Mục lục

[Chương 1 - Giới thiệu chung 2](#_Toc308591605)

[1.1 Tầm quan trọng của đề tài: 2](#_Toc308591606)

[1.2 Phạm vi giới hạn thực hiện đề tài: 2](#_Toc308591607)

[1.3 Cấu trúc luận văn: 2](#_Toc308591608)

[Chương 2 - Kiến thức nền tảng 3](#_Toc308591610)

[2.1 Ngôn ngữ COBOL: 3](#_Toc308591613)

[2.1.1 Quan hệ giữa các thành phần dữ liệu: 3](#_Toc308591614)

[2.1.2 Khai báo và định dạng dữ liệu: 5](#_Toc308591615)

[2.1.3 Các thao tác trên dữ liệu: 9](#_Toc308591616)

[2.2 Ngôn ngữ JAVA: 10](#_Toc308591617)

[2.2.1 Các kiểu dữ liệu cơ bản trong ngôn ngữ JAVA: 10](#_Toc308591618)

[2.2.2 Các thao tác trên dữ liệu: 11](#_Toc308591619)

[2.3 Chương trình RES: 11](#_Toc308591620)

[Chương 3 - Phương pháp hiện thực 12](#_Toc308591621)

[3.1 Phân tích vấn đề: 12](#_Toc308591625)

[3.2 Phương án được sử dụng: 13](#_Toc308591626)

[3.2.1 Chuyển đổi thông thường: 14](#_Toc308591627)

[3.2.2 Chuyển đổi mở rộng sinh ra kiểu dữ liệu JAVA. 16](#_Toc308591628)

[3.3 Khó khăn trong quá trình thực hiện: 18](#_Toc308591629)

[3.4 Các phần mở rộng: 18](#_Toc308591630)

[Chương 4 - Đánh giá và kết quả 18](#_Toc308591631)

[4.1 Phương pháp đánh giá: 18](#_Toc308591633)

[4.2 Kết quả: 18](#_Toc308591634)

[Chương 5 - Tổng kết 18](#_Toc308591635)

# Giới thiệu chung

* 1. Tầm quan trọng của đề tài:

Ngôn ngữ **CO**mmon **B**usiness **O**riented **L**anguage (viết tắt là COBOL) đã được phát triển (năm 1959) và đưa vào sử dụng khá lâu (năm 1960).

Mục đích chủ yếu của ngôn ngữ COBOL là hướng tới việc xây dựng các hệ thống thương mại hơn là các hệ thống tính toán khoa học. Vì vậy mà số lượng phần mềm viết bằng ngôn ngữ này đang được sử dụng trong hệ thống của các ngân hàng, công ty tài chính vẫn còn khá cao.

Một nghiên cứu năm 2007 của *David Norfolk and Martin Banks* [1] cho thấy các giao dịch tài chính dựa trên những chương trình viết bằng ngôn ngữ COBOL chiếm đến 75% tổng số các giao dịch.

1. Mặc dù vẫn được sử dụng nhưng các hệ thống viết bằng ngôn ngữ COBOL cũng tồn tại những vấn đề:
   * Chỉ chạy trên một nền tảng nhất định (vd here). Bên cạnh đó một số đặc tả ngôn ngữ COBOL cũ cũng hỗ trợ rất hạn chế việc truy xuất cơ sở dữ liệu mà chủ yếu làm việc trên các tập tin dữ liệu (data files).
   * Số lượng lập trình viên am hiểu ngôn ngữ hiện nay không nhiều. (vd here)
   * Nhu cầu tích hợp hệ thống cũvới các hệ thống mới hơn được viết bằng các ngôn ngữ khác (JAVA, C#, VB…). (vd here)
2. Hiện nay trên thế giới, có hai giải pháp được đưa ra để giải quyết cho nhu cầu trên:
   * Bổ sung vào ngôn ngữ COBOL các tính năng để có thể tích hợp được với các hệ thống viết bằng ngôn ngữ khác. Ví dụ:
   * Chuyển đổi toàn bộ mã nguồn chương trình COBOL có sẵn sang ngôn ngữ khác (có thể tự động hoặc không).
3. Với những nhu cầu bức thiết kể trên, định hướng của để tài sẽ hiện thực giải pháp chuyển đổi mã nguồn chương trình COBOL có sẵn sang ngôn ngữ khác một cách tự động. Tuy nhiên, đề tài chỉ dừng lại ở mức chuyển đổi phần cấu trúc dữ liệu. Ngôn ngữ đích được chọn là JAVA với những ưu điểm:
   * Những chương trình viết bằng ngôn ngữ JAVA có thể chạy ở nhiều nền tảng khác nhau mà không cần sửa đổi mã nguồn.
   * JAVA là một ngôn ngữ mạnh dùng để viết các hệ thống quản lý.
   * Số lập trình viên thông thạo ngôn ngữ JAVA khá phổ biến, có thể dễ dàng hiểu được cũng như sửa đổi mã nguồn sinh ra.
   1. Phạm vi giới hạn thực hiện đề tài:
   2. Cấu trúc luận văn:

# Kiến thức nền tảng

* 1. Ngôn ngữ COBOL:

COBOL là ngôn ngữ lập trình thủ tục có cú pháp được thiết kế gần giống với tiếng Anh thông thường nên rất đơn giản, dễ hiểu.

Hiện nay tồn tại các tiêu chuẩn về đặc tả ngôn ngữ COBOL như: COBOL68, COBOL74, COBOL85, COBOL2002. Đề tài chỉ tập trung giới thiệu và chuyển đổi chủ yếu các chương trình viết theo chuẩn ngôn ngữ COBOL85.

Một chương trình COBOL có cấu trúc phân cấp, bao gồm 4 phần chính:

* + ***Identification Division:*** chứa các thông tin chung về chương trình như: tên chương trình, ngày viết, tên tác giả…
  + ***Environment Division:*** chứa các thông tin về cấu hình môi trường mà chương trình sẽ được thực thi như: loại kiến trúc máy tính, các tên đặc biệt, các tập tin được sử dụng trong chương trình…
  + ***Data Division:*** chứa định nghĩa các biến sẽ được sử dụng trong chương trình, ngoài ra còn chứa định nghĩa các biến được truyền qua các lệnh gọi, các cấu trúc bản ghi tập tin (file record). Đây là phần quan trọng trực tiếp liên quan đến mục tiêu thực hiện của đề tài và sẽ được mô tả kĩ trong phần sau.
  + ***Procedure Division:*** chứa các câu lệnh thể hiện giải thuật của chương trình.

Tùy theo bộ biên dịch (compiler) mà một chương trình COBOL bắt buộc chứa cả 4 phần trên hoặc chỉ cần chứa *IDENTIFICATION DIVISION* và *PROCEDURE DIVISION*.

Các biến được sử dụng nội bộ trong chương trình COBOL sẽ được khai báo trong mục *Working Data Section* nằm trong phần *Data Division*. Cú pháp khai báo biến:

Ví dụ:

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 EMPLOYEE-RECORD.**

**05 EMPLOYEE-NAME**

**10 FIRST PICTURE X(10)**

**10 LAST PICTURE X(10)**

**05 EMPLOYEE-ADDRESS.**

**10 STREET PICTURE X(10).**

**10 CITY PICTURE X(10).**

* + 1. Quan hệ giữa các thành phần dữ liệu:

Ngôn ngữ COBOL hỗ trợ định nghĩa kiểu dữ liệu dạng phân cấp, có nghĩa là:

* + Một thành phần cơ bản là một biến được định nghĩa mang kiểu dữ liệu như số, kí tự chữ, chuỗi.
  + Các thành phần dữ liệu cơ bản (element data) có thể được gộp thành một nhóm.
  + Một hay nhiều nhóm có thể được gộp thành nhóm lớn hơn.

Để phục vụ việc quản lý các biến theo mô hình trên, ngôn ngữ COBOL đưa ra hệ thống đánh dấu cấp (level indicator) và số thứ tự cấp (level number):

* Hệ thống đánh dấu cấp:

Đây là hệ thống được dùng trong việc định nghĩa các tập tin được sử dụng trong chương trình. Hệ thống gồm hai mức FD và SD. FD dùng để định nghĩa cho một tập tin thông thường, SD dùng để định nghĩa cho tập tin cần sắp xếp (sort-merge file). Hệ thống này chỉ được dùng trong phần FILE SECTION.

* Hệ thống số thứ tự cấp:

Hệ thống số thứ tự cấp được sử dụng nhằm mục đích xác định quan hệ cha con giữa hai nhóm cũng như phân biệt giữa nhóm và thành phần cơ bản.

Số thứ tự cấp là một số có một hoặc hai chữ số, giá trị từ 01 đến 49, hoặc một trong ba giá trị đặc biệt: 66, 77, 88. Cụ thể như sau:

* **01**: số thứ tự chỉ định một thành phần dữ liệu, có thể là một thành phần nhóm hoặc một thành phần cơ bản. 01 là cấp độ cao nhất trong hệ thống số thứ tự cấp.
* **02-49**: số thứ tự chỉ định các nhóm hoặc thành phần cơ bản bên trong một thành phần dữ liệu. Một thành phần sẽ nằm bên trong thành phần khác nếu số thứ tự cấp của nó có giá trị lớn hơn số thứ tự cấp của thành phần kia. Những thành phần cùng cấp phải có cùng số thứ tự.
* **66**: số thứ tự áp dụng cho những thành phần sử dụng mệnh đề RENAMES.
* **77**: xác định một thành phần cơ bản độc lập (không nằm trong một nhóm, không có các thành phần nhỏ hơn).
* **88**: số thứ tự chỉ áp dụng cho các thành phần điều kiện. Thành phần điều kiện là một thành phần dữ liệu bổ trợ cho một thành phần khác. Mỗi thành phần điều kiện sẽ tương ứng với một mệnh đề *if*  so sánh giá trị trong mệnh đề *VALUE* thành phần điều kiện với giá trị của thành phần nó bổ trợ.

Ví dụ:

**01 EMPLOYEE-RECORD.**

**05 EMPLOYEE-NAME**

**10 FIRST PICTURE X(10)**

**10 LAST PICTURE X(10)**

**05 EMPLOYEE-POSITION PICTURE 9(1).**

**88 MANAGER VALUE 0.**

**88 EMPLOYEE VALUE 1.**

Với ví dụ trên có hai thành phần điều kiện là MANAGER và EMPLOYEE bổ trợ cho biến EMPLOYEE-POSITION. Hai thành phần có giá trị trong mệnh đề VALUE lần lượt là 0 và 1. Vì vậy, nếu biến EMPLOYEE-POSITION có giá trị là 0, truy xuất giá trị của MANAGER sẽ trả về True. Tương ứng, khi EMPLOYEE-POSITION có giá trị là 1, truy xuất giá trị của EMPLOYEE sẽ trả về True.

* + 1. Khai báo và định dạng dữ liệu:

Ngôn ngữ COBOL hỗ trợ các kiểu dữ liệu cơ bản sau:

**Alphabetic** (kí tự chữ): chỉ chứa các kí tự trong bảng chữ cái tiếng Anh và dấu khoảng trắng (space).

**Numeric** (số): chỉ chứa các kí tự số, tối đa là 18 chữ số.

**Alphanumeric** (chuỗi): chỉ chứa các kí tự là chữ hoặc số.

**Numeric-edited** (số được định dạng lại): là số nhưng được định dạng lại theo mẫu qui định trong câu lệnh khai báo.

**Alphanumerc-edited** (chuỗi được định dạng lại): là chuỗi nhưng được định dạng lại theo mẫu qui định trong câu lệnh khai báo.

Các mệnh đề PICTURE, USAGE, OCCURS trong câu khai báo sẽ qui định kiểu dữ liệu, định dạng cũng như kích thước của một thành phần dữ liệu trong chương trình:

* Mệnh đề PICTURE:

Cú pháp:

**PIC**[**PICTURE**] [**IS**] *character-string*

Mệnh đề PICTURE giúp xác định một số tính chất chung và những hiệu chỉnh về định dạng cho một thành phần dữ liệu cơ bản. Tất cả các thành phần dữ liệu cơ bản, trừ những thành phần có chứa mệnh đề USAGE IS INDEX, đều phải được khai báo bằng từ khóa PICTURE.

*Character-string* là chuỗi kí tự để chỉ định định dạng và kiểu dữ liệu cho thành phần dữ liệu. Các kí tự được sử dụng trong *character-string* bao gồm:

|  |  |
| --- | --- |
| **Kí tự** | **Ý nghĩa** |
| A | Vị trí có thể chứa 1 kí tự hoặc khoảng trắng (space) |
| X | Vị trí có thể chứa 1 kí tự bất kì trong bảng kí tự (character set) |
| 9 | Vị trí có thể chứa 1 kí tự là số |
| S | Chỉ định số có dấu, phải là kí tự đầu tiên của chuỗi character-string, không tính vào kích thước của dữ liệu. |
| V | Chỉ định vị trí của dấu chấm thập phân, không tính vào kích thước dữ liệu. |
| P | Đánh dấu vị trí làm tròn thập phân. Một hoặc nhiều kí tự P phải xuất hiện liên tục ở tận cùng bên trái, hoặc bên phải chuỗi. |
| B | Vị trí có thể chứa 1 kí tự khoảng trắng |
| Z | Thay thế số 0 vô nghĩatrong giá trị số, nếu vị trí có số 0 sẽ được thay bằng khoảng trắng |
| \* | Tương tự như Z, thay thế số 0 bằng kí tự ‘\*’ |
| 0 | Số 0 sẽ được chèn vào vị trí này |
| / | Kí tự ‘/’ sẽ được chèn vào vị trí này |
| +  -  CR DB | Kí tự hiệu chỉnh cho số có dấu. CR tương đương với -, DB tương đương với + |
| . | Kí tự hiệu chỉnh cho dấu chấm thập phân |
| , | Kí tự ‘,’ sẽ được chèn vào vị trí này |
| $ | Kí hiệu tiền tệ (currency symbol) sẽ được chèn vào vị trí này |

Bảng 2.1 Bảng các kí tự dùng trong character-string

Mỗi kiểu dữ liệu chấp nhận mệnh đề PICTURE với các chuỗi khác nhau:

|  |  |  |
| --- | --- | --- |
| **Kiểu dữ liệu** | **Các kí tự chấp nhận** | **Chú thích thêm** |
| Alphabetic | A |  |
| Alphanumeric | A X 9 | Không chấp nhận chuỗi chỉ chứa toàn A hoặc toàn 9 |
| Numeric | 9 P S V | Chiều dài tối đa 18 |
| Numeric-edited | B P V Z 9 0 / , . + - CR DB \* $ | Nếu chỉ chứa P V 9 thì phải khai báo BLANK WHEN ZERO |
| Alphanumeric edited | A X 9 B 0 / | Chuỗi phải chứa ít nhất một A hoặc X, và ít nhất một 9 hoặc B hoặc 0 |

Bảng 2.2 Bảng

***Ví dụ***: các PICTURE cho giá trị số:

|  |  |
| --- | --- |
| **PICTURE** | **Giá trị hợp lệ** |
| 9999 | 0 đến 9999 |
| S99 | -99 đến +99 |
| S999V9 | -999.99 đến +999.99 |
| PPP999 | 0 đến .000999 |
| S999PPP | -1000 đến -999000,  +1000 đến +999000, và 0 |

***Ví dụ***: các PICTURE để hiệu chỉnh chuỗi:

|  |  |  |
| --- | --- | --- |
| **PICTURE** | **Dữ liệu đầu vào** | **Dữ liệu hiệu chỉnh** |
| X(10)/XX | ALPHANUMER01 | ALPHANUMER/01 |
| X(5)BX(7) | ALPHANUMERIC | ALPHA NUMERIC |
| 99,B999,B000 | 1234 | 01, 234, 000 |
| 99,999 | 12345 | 12,345 |
| 9999.99 | 123.45 | 0123.45 |
| 999.99+ | +1234.567  -1234.567 | 234.56+  234.56- |
| $$$9.99 | .12  -0.12 | $0.12  $0.12 |
| $999.99 | +123.45  -123.45 | $123.45  $123.45 |
| -$999.99 | +123.45  -123.45 | $123.45  -$123.45 |
| Z,ZZZ.ZZ+ | +0123.45 | 123.45+ |
| \*\*\*\*.99 | 0012.23 | \*\*12.23 |

* Mệnh đề USAGE

Mệnh đề USAGE cho biết định dạng dữ liệu được lưu trữ thực tế trong vùng nhớ. Nếu không chỉ định sẽ lấy giá trị mặc định là DISPLAY.

Cú pháp: **USAGE** [**IS**] *usage*

Các giá trị có thể có của *usage*: DISPLAY, BINARY, COMPUTATIONAL, COMP, PACKED-DECIMAL, INDEX.

Trừ những thành phần có kiểu dữ liệu Numeric, những thành phần có kiểu dữ liệu khác chỉ có thể có USAGE DISPLAY.

Mệnh đề USAGE không được sử dụng với các thành phần có level number là 66 hoặc 88. Nếu sử dụng cho 1 thành phần nhóm (group item) sẽ áp dụng cho từng thành phần cơ bản bên trong nhóm, lúc này USAGE của các thành phần cơ bản không được mâu thuẫn với USAGE đã được chỉ định của nhóm.

* **DISPLAY**: dữ liệu được lưu trữ theo dạng kí tự, mỗi kí tự tương ứng với 1 byte. Với kiểu dữ liệu *alphabetic* hay *alphanumeric*, mỗi kí tự sẽ được lưu trữ bằng mã ASCII hay mã EBDIC của kí tự đó (tùy vào cấu hình của chương trình).

Với kiểu dữ liệu numeric, mỗi chữ số sẽ được lưu trữ trong 1 byte với 4 bit cao được gọi là *zone bits*, 4 bit thấp chứa giá trị số BCD (Binary Coded Decimal) của chữ số đó. Nếu là số có dấu, 4 bit cao của byte thấp nhất được sử dụng để lưu trữ dấu, số âm sẽ lưu trữ giá trị 0xD, số dương sẽ lưu trữ giá trị 0xF.

* **BINARY**: dữ liệu được lưu trữ theo dạng nhị phân. Kích thước vùng dữ liệu được quyết định dựa trên số lượng chữ số thập phân chỉ định trong mệnh đề PICTURE. Bit ngoài cùng bên trái là bit dấu.

|  |  |
| --- | --- |
| **Số lượng chữ số trong mệnh đề PICTURE** | **Kích thước vùng nhớ (byte)** |
| 1 tới 4 | 2 |
| 5 tới 9 | 4 |
| 10 tới 18 | 8 |

* **COMPUTATIONAL, COMP, PACKED-DECIMAL**: dữ liệu được lưu trữ dạng 2 chữ số/byte, ngoại trừ byte cuối cùng được sử dụng để lưu trữ dấu và một chữ số. Phần qui định dấu bao gồm 4 bit cuối của byte cuối cùng sẽ lưu giá trị 0xF với số dương và 0xD với số âm.
* **INDEX**: thành phần sử dụng USAGE INDEX là vùng dữ liệu 4 byte, lưu trữ giá trị thông qua lệnh SET. Việc truy xuất dữ liệu này chỉ được phép với các lệnh SEARCH, SET, và mệnh đề USING.

***Ví dụ***: với tập kí tự EBCDIC, zone bits là 0xF

|  |  |  |
| --- | --- | --- |
| **PICTURE** | **Dữ liệu** | **Lưu trữ trong vùng nhớ** |
| PIC S9999 DISPLAY | 1234  +1234  -1234 | F1 F2 F3 F4  F1 F2 F3 F4  F1 F2 F3 D4 |
| PIC 9999 DISPLAY | 1234  +1234  -1234 | F1 F2 F3 F4  F1 F2 F3 F4  F1 F2 F3 F4 |
| PIC S9999 COMP | 1234  +1234  -1234 | 01 23 4F  01 23 4F  01 23 4D |
| PIC 9999 COMP | 1234  +1234  -1234 | 01 23 4F  01 23 4F  01 23 4F |
| PIC S9999 BINARY | +1234  -1234 | 04 D2  FB 2E |
| PIC 9999 BINARY | +1234  -1234 | 04 D2  04 D2 |

* Mệnh đề OCCURS

COBOL cho phép định nghĩa nhiều phần tử dữ liệu liên tiếp có cấu trúc giống nhau, gọi là bảng (table). Có thể truy xuất toàn bộ bảng hoặc một phần tử trong bảng. Bảng được định nghĩa bằng mệnh đề OCCURS.

Bảng có thể có một chiều hoặc nhiều chiều (multi-dimension), trong COBOL một bảng có thể có tối đa 7 chiều.

* Bảng chiều dài cố định: khai báo 1 bảng chiều dài cố định bằng mệnh đề OCCURS và số lượng phần tử trong bảng. Không áp dụng với các dữ liệu có level number 01, 66, 77, 88.

***Ví dụ***: bảng 1 chiều

**01 TABLE-ONE.**

**05 ELEMENT-ONE OCCURS 3 TIMES.**

**10 ELEMENT-A PIC X(4).**

**10 ELEMENT-B PIC 9(4).**

***Ví dụ***: bảng nhiều chiều

**01 TABLE-THREE.**

**05 ELEMENT-ONE OCCURS 3 TIMES.**

**10 ELEMENT-TWO OCCURS 3 TIMES.**

**15 ELEMENT-THREE OCCURS 2 TIMES PICTURE X(8).**

* Bảng chiều dài thay đổi: khai báo 1 bảng chiều dài thay đổi bằng mệnh đề OCCURS DEPENDING ON

Cú pháp:

**OCCURS** *minimum* **TO** *maximum* [**TIMES**]

**DEPENDING** [**ON**] *data-name*.

Kích thước của bảng sẽ cố định (chứa được *maximum* phần tử), chỉ có số lượng phần tử trong bảng sẽ thay đổi. Khi bảng được truy xuất lần đầu tiên, giá trị của *data-name* phải là giá trị hợp lệ giữa *minimum* và *maximum*.

Mệnh đề OCCURS DEPENDING ON không được sử dụng bên trong một mệnh đề OCCURS khác.

***Ví dụ***:

**01 MAIN-AREA.**

**03 REC-1.**

**05 FIELD-1 PIC 9.**

**05 FIELD-2 OCCURS 1 TO 5 TIMES**

**DEPENDING ON FIELD-1 PIC X(05).**

* + 1. Các thao tác trên dữ liệu:
    - Định dạng lại dữ liệu theo mẫu:
    - Canh chỉnh dữ liệu:

Việc canh chỉnh dữ liệu chỉ áp dụng trên thành phần dữ liệu cơ bản. Mỗi kiểu dữ liệu có những quy tắc riêng về canh chỉnh.

* **Numeric**: dữ liệu được canh chỉnh dựa trên dấu chấm thập phân. Nếu dấu chấm thập phân không được chỉ định, dữ liệu sẽ được xử lí giống với dữ liệu có dấu chấm thập phân ở tận cùng bên phải. Nếu cần thiết, dữ liệu có thể bị cắt ngắn, hoặc chèn bằng số 0 ở hai phía của dấu chấm, tùy thuộc vào giá trị và định dạng cụ thể.
* **Numeric-edited**: tương tự numeric, dữ liệu được canh chỉnh theo dấu chấm thập phân. Nếu đã có hiệu chỉnh không hiển thị số 0 vô nghĩa, sẽ không thực hiện việc chèn số 0.
* **Alphabetic, alphanumeric, alphanumeric-edited**: dữ liệu được canh chỉnh từ trái sang phải, nếu cần thiết dữ liệu có thể bị cắt ngắn, hoặc chèn bằng khoảng trắng. Nếu mệnh đề JUSTIFIED được chỉ định, dữ liệu được canh chỉnh theo quy tắc của JUSTIFIED.
  + - Tính toán số học trên dữ liệu:

Các lệnh về tính toán số học gồm có ADD, SUBTRACT, DIVIDE, MULTIPLY, và COMPUTE.

Các toán hạng không cần phải có chung định dạng về dữ liệu. Trong quá trình tính toán, trình biên dịch sẽ thực hiện việc chuyển đổi và canh chỉnh dữ liệu.

***Ví dụ***: với dữ liệu được định nghĩa như sau

**A PICTURE 9(7)V9(5).**

**B PICTURE 9(11)V99.**

**C PICTURE 9(12)V9(3).**

Nếu thực hiện câu lệnh **ADD A B TO C** sẽ cần có một toán hạng trung gian được định nghĩa như sau:

**Composite-of-Operands PICTURE 9(12)V9(5).**

Các lệnh ADD, SUBTRACT, DIVIDE, MULTIPLY có cùng cách xử lí. Riêng lệnh COMPUTE có thể tính toán trên một biểu thức, vì vậy có những quy tắc xác định toán hạng trung gian cụ thể hơn. Các lệnh ADD, SUBTRACT, DIVIDE, MULTIPLY được khuyến khích sử dụng thay vì COMPUTE để giảm khả năng sai lệch kết quả do sử dụng toán hạng trung gian.

* 1. Ngôn ngữ JAVA:

Khác với ngôn ngữ COBOL, JAVA là một ngôn ngữ lập trình đối tượng. Ngôn ngữ JAVA được bắt đầu phát triển từ năm 1995 và phiên bản hiện tại đến thời điểm này là JAVA 7.

Các tập tin mã nguồn JAVA sẽ được biên dịch sang mã *bytecode*. Mã này sẽ được chạy trên máy ảo JVM (Java Virtual Machine) tương ứng với từng nền tảng nhất định.

Vì vậy một chương trình được viết bằng ngôn ngữ này có thể chạy được trên nhiều nền tảng khác nhau.

Một ưu điểm khác của ngôn ngữ JAVA là bộ thư viện đi kèm của ngôn ngữ này hỗ trợ rất nhiều các lớp, phương thức có sẵn nhằm tạo sự thoải mái cho lập trình viên khi sử dụng.

Về mặt quản lý các đối tượng sử dụng trong một chương trình, máy ảo JAVA có một bộ phận “thu gom rác” (garbage collector) riêng. Bộ phận này sẽ tự động chạy và tiến hành thu gom các “đối tượng chết” (dead objects – các đối tượng không còn liên kết tới) và tiến hành xóa bỏ chúng.

* + 1. Các kiểu dữ liệu cơ bản trong ngôn ngữ JAVA:
* byte: có độ dài 1 byte, dùng để lưu trữ các số nguyên từ -128 đến 127
* short: có độ dài 2 bytes, dùng để lưu trữ các số nguyên từ -32,368 đến 32,367
* int: có độ dài 4 bytes, dùng để lưu trữ các số nguyên từ -2,147,483,648 đến 2,147,483,647
* long: có độ dài 8 bytes, dùng để lưu trữ các số nguyên từ -9,223,372,036,854,775,808 đến 9,223,372,036,854,775,807
* float: có độ dài 4 bytes, dùng để biểu diễn số thực dấu chấm động theo chuẩn IEEE 754.
* double: có độ dài 8 bytes, dùng để biểu diễn số thực dấu chấm động theo chuẩn IEEE 754.
* boolean: dùng để biểu diễn các giá trị đúng/sai trong chương trình, chỉ có hai giá trị là true/false.
* char: có độ dài 2 bytes, dùng để biểu diễn một kí tự theo bảng mã Unicode.
* String: ngoài tám kiểu dữ liệu nguyên thủy trên (primitive data types), ngôn ngữ JAVA còn hỗ trợ một kiểu dữ liệu đặc biệt là String (chuỗi). Đây thực chất là một lớp (class) nhưng các đối tượng thuộc lớp này có tính chất không thể thay đổi (immutable). Nghĩa là sau khi được tạo ra, giá trị của các đối tượng này không thể thay đổi được.
* Mảng: mảng thực chất là một đối tượng chứa nhiều các giá trị của cùng một kiểu dữ liệu xác định. Một mảng bao gồm nhiều thành viên, mỗi một thành viên trong mảng sẽ có một chỉ số (index) để có thể truy cập đến giá trị của chính mình trong mảng.
  + 1. Các thao tác trên dữ liệu:
  1. Chương trình RES:

RES là chương trình viết bằng JAVA nhằm mục đích chuyển đổi mã nguồn ngôn ngữ COBOL sang ngôn ngữ JAVA một cách tự động.

RES được viết từ tháng 4 năm 2009 và vẫn trong giai đoạn pre-beta, tuy nhiên hiện nay đã ngừng phát triển tiếp.

Một số tính năng mà RES hỗ trợ:

* Thực hiện chuyển đổi mã nguồn COBOL sang mã nguồn JAVA một cách tự động (theo chuẩn COBOL85, COBOL2002…)
* Hỗ trợ linh hoạt các định dạng mã nguồn của COBOL (tự do hoặc theo chuẩn về hàng cột, khoảng cách…)
* Hỗ trợ đầy đủ việc chuyển đổi các kiểu dữ liệu của COBOL sang JAVA. Hỗ trợ cả những câu lệnh đặc biệt như Redefines và Renames. Đặc biệt hỗ trợ các chế độ chuyển đổi khác nhau tùy theo mục đích sử dụng.
* Hỗ trợ chuyển đổi các câu lệnh cơ bản và các phép toán trên kiểu dữ liệu:
* If, Perform, Goto, Move, Call…
* Add, Subtract, Multiply, Divide, Compute…

Chương trình RES hỗ trợ cho người sử dụng ba chế chuyển đổi khác nhau:

# Phương pháp hiện thực

* 1. Phân tích vấn đề:

Vấn đề lớn nhất trong việc chuyển đổi các cấu trúc dữ liệu từ ngôn ngữ COBOL sang JAVA chính là sự khác biệt giữa hai cách quản lý vùng nhớ cho một chương trình của hai ngôn ngữ. Cụ thể:

* + Các biến trong một chương trình COBOL được cấp phát thành một vùng nhớ liên tục theo thứ tự từ trên xuống. Ngôn ngữ COBOL cũng hỗ trợ các mệnh đề (clause) REDEFINES, RENAMES để lợi dụng ưu điểm của cách cấp phát này.
  + Trong khi đó ngôn ngữ JAVA không hỗ trợ kiểu cấp phát vùng nhớ như trên. Các đối tượng trong một chương trình JAVA được cấp phát trong vùng nhớ *heap*, và có thể bị thu dọn, cấp phát lại trong quá trình chạy.

Với sự khác biệt như trên nên việc chuyển đổi cũng sẽ có những cách tiếp cận khác nhau:

* + Mô phỏng quá trình thực hiện cấp phát và quản lý bộ nhớ của ngôn ngữ COBOL.
  + Thực hiện quá trình chuyển đổi tương đương (một - một) giữa các kiểu dữ liệu của hai ngôn ngữ.

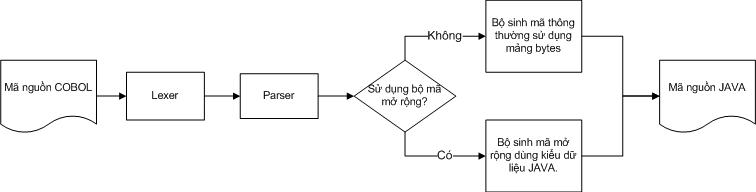
Cả hai cách hiện thực trên đều có những ưu và nhược điểm nhất định phù hợp cho những mục đích và ngữ cảnh chuyển đổi khác nhau. Phân tích cụ thể từng cách hiện thực:

* Phương pháp mô phỏng quá trình quản lý bộ nhớ của ngôn ngữ COBOL: ý tưởng chính cho việc mô phỏng này là thực hiện lưu trữ dữ liệu trong một cấu trúc dữ liệu có tính liên tục và hiện thực các phương thức hỗ trợ để gán và lấy dữ liệu.
  + Cách làm này đòi hỏi phải xây dựng một thư viện lập trình (đi kèm theo mã nguồn sinh ra) hỗ trợ các chức năng:
    - Quản lý một khối dữ liệu liên tục theo đặc tả của ngôn ngữ COBOL. Một cách hiện thực đơn giản là dùng một mảng byte để lưu tất cả các biến trong một chương trình COBOL.
    - Chuyển đổi từ dạng dữ liệu trung gian (ví dụ mảng byte) sang dạng dữ liệu chuẩn của JAVA (short, int, long, String) và ngược lại. Chức năng này đảm bảo cho mã nguồn sinh ra gần gũi hơn với những lập trình viên JAVA.
  + Ưu điểm của cách chuyển đổi này sẽ mang lại sự dễ dàng khi hiện thực các phép gán, lấy dữ liệu và các mệnh đề đặc biệt (REDEFINES, RENAMES).
  + Tuy nhiên, do có sự chuyển đổi qua lại giữa dạng dữ liệu trung gian và kiểu dữ liệu chuẩn của JAVA, hiệu suất của mã nguồn sinh ra sẽ bị ảnh hưởng.
* Phương pháp thực hiện chuyển đổi tương đương: ý tưởng chính của phương pháp này là tìm kiếm những kiểu dữ liệu tương đương giữa hai ngôn ngữ từ đó xây dựng phép chuyển đổi phù hợp.
  + Ưu điểm của cách làm này là không cần xây dựng thư viện lập trình hỗ trợ.
  + Mã nguồn sinh ra sẽ tương tự như một chương trình viết bằng JAVA thông thường.
  + Hiệu suất của mã nguồn sinh ra khi chạy sẽ tốt hơn, do không sử dụng dữ liệu trung gian và các phép chuyển đổi trên dữ liệu này.
  + Tuy vậy, việc hiện thực các mệnh đề đặc biệt như REDEFINES hay RENAMES sẽ trở nên khó khăn hơn (do các biến chuyển sang biến mang kiểu dữ liệu JAVA sẽ không được cấp phát liên tục)

* 1. Phương án được sử dụng:

Với những ưu và nhược điểm của hai phương án được trình bày ở mục 3.1, cả hai phương án đều được hiện thực trong nội dung đề tài nhưng với những thay đổi nhất định cho phù hợp với những ngữ cảnh chuyển đổi nhất định. Hai phương án này được hiện thực theo thứ tự thành hai lựa chọn cho người sử dụng: chuyển đổi thông thường và chuyển đổi mở rộng.

Mô hình quá trình hoạt động tổng quát của chương trình:



Do định hướng đề tài tập trung vào quá trình chuyển đổi mã nguồn COBOL sang JAVA một cách tự động, phần lexer và parser (*dịch ra*) được sử dụng lại từ mã nguồn của chương trình mã nguồn mở RES. Kết quả nhận được sau quá trình này là cây AST (*giải thích…)* sẽ được sử dụng cho quá trình chuyển mã.

Phép chuyển đổi các phần cơ bản được thực hiện chung trong cả hai bộ sinh mã như sau:

* + Một chương trình COBOL sẽ được chuyển thành một lớp (class) tương ứng trong ngôn ngữ JAVA. Lớp này có chứa hàm *main*, là nơi chứa các đoạn code thực thi của chương trình (được chuyển đổi từ mục Procedure Division).
  + Mỗi nhóm (group) của ngôn ngữ COBOL sẽ được chuyển thành một lớp trong ngôn ngữ JAVA. Các nhóm con trong một nhóm cha sẽ được chuyển thành các lớp con tương ứng (inner class) nằm trong lớp cha. Các lớp này không chứa hàm main.
  + Các biến thành phần sẽ không được truy xuất trực tiếp mà được chuyển thành các phương thức lấy/gán (get/set) giá trị dữ liệu.
  + Đối với biến được định nghĩa với số thứ tự cấp là 88, đây là những biến có giá trị phụ thuộc vào biến mà nó bổ trợ. Vì vậy mà cách chuyển đổi cho những biến kiểu này cũng có sự khác biệt :
    - Với phép lấy dữ liệu, mỗi dòng định nghĩa với số thứ tự cấp 88 sẽ được chuyển thành một phương thức có giá trị trả về là đúng hoặc sai.
    - Với phép gán dữ liệu, mỗi dòng định nghĩa sẽ được chuyển tương ứng thành một phương thức có nhiệm vụ gán giá trị nhận được vào cho biến mà nó hổ trợ.

Ví dụ:

**01 EMPLOYEE-RECORD.**

**05 EMPLOYEE-NAME PICTURE X(100).**

**05 EMPLOYEE-POSITION PICTURE 9(1).**

**88 MANAGER VALUE 0.**

**88 EMPLOYEE VALUE 1.**

Ví dụ trên có hai biến được định nghĩa có số thứ tự cấp là 88: MANAGER và EMPLOYEE. Giá trị của hai biến này sẽ phụ thuộc vào giá trị của biến EMPLOYEE-POSITION. Đoạn mã chuyển đổi dành cho biến MANAGER và biến EMPLOYEE sẽ được sinh ra như sau:

// 88 MANAGER VALUE 0.

**public** **boolean** getMANAGER() {

**short** val = getEMPLOYEE\_POSITION();

**return** Compare.*equal*(val, 0);

}

**public** **void** setMANAGER() {

setEMPLOYEE\_POSITION(0);

}

// 88 EMPLOYEE VALUE 1.

**public** **boolean** getEMPLOYEE() {

**short** val = getEMPLOYEE\_POSITION();

**return** Compare.*equal*(val, 1);

}

**public** **void** setEMPLOYEE() {

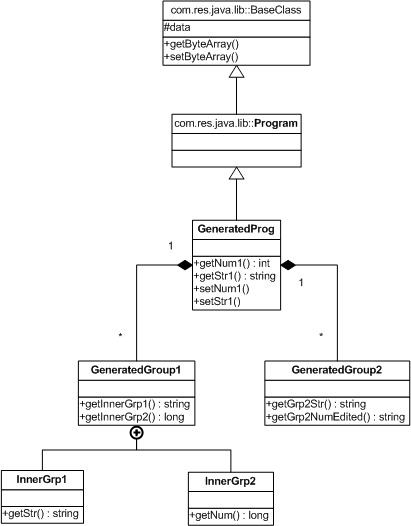
setEMPLOYEE\_POSITION(1);

}

Để việc sinh mã được dễ dàng, ­­­trong đề tài có hiện thực thêm một thư viện lập trình bổ trợ với các chức năng:

* + Hỗ trợ các phương thức chuyển đổi giữa các định dạng dữ liệu: mảng bytes sang kiểu dữ liệu JAVA và ngược lại.
  + Hỗ trợ các phương thức thực hiện định dạng dữ liệu (dành cho kiểu dữ liệu numeric-edited và alphanumeric-edited).
  + Thư viện này sẽ được tích hợp (import) trong mã nguồn sinh ra sau khi chuyển đổi, nhằm giảm bớt số dòng lệnh phải sinh ra, mã nguồn sinh ra sẽ tập trung vào những đoạn mã tương đương giữa hai ngôn ngữ. Thư viện này được hiện thực trong gói (package) com.res.java.lib.
  + Các lớp chính trong bộ thư viện này bao gồm:
    - BaseClass: chứa các phương thức chuyển đổi từ mảng bytes sang kiểu dữ liệu JAVA và ngược lại. Lớp này còn chứa một trường (field) kiểu mảng bytes là data, các phương thức chuyển đổi sẽ được thực hiện trên trường này.
    - Program: chứa phần hiện thực bằng mã nguồn JAVA của các câu lệnh COBOL.
    - EditedVar: chứa các phương thức để định dạng dữ liệu cho kiểu dữ liệu numeric-edited và alphanumeric-edited.

Sơ đồ lớp tổng quát:



Những chương trình COBOL khi được chuyển đổi qua ngôn ngữ JAVA sẽ có sơ đồ lớp tổng quát như trên. Mỗi chương trình COBOL được chuyển đổi thành một lớp (class) chính, lớp này sẽ thừa kế từ lớp Program. Mỗi biến mang kiểu dữ liệu nhóm được chuyển thành một lớp (class), lớp này được thừa kế từ lớp BaseClass. Các nhóm con sẽ được chuyển thành lớp nằm trong (inner class) của lớp cha.

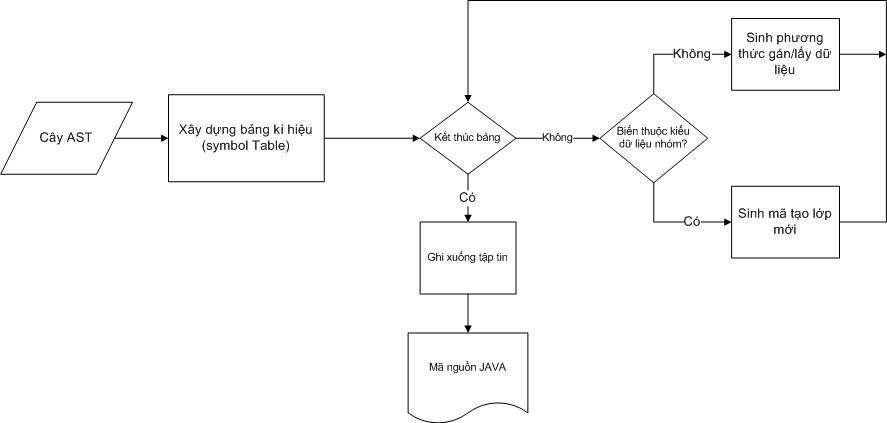
Với sơ đồ trên, vai trò các lớp tương ứng như sau:

* + GeneratedProg: đại diện cho lớp chính sinh ra tương ứng với mỗi chương trình COBOL.
  + GeneratedGroup1: đại diện cho lớp sinh ra từ kiểu dữ liệu nhóm.
  + GeneratedGroup2: đại diện cho lớp sinh ra từ kiểu dữ liệu nhóm.
  + InnerGrp1, InnerGrp2: đại diện cho lớp nằm trong sinh ra bởi kiểu dữ liệu nhóm (nhóm con nằm trong nhóm cha là GeneratedGroup1).
    1. Chuyển đổi thông thường:

Đây là chế độ chuyển đổi mô phỏng quá trình cấp phát và quản lý bộ nhớ của ngôn ngữ COBOL thông qua việc sử dụng một mảng bytes để lưu trữ dữ liệu. Các tính chất của mảng bytes này:

* + Mảng bytes này mang tầm vực *protected* (chỉ có thể truy xuất bởi các lớp con). Với hiện thực trong đề tài này, mảng bytes này chính là trường *data* trong lớp BaseClass, các lớp được sinh ra từ chương trình COBOL sẽ kế thừa từ lớp BaseClass.
  + Mỗi chương trình COBOL được chuyển sang một lớp của JAVA sẽ chỉ có duy nhất một mảng bytes chứa toàn bộ dữ liệu của chương trình đó.
  + Với những biến được định nghĩa với kiểu dữ liệu nhóm nhưng có sự phân cấp, mảng bytes của lớp con sẽ chỉ là con trỏ chỉ tới mảng bytes của lớp cha.
  + Độ dài của mảng bytes sẽ được tính toán trong khi sinh mã.

Sơ đồ hoạt động chung của chế độ chuyển đổi thông thường:



Cách hiện thực này cần có các phương thức hỗ trợ cho việc chuyển đổi dữ liệu trong mảng byte sang các kiểu dữ liệu JAVA thân thiện với lập trình viên JAVA (int, String, long…). Các phương thức này được hiện thực trong lớp BaseClass. Chi tiết việc chuyển đổi được thực hiện theo bảng sau:

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu dữ liệu** | **Dạng khai báo** |  | **Mã JAVA tương ứng** |
| Số nguyên  (numeric) | PIC 9(1 🡪 4)  PIC S9(1 🡪 4) | 🡪 | short getData(); |
| PIC 9(5 🡪 9)  PIC S9(5 🡪 9) | int getData(); |
| PIC 9(10 🡪 18)  PIC S9(10 🡪 18) | long getData(); |
| Số thực  (numeric) | PIC 9999V99  PIC S9999V99 | BigDecimal getData(); |
| Số được định dạng (numeric-edited) | PIC 999/999.999  PIC ZZZ,ZZZ.999 | String getData(); |
| Kí tự  (alphabetic) | PIC A(1 🡪 ) | String getData(); |
| Chuỗi  (alphanumeric) | PIC X(1 🡪 ) | String getData(); |
| Chuỗi được định dạng (alphanumeric-edited) | PIC X(4)/X(2)BX(2) | String getData(); |

Bảng .

Các kiểu dữ liệu số thực không có các phương thức đưa về kiểu float hay double trong JAVA là do kiểu số thực của ngôn ngữ COBOL không phải là kiểu số thực dấu chấm động. Trong khi đó, kiểu dữ liệu BigDecimal có hiện thực phù hợp cho phép biểu diễn số thực một cách chính xác theo đặc tả của ngôn ngữ COBOL.

Trong khi đó, với các kiểu dữ liệu được định dạng (numeric-edited và alphanumeric-edited), việc định dạng dữ liệu theo mẫu khai báo sẽ được thực hiện bằng lớp EditedVar. (Ví dụ here)

Đối với kiểu dữ liệu bảng (Table), ngoài hai phương thức để lấy/gán giá trị của mảng còn có phương thức để lấy giá trị của các phần tử trong mảng dựa vào chỉ số phần tử.

Hiện thực mệnh đề REDEFINES:

* + Đối với biến mang kiểu dữ liệu thành phần (numeric, numeric-edited, alphanumeric, alphanumeric-edited) trong câu khai báo có mệnh đề REDEFINES đối với một biến khác, các phương thức lấy/gán (get/set) dữ liệu của biến này sẽ được thực hiện trên cùng đoạn mảng bytes tương ứng với biến mà nó định nghĩa lại (redefines).
  + Đối với biến mang kiểu dữ liệu nhóm trong câu khai báo có mệnh đề REDEFINES đối với một biến khác, biến này sẽ được chuyển đổi thành một lớp như bình thường. Tuy nhiên mảng bytes của lớp này sẽ là con trỏ, trỏ đến đoạn mảng bytes của biến mà nó định nghĩa lại (redefines).

Ví dụ:

**01 EMPLOYEE-RECORD.**

**05 EMP-ID PICTURE X(10).**

**05 DEP-ID REDEFINES EMP-ID PICTURE X(3).**

**01 DETAIL-RECORD REDEFINES EMPLOYEE-RECORD.**

**05 FIRST-PART-ID PICTURE X(3).**

**05 SECOND-PART-ID PICTURE X(7).**

Trong đoạn code COBOL ở trên, EMP-ID là một biến thành phần mang kiểu dữ liệu *alphanumeric* với độ dài là 10. Biến DEP-ID định nghĩa lại biến EMP-ID và cũng có kiểu dữ liệu *alphanumeric* với độ dài là 3. Giá trị trả về khi truy xuất biến DEP-ID sẽ là 3 kí tự đầu được chứa trong biến EMP-ID,

Vì EMP-ID là biến khởi đầu của nhóm EMPLOYEE-GROUP và có độ dài bằng 10 nên phương thức lấy giá trị của biến này sẽ chuyển đổi các giá trị trên mảng bytes bắt đầu từ vị trí 0 đến vị trí 9. Biến DEP-ID có giá trị là 3 kí tự đầu của EMP-ID nên phương thức lấy giá trị của biến này sẽ làm việc từ vị trí 0 đến 2 của mảng bytes trên. Đoạn code JAVA sẽ được sinh ra như sau:

//05 EMP-ID PICTURE X(10).

**public** String getEMP\_ID() {

**return** getStringDisplay(0, 10);

}

// 05 DEP-ID REDEFINES EMP-ID PICTURE X(3).

**public** String getDEP\_ID() {

**return** getStringDisplay(0, 3);

}

**public** **void** setDEP\_ID(**long** input) {

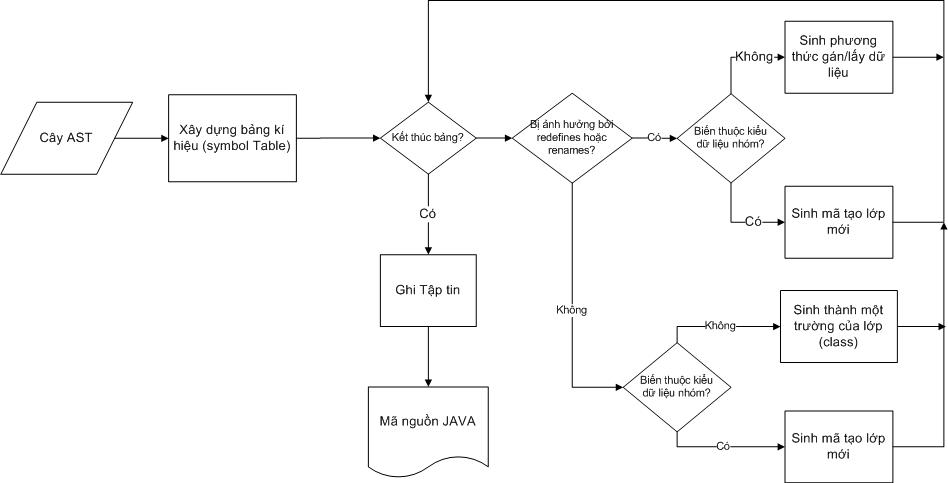
setStringDisplay(unsignedValue(input), 0, 3, **false**);

}

* + 1. Chuyển đổi mở rộng sinh ra kiểu dữ liệu JAVA.

Đây là chế độ chuyển đổi mà các kiểu dữ liệu COBOL sẽ được đưa về kiểu dữ liệu JAVA tương đương. Ở chế độ này, thay vì sử dụng một mảng byte để lưu trữ dữ liệu, các biến thành phần được chuyển đổi tương đương một trường (field) của lớp (class) chứa nó ( nhóm cha).

Sơ đồ hoạt động chung của các chuyển đổi này:



Tuy nhiên cách hiện thực này chỉ áp dụng được cho những biến kiểu nhóm (group) hoặc biến thành phần không xuất hiện trong bất cứ câu mệnh đề REDEFINES hoặc RENAMES nào. Với những trường hợp có xuất hiện mệnh đề (clause) REDEFINES hoặc RENAMES, việc chuyển đổi sẽ được thực hiện như cách chuyển đổi thông thường.

Việc chuyển đổi tương đương được thực hiện như sau:

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu dữ liệu** | **Dạng khai báo** |  | **Kiểu chuyển đổi** |
| Số nguyên  (numeric) | PIC 9(1 🡪 4)  PIC S9(1 🡪 4) | 🡪 | short |
| PIC 9(5 🡪 9)  PIC S9(5 🡪 9) | int |
| PIC 9(10 🡪 18)  PIC S9(10 🡪 18) | long |
| Số thực  (numeric) | PIC 9999V99  PIC S9999V99 | BigDecimal |
| Số được định dạng (numeric-edited) | PIC 999/999.999  PIC ZZZ,ZZZ.999 | String |
| Kí tự  (alphabetic) | PIC A(5) | String |
| Chuỗi (alphanumeric) | PIC X(9)9(2) | String |
| Chuỗi được định dạng (alphanumeric-edited) | PIC X(4)/X(2)BX(2) | String |

Bảng .

Đối với kiểu dữ liệu nhóm, việc chuyển đổi sẽ được thực hiện tương tự như cách chuyển đổi thông thường, nghĩa là mỗi biến kiểu nhóm sẽ được chuyển đổi thành một lớp (class). Với các nhóm có sự phân cấp, các nhóm con sẽ được chuyển đổi thành lớp con (inner class) nằm trong lớp cha, các lớp con này cũng thừa kế từ lớp BaseClass. Tuy nhiên các phương thức lấy/gán (get/set) các biến dữ liệu thành phần của nhóm không dựa trên mảng bytes (lữ trữ và chuyển đổi sang kiểu dữ liệu JAVA) mà sử dụng các biến kiểu dữ liệu JAVA. Ví dụ:

**01 EMPLOYEE-RECORD.**

**05 EMP-ID PICTURE X(10).**

**05 EMP-NAME PICTURE X(100).**

Đoạn mã JAVA tương ứng:

// 05 EMP-ID PICTURE X(10).

**private** StringField eMP\_ID = **new** StringField(0, 10, **false**);

**public** String getEMP\_ID() {

**return** eMP\_ID.getValue();

}

**public** **void** setEMP\_ID(String input) {

eMP\_ID.setValue(input);

}

// 05 EMP-NAME PICTURE X(100).

**private** StringField eMP\_NAME = **new** StringField(10, 100, **false**);

**public** String getEMP\_NAME() {

**return** eMP\_NAME.getValue();

}

**public** **void** setEMP\_NAME(String input) {

eMP\_NAME.setValue(input);

}

Tuy nhiên, phân tích ở mục 1.1.2 đã cho thấy việc chuyển đổi tương đương đơn thuần như vậy sẽ gặp khó khăn khi thực hiện các phép gán hay lấy giá trị của các biến kiểu nhóm (group). Do đó, cách hiện thực được lựa chọn là vẫn giữ lại mảng byte dữ liệu. Mảng byte này sẽ được sử dụng trong việc hiện thực các phép gán/ lấy dữ liệu của các biến kiểu nhóm (group). Cụ thể:

* + - Mỗi phép lấy dữ liệu của biến kiểu nhóm: các biến thành phần con sẽ được chuyển đổi từ kiểu dữ liệu JAVA về kiểu lưu trữ trong mảng byte. Dữ liệu trả về sẽ chính là mảng byte này.
    - Mỗi phép gán các biến kiểu nhóm: *🡪 ví dụ*
      * Nếu là phép gán hai nhóm cùng kiểu (cùng là thể hiện của một đối tượng): thực hiện việc gán từng thành phần của nhóm tương ứng cho nhau. Trường hợp này không sử dụng mảng bytes trung gian.
      * Nếu là phép gán hai nhóm khác kiểu: thực hiện việc copy (sao chép) mảng byte của nhóm này vào nhóm kia (các giá trị này lấy được sao khi thực hiện phép lấy dữ liệu kiểu nhóm).

Với cách hiện thực này, việc gán và lấy dữ liệu có thể chậm hơn cách hiện thực ở 1.2.1. Tuy nhiên, việc truy xuất các biến thành phần được chuyển đổi trực tiếp thành các kiểu dữ liệu của JAVA sẽ đạt hiệu suất cao hơn. (Các phép đánh giá và so sánh hai kiểu chuyển đổi này sẽ được trình bày cụ thể ở Chương 5).

Ví dụ:

**01 EMPLOYEE-RECORD.**

**05 EMP-ID PICTURE X(10).**

**05 EMP-NAME PICTURE X(100).**

Với đoạn mã COBOL tương tự ví dụ trước, tuy các biến thành phần được hiện thực bằng cách sử dụng kiểu dữ liệu JAVA. Nhưng lớp EMPLOYEE\_RECORD vẫn có những phương thức \_setToBytes() và \_getFromBytes() để thực hiện lấy/gán (get/set) giá trị của nhóm từ mảng bytes dữ liệu. Vì giá trị của một nhóm được ngôn ngữ COBOL xem như một giá trị mang kiểu dữ liệu *alphanumeric* nên phương thức \_setValue() sẽ nhận thông số đầu vào là một chuỗi. Đoạn mã JAVA được chuyển đổi tương ứng:

**public** **class** EMPLOYEE\_RECORD **extends** BaseClass {

**public** **void** \_setValue(String input) {

setStringDisplay(input, 0, 110, **false**);

\_getFromBytes();

}

**public** **void** \_setToBytes() {

eMP\_ID.setCurrentValueToBytes();

eMP\_NAME.setCurrentValueToBytes();

}

**public** **void** \_getFromBytes() {

eMP\_ID.getCurrentValueFromBytes();

eMP\_NAME.getCurrentValueFromBytes();

}

}

* 1. Khó khăn trong quá trình thực hiện:

Mọi phép chuyển đổi từ mã nguồn COBOL sang JAVA trong đề tài luôn hướng tới việc đảm bảo tính chính xác:

* + Sự nhập nhàng trong kiểu dữ liệu số và chữ với những biến được định nghĩa với mệnh đề USAGE DISPLAY.
  + Vị trí của các bit/byte dấu trong quá trình lấy dữ liệu từ mảng bytes.
  1. Phần mở rộng:

Bên cạnh chuyển đổi phần dữ liệu từ ngôn ngữ COBOL sang JAVA, đề tài còn thực hiện chuyển đổi thêm một số câu lệnh. Các câu lệnh

* + DISPLAY:
  + MOVE:

# Công cụ được hiện thực

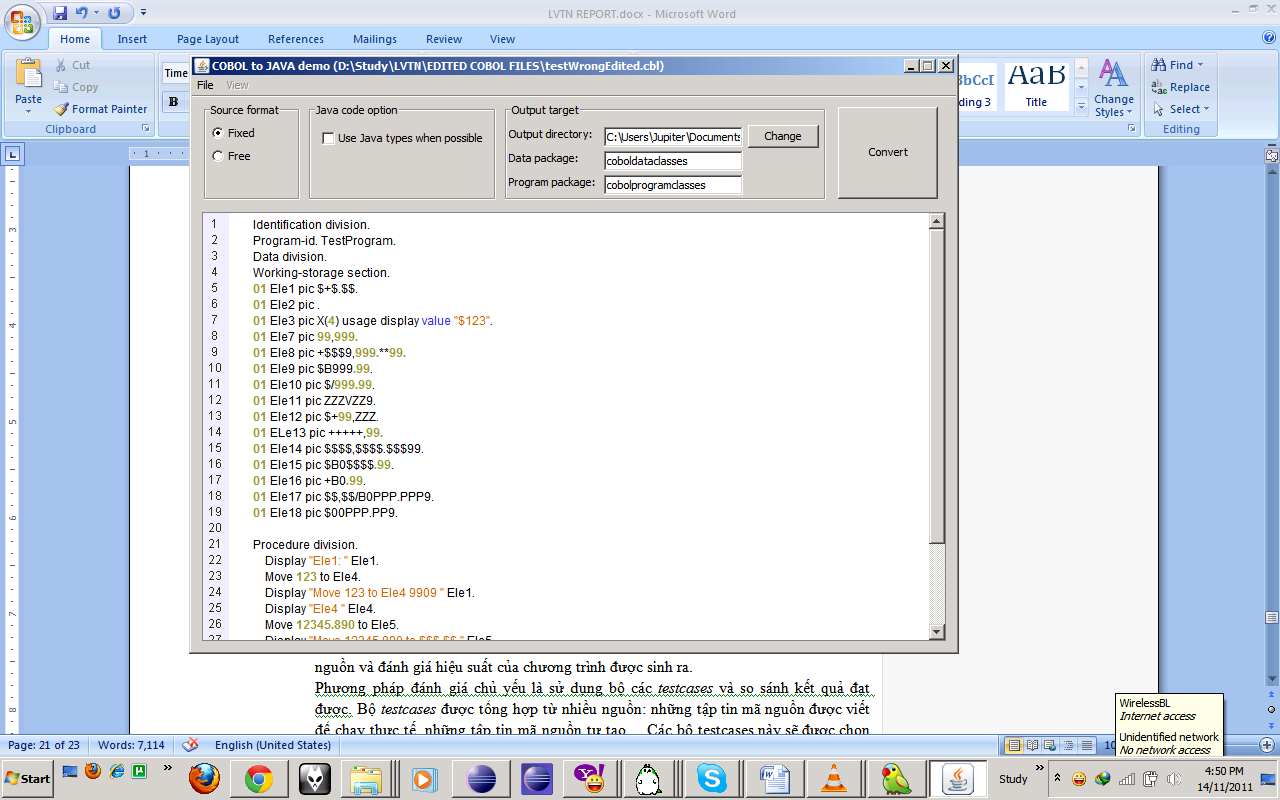
* 1. Giới thiệu về công cụ:

Nhằm tăng tính ứng dụng của đề tài, trong luận văn còn xây dựng thêm một công cụ với giao diện đơn giản hỗ trợ người dùng trong quá trình chuyển mã.

Các tính năng chính của công cụ:

* + Hỗ trợ tinh chỉnh các tùy chọn trong quá trình chuyển mã: vị trí đích, tên gói (package) sau khi chuyển mã, bật/tắt tùy chọn chuyển mã sử dụng kiểu JAVA.
  + Cho phép chỉnh sửa mã nguồn COBOL ngay trong giao diện của chương trình.
  + Cung cấp giao diện so sánh mã nguồn trước và sau khi chuyển mã.

Giao diện chính của chương trình:



* 1. Hướng dẫn sử dụng:

# Đánh giá và kết quả

* 1. Phương pháp đánh giá:

Công việc đánh giá kết quả đạt được gồm hai phần: đánh giá tính đúng đắn của mã nguồn và đánh giá hiệu suất của chương trình được sinh ra.

Phương pháp đánh giá chủ yếu là sử dụng bộ các *testcases* và so sánh kết quả đạt được. Bộ *testcases* được tổng hợp từ nhiều nguồn: những tập tin mã nguồn được viết để chạy thực tế, những tập tin mã nguồn tự tạo… Các bộ testcases này sẽ được chọn lọc trước khi sử dụng. Quá trình chọn lọc sẽ loại bỏ bớt những đoạn mã nguồn COBOL không liên quan đến mục tiêu chuyển đổi của đề tài, chỉ giữ lại những đoạn mã nguồn như: khai báo biến, các phép lấy/gándữ liệu, các câu lệnh đã được hiện thực trong phần mở rộng (DISPLAY, MOVE ..)

Để đánh giá tính đúng đắn của mã nguồn JAVA, mã nguồn JAVA sinh ra sẽ được biên dịch, chạy và so sánh với kết quả đạt được từ việc biên dịch mã nguồn COBOL bằng chương trình OpenCobol. Bên cạnh đó, đặc tả kĩ thuật của ngôn ngữ COBOL cũng sẽ được sử dụng để thực hiện việc đối chiếu.

Đối với việc đánh giá hiệu suất của chương trình sinh ra, hai công việc sẽ được thực hiện:

* So sánh hiệu suất giữa kết quả sinh ra của hai chế độ chuyển đổi.
* So sánh hiệu suất giữa kết quả sinh ra với kết quả của một giải pháp có sẵn (chương trình được chọn trong đề tài này là RES).
  1. Mô tả bộ testcases sử dụng:
  2. Kết quả:

# Tổng kết