham biến.
- Một con trỏ đến bản ghi tham số tiếp theo.

Các khai báo sau mô tả các bản ghi của cây tên, danh sách định nghĩa và danh sách tham số. Ta có kiểu *e_kind* là đánh số tất cả các kiểu có thể có của tên (nhờ kĩ thuật enum). Nghĩa của các giá trị của nó như sau:

- | | |
|-----------------------|-----------------------------------|
| • <i>unknown_kind</i> | không hiểu |
| • <i>const_kind</i> | tên hằng |
| • <i>svar_kind</i> | tên biến đơn |
| • <i>avar_kind</i> | tên biến mảng |
| • <i>proc_kind</i> | tên thủ tục |
| • <i>vpar_kind</i> | tên tham số giá trị (tham trị) |
| • <i>rpar_kind</i> | tên tham số truy nhập (tham biến) |

Kiểu *e_access* định nghĩa các phương pháp truy cập đến các tham số như sau:

- | | |
|---------------------|---------|
| • <i>val_access</i> | tham tr |
| • <i>ref_access</i> | tham bi |

Các khai báo mới là:

```
typedef enum {
    no_type, unknown_type, int_type
} e_type;

typedef enum {
    unknown_kind, const_kind, svar_kind, avar_kind, proc_kind, vpar_kind, rpar_kind
} e_kind

typedef enum {
    val_access, ref_access
} e_access

typedef struct par_rec {
    e_access      access;
    struct par_rec *next;
} par_rec;

typedef struct def_rec {
    int           sn, dpl_or_value_or_index;
    e_type        type;
    e_kind        kind;
    int           dim_or_nr_of_pars;
    par_rec      *par_list;
    struct def_rec *next;
} def_rec;

typedef struct ident_rec {
    string        ident;
    def_rec       *def_list;
    struct ident_rec *left, *right;
} ident_rec;
```

2. Các thủ tục và hàm

Do thêm mảng và tham số vào SLANG, ta buộc phải xem xét lại các thủ tục và hàm trong các giai đoạn khác nhau của chương trình dịch. Trong các bảng dưới đây là các nhiệm vụ của chúng, trong đó một số được xác định lại, một số được thêm mới trong khi một số lại bị bỏ đi. Các tham số **in** là các tham số vào, **out** là các tham số lấy ra, còn **inout** vừa làm nhiệm vụ đưa vào vừa làm nhiệm vụ lấy ra.

Các chức năng giúp quản lý bảng ký hiệu

Trong bảng sau ta thấy hàm *create_definition* có thêm tham số *dim_or_nr_of_pars* dùng để biểu diễn các chiều của một mảng hoặc số tham

số của một thủ tục. Còn *find_definition* có thêm tham số *is_bound* có giá trị là true nếu toán hạng là thành phần của một biểu thức buộc. Trong trường hợp này, ta phải tìm một hằng hoặc biến trong khối bao quanh. Việc quản lý bảng ký hiệu cần các thủ tục /hàm cho mảng và tham số. Thủ tục *get_dimension* trả lại số chiều của một mảng. Còn *get_nr_of_parameters* trả lại số tham số của một thủ tục. Có các chức năng dùng để thêm các bản ghi tham số và tìm kiếm các bản ghi tham số trong bảng ký hiệu. Một chức năng khác trả lại kiểu truy cập *access_kind* của một bản ghi tham số - cho biết đó là tham trị hay tham biến.

thủ tục /hàm	ý nghĩa
<code>enter_scope()</code>	tạo một bản ghi phạm vi mới và tăng mức phạm vi
<code>exit_scope()</code>	xoá các bản ghi định nghĩa của tất cả các tên trong phạm vi hiện tại; xoá bản ghi phạm vi hiện tại và giảm mức phạm vi
<code>current_scope(out level: integer)</code>	trả lại trong level mức phạm vi của khối hiện tại đang được phân tích
<code>create_definition(out def: def_ptr; in dpl_or_value_or_index: integer; in type: e_type; in kind: e_kind; in dim_or_nr_of_pars: integer)</code>	tạo một bản ghi định nghĩa mới và khởi tạo các trường của nó với các giá trị của <code>dpl_or_value_or_index</code> , <code>type</code> , <code>kind</code> và <code>dim_or_nr_of_pars</code> ; tham số <code>sn</code> ẩn; tham số <code>ra</code> <code>def</code> chỉ đến bản ghi được tạo ra.
<code>insert_definition(in name: string; in def: def_ptr)</code>	thêm một tên <code>name</code> vào cây tên và thêm bản ghi (được trả bằng <code>def</code>) vào danh sách định nghĩa của bảng ký hiệu
<code>find_definition(out def: def_ptr; in name: string; in is_bound: Boolean)</code>	trả lại thông qua con trả <code>def</code> trả đến một bản ghi định nghĩa của tên <code>name</code> ; nếu <code>is_bound</code> là false thì định nghĩa hiện tại được lấy; nếu ngược lại thì một tham số được thực hiện từ khối hiện tại và một hằng số hoặc biến từ một khối bao quanh.
<code>get_sn_dpl(out sn, dpl: integer; in def: def_ptr)</code>	trả lại trong <code>sn</code> số phân đoạn dữ liệu và trong <code>dpl</code> chuyển dịch của tên định nghĩa trong bản ghi định nghĩa trả bởi <code>def</code> .
<code>get_address(out address: integer; in def: def_ptr)</code>	trả lại trong <code>address</code> chỉ số trong bảng địa chỉ của thủ tục định nghĩa trong bản ghi định nghĩa trả bởi <code>def</code> .

<code>get_nr_of_parameters(out n: integer; in def: def_ptr)</code>	trả lại trong <i>n</i> số tham số của thủ tục định nghĩa trong bản ghi định nghĩa trả bởi def.
<code>get_dimension(out n: integer; in def: def_ptr)</code>	trả lại trong <i>n</i> số chiều của biến mảng định nghĩa trong bản ghi định nghĩa trả bởi def.
<code>add_parameters(in def: def_ptr; in access: e_access)</code>	tạo ra một bản ghi tham số mới, khởi đầu các trường của nó với giá trị của access, thêm bản ghi này vào danh sách các bản ghi tham số của bản ghi thủ tục trả bởi def và tăng số tham số.
<code>find_first_parameters(out def_par; par_ptr, in def proc: def_ptr)</code>	trả lại trong def_par một con trỏ đến bản ghi tham số của tham số thứ nhất của bản ghi thủ tục trả bởi def proc.
<code>find_next_parameters(inout def_par; par_ptr)</code>	trả lại trong def_par một con trỏ đến bản ghi tham số của tham số tiếp theo của bản ghi thủ tục trả bởi def_par.
<code>get_kind_of_parameters(out is_ref_arg: Boolean; in def: par_ptr)</code>	trả lại trong is_ref_arg kiểu truy nhập access_kind của bản ghi tham số trả bởi def_par, là rpart_kind hay không.

Các chức năng kiểm tra các điều kiện ràng buộc

Các chức năng trong bảng sau được sửa phức tạp hơn cũ do thêm các trường hợp. Chức năng *check_if_false* dùng để kiểm tra một giá trị boolean. Chức năng *check_if_too_many* và *check_if_too_few* lại được dùng để kiểm tra xem một tham số của một thủ tục được gọi có bằng số tham số có trong khai báo của thủ tục đó hay không. Chúng cũng có thể được dùng để kiểm tra xem số chỉ số của một biến chỉ số có bằng số chiều (nghĩa là số cặp buộc) của khai báo mảng tương ứng hay không. Kiểm tra số tham số của một thủ tục được gọi thực hiện như sau: Các chức năng *check_if_too_many* và *check_if_too_few* có một tham số *nr*. Giá trị khởi đầu của *nr* là số tham số của khai báo thủ tục. Giá trị này được giảm đi 1 mỗi khi một tham số của thủ tục được phân tích. Chức năng *check_if_too_many* được dùng để kiểm tra xem có nhiều tham số hơn khai báo không. Trường hợp này xảy ra khi *nr* đã giảm về 0 và vẫn còn tham số để phân tích. Chức năng *check_if_too_few* được dùng để kiểm tra xem số tham số có ít hơn khai báo không và trường hợp này xảy ra khi không còn tham số nào để phân tích nữa mà *nr* vẫn chưa giảm đến 0. Theo cách tương tự ta kiểm tra số chỉ số trong biến mảng.

thủ tục /hàm.	ý nghĩa
check_kind(in kind: e_kind; in def: def_ptr; in truthval: Boolean)	kiểm tra kiểu của bản ghi trả bởi def xem nó bằng hay khác <i>kind</i> .
check_if_false(in truthval: Boolean)	kiểm tra giá trị của <i>truthval</i> là false.
Check_if_too_many(in nr: integer)	kiểm tra xem có nhiều tham số hơn khai báo hoặc có nhiều chỉ số hơn các cặp buộc hay không; nr là số chỉ số hoặc tham số cần thiết.
Check_if_too_few(in nr: integer)	kiểm tra xem có nhiều tham số hơn khai báo hoặc có nhiều cặp buộc hơn các chỉ số hay không; nr là số chỉ số hoặc tham số cần thiết.

Các chức năng làm việc với bảng địa chỉ và bảng độ dài

So với trước, không có chức năng nào được sửa đổi ở đây.

thủ tục /hàm	ý nghĩa
get_label(out index: integer)	trả lại vào <i>index</i> một vị trí còn trống trong bảng địa chỉ.
get_index (out index: integer)	trả lại vào <i>index</i> một vị trí còn trống trong bảng độ dài.
emit_label(in index: integer)	giá trị hiện tại của <i>ic</i> được lưu tại vị trí <i>index</i> trong bảng địa chỉ.
enter_length(in index, length: integer)	giá trị của <i>length</i> được lưu tại vị trí <i>index</i> trong bảng địa chỉ.

Các chức năng giúp sinh mã

Chỉ có hai chức năng mới thêm là *emit_descr* và *emit_stargs*. Chức năng thứ nhất sẽ sinh ra chỉ thị *descr*, còn chức năng thứ hai sẽ sinh ra chỉ thị *stvar* nhằm lấy các tham số ra khỏi ngăn xếp và lưu chúng vào phân đoạn dữ liệu theo thứ tự ngược. Chú ý là chức năng *emit_load* phức tạp hơn do không những phải phân biệt biến đơn và hàng số mà còn các biến chỉ số, tham trị và tham biến. Một chức năng tương tự *emit_store* cũng được thêm vào. Chức năng *emit_stvar* không cần nữa và được loại đi.

thủ tục /hàm	Ý nghĩa
initialise_vimcode()	khởi tạo việc sinh mã cho VIM; mở file mã.
Finalise_vimcode()	sinh ra chỉ thị halt; ghi bảng địa chỉ và bảng độ dài vào các file ra và đóng chúng lại.
emit_crseg(in level, index: integer)	sinh ra chỉ thị crseg với các giá trị level và index làm tham số.
emit_descr(in sn, dpl, n: integer)	sinh ra chỉ thị descr với các giá trị sn, dpl và n làm tham số.
emit_stargs(in n: integer)	sinh ra n chỉ thị stvar; các chỉ thị này lấy các tham số từ ngăn xếp và lưu chúng trong phân đoạn dữ liệu theo thứ tự ngược lại.
emit(incode: operation)	tạo ra một chỉ thị không có tham số.
emit_jump(in code: operation; in index: integer)	sinh ra một lệnh nhảy với giá trị của index làm chỉ số trong bảng địa chỉ.
emit_load(in def: def_ptr; in is_ref_arg: Boolean)	xác định xem con trả def chỉ đến một bản ghi của một hằng, một biến đơn hay một biến chỉ số, hoặc một tham trị hay một tham biến, đưa vào is_ref_arg và sinh ra một trong các chỉ thị ldcon, ldvar, idxvar, varaddr hoặc xvaraddr, hoặc kết hợp của một chỉ thị ldvar và một ldind; giá trị hoặc số phân đoạn và chuyển dịch lấy từ bản ghi trả bởi def.
emit_store(in def: def_ptr)	xác định xem con trả def chỉ đến một bản ghi của một biến đơn hay một biến chỉ số, hoặc một tham biến, đưa vào is_ref_arg và sinh ra một trong các chỉ thị stvar hay stxvar, hoặc kết hợp của một chỉ thị ldvar và một stind; số phân đoạn và chuyển dịch lấy từ bản ghi trả bởi def.
emit_ldcon(in value: integer)	sinh ra một chỉ thị ldcon với tham số là giá trị value.

Các chức năng nhằm đánh giá thuộc tính

Đây là các chức năng mới nhằm đánh giá thuộc tính (lượng giá).

thư tục /hàm	ý nghĩa
assign_int(out dest: integer; in source: integer)	gán giá trị số nguyên của <i>source</i> vào <i>dest</i> .
clear_int(out dest: integer)	gán giá trị 0 vào <i>dest</i> .
increment_int(inout dest: integer; in value: integer)	tăng giá trị số nguyên của <i>dest</i> một khoảng là giá trị của <i>value</i> .
decrement_int(inout dest: integer; in value: integer)	giảm giá trị số nguyên của <i>dest</i> một khoảng là giá trị của <i>value</i> .
assign_true(out dest: Boolean)	gán giá trị true vào <i>dest</i> .
assign_false(out dest: Boolean)	gán giá trị false vào <i>dest</i> .
assign_bool(out dest: Boolean; in source: integer)	gán giá trị Boolean của <i>source</i> vào <i>dest</i> .
assign_op(out dest: operation; in source: operation)	gán giá trị toán tử của <i>source</i> vào <i>dest</i> .
clear_op(out dest: operation)	gán giá trị toán tử noop vào <i>dest</i> .
length_of_descriptor(out size: integer; in n: integer)	gán giá trị $2 * n + 2$ vào <i>size</i> .

Bài tập

1. Cho các chương trình nguồn trong SLANG như sau:

a)

```

begin var int i, j, k[1..2];
    proc p(int i; var int a, b)
        begin var int j;
            j := b; a:= i; b := j - 1
        end;
        i := 1; j := 2; k[1] := 1; k[2*i] := 2;
        call p( k[k[j-1]], k[k[2]], k[k[2]] )
    end.

```

b)

```

begin var int a;
    proc proc1 (var int b)
        begin
            proc proc2 (var int c)
                begin c := 2*c end;
                call proc2 (b);
                b := b + 1
            end;

```

```

read (a);
call proc1 (a);
write (a)
end.

```

c)

```

begin var int n;
    proc exchangr (var int a, b)
        begin var int c;
            c := a; a := b; b := c
        end;
    proc sort (int n)
        begin var int i, j, a[1..n];
            i := 1;
            while i <= n do read(a[i]); i := i+1 od;
            i := 2;
            while i <= n do
                if a[i] < a[i-1] then
                    call exchange (a[i], a[i-1]);
                    j := i-1;
                    while j >= 2 do
                        if a[j] < a[j-1] then
                            call exchange(a[j],a[j-1])
                        fi;
                        j := j-1;
                    od
                fi;
                i := i+1
            od;
            i := 1;
            while i <= n do write (a[i]); i := i +1 od
        end;
        read (n);
        call sort(n);
    end.

```

Bạn hãy chuyển chúng về mã VIM.

2. Bạn hãy tự viết các chương trình khác nhau (tùy ý) bằng ngôn ngữ lập trình SLANG đã mở rộng và chuyển chúng về mã VIM.

Bài tập lập trình

Hoàn thiện chương trình dịch cho ngôn ngữ lập trình SLANG có mảng và tham số.

CÁC BÀI TẬP TỔNG HỢP

Dưới đây là các bài tập đòi hỏi bạn phải vận dụng những kiến thức tổng hợp thu được từ giáo trình cũng như những hiểu biết khác. Phần lớn các bài tập đều ở dạng những dự án nhỏ mà bạn có thể tự độc lập nghiên cứu, phát triển.

I. CÁC BÀI TẬP HOÀN THIỆN CHƯƠNG TRÌNH DỊCH NGÔN NGỮ SLANG

Bài tập 1

Cài tiến SLC và VIM

Bạn hãy cài tiến chương trình dịch SLC và chương trình thông dịch mô phỏng máy tính ảo VIM sao cho không cần các file chứa bảng địa chỉ và file chứa bảng độ dài (chỉ cần một file đích duy nhất).

Hướng dẫn: Để bỏ được file bảng độ dài và file bảng địa chỉ bạn phải thêm một lần duyệt sinh mã nữa.

Bài tập 2

Kết hợp SLC và VIM

Bạn hãy kết hợp chương trình dịch SLC và chương trình thông dịch mô phỏng máy tính ảo VIM thành một hệ thống dịch - chạy thống nhất (không cần ghi ra file).

Bài tập 3

Mở rộng SLANG

Bạn hãy tự mở rộng SLANG có những khả năng sau và viết chương trình dịch cho nó (không nhất thiết phải thêm mọi khả năng):

- Hàm.
- Con trỏ.
- Bản ghi..

- Số thực (dấu chấm tĩnh).

- Xâu.

Bài tập 4

Định vị lỗi

Bạn hãy tổ chức lại chương trình dịch cho SLANG sao cho khi báo lỗi, chương trình sẽ in thêm số dòng và số cột của nơi gây ra lỗi.

Bài tập 5

Thêm khả năng gỡ rối (debug) cho VIM

Hướng dẫn: Bạn hãy thêm các khả năng vào chương trình mô phỏng VIM sao cho có thể chạy từng lệnh của chương trình mã VIM và hiện tất cả các thông tin cần thiết phục vụ cho việc phát hiện lỗi sai (gỡ rối).

Bài tập 6

Thêm khả năng gỡ rối (debug) cho SLC

Hướng dẫn: Bạn hãy tham khảo bài tập 2 và bài tập 5, cải tiến SLC sao cho người dùng có thể chạy và theo dõi (debug) các lệnh của SLANG (bài tập trên là chạy và theo dõi các lệnh của VIM).

Bài tập 7

Thêm thanh ghi cho VIM

Bạn hãy thêm các thanh ghi số học AX, BX, CX, DX cho máy tính ảo VIM và các chỉ thị làm việc với các thanh ghi này. Sau đó, bạn cải tiến SLC để nó sinh các chỉ thị dùng thanh ghi sao cho dùng tối đa chúng.

Hướng dẫn: Đối với máy tính thật, thêm thanh ghi là thêm khả năng tăng tốc độ tính toán do thời gian truy cập và thực hiện phép toán nhanh hơn với bộ nhớ (tuy điều này không thật đúng với máy tính VIM do được mô phỏng bằng phần mềm, nhưng việc thêm thanh ghi vào máy tính ảo VIM làm nó giống thật hơn). Bạn phải tìm mọi cách dùng tối đa các chỉ thị này. Bạn cũng nên tham khảo các chỉ thị của bộ vi xử lý 80x86 để biết cách sáng tạo và sử dụng các chỉ thị làm việc với thanh ghi.

Bài tập 8

Dịch ra mã máy IBM -PC

Dịch các chương trình SLANG sao cho chương trình đích có thể chạy trực tiếp trong hệ điều hành MS - DOS (hoặc một hệ điều hành bất kì) trên các máy tính họ 80x86 (hoặc một máy tính thực sự bất kì).

Hướng dẫn: Bạn cần sửa phần sinh mã sao cho có thể sinh trực tiếp mã 80x86. Có nhiều công việc được thực hiện bằng một chỉ thị lệnh trong VIM thì bạn phải thực hiện bằng nhiều chỉ thị lệnh trong 80x86 (và ngược lại). Bạn cũng cần tìm hiểu cách tổ chức file chạy của MS - DOS (dạng file COM là đơn giản nhất).

Bài tập 9

Phân tích từ vựng dùng DFA

Bạn hãy viết phần phân tích từ vựng của SLC theo phương pháp mô phỏng ôtômat hữu hạn đơn định (DFA).

Bài tập 10

Phân tích cú pháp dùng PDA

Bạn hãy viết phần phân tích cú pháp của SLC theo phương pháp mô phỏng ôtômat đầy xuống (PDA).

Hướng dẫn: Bạn hãy viết chương trình mô phỏng ôtômat đầy xuống dùng cho văn phạm LL (1). Mấu chốt ở đây là bạn phải mã hoá được các sản xuất của SLANG thành bảng phân tích. Đầu ra là cây phân tích ở dạng ẩn.

Bài tập 11

Cú pháp điều khiển

Bạn hãy viết phần duyệt cú pháp điều khiển của SLC sau khi nhận được cây phân tích từ bài tập trên.

Bài tập 12

Xây dựng một hệ thống dịch hoàn chỉnh cho ngôn ngữ SLANG

Hướng dẫn: Bạn hãy lấy kết quả thu được từ những bài tập trên, thêm các phần soạn thảo, menu, bảng chọn... để thành một hệ thống hoàn chỉnh: soạn thảo, chương trình -dịch-chạy-gõ rồi.

II. CÁC BÀI TẬP VIẾT CHƯƠNG TRÌNH DỊCH

Bài tập 13

Viết một chương trình biên dịch hoàn chỉnh cho một ngôn ngữ bậc cao

Tìm hiểu hoặc thiết kế mới một ngôn ngữ lập trình, sau đó viết chương trình dịch cho nó (cho bất cứ ngôn ngữ lập trình nào và bằng bất cứ công cụ lập trình gì cũng được).

Hướng dẫn: Các thông tin về một số ngôn ngữ lập trình cho trong phần phụ lục A.

Bài tập 14

Viết chương trình thông dịch mô phỏng bộ vi xử lý họ 80x86

Hướng dẫn: Bạn hãy chọn mô phỏng bộ vi xử lý 8088 (tiền bối của 80x86) do tập thanh ghi và tập lệnh nhỏ. Bạn cũng có thể giới hạn bót tập lệnh chỉ bao gồm những lệnh thường dùng.

Bài tập 15

Viết chương trình dịch Assembler

Viết chương trình chuyển đổi các chương trình nguồn Assembly sang mã máy (tạo file COM chạy trên hệ điều hành MS DOS).

Hướng dẫn: Bạn hãy tự giới hạn trong các lệnh thường dùng, chương trình nguồn nhỏ và không dùng các chỉ thị điều khiển dịch (các lệnh nói cho chương trình dịch điều chỉnh cách làm việc) và bỏ đi một số cấu trúc quá phức tạp.

Lời khuyên: Bạn hãy tham khảo các chương trình Assember như MASM, TASM,... và lệnh A, U của chương trình Debug.exe.

Bài tập 16

Viết chương trình UnAssembler

Viết chương trình chuyển đổi các chương trình mã máy (dạng file COM chạy trên máy IBM -PC) thành chương trình nguồn Assembly.

Hướng dẫn: Bạn hãy giới hạn chương trình mã máy nhỏ và chỉ có các lệnh bạn biết.

Lời khuyên: Tham khảo lệnh A, U của Debug; chương trình dịch ngược Sour.exe.

Bài tập 17

Viết chương trình thông dịch cho BASIC (hoặc một ngôn ngữ lập trình loại thông dịch bất kỳ)

Hướng dẫn: Bạn hãy tự tìm hiểu ngôn ngữ lập trình BASIC (hoặc một ngôn ngữ lập trình loại thông dịch bất kỳ) và viết chương trình thông dịch cho nó. Tham khảo thêm bài tập 23.

Bài tập 18

Chương trình chuyển đổi từ Pascal sang C

Viết chương trình chuyển đổi một chương trình nguồn viết bằng Pascal sang C (hoặc ngược lại).

Hướng dẫn: Xây dựng một chương trình chuyển đổi hoàn chỉnh giữa hai ngôn ngữ rất khó. Bạn hãy tự giới hạn trong những cấu trúc ngôn ngữ thông thường và bỏ không chuyển những cấu trúc quá phức tạp.

Bài tập 19

Cũng như bài tập trên nhưng cho các cặp ngôn ngữ lập trình bất kỳ do bạn tự chọn.

Bài tập 20

Viết chương trình sinh phần phân tích từ vựng.

Hướng dẫn: Thông thường, để dịch một ngôn ngữ lập trình, ta phải viết phần phân tích từ vựng cho nó. Khi viết chương trình dịch cho một ngôn ngữ lập trình khác, ta lại phải viết lại phần này. Cách hay nhất là ta viết một chương trình đặc biệt (gọi là chương trình sinh phân tích từ vựng) để tạo tự động phần phân tích từ vựng. Để có chương trình phân tích từ vựng cho một ngôn ngữ lập trình nào đó, ta chỉ cần cho vào chương trình sinh đặc tả từ vựng của ngôn ngữ đó (các mẫu và luật cấu thành từ vựng), đầu ra sẽ là chương trình phân tích từ vựng cho đúng ngôn ngữ lập trình này (thường ở dạng nguồn).

Bài tập 21

Viết chương trình sinh phần phân tích cú pháp.

Hướng dẫn: Thông thường, để dịch một ngôn ngữ lập trình, ta phải viết phần phân tích cú pháp cho nó. Khi viết chương trình dịch cho một ngôn

ngữ lập trình khác, ta lại phải viết lại phần này. Cách hay nhất là ta viết một chương trình đặc biệt (gọi là chương trình sinh phân tích cú pháp) để tạo tự động phần phân tích cú pháp. Để có chương trình phân tích cú pháp cho một ngôn ngữ lập trình nào đó, ta chỉ cần cho vào chương trình sinh đặc tả cú pháp của ngôn ngữ đó (các luật cú pháp), đầu ra sẽ là chương trình phân tích cú pháp cho đúng ngôn ngữ lập trình này (thường ở dạng nguồn).

Bài tập 22

Viết chương trình dịch chương trình dịch (compiler compiler).

Giải thích: Bài tập này là mở rộng và hoàn thiện của hai bài tập trên. Các kết quả thu được sẽ gộp với một số phần nữa như phân tích ngữ nghĩa, sinh mã để thành một chương trình dịch hoàn chỉnh cho một ngôn ngữ lập trình.

III. CÁC ỨNG DỤNG KHÁC CỦA CHƯƠNG TRÌNH DỊCH

Bài tập 23

Mô phỏng hệ điều hành MS DOS.

Viết một chương trình thông dịch giả (mô phỏng) một số lệnh của hệ điều hành MS DOS (dùng cho mục đích học thực hành sử dụng hệ điều hành này).

Hướng dẫn: Quan sát nhiều lệnh của hệ điều hành này trong các sách học, ta thấy chúng dùng dạng ký hiệu BNF và thực chất là một ngôn ngữ phi ngữ cảnh. Ví dụ hướng dẫn về lệnh COPY và REN như sau (xem bảng cách gọi COPY /?, REN/? tại dấu nháy của MSDOS từ ver 5 trở lên hoặc trong chương trình HELP có sẵn trong thư mục DOS):

COPY [/A | /B] source [/A | /B] [+ source [/A | /B] [+ ...]] [destination
[/A | /B]] [/V] [/Y | /-Y]

REN [drive:][path][directoryname1 | filename1] [directoryname2 |
filename2]

Để thực hiện nhiệm vụ, bạn cần phải viết các phần phân tích từ vựng, phân tích cú pháp và phân tích ngữ nghĩa cho ngôn ngữ này. Sau khi một câu lệnh được xử lý, nếu sai nó sẽ đưa được các thông báo thích hợp; nếu đúng, nó sẽ gọi các chức năng cần thiết để thực hiện nhiệm vụ chi tiết. Tham khảo thêm bài tập 17.

Bài tập 24

Hiện chương trình nguồn theo cú pháp

Bạn hãy viết một chương trình đọc và hiện chương trình nguồn viết trong một ngôn ngữ lập trình bất kì. Hiện những từ /cụm từ làm những nhiệm vụ khác nhau bằng những màu khác nhau. Ví dụ: từ khoá màu xanh biển, chú thích màu xám, chõ sai màu đỏ, những phần khác màu vàng.

Hướng dẫn: Bạn hãy tham khảo phụ lục B.

Bài tập 25

Viết chương trình chuyển đổi file văn bản dạng RTF sang HTML (hoặc ngược lại).

Hướng dẫn: Các file dạng RTF và HTML là các file dạng văn bản (text) có thêm các thông tin bổ sung làm định dạng cho văn bản và được viết như một ngôn ngữ lập trình (RTF, HTML đều được gọi là các ngôn ngữ mô tả trang). Bài tập này đòi hỏi bạn phải biết nhiều về cấu trúc RTF và HTML và viết được phần phân tích từ vựng và cú pháp cho chúng. Bạn hãy tham khảo phần phụ lục B.

Bài tập 26

Viết chương trình chuyển đổi file văn bản giữa các cặp ngôn ngữ mô tả trang RTF, HTML, TEX, PostScript.

Hướng dẫn: Giống bài tập trên nhưng bạn được chọn giữa các cặp ngôn ngữ mô tả trang bất kì.

Bài tập 27

Viết chương trình chuyển đổi file văn bản của VNI (DOS), BKED, VietRes sang dạng RTF, HTML, TEX, PostScript (các ngôn ngữ mô tả trang) và ngược lại.

Hướng dẫn: Bạn chỉ cần chọn một loại văn bản tiếng Việt và một ngôn ngữ mô tả trang để thực hiện công việc. Yêu cầu ở đây là phải giữ được tối đa các định dạng chữ (đổi sang mã tiếng Việt thích hợp, giữ được những chõ in đậm, nghiêng...).

Bài tập 28

Hiện nội dung file dạng RTF, HTML, TEX

Viết chương trình hiện nội dung văn bản của những file dạng RTF (hoặc HTML, TEX) theo đúng định dạng của nó.

Hướng dẫn: Bài tập này đòi hỏi bạn phải biết nhiều về cấu trúc RTF (hoặc HTML, TEX) và viết được phần phân tích từ vựng cho nó. Chỗ khó của bài này là bạn cần nắm được nhiều kĩ thuật đồ họa dùng để thể hiện các kiểu font chữ và kích cỡ khác nhau. Bạn hãy tham khảo phần phụ lục B.

Lời khuyên: Nếu bạn biết lập trình trong MS -Windows thì công việc sẽ dễ hơn.

Bài tập 29

Trình bày lại chương trình nguồn cho đẹp.

Hướng dẫn: Các chương trình nguồn viết ra thường không được trình bày đẹp và thống nhất. Ví dụ, khoảng cách giữa các chữ, ký hiệu rất tuỳ tiện (dạng $a:= b$ và $a:=b$), chỗ xuống dòng, thụt đầu dòng cũng không đúng hoặc không thống nhất. Bạn cần đề ra các quy tắc thế nào là một chương trình nguồn được trình bày đẹp và viết chương trình để căn chỉnh theo nó (chú ý là các chỗ thụt đầu dòng khác nhau tuỳ theo cấp độ lồng nhau, bạn cần phải phân tích cú pháp mới làm được đúng).

Bài tập 30

Xử lý ngôn ngữ tự nhiên

Hướng dẫn: Có rất nhiều bài toán và mức độ xử lý ngôn ngữ tự nhiên rất khác nhau. Ở đây, bạn được yêu cầu áp dụng những kĩ thuật của chương trình dịch để xử lý chúng. Ví dụ, để kiểm tra xem một câu trong một ngôn ngữ (như tiếng Anh) có các từ vựng viết đúng không ta cần áp dụng nhiều các kĩ thuật phân tích từ vựng (chú ý là mẫu trong trường hợp này là thông tin chứa trong từ điển) và có thể rút ra từ loại của các từ đó. Để kiểm tra xem một câu có đúng ngữ pháp không, từ các từ loại thu được và các luật ngữ pháp được đưa vào bộ phân tích cú pháp và kết quả chính là câu trả lời.

Phụ lục A

MÔ TẢ MỘT SỐ NGÔN NGỮ LẬP TRÌNH

Trong phần phụ lục này bạn sẽ được cung cấp văn phạm của một số ngôn ngữ lập trình để làm cơ sở tham khảo.

Chú ý là các quy ước ở đây hơi khác nhau và khác với các quy ước trong các chương của giáo trình. Các từ in nghiêng là các ký hiệu không kết thúc (biến). Các từ in đậm và các ký hiệu như +, -, *, /, =... là các từ tố do bộ phân tích từ vựng trả về. Cách viết trực tiếp các ký hiệu này giúp cho các luật văn phạm trông gọn và dễ hiểu hơn (nên thường được dùng trong các sách nói về ngôn ngữ lập trình). Một số ký hiệu khác như →, ::=, {, }, | ... là các ký hiệu của BNF (đây là một biến thể khác của BNF - không phải là BNF theo gốc ban đầu).

Ở đây cũng chỉ có các luật cú pháp mà không có các ràng buộc và các luật ngữ nghĩa. Bạn có thể dựa vào kinh nghiệm để tự mình đề ra các ràng buộc và luật ngữ nghĩa thích hợp cho chúng.

1. PL/0

Văn phạm PL/0 là văn phạm đơn giản nhất được giới thiệu ở đây. Nó đơn giản hơn nhiều so với văn phạm SLANG và được biết là loại LL (1) (bạn hãy tự kiểm chứng điều này). Tuy vậy, PL/0 là một ngôn ngữ thủ tục hoàn chỉnh, có tương đối đủ các cấu trúc cơ bản. Nó rất thích hợp cho những dự án viết chương trình dịch cỡ nhỏ và trung. Văn phạm PL/0 được cho ở hai dạng khác nhau: các luật sản xuất và các đồ thị chuyển.

chươngtrình →
khối

khối →
khaibáohàng
khaibáobiển

các chương trình con
các câu lệnh tổng hợp

danh sách tên →

tên

| danh sách tên, tên

hằng →

tên = số

| hằng, tên = số

khai báo hằng →

const hằng ;

| ε

khai báo biến →

var danh sách tên ;

| ε

các chương trình con →

procedure tên ; khối ;

các chương trình con procedure tên ; khối ;

| ε

các lệnh tổng hợp →

câu lệnh

| câu lệnh tổng hợp; câu lệnh

| ε

câu lệnh →

biến phép gán biến thức

| call tên

| các lệnh tổng hợp

| if điều kiện then câu lệnh

| while điều kiện do câu lệnh

điều kiện →

biến thức = biến thức

| biến thức <> biến thức

| biến thức < biến thức

| biến thức > biến thức

| $biểu\ thức \leq biểu\ thức$
 | $biểu\ thức \geq biểu\ thức$

$biểu\ thức \rightarrow$
 dấu $biểu\ thức$

$biểu\ thức$ →
 hạng thức
 | $biểu\ thức$ **công** **trù** $hạng\ thức$

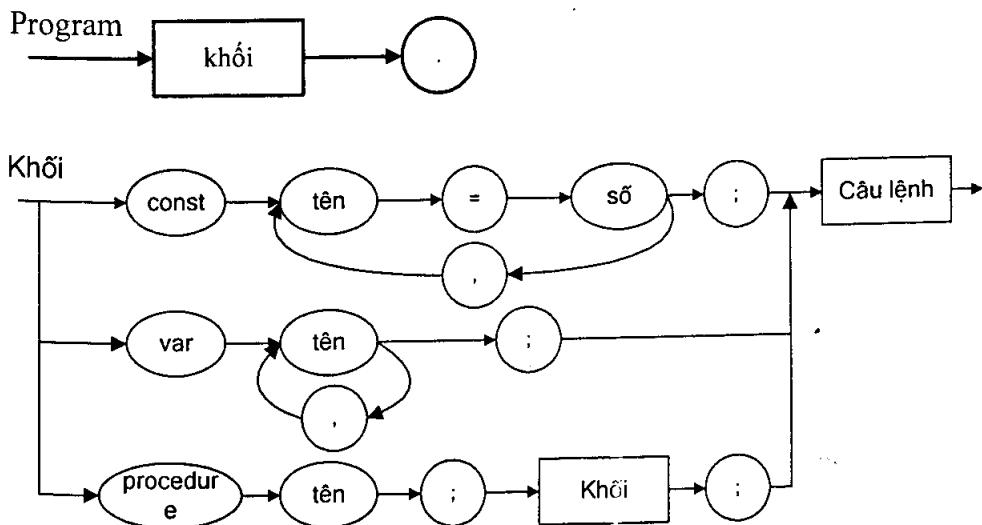
$hạng\ thức \rightarrow$
 thùatô
 | $hạng\ thức$ **nhân** **chia** thùatô

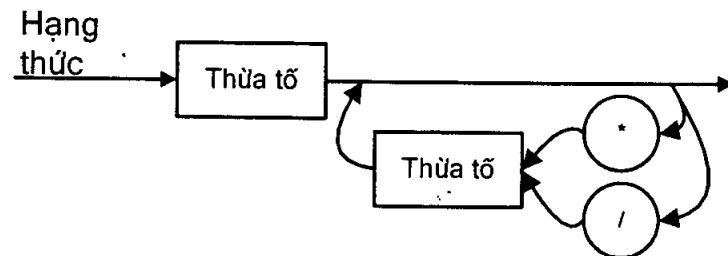
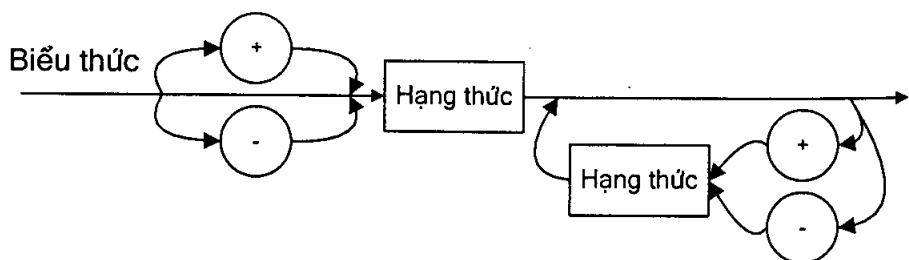
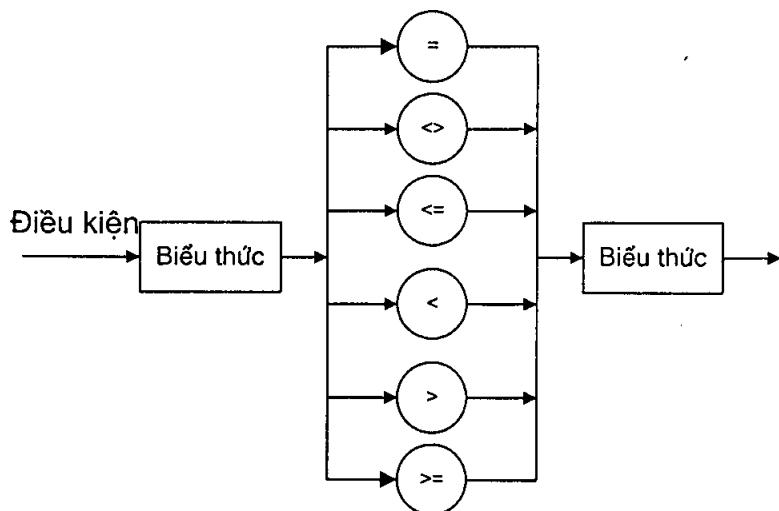
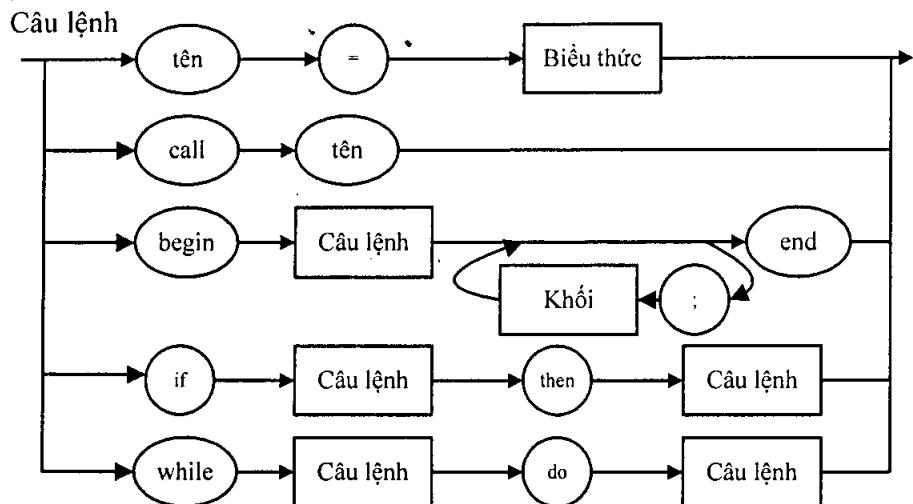
$thùatô \rightarrow$
 tên
 | số
 | ($biểu\ thức$)

$dấu \rightarrow$
 + | -

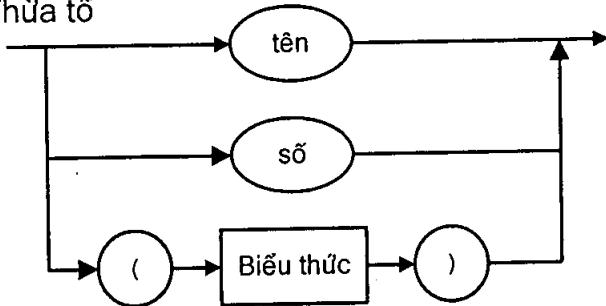
Biểu đồ chuyển của văn phạm này như sau:

Các khói tròn là các ký hiệu kết thúc, các khói vuông là các ký hiệu không kết thúc.





Thừa tố



Văn phạm PL /0 cho phép viết được các chương trình đơn giản như ví dụ sau:

```
Const m = 7, n = 82;  
Var x, y, z, q, r;  
Procedure Multiply;  
Var a, b;  
begin  
    a := b; b := y; z := 0;  
    while b > 0 do  
        begin  
            if b = a then z := z + a;  
            a := 2 * a; b := b /2  
        end;  
    end;  
Procedure Divice;  
Var w;  
begin  
    r := x; q := 0; w := y;  
    while w <= r do w := 2 * w;  
    while w > y do  
        begin  
            q := 2 * q; w := w / 2;  
            if w <= z then  
                begin  
                    r := r - w; q := q + 1  
                end;  
        end;  
    end;  
begin  
    x := m; y := n; call multiply;  
    x := 25; y := 0; call Divide;  
end.
```

2. Văn phạm Pascal -S

Pascal-S là một ngôn ngữ lập trình dạng tập con của Pascal chuẩn. Do vậy nó có được các cấu trúc, phong cách, khả năng gần giống với Pascal (chỉ trừ một số cấu trúc quá phức tạp bị bỏ đi). Nó rất thích hợp cho những dự án viết chương trình dịch cỡ trung và lớn. Chú ý là văn phạm này là LALR (1). Nếu bạn muốn phân tích theo kiểu LL₁(1) (hoặc một phương pháp phân tích bất kì) thì phải kiểm tra và sửa lại nếu cần thiết.

program →

program **tên** (*danh sách tên*) ;
các khai báo
các khai báo chương trình con
các lệnh tổng hợp

danh sách tên →

tên
| *danh sách tên*, **tên**

các khai báo →

các khai báo **var** *danh sách tên* : *kiểu*;
| ε

kiểu →

kiểu chuẩn
| **array** [*số* .. *số*] **of** *kiểu chuẩn*

kiểu chuẩn →

integer
| **real**

các khai báo chương trình con →

các khai báo chương trình con *khai báo chương trình con*;

khai báo chương trình con →

đầu chương trình con *các khai báo* *các lệnh tổng hợp*

đầu chương trình con →

function **tên** *các tham số*: *kiểu chuẩn* ;
| **procedure** **tên** *các tham số*;

cácthamsô →
(*danh sach thamsô*)
| ε

danh sach thamsô →
danh sach ten : *kiieu*
| *danh sach thamsô*; *danh sach ten*: *kiieu*

cac lenh tong hop →
begin
cac lenh tong hop_tuy co
end

cac lenh tong hop_tuy co →
danh sach cau lenh
| ε

danh sach cau lenh →
cau lenh
| *danh sach cau lenh*; *cau lenh*

cau lenh →
biến **phép gán** *bieuthuc*
| *cau lenh_thu tuc*
| *cac lenh tong hop*
| **if** *bieuthuc* **then** *cau lenh* **else** *cau lenh*
| **while** *bieuthuc* **do** *cau lenh*

biến →
tên
| **tên** [*bieuthuc*]

cau lenh_thu tuc →
tên
| **tên** (*danh sach bieuthuc*)

danh sach bieuthuc →
bieuthuc
| *danh sach bieuthuc*, *bieuthuc*

biểu thức →

biểu thức đơn

| *biểu thức đơn* quanh *hệ biểu thức đơn*

biểu thức đơn →

hạng thức

| *dấu hạng thức*

| *biểu thức đơn cộng trừ hạng thức*

hạng thức →

thìatô

| *hạng thức nhanchia thìatô*

thìatô →

tên

| *tên* (*danh sách biểu thức*)

| *số*

| (*biểu thức*)

| *not* *thìatô*

dấu →

+ | -

Văn phạm này cho phép viết được các chương trình đơn giản như ví dụ sau:

```
program example(input, output);
var x, y: integer;
{ Hàm con gọi đệ quy }
function gcd(a, b: integer): integer;
begin
    if b = 0 then gcd := a
    else gcd := gcd(b, a mod b)
end;

begin
    read(x, y);
    write(gcd(x, y))
end.
```

3. Pascal

Sau đây là cú pháp của ngôn ngữ Pascal thực sự. Ngôn ngữ này do N.Wirth phát triển vào cuối những năm 1960 và rất quen thuộc với chúng ta. Nó rất thích hợp cho những dự án viết chương trình dịch cỡ lớn.

program → *program_heading* ; *program_block*
program_block → *block*
program_heading → **program** *identifier* (*file_identifier* { , *file_identifier* }) ;
file_identifier → *identifier*
identifier → *letter* { *letter_or_digit* }
letter_or_digit → *letter* | *digit*
letter → a | b | ... | z
digit → 0 | 1 | ... | 9
block → *label_declaration_part* *constant_definition_part*
type_definition_part
 variable_declaration_part
procedure_and_function_declaration_part
 statement_part
label_declaration_part → empty | **label** *label* { , *label* } ;
label → *unsigned_integer*
constant_definition_part → empty | **const** *constant_definition* { ;
 constant_definition } ;
constant_definition → *identifier* = *constant*
constant → *unsigned_number* | *sign* *unsigned_number* | *constant_identifier*
 | *sign* *constant_identifier* | *string*
unsigned_number → *unsigned_integer* | *unsigned_real*
unsigned_integer → *digit* { *digit* }
unsigned_real → *unsigned_integer*. *digit* { *digit* }
 | *unsigned_integer*. *digit* { *digit* } E *scale_factor*
 | *unsigned_integer* E *scale_factor*
scale_factor → *unsigned_integer* | *sign* *unsigned_integer*
sign → + | -
constant_identifier → *identifier*
string → ' *character* { *character* } '
type_definition_part → empty | **type** *type_definition* { ; *type_definition* } ;
type_definition → *identifier* = *type*
type → *simple_type* | *structured_type* | *pointer_type*
simple_type → *scalar_type* | *subrange_type* | *type_identifier*
scalar_type → *identifier* { , *identifier* } ;

`subrange_type → constant .. constant`
 `type_identifier → identifier`
 `structured_type → unpacked_structured_type | packed`
 `unpacked_structured_type`
 `unpacked_structured_type → array_type | record_type | set_type | file_type`
 `array_type → array [index_type { , index_type }] of component_type`
 `index_type → simple_type`
 `component_type → type`
 `record_type → record field_list end`
 `field_list → fixed_part | fixed_part ; variant_part | variant_part`
 `fixed_part → record_section { ; record_section }`
 `record_section → field_identifier { , field_identifier } : type | empty`
 `variant_part → case tag_field type_identifier of variant { ; variant }`
 `tag_field → field_identifier : | empty`
 `variant → case_label_list : (field_list) | empty`
 `case_label_list → case_label { , case_label }`
 `case_label → constant`
 `set_type → set of base_type`
 `base_type → simple_type`
 `file_type → file of type`
 `pointer_type → ^ type_identifier`
 `variable_declaration_part → empty | var variable_declaration { ; variable_declaration }`
 `variable_declaration → identifier { , identifier } : type`
 `procedure_and_function_declaration_part → {procedure_and_function_declaration ; }`
 `procedure_and_function_declaration → procedure_declaration | function_declaration`
 `procedure_declaration → procedure_heading block`
 `procedure_heading → procedure identifier ;`
 `| procedure identifier (formal_parameter_section { , formal_parameter_section }) ;`
 `formal_parameter_section → parameter_group | var parameter_group`
 `| function parameter_group`
 `| procedure identifier { , identifier }`
 `parameter_group → identifier { , identifier } : type_identifier`
 `function_declaration → function_heading block`
 `function_heading → function identifier : result_type ;`
 `| function identifier (formal_parameter_section`

```

{ , formal_parameter_section } ) : result_type.
;

result_type → type_identifier
statement_part → compound_statement
statement → unlabeled_statement | label : unlabeled_statement
unlabeled_statement → simple_statement | structured_statement
simple_statement → assignment_statement | procedure_statement
| goto_statement | empty_statement
assignment_statement → variable := expression
| function_identifier := expression
variable → entire_variable | component_variable | referenced_variable
entire_variable → variable_identifier
variable_identifier → identifier
component_variable → indexed_variable | field_designator | file_buffer
indexed_variable → array_variable [ expression { , expression } ]
array_variable → variable
field_designator → record_variable . field_identifier
record_variable → variable
field_identifier → variable
file_buffer → file_variable ^
file_variable → variable
referenced_variable → pointer_variable ^
pointer_variable → variable
expression → simple_expression
| simple_expression relational_operator simple_expression
relational_operator → = | <> | < | <= | >= | > | in
simple_expression → term | sign term | simple_expression adding_operator
term
adding_operator → + | - | or
term → factor | term multiplying_operator factor
multiplying_operator → * | / | div | mod | and
factor → variable | unsigned_constant | ( expression ) | function_designator
| set | not factor
unsigned_constant → unsigned_number | string | constant_identifier | nil
function_designator → function_identifier
| function_identifier ( actual_parameter { , actual_parameter } )
function_identifier → identifier
set → [ element_list ]
element_list → element { , element } | empty
element → expression | expression .. expression

```

$\text{procedure_statement} \rightarrow \text{procedure_identifier}$
 $| \text{procedure_identifier} (\text{actual_parameter} \{ , \text{actual_parameter} \})$
 $\text{procedure_identifier} \rightarrow \text{identifier}$
 $\text{actual_parameter} \rightarrow \text{expression} | \text{variable}$
 $| \text{procedure_identifier} | \text{function_identifier}$
 $\text{go_to_statement} \rightarrow \text{goto label}$
 $\text{empty_statement} \rightarrow \text{empty}$
 $\text{empty} \rightarrow \epsilon$
 $\text{structured_statement} \rightarrow \text{compound_statement} | \text{conditional_statement}$
 $| \text{repetitive_statement} | \text{with_statement}$
 $\text{compound_statement} \rightarrow \text{begin statement} \{ ; \text{statement} \} \text{ end}$
 $\text{conditional_statement} \rightarrow \text{if_statement} | \text{case_statement}$
 $\text{if_statement} \rightarrow \text{if expression then statement}$
 $| \text{if expression then statement else statement}$
 $\text{case_statement} \rightarrow \text{case expression of case_list_element} \{ ; \text{case_list_element} \} \text{ end}$
 $\text{case_list_element} \rightarrow \text{case_label_list} : \text{statement} | \text{empty}$
 $\text{case_label_list} \rightarrow \text{case_label} \{ , \text{case_label} \}$
 $\text{repetitive_statement} \rightarrow \text{while_statement} | \text{repeat_statement} | \text{for_statement}$
 $\text{while_statement} \rightarrow \text{while expression do statement}$
 $\text{repeat_statement} \rightarrow \text{repeat statement} \{ ; \text{statement} \} \text{ until expression}$
 $\text{for_statement} \rightarrow \text{for control_variable} := \text{for_list} \text{ do statement}$
 $\text{for_list} \rightarrow \text{initial_value to final_value} | \text{initial_value downto final_value}$
 $\text{control_variable} \rightarrow \text{identifier}$
 $\text{initial_value} \rightarrow \text{expression}$
 $\text{final_value} \rightarrow \text{expression}$
 $\text{with_statement} \rightarrow \text{with record_variable_list do statement}$
 $\text{record_variable_list} \rightarrow \text{record_variable} \{ , \text{record_variable} \}$

4. Oard

Phần này sẽ mô tả về ngôn ngữ lập trình Oard. Chúng tôi sẽ cố gắng cung cấp đủ các chi tiết để bạn có thể xây dựng được một chương trình dịch cho nó.

Oard là một ngôn ngữ lập trình dạng đơn giản của Pascal, cho phép làm việc với số nguyên, số thực và các xâu đơn giản. Nó có một số đặc tính giúp cho người dùng có thể viết được những chương trình có ích và là một minh họa rất tốt cho các ngôn ngữ lập trình hiện đại. Nó tránh dùng những thứ

phức tạp như cấu trúc khối nhung vẫn cho viết các thủ tục đệ quy mà đó thường là những vấn đề cơ bản của việc thiết kế chương trình dịch.

Oard có ba kiểu dữ liệu cơ sở: số nguyên, số thực dấu chấm động và ký tự. Mỗi kiểu này có thể được dùng trong mảng một chiều.

Các cấu trúc điều khiển trong Oard được giới hạn nhiều. Nó có câu lệnh if, while và câu lệnh tổng hợp. Các thủ tục có thể đệ quy và chúng có thể được khai báo với chương trình chính nhưng không trong thủ tục khác.

Chú ý là các ký hiệu dùng trong định nghĩa Oard dưới đây được giữ nguyên theo thực tế. Bạn hãy tự tìm hiểu ý nghĩa của chúng.

Các đặc tính từ vựng của Oard:

1. Trong Oard, các khoảng trắng cũng quan trọng.

2. Trong Oard, các từ khoá luôn được viết hoa. Các từ khoá bao gồm:

AND, ARRAY, BEGIN, CHARACTER, DO, ELSE, END, FLOAT,
FUNCTION, IF, INTEGER, MOD, NOT, OF, OR, PROCEDURE,
PROGRAM, READ, RETURN, THEN, VAR, WHILE, WRITE.

Chú ý: do phân biệt chữ hoa nên END là từ khoá nhưng end có thể dùng làm một biến.

3. Còn sau đây là một số ký hiệu có nghĩa (xem định nghĩa văn phạm để hiểu ý nghĩa)

{ } ' < > = + - * [] () . , ; :

4. Các chú thích được đặt giữa cặp dấu {} và bị xoá bỏ. Các chú thích có thể xuất hiện ở mọi nơi.

5. Tên được dùng với cả chữ hoa và số và được định nghĩa như sau:

<letter> ::= a | b | c | ... | z | A | B | ... | Z

<digit> ::= 0 | 1 | 2 | ... | 9

<identifier> ::= <letter> (<letter> | <digit>)*

Khi thực hiện, bạn có thể giới hạn độ dài của một tên không quá 31 ký tự.

6. Hằng số được định nghĩa như sau:

<constant> ::= <intnum> | <floatnum> | <char const>

<positive> ::= 1 | 2 | 3 | ... | 9

<sign> ::= + | - |

<intnum> ::= <positive> <digit>* | 0

```

<floatnum> ::= <intnum> .
    | <intnum> . <intnum>
    | <intnum> . E <sign&ft; <intnum>
    | <intnum> . <intnum> E <sign&ft; <intnum>
<char const> ::= '<letter>'
```

Các hàng xâu được dùng trong câu lệnh WRITE:

```
<string const> ::= '<letter>*''
```

7. Các toán tử quan hệ được định nghĩa như sau:

```
<relop> ::= < | <= | >= | > | = | <>
```

8. Các phép toán:

```

<addop> ::= + | -
<mulop> ::= * | MOD
<logop> ::= OR | AND
```

Cú pháp của Oard

Phần này là các mô tả cú pháp của Oard ở dạng BNF.

```

<program>   ::=
    PROGRAM <identifier>; <decls> <subprogram decls>
    <compound statement>
<decls>      ::=
    VAR <decl list>
    | <empty>
<decl list>   ::=
    <identifier list> : <type> ;
    | <decl list> <identifier list> : <type> ;
<identifier list> ::=
    <identifier>
    | <identifier list> , <identifier>
<type>        ::=
    <standard type>
    | <array type>
<standard type> ::=
    INTEGER
    | FLOAT
    | CHARACTER
```

```

<array type> ::= ARRAY [ <dim> ] OF <standard type>
<dim> ::= <intnum> .. <intnum>
| <char constant> .. <char constant>
<subprogram decls> ::= <subprogram decls> <subprogram decl> ;
| <empty>
<subprogram decl> ::= <subprogram head> <decls> <compound statement>
<subprogram head> ::= FUNCTION <identifier> <arguments>:} <standard type>
{ \tt ;
| PROCEDURE <identifier> <arguments> ;
<arguments> ::= () <parameter list> { \tt )
| <empty>
<parameter list> ::= <identifier list> : <type>
| <parameter list> ; <identifier list> : <type>
<statement> ::= <assignment>
| <if statement>
| <while statement>
| <procedure invocation>
| <i-o statement>
| <compound statement>
| <return statement>
<assignment> ::= <variable> := <expr>
<if statement> ::= IF <expr> THEN <restricted statement> ELSE <statement>
| IF <expr> THEN <statement>
<restricted statement> ::= <assignment>
| <while statement>
| <i-o statement>
| <procedure invocation>

```

- | <compound statement>
- | <return statement>
- | IF <expr> THEN <restricted statement> ELSE <restricted statement>

<while statement> ::=

- WHILE <expr> DO <restricted statement>

<procedure invocation> ::=

- <identifier>()
- | <identifier> (<expr list>)

<i-o statement> ::=

- READ (<variable>)
- | WRITE (<expr>)
- | WRITE (<string constant>)

<compound statement> ::=

- BEGIN <statement list> END

<statement list> ::=

- <statement>
- | <statement list> ; <statement>

<return statement> ::=

- RETURN <expr>

<expr list> ::=

- <expr>
- | <expr list> , <expr>

<expr> ::=

- <simple expr>
- | <expr> <logop> <simple expr>
- | NOT <simple expr>

<simple expr> ::=

- <add expr>
- | <simple expr> <relop> <add expr>

<add expr> ::=

- <mul expr>
- | <add expr> <addop> <mul expr>

<mul expr> ::=

- <factor>
- | <mul expr> <mulop> <factor>

<factor> ::=

- <variable>

```

| <constant>
| ( <expr> )
| <procedure invocation>
<variable> ::= 
    <identifier>
    | <identifier> [ <expr> ]

```

Ví dụ

Dưới đây là một ví dụ về chương trình viết trong Oard. Chương trình này sẽ đọc một cặp số nguyên vào và in ra ước số chung lớn nhất của chúng.

```

PROGRAM example;
VAR x, y : INTEGER;
FUNCTION gcd (a,b: INTEGER):INTEGER;
BEGIN
  IF b=0 THEN
    RETURN a
  ELSE
    RETURN gcd(b, a MOD b)
END;

BEGIN
  READ (x);
  READ (y);
  WHILE (x <> 0) OR (y <> 0) DO
    BEGIN
      WRITE (gcd (x,y));
      READ (x);
      READ (y)
    END
  END

```

Phụ lục B

ĐỊNH DẠNG CHƯƠNG TRÌNH NGUỒN trong MS-word

I. ĐẶT VÂN ĐỀ

Để in (hoặc xem) các chương trình nguồn ở dạng dẽ nhìn, người ta thường in các thành phần (từ hoặc cụm từ) khác nhau với màu sắc hoặc đặc tính khác nhau. Ví dụ, các từ khoá thường in bằng kiểu chữ đậm, các chú thích in nghiêng. Các cấu trúc sai có thể in gạch chân... Một số chương trình dịch có trên thị trường có thể giúp ta in như vậy (như Borland C 3.0, Visial C 4.1...). Việc dùng các công cụ có sẵn này đôi khi có điều bất tiện là bạn buộc phải in chương trình trực tiếp, thẳng ra máy in. Nếu bạn muốn lồng chương trình nguồn vào file văn bản như trong MS Word (nhằm sửa đổi, chú thích, định dạng lại trang in...) để in cùng với các phần văn bản khác thì không được.

Trong phần này, ta sẽ xem cách viết các chương trình giúp ta định dạng lại một chương trình nguồn và đưa vào văn bản MS Word. Các kĩ thuật dùng ở đây chủ yếu dựa vào kĩ thuật xây dựng bộ phân tích từ vựng.

II. THỰC HIỆN

Lý tưởng nhất là ta viết một chương trình dùng để chuyển trực tiếp file chương trình nguồn của một ngôn ngữ lập trình nào đó thành dạng file DOC thông thường của MS Word. Nhưng cách này có một trở ngại là ta rất khó biết được thông tin về cấu trúc của file DOC (và nếu biết thì cấu trúc của file này cũng quá phức tạp). Cách đơn giản hơn là ta chuyển sang file dạng RTF (Rich Text Format) mà Word có thể hiểu được. File dạng RTF là file văn bản (text) có thêm các ký hiệu quy ước để định dạng văn bản. Các quy ước này thực chất tạo cho file RTF có dạng là một ngôn ngữ lập trình hoàn chỉnh.

Ví dụ

Để tạo một file có dòng chữ "XIN CHAO BAN" có chữ "CHAO" in nghiêng, ta tạo một file mới (bằng một công cụ soạn thảo đơn giản bất kì như TP, TC, EDIT...) và gõ vào nội dung như sau:

{\rtf1 XIN {\i CHAO} BAN}

Bạn hãy ghi file này với tên CHAO.RTF. Bây giờ bạn hãy gọi MS Word, mở file (open) và nhập vào tên CHAO.RTF. MS Word sẽ hiện dòng chữ theo đúng mong muốn.

Một số quy ước của file RTF

Số quy ước của RTF có rất nhiều, lên đến hàng trăm. Rất may cho ta là chỉ cần một số quy ước đơn giản cũng thực hiện được nhiệm vụ.

Mở đầu và kết thúc một văn bản dạng RTF ta phải có:

```
\rtf1  
...  
}
```

Để in đậm cần bao quanh chỗ cần in đậm như sau:

```
{\b Nội dung in đậm }
```

Để in nghiêng:

```
{\i Nội dung in nghiêng }
```

Để in gạch chân:

```
{\ul Nội dung in gạch chân }
```

Thay cho các ký hiệu xuống dòng dùng:

```
\par
```

Trong file RTF, các ký hiệu \{ \} đều là các ký hiệu điều khiển. Để giữ nó thành một ký hiệu văn bản ta phải thêm một ký hiệu đánh dấu \ ngay trước chúng. Ví dụ: xâu C:\DOS\A.TXT sẽ được in ra là C:DOS\A.TXT.

Chương trình sau sẽ định dạng các file nguồn C và ghi thành một file khác dạng RTF mà MS -WORD hiểu được. Bạn hãy chú ý những chỗ xử lý chú thích, xâu, hàng chữ là tương đối phức tạp.

```
#include <string.h>  
#include <stdio.h>  
#include <conio.h>  
  
#define max_ident_length 255  
typedef char string[max_ident_length + 1];  
  
char *keyword[] =
```

```

{"asm","auto","break","case","cdecl","char","class","const",
 "continue","default","delete","do","double","else","enum",
 "_export","extern","far","float","for","friend","goto","huge",
 "if","inline","int","interrupt","_loadds","long","near","new",
 "operator","pascal","private","protected","public","register",
 "return","_saveregs","_seg","short","signed","sizeof",
 "static","struct","switch","template","this","typedef",
 "union","unsigned","virtual","void","volatile","while",
 NULL
};

typedef enum {
    true = 1, false = 0
} boolean;

string ident_lexeme;
int ch;
boolean retract;

FILE *ifp, *ofp;

void next_character(void)
{
    ch = fgetc(ifp);
}

int look_up(string key_lexeme)
{
    int i;
    for (i=0; keyword[i] != NULL; i++)
        if (0==strcmp(key_lexeme, keyword[i]))
            return 1;
    return 0;
}

void identifier(void)
{
    int i = 0;

    do {
        ident_lexeme[i] = ch; i++;
        next_character();
    } while (isalpha(ch) || isdigit(ch) || ch=='_');
    ident_lexeme[i] = 0;

    if(look_up(ident_lexeme))
        fprintf(ofp, "\b %s", ident_lexeme);
    else fprintf(ofp, "%s", ident_lexeme);

    retract = true;
}

```

```

}

void comment(void)
{
    char c;

    fprintf(ofp, "\i /*");
    do {
        c = ch;
        next_character();
        if (ch == EOF) return;
        if (ch=='{' || ch=='}') fputc('\', ofp);
        if (ch == 10) fprintf(ofp, "\par "); /* Xuong dong */
        else fputc(ch, ofp);
    } while (c != '*' || ch!='/');

    fputc(ch, ofp); fputc('}', ofp);
}

void pstring(void)
{
    char c;

    c = ch;
    fputc(ch, ofp);
    do {
        next_character();
        if (ch == EOF) return;
        if (ch=='{' || ch=='}') fputc('\', ofp);
        if (ch == 10) fprintf(ofp, "\par "); /* Xuong dong */
        else fputc(ch, ofp);
    } while (ch!=c);
}

void next_token(void)
{
    retract = false;

    while (ch != EOF && !isalpha(ch) && ch != '/'
          && ch != "" && ch != 39) {
        if (ch=='{' || ch=='}') fputc('\', ofp);
        fputc(ch, ofp);
        if (ch==10)      fprintf(ofp, "\par ");
        next_character();
    }

    if (ch=='/') {
        next_character();
        if (ch == '*') comment();
        else { fputc('/', ofp); fputc(ch, ofp); }
    }
}

```

```

} else if (ch==" " || ch==39) {
    pstring();
} else if (isalpha(ch) || ch=='_') identifier();

if(ch==EOF) return;
if (!retract) next_character();
}

void main(void)
{
    char ifilename[50], ofilename[50];

    printf("Nhập vào tên chương trình C: "); gets(ifilename);
    if ((ifp = fopen(ifilename, "rt"))== NULL) {
        printf("Không mở được file để đọc."); exit(1);
    }

    printf("Nhập vào tên file RTF (đuôi RTF): "); gets(ofilename);
    if ((ofp = fopen(ofilename, "wt"))== NULL) {
        printf("Không mở được file để ghi ra."); exit(1);
    }

    fprintf(ofp,"\\rtf1");
    next_character();
    while (ch != EOF) next_token();

    fprintf(ofp,"}"); fclose(ifp); fclose(ofp);
}

```

III. ĐỀ NGHỊ PHÁT TRIỂN

Bạn có thể tự phát triển tiếp theo những hướng sau:

1. Cho in những cấu trúc lỗi (lỗi từ tố, cú pháp, ngữ nghĩa) ở dạng gạch chân.
2. Cho phép chọn và định dạng chương trình nguồn của nhiều ngôn ngữ lập trình khác nhau như C, Pascal, Basic, Foxpro...
3. In các cấu trúc khác nhau với nhiều font chữ khác nhau.
4. Sửa chương trình nguồn thành dạng chuẩn (theo ý bạn). Ví dụ:
 - Sửa các chỗ thụt đầu dòng theo đúng cấp của dòng đó.
 - Dẫn những chỗ cần thiết, như a:= b sửa thành a := b.
 - Bỏ những ký tự thừa như khoảng trắng thừa.

Phụ lục C

CÁC CHƯƠNG TRÌNH NGUỒN PASCAL

Sau đây là các đoạn chương trình nguồn viết bằng Pascal tương đương với các chương trình C minh họa trong các chương của giáo trình.

Chương 2, trang 181 (bộ thông dịch VIM - version đơn giản)

```
const
  max_proc = 1000;
  max_data = 100;
  max_stack = 100;

type
  { Liệt kê các chỉ thị của VIM và mã hóa chúng bằng cách gán cho mỗi
    lệnh một mã khác nhau }
  operation = (neg, abs_, add, sub, mul, dvi, mdl, eq, ne, lt, le, gt, ge,
    ldcon, ldvar, stvar, jump, jift, jiff, rdint, wrint, halt_);
  { Phần tham số, tùy thuộc vào chỉ thị có thể là giá trị (như lệnh ldcom)
    hoặc địa chỉ (như các lệnh ldvar, stvar, jump, jift, jiff) }

union = record
  case byte of
    0: (value : integer);
    1: (address : integer);
  end;

{ Khai báo cấu trúc của chỉ thị }
instruction = record
  code : operation; { Phần mã cho biết đó là chỉ thị nào }
  u : union;
end;

var
  prog : array[0..max_proc - 1] of instruction;
  data : array[0..max_data - 1] of integer;
  stack : array[0..max_stack - 1] of integer;
  sp : integer;

procedure read_code;
begin
```

{ Phần này đọc chương trình mã máy VIM vào và đặt trong mảng prog }

```

end;

procedure push(x: integer);
begin
    stack[sp] := x; inc(sp);
end;

procedure pop(var x: integer);
begin
    dec(sp); x := stack[sp];
end;

procedure read_int(var x: integer);
begin
    { Đọc một số nguyên từ bàn phím và gán vào tham số x }
end;

procedure write_int(x: integer);
begin
    { In ra màn hình giá trị số nguyên x }
end;

var
    instr: ^instruction;
    ic, next, left_operand, right_operand, operand, result: integer;
    truthval, stop: boolean;

begin
    read_code;
    ic := 0; sp := 0; stop := false;

    while not stop do begin
        next := ic + 1;
        instr := addr(prog[ic]);

        case instr^.code of
            neg, abs_:
                begin
                    pop(operand);
                    case (instr^.code) of
                        neg: result := -operand;

                        abs_: if (operand < 0) then result := -operand
                                else result := operand;
                    end;
                    push(result);
                end;

            add, sub, mul, dvi, md1:

```

```

begin
  pop(right_operand); pop(left_operand);
  case (instr^.code) of
    add: result := left_operand + right_operand;
    sub: result := left_operand - right_operand;
    mul: result := left_operand * right_operand;
    div: result := left_operand div right_operand;
    mod: result := left_operand mod right_operand;
  end;
  push(result);
end;

eq, ne, lt, le, gt, ge:
begin
  pop(right_operand); pop(left_operand);
  case (instr^.code) of
    eq:   truthval := (left_operand = right_operand);
    ne:   truthval := (left_operand <> right_operand);
    lt:   truthval := (left_operand < right_operand);
    le:   truthval := (left_operand <= right_operand);
    gt:   truthval := (left_operand > right_operand);
    ge:   truthval := (left_operand >= right_operand);
  end;
  if truthval then result := 1 else result := 0;
  push(result);
end;

ldcon: push(instr^.u.value);
ldvar: push(data[instr^.u.address]);
stvar: pop(data[instr^.u.address]);
jump: next := instr^.u.address;
jift: begin pop(operand);
        if (operand = 1) then next := instr^.u.address;
      end;
jiff: begin pop(operand);
        if (operand = 0) then next := instr^.u.address;
      end;
rdint: begin read_int(operand); push(operand); end;
wrint: begin pop(operand); write_int(operand); end;
halt_: stop := true;
end;
ic := next;
end;
end.

```

Chương 2, trang 203 (Bộ thông dịch VIM - version thực sự)

```

const
  max_proc = 1000;      { Độ dài bộ nhớ chương trình }
  max_table = 1000;     { Độ dài tối đa các loại bảng }

```

```

max_data = 100;           { Độ dài bộ nhớ dữ liệu }
max_depth = 15;          { Độ sâu gọi lồng nhau tối đa }
max_stack = 100;          { Độ sâu tối đa của các ngăn xếp }

type
{ Liệt kê các chi thị của VIM và mã hoá chúng bằng cách gán cho mỗi
lệnh một mã khác nhau }
operation = ( neg, abs_, add, sub, mul, dvi, mdl, eq, ne, lt, le, gt,
              ge, ldcon, ldvar, stvar, jump, jift, jiff, call, crseg,
              dlseg, return , rdint, wrint, halt_ );

union1 =      record
               sn1, dpl: integer;           { ldvar, stvar }
               end;

union2 =      record
               sn2, index2: integer;       { srseg }
               end;

union = record
         case byte of
             0: (value : integer);
             1: (u1 : union1);
             2: (index1 : integer);
             3: (u2 : union2);
         end;

{ Khai báo cấu trúc của chi thị }

instruction = record
               code : operation;
               u : union;
               end;

var
prog        : array[0..max_proc - 1] of instruction;
CSG         : array[0..max_depth - 1] of integer;
address_table : array[0..max_table - 1] of integer;
length_table : array[0..max_table - 1] of integer;
data         : array[0..max_data - 1] of integer;
stack        : array[0..max_stack - 1] of integer;
return_stack : array[0..max_stack - 1] of integer;
sp, dsp, rsp, csn: integer;

procedure read_code;
var fp: file; fname: string[100];
begin
{ Đọc chương trình VIM từ file vào mảng prog }
write('Nhập tên chương trình VIM: '); readln(fname);
assign(fp, fname); reset(fp, 1);
blockread(fp, prog, filesize(fp)); close(fp);

```

```

{ Đọc toàn bộ file và ghi vào vùng nhớ chỉ bởi prog }
write('Nhập tên file bang dia chi: '); readln(fname);
assign(fp, fname); reset(fp, 1);
blockread(fp, address_table, filesize(fp)); close(fp);

{ Đọc bảng địa chỉ từ file vào mảng length_table }
write('Nhập tên file bang do dai: '); readln(fname);
assign(fp, fname); reset(fp, 1);
blockread(fp, length_table, filesize(fp)); close(fp);
end;

procedure push(x: integer);
begin
  stack[sp] := x; inc(sp);
end;

procedure pop(var x: integer);
begin
  dec(sp); x := stack[sp];
end;

procedure push_return(x: integer);
begin
  return_stack[rsp] := x; inc(rsp);
end;

procedure pop_return(var x: integer);
begin
  dec(rsp); x := return_stack[rsp];
end;

procedure read_int(var x: integer);
begin
  write('Nhập vào một số nguyên: '); read(x);
end;

procedure write_int(x: integer);
begin
  write(x, ' ');
end;

procedure creat_segment(sn, data_length: integer);
begin
  data[dsp] := sn;           inc(dsp);
  data[dsp] := data_length;   inc(dsp);
  data[dsp] := CSG[sn - 1];   inc(dsp);
  data[dsp] := CSG[csn];     inc(dsp);
  csn := sn; CSG[csn] := dsp; dsp := dsp + data_length;
end;

```

```

procedure delete_segment;
var      current_segment, call_env, sn_caller, i, data_length,
         static_link, dynamic_link: integer;
begin
  current_segment := CSG[csn];
  data_length := data[current_segment - 3];
  static_link := data[current_segment - 2];
  dynamic_link := data[current_segment - 1];

  if static_link = dynamic_link then
    dec(csn) { Noi gọi trong cùng nhóm phân đoạn }
  else begin { Noi gọi khác nhóm phân đoạn }
    call_env := dynamic_link;
    sn_caller := data[dynamic_link - 4];
    for i := sn_caller downto csn do begin
      { Lấy môi trường nơi gọi }
      CSG[i] := call_env; call_env := data[call_env - 2];
    end;
    csn := sn_caller;
  end;
  dsp := dsp - (data_length + 4);
end;

var
  instr: ^instruction;
  ic, next, left_operand, right_operand, operand, result, length: integer;
  truthval, stop: boolean;

begin
  read_code;
  ic := 0; sp := 0; dsp := 0; rsp := 0; csn := 0; CSG[csn] := 0;
  stop := false;

  while not stop do begin
    next := ic + 1;
    instr := addr(prog[ic]);

    case instr^.code of
      neg, abs_:
        begin
          pop(operand);
          case (instr^.code) of
            neg: result := -operand;
            abs_: if (operand < 0) then result := -operand
                               else result := operand;
          end;
          push(result);
        end;
    end;
  end;

```

```

add, sub, mul, dvi, mdl;
begin
  pop(right_operand); pop(left_operand);
  case (instr^.code) of
    add: result := left_operand + right_operand;
    sub: result := left_operand - right_operand;
    mul: result := left_operand * right_operand;
    dvi: result := left_operand div right_operand;
    mdl: result := left_operand mod right_operand;
  end;
  push(result);
end;

eq, ne, lt, le, gt, ge:
begin
  pop(right_operand); pop(left_operand);
  case (instr^.code) of
    eq: truthval := (left_operand = right_operand);
    ne: truthval := (left_operand <> right_operand);
    lt: truthval := (left_operand < right_operand);
    le: truthval := (left_operand <= right_operand);
    gt: truthval := (left_operand > right_operand);
    ge: truthval := (left_operand >= right_operand);
  end;
  if truthval then result := 1 else result := 0;
  push(result);
end;

ldcon: push(instr^.u.value);
ldvar: push(data[CSG[instr^.u.u1.sn1] + instr^.u.u1.dpl]);
stvar: pop(data[CSG[instr^.u.u1.sn1] + instr^.u.u1.dpl]);
jump: next := address_table[instr^.u.index1];
jift: begin pop(operand);
         if operand = 1 then next:=address_table[instr^.u.index1];
       end;
jiff: begin pop(operand);
         if operand = 0 then next:=address_table[instr^.u.index1];
       end;
call: begin push_return(next);
         next := address_table[instr^.u.index1];
       end;
crseg: begin length := length_table[instr^.u.u2.index2];
         creat_segment(instr^.u.u2.sn2, length);
       end;
dlseg: delete_segment;
return: pop_return(next);
rdint: begin read_int(operand); push(operand); end;
wrint: begin pop(operand); write_int(operand); end;
halt_: stop := true;
end;
ic := next;

```

```

end;
end.

Chương 3, trang 214 (Phân tích từ vựng - chưa hoàn chỉnh)

const
  KNUM = 17; { Số lượng từ khoá }
  TNUM = 37; { Số lượng từ tố }
  EOF_ = char(0); { Dấu hiệu đánh dấu hết file }

type
{ Liệt kê và đánh số tự động các từ tố }
  token = (ident_token, num_token, begin_token, end_token, int_token,
            var_token, procedure_token, call_token, read_token, write_token,
            if_token, then_token, else_token, fi_token, while_token, do_token,
            od_token, negate_token, absolute_token, open_token, close_token,
            list_token, period_token, separator_token, becomes_token, plus_token,
            minus_token, times_token, over_token, modulo_token, equal_token,
            not_equal_token, less_than_token, less_or_equal_token,
            greater_than_token, greater_or_equal_token, err_token );

var
  ch : char; { Ký tự đọc vào từ chương trình nguồn }
  retract : boolean; { Cờ cho biết đã đọc quá }
  look_ahead : token; { Từ tố của từ vị vừa phân tích }
  ident_lexeme : string; { Xâu chứa từ vị tên }
  num_value : integer; { Giá trị của từ vị số }

procedure initialise_scanner;
begin
{ Đặt các lệnh khởi đầu ở đây }
  ch := '';
end;

procedure next_character;
begin
{ Đọc từ chương trình nguồn vào một ký tự và đặt trong biến ch }
end;

function look_up(ident_lexeme: string): token;
begin
{ Trả lại từ tố của từ khoá đang nằm trong biến ident_lexeme }
end;

{ Phân tích tên và kiểm tra xem có phải là từ khoá không. Trả về từ tố ident_token hoặc từ tố theo từ khoá, xâu tên được đặt trong xâu ident_lexeme }
procedure identifier;
begin
  ident_lexeme := "";

```

```

repeat
  ident_lexeme := ident_lexeme + ch; next_character;
until ((ch < 'A') or (ch > 'z')) and ((ch < '0') or (ch > '9'));

look_ahead := look_up(ident_lexeme); { Tìm và trả về từ tố nếu đó là từ khoá }
{ Nếu không phải là từ khoá thì đó chỉ là một tên }
if look_ahead = err_token then look_ahead := ident_token;
retract := true;
end;

{ Đoán nhận số. Hàm này cũng tính luôn giá trị của số đó và đặt vào biến num_value }
procedure number;
begin
  num_value := 0;
  repeat
    num_value := num_value * 10 + byte(ch) - byte('0'); next_character;
  until (ch < '0') or (ch > '9');
  look_ahead := num_token;
  retract := true;
end;
{ Loại bỏ các chú thích }
procedure comment;
begin
  repeat next_character; until ch = '}';
end;
{ Thủ tục chính của phân tích từ vựng: tìm từ tố tiếp theo }
procedure next_token;
begin
  retract := false;
  { Loại bỏ các ký hiệu cách, tab, xuống dòng, chú thích }
  while (((ch <= ' ') or (ch = '{')) and (ch < EOF_)) do begin
    if ch = '{' then comment; next_character;
  end;
  if ch = EOF_ then exit;
  case ch of
    '{': look_ahead := open_token;
    '}': look_ahead := close_token;
    ',': look_ahead := list_token;
    '.': look_ahead := period_token;
    ';': look_ahead := separator_token;
    ':': begin next_character;
      if ch = '=' then look_ahead := becomes_token;
    end;
    '+': look_ahead := plus_token;
  end;
end;

```

```

'~': look_ahead := minus_token;

'*': look_ahead := times_token;

'/': look_ahead := over_token;

'|': look_ahead := modulo_token;

'=': look_ahead := equal_token;

'<': begin next_character;
      if ch = '>' then look_ahead := not_equal_token
      else if ch = '=' then look_ahead := less_or_equal_token
      else begin look_ahead := less_than_token; retract := true; end;
      end;

'>': begin next_character;
      if ch = '=' then look_ahead := greater_or_equal_token
      else
        begin look_ahead := greater_than_token; retract := true; end;
      end
    else {case}
      if (ch >= 'A') and (ch <= 'z') then identifier {isalpha(ch)}
      else if (ch >= '0') and (ch <= '9') then number; {isdigit(ch)}
    end;
  end;

if not retract then next_character;
end;

```

Chương 3, trang 218 (Hoàn thiện phần phân tích từ vựng)

```

type
  keyword = record
    erep: token;           {Từ tố của từ khoá}
    irep: string[8];       {Mẫu từ khoá}
  end;
const
  {Bảng các từ khoá và từ tố tương ứng. Có tất cả 17 từ khoá}
  keywtb: array[1..KWNUM] of keyword =
    (erep: begin_token; irep: 'BEGIN'), (erep: end_token; irep: 'END'),
    (erep: int_token; irep: 'INT'), (erep: var_token; irep: 'VAR'),
    (erep: procedure_token; irep: 'PROC'), (erep: call_token; irep: 'CALL'),
    (erep: read_token; irep: 'READ'), (erep: write_token; irep: 'WRITE'),
    (erep: if_token; irep: 'IF'), (erep: then_token; irep: 'THEN'),
    (erep: else_token; irep: 'ELSE'), (erep: fi_token; irep: 'FI'),
    (erep: while_token; irep: 'WHILE'), (erep: do_token; irep: 'DO'),
    (erep: od_token; irep: 'OD'), (erep: negate_token; irep: 'NEG'),
    (erep: absolute_token; irep: 'ABS')

```

```

);
var
  fp: text;
procedure initialise_scanner;
var filename: string[20];
begin
  write('Nhap vao ten chuong trinh nguon: '); readln(filename);
  assign(fp, filename); reset(fp);
  ch := '';
end;
procedure next_character;
begin
  if eof(fp) then begin close(fp); ch := EOF_; end
  else begin read(fp, ch); ch := upcase(ch); end;
end;
function look_up(ident_lexeme: string): token;
var i: integer;
begin
  look_up := err_token;
  for i := 1 to KWNUM do
    if ident_lexeme = keywtb[i].irep then look_up := keywtb[i].erep;
end;

```

Chương 3, trang 219 (Kiểm tra, thử nghiệm phân tích từ vựng)

```

const
  { Khai báo tên các từ tố nhằm in theo dõi }
  tokenname: array[0..TNUM - 1] of string =
    {
      'ident_token', 'num_token', 'begin_token', 'end_token', 'int_token',
      'var_token', 'procedure_token', 'call_token', 'read_token',
      'write_token', 'if_token', 'then_token', 'else_token', 'fi_token',
      'while_token', 'do_token', 'od_token', 'negate_token',
      'absolute_token', 'open_token', 'close_token', 'list_token',
      'period_token', 'separator_token', 'becomes_token', 'plus_token',
      'minus_token', 'times_token', 'over_token', 'modulo_token',
      'equal_token', 'not_equal_token', 'less_than_token',
      'less_or_equal_token', 'greater_than_token', 'greater_or_equal_token',
      'err_token'
    };

```

{ Phần thân chương trình để thử nghiệm bộ phân tích từ vựng. Khi kết nối với phần khác bạn có thể xoá phần này đi }

```

begin
  initialise_scanner;
  { Chu động gọi phân tích từ tố cho toàn chương trình SLANG }
  while ch <> EOF_ do begin
    next_token;
    if look_ahead = ident_token then
      writeln('<', tokenname[ord(look_ahead)], ', ', ident_lexeme, '>')
  end;

```

```

else
    if look_ahead = num_token then
        writeln('<', tokenname[ord(look_ahead)], ',', num_value, '>')
    else writeln('<', tokenname[ord(look_ahead)], ',', '>');
    if look_ahead = period_token then break; { Chỉ làm việc đến dấu
                                                chấm hết chương trình và bỏ phần còn lại}
end;
end.

```

Chương 4, trang 227 (Bảng luật xây dựng bộ phân tích cú pháp)

Bảng các luật cấu trúc bộ phân tích bằng ngôn ngữ Pascal:

TT	Biểu thức chính quy	Module phân tích
1	a (ký hiệu kết thúc)	if look_ahead = a then next_token;
2	A (ký hiệu không kết thúc)	A ; (nghĩa là gọi thủ tục ứng với A)
3	$E_1 E_2 \dots E_n \ (n > 1)$	<pre> if look_ahead ∈ DIRSET(E_1) then begin Π (E_1) end else if look_ahead ∈ DIRSET(E_2) then begin Π (E_2) ... end else if look_ahead ∈ DIRSET(E_n) then begin Π (E_n) end; </pre>
4	$E_1 E_2 \dots E_n$	$\Pi (E_1) \Pi (E_2) \dots \Pi (E_n)$
5	E^*	<pre> while look_ahead ∈ FIRST(E) do begin Π (E) end; </pre>
6	E	$\Pi (E)$
7	$[E]$	if look_ahead ∈ FIRST(E) then $\Pi (E)$
8	E^+	<pre> Π (E) while look_ahead ∈ FIRST(E) do begin Π (E) end; </pre>
9	$E_1 (E_2 E_1)^*$	<pre> Π (E_1) while look_ahead ∈ FIRST(E_2) do begin Π (E_2) Π (E_1) end; </pre>

		end;
--	--	------

Chương 4, trang 230 (Phân tích cú pháp)

{ Khai báo thủ tục trước khi dùng }

```

procedure block;           forward;
procedure declaration_part; forward;
procedure constant_declarer; forward;
procedure type_declarer;   forward;
procedure variable_declaration; forward;
procedure procedure_declaration; forward;
procedure statement_part;   forward;
procedure statement;        forward;
procedure assignment_statement; forward;
procedure left_part;        forward;
procedure if_statement;     forward;
procedure while_statement;  forward;
procedure call_statement;   forward;
procedure read_statement;   forward;
procedure write_statement;  forward;
procedure expression;       forward;
procedure term;             forward;
procedure factor;           forward;
procedure operand;          forward;
procedure add_operator;     forward;
procedure multiply_operator; forward;
procedure unary_operator;   forward;
procedure relation;         forward;
procedure relational_operator; forward;

```

{*** Phân phân tích từ vựng nằm ở đây ***}

{*** Phân phân tích cú pháp ***}

```

procedure program_declaration;
begin
  block;
  if look_ahead = period_token then next_token;
end;

```

```

procedure block;
begin
  if look_ahead = begin_token then next_token;
  declaration_part;
  statement_part;
  if look_ahead = end_token then next_token;
end;

```

```

procedure declaration_part;
begin
  while (look_ahead = int_token) or (look_ahead = var_token) do begin
    if look_ahead = int_token then constant_declaration
    else if look_ahead = var_token then variable_declaration;
  end;
  while look_ahead = procedure_token do procedure_declaration;
end;

procedure constant_declaration;
begin
  type_declarer;
  if look_ahead = ident_token then next_token;
  if look_ahead = equal_token then next_token;
  if look_ahead = num_token then next_token;
  while (look_ahead = list_token) do begin
    if look_ahead = list_token then next_token;
    if look_ahead = ident_token then next_token;
    if look_ahead = equal_token then next_token;
    if look_ahead = num_token then next_token;
  end;
  if look_ahead = separator_token then next_token;
end;

procedure type_declarer;
begin
  if look_ahead = int_token then next_token;
end;

procedure variable_declaration;
begin
  if look_ahead = var_token then next_token;
  type_declarer;
  if look_ahead = ident_token then next_token;
  while (look_ahead = list_token) do begin
    if look_ahead = list_token then next_token;
    if look_ahead = ident_token then next_token;
  end;
  if look_ahead = separator_token then next_token;
end;

procedure procedure_declaration;
begin
  if look_ahead = procedure_token then next_token;
  if look_ahead = ident_token then next_token;
  block;
  if look_ahead = separator_token then next_token;
end;

procedure statement_part;

```

```

begin
  statement;
  while look_ahead = separator_token do begin
    if look_ahead = separator_token then next_token;
    statement;
  end;
end;

procedure statement;
begin
  if look_ahead = ident_token then assignment_statement
  else if look_ahead = if_token then if_statement
  else if look_ahead = while_token then while_statement
  else if look_ahead = call_token then call_statement
  else if look_ahead = read_token then read_statement
  else if look_ahead = write_token then write_statement;
end;

procedure assignment_statement;
begin
  left_part;
  if look_ahead = becomes_token then next_token;
  expression;
end;

procedure left_part;
begin
  if look_ahead = ident_token then next_token;
end;

procedure if_statement;
begin
  if look_ahead = if_token then next_token;
  relation;
  if look_ahead = then_token then next_token;
  statement_part;
  if look_ahead = else_token then begin
    if look_ahead = else_token then next_token;
    statement_part;
  end;
  if look_ahead = fi_token then next_token;
end;

procedure while_statement;
begin
  if look_ahead = while_token then next_token;
  relation;
  if look_ahead = do_token then next_token;
  statement_part;
  if look_ahead = od_token then next_token;

```

```

end;

procedure call_statement;
begin
  if look_ahead = call_token then next_token;
  if look_ahead = ident_token then next_token;
end;

procedure read_statement;
begin
  if look_ahead = read_token then next_token;
  if look_ahead = open_token then next_token;
  if look_ahead = ident_token then next_token;
  while look_ahead = list_token do begin
    if look_ahead = list_token then next_token;
    if look_ahead = ident_token then next_token;
  end;
  if look_ahead = close_token then next_token;
end;

procedure write_statement;
begin
  if look_ahead = write_token then next_token;
  if look_ahead = open_token then next_token;
  expression;
  while look_ahead = list_token do begin
    if look_ahead = list_token then next_token;
    expression;
  end;
  if look_ahead = close_token then next_token;
end;

procedure expression;
begin
  term;
  while (look_ahead = minus_token) or (look_ahead = plus_token) do begin
    add_operator;
    term;
  end;
end;

procedure term;
begin
  factor;
  while (look_ahead = modulo_token) or (look_ahead = over_token)
    or (look_ahead = times_token) do
  begin
    multiply_operator;
    factor;
  end;

```

```

end;

procedure factor;
begin
  if (look_ahead = absolute_token ) or (look_ahead = negate_token) then
    begin unary_operator; end;
    operand;
  end;

procedure operand;
begin
  if look_ahead = ident_token then
    begin if look_ahead = ident_token then next_token; end
  else if look_ahead = num_token then
    begin if look_ahead = num_token then next_token; end
  else if look_ahead = open_token then begin
    if look_ahead = open_token then next_token;
    expression;
    if look_ahead = close_token then next_token;
  end;
end;

procedure add_operator;
begin
  if look_ahead = plus_token then begin
    if look_ahead = plus_token then next_token;
  end
  else if look_ahead = minus_token then begin
    if look_ahead = minus_token then next_token;
  end;
end;

procedure multiply_operator;
begin
  if look_ahead = times_token then begin
    if look_ahead = times_token then next_token;
  end
  else if look_ahead = over_token then begin
    if look_ahead = over_token then next_token;
  end
  else if look_ahead = modulo_token then begin
    if look_ahead = modulo_token then next_token;
  end;
end;

procedure unary_operator;
begin
  if look_ahead = negate_token then begin
    if look_ahead = negate_token then next_token;
  end

```

```

else if look_ahead = absolute_token then begin
    if look_ahead = absolute_token then next_token;
end;
end;

procedure relation;
begin
    expression;
    relational_operator;
    expression;
end;

procedure relational_operator;
begin
    if look_ahead = equal_token then begin
        if look_ahead = equal_token then next_token;
    end
    else if look_ahead = not_equal_token then begin
        if look_ahead = not_equal_token then next_token;
    end
    else if look_ahead = less_than_token then begin
        if look_ahead = less_than_token then next_token;
    end
    else if look_ahead = less_or_equal_token then begin
        if look_ahead = less_or_equal_token then next_token;
    end
    else if look_ahead = greater_than_token then begin
        if look_ahead = greater_than_token then next_token;
    end
    else if look_ahead = greater_or_equal_token then begin
        if look_ahead = greater_or_equal_token then next_token;
    end;
end;

```

{ *Thân chương trình chính* }

```

begin
    initialise_scanner;
    next_token;
    program_declaration;
end.

```

Thử nghiệm chương trình (trang 237):

```

procedure program_declaration;
begin
    ⇒ writeln('program_declaration');
    block;
    if look_ahead = period_token then next_token;
end;

```

```

procedure block;
begin
⇒ writeln('block');
  if look_ahead = begin_token then next_token;
  declaration_part;
  statement_part;
  if look_ahead = end_token then next_token;
end;

procedure declaration_part;
begin
⇒ writeln('declaration_part');

...

```

Chương 5, trang 247 (Các cấu trúc, biến dùng xây dựng bằng ký hiệu)

```

type
  e_type = (no_type, unknown_type, int_type);
  e_kind = (unknown_kind, const_kind, var_kind, proc_kind);

  def_ptr = ^def_rec;
  def_rec = record {Khai báo bản ghi định nghĩa}
    sn, dpl_or_value_or_index : integer;
    type_ : e_type;
    kind : e_kind;
    next : def_ptr;
  end;

  ident_ptr = ^ident_rec;
  ident_rec = record {Khai báo bản ghi định danh}
    ident : string;
    def_list : def_ptr;
    left, right : ident_ptr;
  end;

  name_ptr = ^name_rec;
  name_rec = record {Khai báo bản ghi tên}
    ident : string;
    next : name_ptr;
  end;

  scope_ptr = ^scope_rec;
  scope_rec = record {Khai báo bản ghi phạm vi}

```

```

name_list : name_ptr;
prev_scope : scope_ptr;
end;

const
UNDECLARED = NIL;

var
ident_tree : ident_ptr;
scope_list : scope_ptr;
scope_level : integer;

```

Chương 5, trang 250 - 251 (Giao thức các chương trình con làm việc với bảng ký hiệu)

```

procedure enter_scope; forward;
procedure exit_scope; forward;
function current_scope: integer; forward;

function create_definition(sn, dpl_or_value_or_index: integer;
                           type_: e_type; kind: e_kind): def_ptr; forward;
procedure insert_definition(ident: string; def: def_ptr); forward;
function find_definition(ident: string): def_ptr; forward;
function get_sn(def: def_ptr): integer; forward;
function get_dpl(def: def_ptr): integer; forward;
function get_value(def: def_ptr): integer; forward;
function get_address(def: def_ptr): integer; forward;
function get_type(def: def_ptr): e_type; forward;
function get_kind(def: def_ptr): e_kind; forward;
procedure report_error(errmsg: string); forward;

```

Chương 5, trang 252 (Các chương trình con phân tích ngữ nghĩa)

```

procedure program_declaration;
begin
⇒ enter_scope;
  block;
⇒ exit_scope;
  if look_ahead = period_token then next_token;
end;

```

```

procedure declaration_part;
⇒ var dpl: integer;
begin

```

```

⇒ dpl := 0;
  while (look_ahead = int_token) or (look_ahead = var_token) do begin
    if look_ahead = int_token then constant_declaration
  ⇒ else if look_ahead = var_token then variable_declaration(dpl);
  end;
  while look_ahead = procedure_token do procedure_declaration;
end;

procedure constant_declaration;
⇒var sn: integer; def: def_ptr ;
begin
⇒ sn := current_scope;
  type_declarer;
  if look_ahead = ident_token then next_token;
  if look_ahead = equal_token then next_token;
  if look_ahead = num_token then next_token;
⇒ def := create_definition(sn, num_value, int_type, const_kind);
⇒ insert_definition(ident_lexeme, def);

while (look_ahead = list_token) do begin
  if look_ahead = list_token then next_token;
  if look_ahead = ident_token then next_token;
  if look_ahead = equal_token then next_token;
  if look_ahead = num_token then next_token;
⇒ def := create_definition(sn, num_value, int_type, const_kind);
⇒ insert_definition(ident_lexeme, def);
  end;
  if look_ahead = separator_token then next_token;
end;

⇒procedure variable_declaration(var dpl: integer);
⇒var sn: integer; def: def_ptr;
begin
⇒ sn := current_scope;
  if look_ahead = var_token then next_token;
  type_declarer;
  if look_ahead = ident_token then next_token;
⇒ def := create_definition(sn, dpl, int_type, var_kind);
⇒ insert_definition(ident_lexeme, def);
⇒ inc(dpl);

while (look_ahead = list_token) do begin
  if look_ahead = list_token then next_token;
  if look_ahead = ident_token then next_token;
⇒ def := create_definition(sn, dpl, int_type, var_kind);
⇒ insert_definition(ident_lexeme, def);
⇒ inc(dpl);
  end;
  if look_ahead = separator_token then next_token;

```

```

end;

procedure procedure_declaration;
⇒var sn, address: integer ; def: def_ptr ;
begin
⇒ sn := current_scope;
    if look_ahead = procedure_token then next_token;
⇒ def := create_definition(sn, address, no_type, proc_kind);
⇒ insert_definition(ident_lexeme, def);
    if look_ahead = ident_token then next_token;
⇒ enter_scope;
    block;
⇒ exit_scope;
    if look_ahead = separator_token then next_token;
end;

procedure left_part;
⇒var def: def_ptr ; kind: e_kind ;
begin
⇒ def := find_definition(ident_lexeme);
⇒ kind := get_kind(def);
⇒ if kind <> var_kind then report_error('Ten khong phai la mot bien');
    if look_ahead = ident_token then next_token;
end;

procedure call_statement;
⇒var def: def_ptr; kind: e_kind ;
begin
    if look_ahead = call_token then next_token;
⇒ def := find_definition(ident_lexeme);
⇒ kind := get_kind(def);
⇒ if kind <> proc_kind then report_error('Ten khong la mot thu tuc');
    if look_ahead = ident_token then next_token;
end;

procedure read_statement;
⇒var def: def_ptr ; kind: e_kind ;
begin
    if look_ahead = read_token then next_token;
    if look_ahead = open_token then next_token;
⇒ def := find_definition(ident_lexeme);
⇒ kind := get_kind(def);
⇒ if kind <> var_kind then report_error('Ten khong la mot bien');
    if look_ahead = ident_token then next_token;

    while (look_ahead = list_token) do begin
        if look_ahead = list_token then next_token;
⇒ def := find_definition(ident_lexeme);
⇒ kind := get_kind(def);

```

```

⇒ if kind <> var_kind then report_error('Ten khong la mot bien');
   if look_ahead = ident_token then next_token;
end;
if look_ahead = close_token then next_token;
end;

procedure operand;
⇒var def: def_ptr; kind: e_kind;
begin
  if look_ahead = ident_token then begin
    ⇒ def := find_definition(ident_lexeme);
    ⇒ kind := get_kind(def);
    ⇒ if (kind = proc_kind) or (kind = unknown_kind) then
        report_error('Ten phai la bien hoac hang so');
        if look_ahead = ident_token then next_token;
      end
    else if look_ahead = num_token then begin
      if look_ahead = num_token then next_token;
    end
    else if look_ahead = open_token then begin
      if look_ahead = open_token then next_token;
      expression;
      if look_ahead = close_token then next_token;
    end;
  end;
end;

```

Chương 5, trang 257 (Hoàn thiện bảng ký hiệu và phân tích ngữ nghĩa)

```

procedure initialise_symboltable;
begin
  scope_list := NIL; ident_tree := NIL; scope_level := 0;
end;

procedure enter_scope;
var scp: scope_ptr;
begin
  { Thủ tục này tạo ra một bản ghi phạm vi mới và chèn vào đầu
  danh sách phạm vi }
  new(scp); scp^.name_list := NIL; scp^.prev_scope := scope_list;
  scope_list := scp; inc(scope_level);
end;

procedure exit_scope;
var      scp: scope_ptr; np1, np2: name_ptr; ip: ident_ptr;
        dp: def_ptr;
begin
  { Phản này tìm và xoá tất cả các bản ghi trong danh sách tên ứng
  với phạm vi trên cùng }
  np1 := scope_list^.name_list;
  while np1 <> NIL do begin

```

```

np2 := np1^.next; ip := ident_tree;

{ Phần này tìm và xoá bản ghi đầu tiên trong danh sách bản ghi
định nghĩa ứng với tên của bản ghi tên - để làm điều này
đòi hỏi phải duyệt cây định danh }

while ip <> NIL do begin
  if ip^.ident = np1^.ident then begin
    dp := ip^.def_list; ip^.def_list := dp^.next; dispose(dp);
    break;
  end;
  if ip^.ident > np1^.ident then ip := ip^.left else ip := ip^.right;
  end;
  dispose(np1); np1 := np2;
end;

{ Xoá bản ghi trên cùng của danh sách phạm vi }
scp := scope_list; scope_list := scope_list^.prev_scope;
dispose(scp); dec(scope_level);
end;

function current_scope: integer;
begin
  current_scope := scope_level;
end;

function create_definition(sn, dpl_or_value_or_index: integer;
                           type_: e_type; kind: e_kind);
var dp: def_ptr;
begin
  new(dp);
  dp^.sn := sn; dp^.dpl_or_value_or_index := dpl_or_value_or_index;
  dp^.type_ := type_; dp^.kind := kind; dp^.next := NIL;
  create_definition := dp;
end;

procedure insert_definition(ident: string; def: def_ptr);
var ip1, ip2: ident_ptr; np: name_ptr;
begin
  { Tìm bản ghi ứng với tên trong cây định danh }
  ip2 := ident_tree;
  while (ip2 <> NIL) do begin
    ip1 := ip2;
    if ip1^.ident = ident then break;
    if ip1^.ident > ident then ip2 := ip1^.left else ip2 := ip1^.right;
  end;

  { Nếu tìm thấy thì thêm bản ghi định nghĩa trả bởi def vào đầu
  danh sách định nghĩa }
  if (ip1^.ident = ident) and (ident_tree <> NIL) then begin
    def^.next := ip1^.def_list; ip1^.def_list := def;
  
```

```

end
else begin { Nếu không thấy thì tạo nút mới cho cây định danh }
  new(ip2); ip2^.ident := ident; ip2^.def_list := def;
  ip2^.left := NIL; ip2^.right := NIL;
  if ident_tree = NIL then ident_tree := ip2
  else if ip1^.ident > ident then
    ip1^.left := ip2 else ip1^.right := ip2;
end;

{ Kiểm tra xem tên đã cho có sẵn trong danh sách tên của mức phạm vi
hiện tại hay không (kiểm tra tính duy nhất) }
np := scope_list^.name_list;
while np <> NIL do begin
  if ident = np^.ident then begin
    report_error('Ten duoc dinh nghia hai lan');
    exit;
  end;
  np := np^.next;
end;

{ Nếu chưa thì thêm vào đầu danh sách tên }
new(np);
np^.ident := ident;
np^.next := scope_list^.name_list;
scope_list^.name_list := np;
end;

function find_definition(ident: string): def_ptr;
var ip1, ip2: ident_ptr;
begin
  { Duyệt cây định danh }
  ip2 := ident_tree;
  while ip2 <> NIL do begin
    ip1 := ip2;
    if ip1^.ident = ident then break;
    if ip1^.ident > ident then ip2 := ip1^.left else ip2 := ip1^.right;
  end;

  if ip1^.ident = ident then begin
    if ip1^.def_list <> NIL then begin
      find_definition := ip1^.def_list;
      exit;
    end;
    report_error('Ten chua duoc dinh nghia');
  end
  else
    report_error('Ten khong co trong cay dinh danh');
  find_definition := UNDECLARED;
end;

```

```

function get_sn(def: def_ptr): integer;
begin
  get_sn := def^.sn;
end;

function get_dpl(def: def_ptr): integer;
begin
  get_dpl := def^.dpl_or_value_or_index;
end;

function get_value(def: def_ptr): integer;
begin
  get_value := def^.dpl_or_value_or_index;
end;

function get_address(def: def_ptr): integer;
begin
  get_address := def^.dpl_or_value_or_index;
end;

function get_type(def: dcf_ptr): e_type;
begin
  get_type := def^.type_;
end;

function get_kind(def: def_ptr): e_kind;
begin
  get_kind := def^.kind;
end;

procedure report_error(errmsg: string);
begin
  writeln(errmsg);
end;

begin
  initialise_scanner;
  initialise_symboltable;
  next_token;
  program_declaration;
end.

```

Chương 6, trang 265 - 273 (Sinh mã)

```

procedure program_declaration;
begin
  ⇒ initialise_vimcode;
  enter_scope;

```

```

block;
exit_scope;
⇒ finalise_vimcode;
if look_ahead = period_token then next_token;
end;

procedure block;
⇒var level, length_index, length: integer ;
begin
  if look_ahead = begin_token then next_token;
  ⇒ declaration_part(length);
  ⇒ level := current_scope;
  ⇒ length_index := get_index;
  ⇒ enter_length(length_index, length);
  ⇒ emit_crseg(level, length_index);
  statement_part;
  ⇒ emit(dlseg);
  if look_ahead = end_token then next_token;
end;

⇒procedure declaration_part(var length: integer);
⇒var dpl, over_proc: integer;
begin
  dpl := 0;

  while (look_ahead = int_token) or (look_ahead = var_token) do begin
    if look_ahead = int_token then constant_declaration
    else if look_ahead = var_token then variable_declaration(dpl);
  end;

  ⇒ length := dpl;
  ⇒ if look_ahead = procedure_token then begin
    ⇒  over_proc := get_label;
    ⇒  emit_jump(jump, over_proc); {Sinh chi thi nhảy jump over_proc}
    ⇒  procedure_declaration;
    ⇒  while look_ahead = procedure_token do procedure_declaration;
    ⇒  emit_label(over_proc); {Đánh nhãn cho over_proc}
  end;
end;

procedure procedure_declaration;
var sn, address: integer; def: def_ptr ;
begin
  sn := current_scope;
  if look_ahead = procedure_token then next_token;
  ⇒ address := get_label;
  ⇒ emit_label(address);
  def := create_definition(sn, address, no_type, proc_kind);
  insert_definition(ident_lexeme, def);

```

```

if look_ahead = ident_token then next_token;
enter_scope;
block;
exit_scope;
⇒ emit(return);
if look_ahead = separator_token then next_token;
end;

procedure assignment_statement;
⇒var sn, dpl: integer;
begin
⇒ left_part(sn, dpl);
if look_ahead = becomes_token then next_token;
expression;
⇒ emit_stvar(sn, dpl);
end;

⇒procedure left_part(var sn, dpl: integer);
var def: def_ptr; kind:e_kind ;
begin
def := find_definition(ident_lexeme);
kind := get_kind(def);
if kind <> var_kind then report_error('Ten khong phai la mot bien');
⇒ sn := get_sn(def);
⇒ dpl := get_dpl(def);
if look_ahead = ident_token then next_token;
end;

procedure if_statement;
⇒var over_then_part, over_else_part:integer ;
begin
if look_ahead = if_token then next_token;
relation;
if look_ahead = then_token then next_token;
⇒ over_then_part := get_label;
⇒ emit_jump(jiff, over_then_part);
statement_part;
if look_ahead = else_token then begin
⇒ over_else_part := get_label;
⇒ emit_jump(jump, over_else_part);
⇒ emit_label(over_then_part);
if look_ahead = else_token then next_token;
statement_part;
⇒ emit_label(over_else_part);
end
⇒ else emit_label(over_then_part);
if look_ahead = fi_token then next_token;
end;

```

```

procedure while_statement;
⇒var begin_while_start, end_while_start:integer ;
begin
  if look_ahead = while_token then next_token;
  ⇒ begin_while_start := get_label;
  ⇒ emit_label(begin_while_start);
    relation;
  ⇒ end_while_start := get_label;
  ⇒ emit_jump(jiff, end_while_start);
    if look_ahead = do_token then next_token;
    statement_part;
    if look_ahead = od_token then next_token;
    ⇒ emit_jump(jump, begin_while_start);
    ⇒ emit_label(end_while_start);
  end;

procedure call_statement;
⇒var def: def_ptr ; kind: e_kind ; address: integer ;
begin
  if look_ahead = call_token then next_token;
  def := find_definition(ident_lexeme);
  kind := get_kind(def);
  if kind <> proc_kind then report_error('Ten khong la mot thu tuc');
  ⇒ address := get_address(def);
  ⇒ emit_jump(call, address);
  if look_ahead = ident_token then next_token;
end;

procedure read_statement;
⇒var def: def_ptr ; kind: e_kind ; sn, dpl: integer ;
begin
  if look_ahead = read_token then next_token;
  if look_ahead = open_token then next_token;
  def := find_definition(ident_lexeme);
  kind := get_kind(def);
  if kind <> var_kind then report_error('Ten khong la mot bien');
  ⇒ sn := get_sn(def);
  ⇒ dpl := get_dpl(def);
  ⇒ emit(rdint);
  ⇒ emit_stvar(sn, dpl);
  if look_ahead = ident_token then next_token;

while (look_ahead = list_token) do begin
  if look_ahead = list_token then next_token;
  def := find_definition(ident_lexeme);
  kind := get_kind(def);
  if kind <> var_kind then report_error('Ten khong la mot bien');
  ⇒ sn := get_sn(def);
  ⇒ dpl := get_dpl(def);

```

```

⇒ emit(rdint);
⇒ emit_stvar(sn, dpl);
  if look_ahead = ident_token then next_token;
end;
  if look_ahead = close_token then next_token;
end;

procedure write_statement;
begin
  if look_ahead = write_token then next_token;
  if look_ahead = open_token then next_token;
  expression;
⇒ emit(wrint);

  while (look_ahead = list_token) do begin
    if look_ahead = list_token then next_token;
    expression;
  ⇒ emit(wrint);
  end;
  if look_ahead = close_token then next_token;
end;

procedure expression;
⇒ var op: add_op ;
begin
  term;

  while (look_ahead = minus_token) or (look_ahead = plus_token) do begin
    add_operator(op);
    term;
  ⇒ if op = plus then emit(add) else if op = minus then emit(sub);
  end;
end;

procedure term;
⇒ var op: mul_op ;
begin
  factor;
  while (look_ahead = modulo_token) or (look_ahead = over_token)
        or (look_ahead = times_token) do
begin
  ⇒ multiply_operator(op);
  factor;
  ⇒ if op = times then emit(mul)
  ⇒ else if op = over then emit(dvi)
  ⇒ else if op = modulo then emit(mod);
  end;
end;

```

```

procedure factor;
⇒var op: unary_op ;
begin
  if (look_ahead = absolute_token ) or (look_ahead = negate_token) then
    begin
      ⇒ unary_operator(op);
      ⇒ operand;
      ⇒ if op = negate then emit(neg) else if op = absolute_ then emit(abs_);
      end
      else operand;
    end;

procedure operand;
⇒var def: def_ptr ; kind: e_kind; sn, dpl, value: integer ;
begin
  if look_ahead = ident_token then begin
    def := find_definition(ident_lexeme);
    ⇒ kind := get_kind(def);
    ⇒ if kind = var_kind then begin
        ⇒ sn := get_sn(def);
        ⇒ dpl := get_dpl(def);
        ⇒ emit_ldvar(sn, dpl);
      end
      ⇒ else if kind = const_kind then begin
        ⇒ value := get_value(def);
        ⇒ emit_ldcon(value);
      end
      ⇒ else if (kind = proc_kind) or (kind = unknown_kind) then
        report_error('Ten phai la bien hoac hang so');
      if look_ahead = ident_token then next_token;
    end
    ⇒ else if look_ahead = num_token then begin
      if look_ahead = num_token then next_token;
      ⇒ emit_ldcon(num_value);
    end
    ⇒ else if look_ahead = open_token then begin
      if look_ahead = open_token then next_token;
      expression;
      if look_ahead = close_token then next_token;
    end;
  end;
end;

⇒procedure add_operator(var op: add_op);
begin
  if look_ahead = plus_token then begin
    ⇒ if look_ahead = plus_token then begin op := plus; next_token; end;
    end
    else if look_ahead = minus_token then begin
      ⇒ if look_ahead = minus_token then begin op := minus; next_token; end;
    end;
  end;

```

```

end;
end;

⇒procedure multiply_operator(var op: mul_op);
begin
  if look_ahead = times_token then begin
    ⇒ if look_ahead = times_token then begin op := times; next_token;end;
    end
    else if look_ahead = over_token then begin
      ⇒ if look_ahead = over_token then begin op := over; next_token;end;
      end
    else if look_ahead = modulo_token then begin
      ⇒ if look_ahead = modulo_token then begin op := modulo; next_token;end;
      end;
  end;

⇒procedure unary_operator(var op: unary_op);
begin
  if look_ahead = negate_token then begin
    ⇒ if look_ahead = negate_token then begin op := negate; next_token;end;
    end
    else if look_ahead = absolute_token then begin
      ⇒ if look_ahead = absolute_token then
      ⇒ begin op := absolute_; next_token; end;
      end;
  end;

procedure relation;
⇒var op: rel_op ;
begin
  expression;
  ⇒ relational_operator(op);
  expression;
  ⇒ if op = equal then emit(eq)
  ⇒ else if op = not_equal then emit(ne)
  ⇒ else if op = less_than then emit(lt)
  ⇒ else if op = less_or_equal then emit(le)
  ⇒ else if op = greater_than then emit(gt)
  ⇒ else if op = greater_or_equal then emit(ge);
end;

⇒procedure relational_operator(var op: rel_op);
begin
  if look_ahead = equal_token then begin
    ⇒ if look_ahead = equal_token then begin op := equal; next_token;end;
    end
    else if look_ahead = not_equal_token then begin
      ⇒ if look_ahead = not_equal_token then
      ⇒ begin op := not_equal; next_token; end;

```

```

end
else if look_ahead = less_than_token then begin
    if look_ahead = less_than_token then
⇒  begin op := less_than; next_token; end;
    end
else if look_ahead = less_or_equal_token then begin
    if look_ahead = less_or_equal_token then
⇒  begin next_token; op := less_or_equal; end;
    end
else if look_ahead = greater_than_token then begin
    if look_ahead = greater_than_token then
⇒  begin next_token; op := greater_than; end;
    end
else if look_ahead = greater_or_equal_token then begin
    if look_ahead = greater_or_equal_token then
⇒  begin next_token; op := greater_or_equal; end;
    end;
end;

```

Chương 6, trang 275 (Hoàn chỉnh phần sinh mã)

type	
operation = (neg, abs_, add, sub, mul, dvi, mdl, eq, ne, lt, le, gt, ge, ldcon, ldvar, stvar, jump, jift, jiff, call, crseg, dlseg, return, rdint, wrint, halt_);	
add_op = (plus, minus);	forward;
mul_op = (times, over, modulo);	forward;
unary_op = (absolute_, negate);	forward;
rel_op = (equal, not_equal, less_than, less_or_equal, greater_than, greater_or_equal);	forward;
procedure block;	forward;
procedure declaration_part(var length: integer);	forward;
procedure constant_declaration;	forward;
procedure type_declarer;	forward;
procedure variable_declaration(var dpl: integer);	forward;
procedure procedure_declaration;	forward;
procedure statement_part;	forward;
procedure statement;	forward;
procedure assignment_statement;	forward;
procedure left_part(var sn, dpl: integer);	forward;
procedure if_statement;	forward;
procedure while_statement;	forward;
procedure call_statement;	forward;
procedure read_statement;	forward;
procedure write_statement;	forward;
procedure expression;	forward;
procedure term;	forward;
procedure factor;	forward;
procedure operand;	forward;
procedure add_operator(var op: add_op);	forward;
procedure multiply_operator(var op: mul_op);	forward;

```

procedure unary_operator(var op: unary_op);           forward;
procedure relation;                                forward;
procedure relational_operator(var op: rel_op);      forward;
procedure enter_scope;                            forward;
procedure exit_scope;                           forward;
function create_definition(sn, dpl_or_value_or_index: integer;
                           type_: e_type; kind: e_kind): def_ptr ;   forward;
procedure insert_definition(ident: string; def: def_ptr);    forward;
function find_definition(ident: string): def_ptr ;       forward;
function get_sn(def: def_ptr): integer;               forward;
function get_dpl(def: def_ptr): integer;             forward;
function get_value(def: def_ptr): integer;            forward;
function get_address(def: def_ptr): integer;          forward;
function get_type(def: def_ptr): e_type;              forward;
function get_kind(def: def_ptr): e_kind;              forward;
procedure report_error(errmsg: string);              forward;
function current_scope: integer;                    forward;
procedure initialise_vimcode;                      forward;
procedure finalise_vimcode;                        forward;
procedure emit_crseg(level, index: integer);        forward;
procedure emit(code: operation);                   forward;
procedure enter_length(index, length: integer);     forward;
procedure emit_jump(code: operation; index: integer); forward;
function get_index:integer ;                       forward;
function get_label:integer ;                      forward;
procedure emit_label(index: integer);              forward;
procedure emit_ldvar(sn, dpl: integer);            forward;
procedure emit_stvar(sn, dpl: integer);            forward;
procedure emit_ldcon(value: integer);             forward;
type
  union1 = record
    sn1, dpl: integer;
  end;
  union2 = record
    sn2, index2: integer;
  end;
  union = record
    case byte of
      0: (value : integer);                  { ldcon }
      1: (u1 : union1);                   { ldvar, stvar }
      2: (index1 : integer);              { jump, jift, jiff, call }
      3: (u2 : union2);                 { crseg }
  end;
  inststruct = record
    code : operation;
    u : union;
  end;
var
  address_table      : array[0..max_table - 1] of integer;
  length_table       : array[0..max_table - 1] of integer;

```

```

len_index, add_index, ic : integer ;
cfp, afp, lfp : FILE ;
instruction : inststruct;

procedure initialise_vimcode;
var fname: string[100];
begin
  write('Ten file ra cua VIM: '); readln(fname);
  assign(cfp, fname); rewrite(cfp, 1);
  write('Ten file bang dia chi: '); readln(fname);
  assign(afp, fname); rewrite(afp, 1);
  write('Ten file bang do dai: '); readln(fname);
  assign(lfp, fname); rewrite(lfp, 1);
  len_index := 0; add_index := 0; ic := 0;
end;
procedure finalise_vimcode;
begin
  emit(halt_);
  blockwrite(afp, address_table, add_index * 2);
  blockwrite(lfp, length_table, len_index * 2);
  close(cfp); close(afp); close(lfp);
end;
procedure emit_crseg(level, index: integer);
begin
  inc(ic);
  instruction.code := crseg;
  instruction.u.u2.sn2 := level;
  instruction.u.u2.index2 := index;
  blockwrite(cfp, instruction, sizeof(instruction));
end;

procedure emit(code: operation);
begin
  inc(ic);
  instruction.code := code;
  blockwrite(cfp, instruction, sizeof(instruction));
end;

procedure emit_jump(code: operation; index: integer);
begin
  inc(ic);
  instruction.code := code;
  instruction.u.index1 := index;
  blockwrite(cfp, instruction, sizeof(instruction));
end;

procedure enter_length(index, length: integer);
begin
  length_table[index] := length;
end;

```

```

function get_index: integer;
begin
  inc(len_index); get_index:=(len_index - 1);
end;

function get_label:integer ;
begin
  inc(add_index); get_label:=(add_index - 1);
end;

procedure emit_label(index: integer);
begin
  address_table[index] := ic;
end;
procedure emit_ldvar(sn, dpl: integer);
begin
  inc(ic);
  instruction.code := ldvar;
  instruction.u.ul.sn1 := sn;
  instruction.u.ul.dpl := dpl;
  blockwrite(cfp, instruction, sizeof(instruction));
end;
procedure emit_stvar(sn, dpl: integer);
begin
  inc(ic);
  instruction.code := stvar;
  instruction.u.ul.sn1 := sn;
  instruction.u.ul.dpl := dpl;
  blockwrite(cfp, instruction, sizeof(instruction));
end;

procedure emit_ldcon(value: integer);
begin
  inc(ic);
  instruction.code := ldcon;
  instruction.u.value := value;
  blockwrite(cfp, instruction, sizeof(instruction));
end;

```

Kiểm tra thử nghiệm phần sinh mã

```

procedure instruction_print;
const
  op_name: array[0..25] of string = ('neg', 'abs_', 'add', 'sub', 'mul',
                                     'dvi', 'mdl', 'eq', 'ne', 'lt', 'le', 'gt', 'ge', 'ldcon', 'ldvar',
                                     'stvar', 'jump', 'jif', 'jiff',
                                     'call', 'crseg', 'dlseg', 'return_', 'rdint',
                                     'wrint', 'halt');
begin
  write(' ', op_name[ord(instruction.code)], ' ');

```

```

case instruction.code of
  ldcon: write(instruction.value);
  ldvar, stvar: write(instruction.u1.sn1, ',', instruction.u1.dpl);
  jump, jift, jiff, call: write(instruction.index1);
  crseg: write(instruction.u2.sn2, ',', instruction.u2.index2);
end;
writeln;
end;
procedure emit_crseg;
begin
  inc(ic);
  instruction.code := crseg;
  instruction.u2.sn2 := level;
  instruction.u2.index2 := index;
  blockwrite(cf, instruction, sizeof(instruction));
⇒ instruction_print;
end;
procedure emit;
begin
  inc(ic);
  instruction.code := code;
  blockwrite(cf, instruction, sizeof(instruction));
⇒ instruction_print;
end;

procedure emit_jump;

begin
  inc(ic);
  instruction.code := code;
  instruction.index1 := index;
  blockwrite(cf, instruction, sizeof(instruction));
⇒ instruction_print;
end;

procedure emit_ldvar;
begin
  inc(ic);
  instruction.code := ldvar;
  instruction.u1.sn1 := sn;
  instruction.u1.dpl := dpl;
  blockwrite(cf, instruction, sizeof(instruction));
⇒ instruction_print;
end;

procedure emit_stvar;
begin
  inc(ic);
  instruction.code := stvar;
  instruction.u1.sn1 := sn;

```

```

instruction.u1.dpl := dpl;
blockwrite(cf, instruction, sizeof(instruction));
⇒ instruction_print;
end;
procedure emit_ldcon;

begin
inc(ic);
instruction.code := ldcon;
instruction.value := value;
blockwrite(cf, instruction, sizeof(instruction));
⇒ instruction_print;
end;

```

Chương 7, trang 286 (Bảng xây dựng bộ phân tích cú pháp có kèm phát hiện lỗi)

TT	Biểu thức chính quy	Module phân tích
1	a (ký hiệu kết thúc)	if look_ahead = a then next_token else report_error ("Thiếu từ tố a ");
2	A (ký hiệu không kết thúc)	A ; (nghĩa là gọi thủ tục ứng với A)
3	$E_1 E_2 \dots E_n$ ($n > 1$)	if look_ahead ∈ DIRSET(E_1) then begin $\Pi(E_1)$ end else if look_ahead ∈ DIRSET(E_2) then begin $\Pi(E_2)$ end else if look_ahead ∈ DIRSET(E_n) then begin $\Pi(E_n)$ end else report_error ("Lựa chọn không đúng");
4	$E_1 E_2 \dots E_n$	$\Pi(E_1) \Pi(E_2) \dots \Pi(E_n)$
5	E^*	while look_ahead ∈ FIRST(E) do begin $\Pi(E)$ end;
6	E	$\Pi(E)$
7	$[E]$	if look_ahead ∈ FIRST(E) then begin $\Pi(E)$ end;
8	E^+	$\Pi(E)$ while look_ahead ∈ FIRST(E) do begin $\Pi(E)$ end;
9	$E_1 (E_2 E_1)^*$	$\Pi(E_1)$ while look_ahead ∈ FIRST(E_2) do begin $\Pi(E_2) \Pi(E_1)$ end;

Chương 7, trang 290 (Bảng xây dựng bộ phân tích cú pháp có kèm phát hiện và khôi phục lỗi)

TT	Biểu thức chính quy	Module phân tích
1	a (ký hiệu kết thúc)	delete_until ($[a] \cup \text{local_markers} \cup \text{global_markers}$); if look_ahead = a then next_token; else report_error ("Thiếu từ tố a ");
2	A (ký hiệu không kết thúc)	A ; (nghĩa là gọi thủ tục ứng với A)
3	$E_1 E_2 \dots E_n$ ($n > 1$)	delete_until ($\text{FIRST}(E_1) \cup \text{FIRST}(E_2) \cup \dots \cup \text{FIRST}(E_n) \cup \text{global_markers}$); if look_ahead $\in \text{DIRSET}(E_1)$ then begin $\Pi(E_1)$ end else if look_ahead $\in \text{DIRSET}(E_2)$ then begin $\Pi(E_2)$ end else if look_ahead $\in \text{DIRSET}(E_n)$ then begin $\Pi(E_n)$ end else report_error ("Lựa chọn không đúng");
4	$E_1 E_2 \dots E_n$	$\Pi(E_1) \Pi(E_2) \dots \Pi(E_n)$
5	E^*	delete_until ($\text{FIRST}(E) \cup \text{FOLLOW}(E) \cup \text{local_markers} \cup \text{global_markers}$); while look_ahead $\in \text{FIRST}(E)$ do begin $\Pi(E)$ end;
6	E	$\Pi(E)$
7	$[E]$	delete_until ($\text{FIRST}(E) \cup \text{FOLLOW}(E) \cup \text{local_markers} \cup \text{global_markers}$); if look_ahead $\in \text{FIRST}(E)$ then begin $\Pi(E)$ end;
8	E^+	$\Pi(E)$ while look_ahead $\in \text{FIRST}(E)$ do begin $\Pi(E)$ end;
9	$E_1 (E_2 E_1)^*$	$\Pi(E_1)$ delete_until ($\text{FIRST}(E_1) \cup \text{FIRST}(E_2) \cup \text{FOLLOW}(E_1) \cup \text{local_markers} \cup \text{global_markers}$); while look_ahead $\in \text{FIRST}(E_2)$ do begin $\Pi(E_2) \Pi(E_1)$ delete_until ($\text{FIRST}(E_1) \cup \text{FIRST}(E_2) \cup \text{FOLLOW}(E_1) \cup \text{local_markers} \cup \text{global_markers}$); end;

Tài liệu tham khảo

- [1] Phạm Hồng Nguyên. *Giáo trình thực hành chương trình dịch*. ĐHQGHN 1998.
- [2] Nguyễn Văn Ba. *Thực hành Kỹ thuật biên dịch*. ĐHBK HN 1993.
- [3] Nguyễn Văn Ba. *Ngôn ngữ hình thức*. ĐHBK Hà Nội 1994.
- [4] Arto Salomaa. *Nhập môn tin học lý thuyết tính toán và các ôtômat*. Bản dịch của Nguyễn Xuân My, Phạm Trà ân. NXB Khoa học và Kỹ thuật, Hà Nội 1992.
- [5] Englewood Cliffs, N.Wirth. *Algorithms+Data structures = programs*. Prentice Hall. Chương 5 - Bản dịch của NXB Thông kê.
- [6] A.V.Aho, Ravi Sethi, D.Ullman. *Compilers - principles, techniques, and tools*. Addison-Wesley publishing company. 1986.
- [7] A.V.Aho, D.Ullman. *The theory of parsing, translation and compiling*. Prentice-Hall, Inc. 1972.
- [8] Niklaus Wirth. *Pascal S - A subset and its implementation*. 1975.
- [9] John Elder. *Compiler Construction*. Prentice Hall, 1994.
- [10] Charles N.Fischer, Richard J.LeBlanc. *Crafting a compiler with C*. The Benjamin/Cummings Publishing Company, Inc, 1991.
- [11] P.Rechenberg, H.Mossenbock. *A compiler generator for microcomputers*. 1988.
- [12] W.P.Cockshott. *A compiler writer's toolbox*. Ellis Horwood Ltd, 1990.
- [13] Richard Bornat. *Understanding and Writing Compilers*. MacMillan, 1979.
- [14] Jim Holmes. *Building your own compiler with C++*. Prentice Hall, 1995.
- [15] H. Alblas, A.Nymeyer. *Practice and principles of compiler building with C*. Prentice Hall, 1996.

- [16] J.E.Hopcroft, DULLMAN. *Introduction to Automata theory, languages and computation*. Addison-Wesley publishing company. 1979.
- [17] D.Appleby, J.J.VandeKopple. *Programming Languages: Paradigm and Practice*. The MacGraw-Hill companies, Inc., 1997.
- [18] P.J.Plauger, Jim Brodie. *Standard C - A reference*. Prntice Hall Ptr. 1996.
- [19] J.Glenn Brookshear. *Computer Science - An overview*. The Benjamin/Cummings Publishing Company, Inc, 1994.
- [20] Peter Naur. *Computing: A Human Activity*. Addison-Wesley publishing company. 1992.
- [21] B.H. Partee, A.Meulen, R.E.Wall. *Mathematical Methods in Linguistics*. Kluwer Academic Publishers, 1990.
- [22] T.A.Sudkamp. *Languages and machines*. Addison-Wesley publishing company. 1997.
- [23] R.Wilhelm, D.Maurer. *Compiler design*. Addison-Wesley publishing company. 1995.
- [24] K.Slonneger, B.L.Kurtz. *Formal syntax and semantics of programming languages*. Addison-Wesley publishing company. 1995.

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

16 Hàng Chuối - Hai Bà Trưng - Hà Nội

Điện thoại: (04) 39714896; (04) 39724770. Fax: (04) 39714899

Chịu trách nhiệm xuất bản:

Giám đốc: PHÙNG QUỐC BẢO

Tổng biên tập: PHẠM THỊ TRÂM

Biên tập: HỮU NGUYÊN

Ché bản: THÁI HÀ

Trình bày bìa: CHÍ KIÊN

GIÁO TRÌNH CHƯƠNG TRÌNH DỊCH

Mã số: 1K-10ĐH2009

In 1.000 cuốn, khổ 16 x 24 cm tại Công ty CP Nhà in KHCN

Số xuất bản: 51 - 2009/CXB/65 - 02/ĐHQGHN, ngày 13/1/2009

Quyết định xuất bản số: 10 KH-TN/XB

In xong và nộp lưu chiểu quý III năm 2009.