Thông tin liên hệ cán bộ trực đề:

- Người trực đề: TRẦN NGỌC BẢO DUY

- Email: duytnb@hcmut.edu.vn

- Phòng họp có thể liên lạc trực tiếp: https://meet.google.com/xmn-qbsh-qww

- Số điện thoại liên lạc: Cô Nguyễn Thị Kim Cương (0985 420 181) hoặc Trần Ngọc Bảo Duy (033 547 4602, chỉ liên lạc về đề thi trong trường hợp thực sự cần thiết).

Sử dụng ngôn ngữ lập trình tuỳ theo lĩnh vực:

- Lĩnh vực khoa học: Fortran, ALGOL 60.
- Lĩnh vực kinh tế: COBOL.
- Trí tuệ nhân tạo: LISP, Prolog.
- Chương trình hệ thống: PL/S, BLISS, Extended ALGOL, C.
- Web: XHTML, JavaScript, PHP.

Đặc tính ngôn ngữ:

- Đơn giản: dễ học.
- Tính trực giao: đặc tính ngôn ngữ không ảnh hưởng lẫn nhau. (Đồ thị 2 chiều khi di chuyển chiều này thì chiều kia ko thay đổi). Ví dụ: C có thể khai báo các biến với nhiều kiểu, đó là sự trực giao. Nhưng C tồn tại void ko thể khai báo biến trừ ngoài con trỏ, làm mất tính trực giao → cần ghi nhớ trường hợp ngoại lệ → việc học khó khăn hơn.
- Hỗ trợ vấn đề trựu tượng (điều khiển, dữ liệu): trừu tượng hoá về điều khiển (nhiều dòng lệnh được thay thế bằng 1 tên, khi dùng tên đó thì có được chuỗi dòng lệnh đó → chương trình con), trừu tượng hoá về dữ liệu (mô tả được dữ liệu).
- Tính an toàn: phần mềm không gây nguy hiểm đối với người dùng khi người dùng sử dụng sai. Khi chúng ta lập trình sai, ngôn ngữ lập trình sẽ giúp chúng ta biết chúng ta sai hoặc phòng ngừa, không làm chương trình gây ra hậu quả nghiệm trọng → ngôn ngữ lập trình có tính an toàn.

Đánh giá ngôn ngữ lập trình:

- Tính dễ đọc.
- Tính dễ viết.
- Tính đáng tin cậy: phát hiện được lỗi sai.
- Chi phí: chi phí lập trình, bảo trì chương trình, nâng cấp, …

Thiết kế nguôn ngữ:

- Kiến trúc máy tính: khi thiết kế cần khớp kiến trúc máy tính để hiện thực đơn giản, hiệu suất tốt hơn.
- Phương pháp luập trình: ngôn ngữ hỗ trợ cho trường pháp lập trình nào: lập trình thủ tục (how – làm thế nào giải quyết vấn đề: machine – based, procedural), lập trình khai báo (what – vấn đề

là gì → có cơ chế tìm lời giải: logic, functional, constraint, query – based), lập trình hướng đối tượng, …

Hiện thực ngôn ngữ:

- Hiểu về kiến trúc máy tính: Von Neumann.
- Ngôn ngữ Imperative (Mệnh lệnh) thiết kế dựa hoàn toàn tên mô hình máy Von Neumann.
  - Dữ liệu và chương trình được lưu trữ trong bộ nhớ.
  - Bộ nhớ được tách khỏi CPU.
  - Lệnh và dữ liệu được truyền từ bộ nhớ vào CPU.
  - Cấu trúc cơ bản của ngôn ngữ mệnh lệnh.
  - Biến được mô hình hoá cho các ô nhớ (có thể đọc, ghi trên đó).
  - Lệnh gán để ghi vào ô nhớ.
  - Tồn tại vòng lặp do mô hình có đầu đọc để di chuyển trên bộ nhớ, dùng vòng lặp sẽ hiệu quả.

Đánh đổi trong việc thiết kế ngôn ngữ lập trình:

- Tính tin cậy và thời gian thực thi: Java kiểm tra nhiều để đảm bảo thực thi đúng thì sẽ tốn thêm thời gian thực thi.
- Tính dễ đọc và tính dễ viết: APL có quá nhiều phép toán → dễ viết nhưng khó nhớ để đọc.
- Tính dễ viết và tính tin cậy: C++ con trỏ quá linh hoạt nhưng có thể gặp lỗi.

Phương pháp hiện thực ngôn ngữ lập trình:

- Trình biên dịch: chương trình được compiler dịch toàn bộ sang mã máy để thực thi.
- Trình thông dịch: chương trình được interpreter dịch và thực thi từng dòng. Dòng kế tiếp được dịch sẽ phụ thuộc và việc thực thi của dòng trước đó.
- Hệ thống thực thi lai: kết hợp giữa biên dịch và thông dịch, vừa có compiler vừa có interpreter. (Java dùng JavaC để dịch sang Java Byte Code → dùng máy ảo Java là interpreter để dịch sang mã máy, C# dịch ra mã trung gian rồi dùng máy ảo của Microsoft để chạy mã đó).
- Just-in-time compiler: bên trong interpreter có compiler, compiler chỉ dịch những phương thức quan trọng. Thay vì chạy từng dòng, có những dòng được sử dụng nhiều lại phải dịch lại khi sử dụng bởi interpreter, compiler sẽ dịch đoạn chương trình đó sang mã máy. Khi cần sử dụng lại phần đó, chương trình sẽ trực tiếp sử dụng mã máy, không tốn thời gian để dịch lại.

Các bước dịch ngôn ngữ lập trình:

❖ Front end: phụ thuộc ngôn ngữ nguồn.
- Kiểm tra từ vựng (Lexical analyzer): tách các ký tự của chương trình ra theo nhóm cho phù hợp. Chuỗi ký tự được tách thành từng phần nhỏ sau đó ghép lại.
- Kiểm tra cú pháp (Syntax analyzer): kiểm tra lỗi cú pháp. Sinh ra astract tree, nếu không có là lỗi.
- Kiểm tra ràng buộc ngữ nghĩa (Semantic analyzer): biến có khai báo trước đó chưa, …
- Sinh mã trung gian (Intermedia Code Generator).

❖ Back end: phụ thuộc ngôn ngữ trong gian và ngôn ngữ được dịch ra.
- Tối ưu mã (Code Optimizer).
- Sinh ra mã (Code Generator).

Các chương trình liên quan chương trình dịch:

- Bộ tiền xử lý (Preprocessor): xử lý các macro, include file trước khi đưa vào chương trình dịch.
- Assembler: dịch mã assembly được bộ dịch dịch ra sang mã máy.
- Linker (Linker Editor): nối các file riêng biệt thành một file thực thi.
- Loader: lấy file từ bộ nhớ ngoài vào bộ nhớ trong, set up các dữ liệu cần thiết để thực thi.
- Debugger: Built-in Debugger nằm trong IDE, chỉ làm việc được trên ngôn ngữ nguồn. Stand-alone Debugger là debugger riêng, làm việc trên mã máy được sinh ra.

Editor: trình để soạn thảo chương trình.

Kỹ thuật phân tích từ vựng trên chương trình dịch

Vai trò:

- Giống như một bộ tách từ: Đầu vào là 1 chuỗi nhiều ký tự. Bộ phân tích từ vựng đề tác các từ ra từ chuỗi liên tục đó.
- Giống như một bộ kiểm tra chính tả: Sau khi tách từ, các từ sai sẽ ra cảnh báo lỗi.
- Giống một bộ phân lớp: Các từ đúng sẽ phân ra làm loại nào.
- Xác định các chuỗi con (lexemes):
- Trả về tokens: lớp, loại của lexeme.
- Bỏ đi khoảng trắng, xuống hàng vì không có tác dụng.
- Ghi nhận lại vị trí các tokens.

i = a = b * 35 + c5; → 10 tokens {i,=,a,=,b,*,35,+,c5,;}

i += a * 24; → 6 tokens {i,+=,a,*,24,;}

inc = -100; → 5 tokens {inc,=,-,100,;}

a[12] = inc * (dec – 1); → 13 tokens {a,[,12,],=,inc,*,(,dec,-,1,),;}

Các comment sẽ không được tính vào token.

Hiện thực bộ phân tích từ vựng: Có quá nhiều loại ký tự → dùng các luật để phân biệt.

Để mô tả các luật về từ vựng, có 2 phương pháp được sử dụng phổ biến:

- Finite Automata (Automata hữu hạn): hay dùng cho cấp máy.
- Regular Expressions (Biểu thức chính quy): cách biểu diễn gần với con người, dễ hiểu, dễ xây dựng luật hơn.

Automata hữu hạn: 1 tập các trạng thái hữu hạn. Khi đọc chuỗi đến được trạng thái kết thúc → chuỗi đúng với automata hữu hạn này.
Automata hữu hạn xác định (DFA):

- Một tập hữu hạn các trạng thái.
- Một tập các ký hiệu.
- Một trạng thái bắt đầu.
- Tập các trạng thái kết thúc.
- Hàm chuyển từ trạng thái này, thông qua hàm chuyển xác định trạng thái mới.

Automate hữu hạn không xác định (NFA): khác Automate hữu hạn xác định vài điểm:

- Chấp nhận chuỗi rỗng, đi từ trạng thái này sang trạng thái khác mà không đọc gì hết.
- Có nhiều trạng thái kế tiếp cho cùng một ký hiệu nhập.
- Giúp automate đơn giản hơn.
- Một NFA hoàn toàn tương đương 1 DFA.

Biểu thức chính quy:

- Các ký hiệu cơ bản khác ( ) | *.
- Ký hiệu (ΦøΘ) cho chuỗi rỗng.
- Đặt 2 biểu thức chính quy kế nhau → nối 2 biểu thức chính quy.
- Hợp a | b → các chuỗi từ a hoặc b.
- a*: lặp a hoặc rỗng.
- ( ) để nhóm.
- a+: 1 hoặc nhiều lần ~ aa*.
- a?: có hoặc không có ~ (a|ø).
- [xyz] = x|y|z
- [x-y]: tất cả các ký hiệu từ x đến y, các ký hiệu liên tục.
- [^x-y]: tất các trừ khoảng ký hiệu bên trong.
- .: nối bất kỳ ký hiệu nào.

VD: (E|e)(+|-|ø)(0|1|2|3|4|5|6|7|8|9)+ → [Ee][+-]?[0-9]+

Sử dụng ANTLR (Another Tool for Language Recognition):

Tên file: <tên file>.g4

lexer grammar <tên file>;

INT: [0-9]+;

HEX: 0[Xx][0-9A-Fa-f]+;

ID: [a-z]+;

WS: [¥t¥r¥n]+ -> skip;

dùng *: tham lam, greedy

dùng *?: không tham lam, non-greedy (chỉ bắt token vừa đủ)

Syntax Analysis: Phân tích văn phạm/cú pháp

Là bước nằm sau bước phân tích từ vựng (Lexical analyzer).

Phân tích cú pháp sẽ yêu cầu token. Khi đó, phân tích từ vựng đọc trong source program, lấy 1 token và trả về cho phân tích cú pháp. Phân tích cú pháp sẽ tiến hành phân tích token đó, khi cần token mới thì lại tiếp tục yêu cầu.

Trong quá trình làm việc giữa 2 phần sẽ hình thành bảng danh hiệu (symbol table).

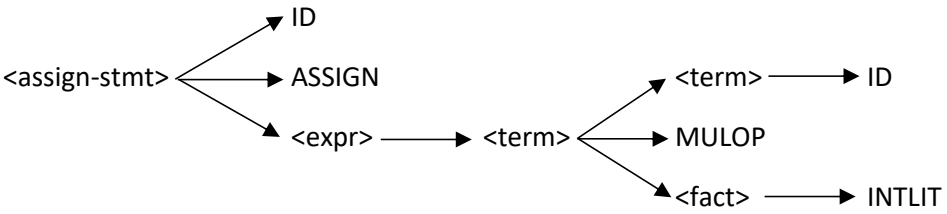
Vai trò của phần tích cú pháp:

- Đọc chuỗi token từ phân tích từ vựng tả ra.
- Nếu chuỗi token đúng thì sẽ tạo ra output là một parse tree hoặc abstract syntax tree (AST – cây cú pháp trừu tượng).
- Nếu chuỗi sai sẽ đưa ra error message.
- Xác định trình tự các token có đúng với các luật văn phạm không.

VD: a = b * 4

Lexer output: ID ASSIGN ID MULOP INTLIT

Parser output:



Biểu thức chính quy không thể mô tả các văn bản có cấu trúc đối xứng hay cấu trúc đệ quy. Ta cần công cụ mô tả mạnh hơn. Ta cần xác định chuỗi token đó là đúng hay sai đối với loại ngôn ngữ này.

➔ Context – free grammar.

Context – free grammar: Văn phạm phi ngữ cảnh gồm:

- **Tập hợp các ký hiệu kết thúc T**.
- **Tập hợp các ký hiệu không kết thúc N**: các khái niệm mình đưa ra để mô tả một cấu trúc của câu. Các ký hiệu không kết thúc của văn phạm phi ngữ cảnh là các ký hiệu xuất hiện bên vế tría của luật sinh. Ký hiệu rỗng (ϵ) dùng để quy ước cho vế phải không có ký hiệu văn phạm nào nên không được xem là ký hiệu kết thúc.
- **Một ký hiệu bắt đầu S thuộc N**: nếu không được đặc tả rõ, ký hiệu bắt đầu của một văn phạm được quy ước là ký hiệu vế trái của luật sinh đầu tiên.
- **Tập các luật sinh P**: luật sinh p thuộc P thoả: X → α. Với X thuộc N và α chuỗi các ký hiệu có thể thuộc cả T và N hoặc chuỗi rỗng.

VD: CFG phép toán nguyên đơn giản.

- T: {ADDOP, MULOP, INTLIT, LB, RB}
- N: {<exp>}
- Ký hiệu bắt đầu S: <exp>
- Tập các luật sinh:
  - <exp> → <exp> ADDOP <exp>
  - <exp> → <exp> MULOP <exp>

➔

<exp> → <exp> ADDOP <exp>
| <exp> MULOP <exp>

- <exp> → INLIT | INTLIT
- <exp> → LB <exp> RB | LB <exp> RB

VD: M → a X b

X → c Y | d

Y → m Y | ϵ

Ký hiệu bắt đầu của văn phạm: M

Ký hiệu không kết thúc của văn phạm: M, X, Y

Ký hiệu kết thúc của văn phạm: {a, b, c, d, m}

Một văn phạm có thể mô tả một ngôn ngữ, dựa vào văn phạm có thể xác định chuỗi nhập có thuộc ngôn ngữ hay không. Hai bài toán thường gặp:

❖ **Quá trình sinh:** Từ văn phạm biết được ngôn ngữ văn phạm vô tả là ngôn ngữ gì, những chuỗi của ngôn ngữ đó là những chuỗi nào → Bài toán sinh/dẫn suất. **Thông qua giải thuật sinh**.

- Từ một ký hiệu bắt đầu, bắt đầu quá trình sinh chuỗi.
- Thay thế các ký hiệu không kết thúc X trong chuỗi với vế phải của luật sinh X → α.
- Lặp lại cho đến khi không còn ký hiệu không kết thúc nào nữa.

Nếu luôn chọn ký hiệu không kết thúc đầu tiên gọi là **leftmostderivation**, nếu luôn chọn bên phải thì **rightmost derivation**, nếu chọn ngẫu nhiên thì là **derivation**.

VD:

<exp> => <exp> MULOP <exp> => INTLIT MULOP <exp> => INTLIT MULOP <exp> ADDOP <exp>

=> INTLIT MULOP INTLIT ADDOP <exp> => INTLIT MULOP INTLIT ADDOP INTLIP

Tập hợp các chuỗi được hình thành từ văn phạm phi ngữ cảnh được gọi là ngôn ngữ phi ngữ cảnh L(G).

❖ **Quá trình nhận dạng:** Cho 1 văn phạm, cho 1 chuỗi. Xác định xem chuỗi đó có được sinh từ văn phạm đó hay không.

- Khi nhận dạng được sẽ tạo 1 cây parse tree: ký hiệu bắt đầu là gốc; luật sinh X → $Y_1Y_2…Y_n$, thêm các node cha là X và $Y_1Y_2…Y_n$ là các node con của X. Các node lá đọc từ trái qua phải chính là chuỗi nhập.

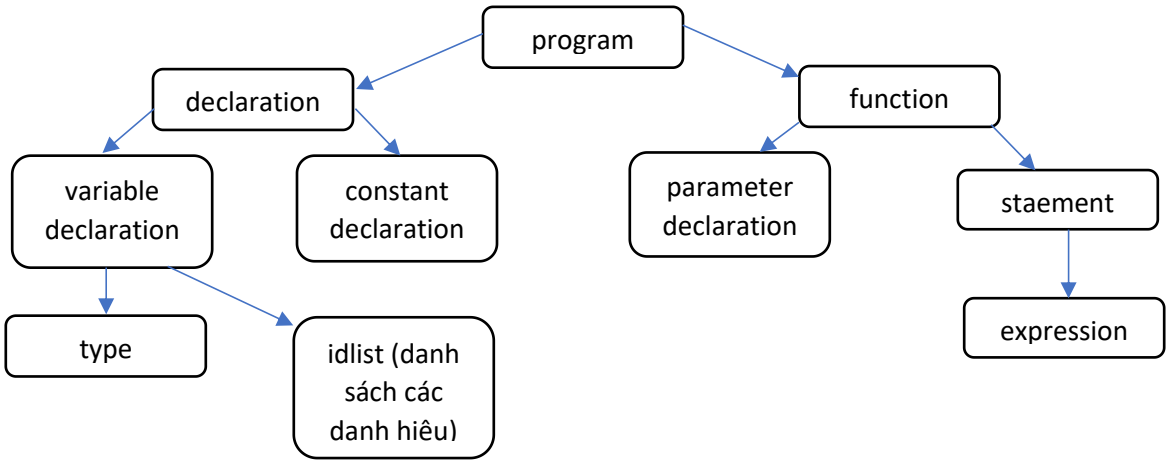
Write a grammar:

❖ **Dựa trên đặc tả ngôn ngữ:** Mỗi ngôn ngữ có đặc tả riêng của nó, đôi khi khác suy nghĩ bình thường. Vì vậy, khi làm không nên dựa vào suy nghĩ thông thường của mình mà phải đọc kỹ đặc tả của ngôn ngữ. Khi có mô tả khác đi, phải tuân theo mô tả đó.

- Smalltalk có độ ưu tiên như nhau cho tất cả phép toán nhị phân.
- C đặc tả >, <, >=, <= có độ ưu tiên cao hơn ==, != trong khi Pascal cho chúng có độ ưu tiên như nhau.

❖ **Cố gắng tìm kiếm cấu trúc phân lớp của ngôn ngữ:** Mỗi ngôn ngữ thường có cấu trúc phân lớp. Cần cố gắng nhìn ra cấu trúc đó → viết văn phạm theo cấu trúc đó thì rất là dễ dàng.

Chương trình C gồm có các khai báo (declaration) và hàm (function):



Các cấu trúc này sẽ chính là những phần nằm bên trái của các tập sinh:

program → … declaration → … variable_decl → … … exp → … …

Trong quá trình làm, chúng ta có thể tạo thêm các cấu trúc khác nhưng những cấu trúc có sẵn sẽ là căn bản để chúng ta phát triển từ đó lên.

- ❖ **Tập trung vào thứ tự xuất hiện của các đơn vị văn phạm (token):** không cần quan tâm về nghĩa hay các ràng buộc khác của chúng. Các ràng buộc về scope (khai báo trước mới được dùng), name resolution (phân giải tên), kiểu dữ liệu các toán hạn có đúng hay không … không thể giải quyết ở bước này.
- ➔ Cần phân biệt phần nào sẽ dành do phân tích văn phạm (thứ tự các token), phần nào dành cho phân tích ngữ nghĩa (scope, thứ tự thực thi phép toán, …).
- ❖ **Cố gắng dùng đệ quy cho các cấu trúc lặp nhiều lần:** Một vài ngôn ngữ cho phép các danh sách vô hạn của các phần tử.
- Khai báo biến có thể có danh sách các định danh (a, c10, b: integer;)
- Khai báo hằng có thể có danh sách các parameter.
- Một block có thể có nhiều lệnh bên trong.

Các mô tả cấu trúc nhiều One vs Many: đặt 1 trước nhiều sẽ là nhiều → dùng đệ quy mô tả cái nhiều dựa trên 1.

<many> → <one> [<separator>] <many>

| <non recursive case>

Đối với vế không đệ quy <non recursive case>: xem vế many ít nhất là bao nhiêu: trong khai báo biến ít nhất là 1; trong parameter và block ít nhất là 0.

- Nếu không có <separator> thì ít nhất có thể là không có gì.
- Nếu có <separator> ít nhất phải là 1.
- ❖ **Dùng đệ quy cho các cấu trúc lồng nhau:** hàm được định nghĩa trong hàm, lệnh trong lệnh, block trong block, … Cấu trúc lồng nhau suất hiện rất nhiều trong ngôn ngữ lập trình → dùng đệ quy để mô tả.

<stmt> → IF <exp> THEN <stmt> ELSE <stmt>

| WHILE <exp> DO <stmt>

| …

Chú ý nhớ làm vế không đệ quy. Chương trình sẽ không kiểm tra lỗi này. Nếu viết sai sẽ dẫn đến lặp vô tận.

Extended Backus – Naur Form (EBNF or RHS):

- Thay vì vế phải chỉ có ký hiệu kết thúc hoặc không kết thúc, được đưa thêm các phép toán của biểu thức chính quy vào.
- Khả năng biểu diễn mạnh.
- Thường được hỗ trợ trong các bộ sinh top – down.

BNF	EBNF (RHS)	ANTLR
<exp> → <exp> '+' <term> | <exp> '-' <term> | <term>	<term> (('+'|'-') <term>)*	exp: term (('+'|'-') term)*;
<else> → ELSE <stmt> | ϵ	(ELSE <stmt>)?	else: ("else" stml)?;
<idlist> → ID ',' idlist | ID	ID (',' ID)*	idlist: ID (",") ID)*;

Một số vấn đề khi viết văn phạm:

❖ **Bị nhập nhằng:** tìm thấy nhiều hơn 1 parse tree cho một trường hợp đúng, có thể làm tối nghĩa hoặc ra nhiều nghĩa khác nhau. Cách sử lý:

- Viết lại văn phạm đó sao cho không còn bị nhập nhằng.

VD: <exp> → <exp> ADDOP <exp> | <exp> MULOP <exp>

| INTLIT

| LB <exp> RB

Đệ quy suất hiện cả 2 phía có thể dẫn đến nhập nhằng → rút về chỉ đệ quy được một phía.

<exp> → <exp> ADDOP < term > | <exp> MULOP < term >

| <term>

<term> → INTLIT

| LB <exp> RB

Bị nhập nhằng:

S → S A | A S → S c S | b

A → A a | a

Không bị nhập nhằng:

S → S A | A S → c S S | b

A → A a | b

- Dùng những công cụ quy định rule để khử nhập nhằng.

Khi dùng ANTLR có thể dùng <assoc=left>, <assoc=right>, default là <assoc=left>. Về các phép toán thì dựa trên thứ tự khai báo, khai báo trước thì ưu tiên cao hơn.

- **Tính kết hợp:** n phép toán có cùng độ ưu tiên kẹp giữa n + 1 toán hạng. Cần xác định phép toán nào thực hiện trước. Kết quả có thể khác nhau khi tính toán thứ tự khác nhau. Tuỳ theo kết hợp trái (left - associated) hay kết hợp phải (right - associated). Nếu kết hợp trái thì tính bên trái trước, đệ quy bên trái. Nếu tính kết hợp phải thì tính phép bên phải trước, đệ quy bên phải. Không có tính kết hợp khi không đệ quy bên nào.

VD: Đệ quy ADDOP dạng kết hợp trái.

<exp> → <exp> ADDOP <term>

Nếu không có tính kết hợp.

<exp> → <term> ADDOP <term>

- **Tính ưu tiên:** để chia các phép toán theo độ ưu tiên, các phép toán độ ưu tiên càng thấp thì viết càng gần ký hiệu bắt đầu.
- **Vấn đề nhập nhằng của mệnh đề IF:**

<stmt> → IF <exp> THEN <stmt> ELSE <stmt>

| IF <exp> THEN <stmt>

| <other>

Ta có biểu thức: IF <exp> THEN IF <exp> THEN <other> ELSE <other>

Ta không xác định được ELSE của IF trong hay IF ngoài.

→ Cần viết lại văn phạm cho không còn nhập nhằng.

<stmt> → <matchStmt>

| <unmatchStmt>

<matchStmt> → IF <exp> THEN <matchStmt> ELSE <matchStmt>

| <other>

<unmatchStmt> → IF <exp> THEN <stmt>

| IF <exp> THEN <matchStmt> ELSE <unmatchStmt>

OOP:

- Mục tiêu: làm sao tận dụng các chương trình đã viết trước đó (tính tái sử dụng - reusable)

- Tính tái sử dụng → giải quyết khủng hoảng phần mềm, các lập trình thủ tục không giúp sử dụng các phần mềm đã tạo ra → dùng OOP để biến các phần mềm thành các đơn vị phần tử độc lập có chức năng.
- Chương trình là sự kết hợp của các khối chức năng, đó là các đối tượng.
- Đối tượng có giao diện riêng, người sử dụng có thể biết nhưng cách hiện thực sẽ bị che dấu (public interface, hiden implementation).
- Phân biệt các đối tượng qua chức năng của nó. Các đối tượng biểu diễn thế giới thực và chúng kết hợp với nhau để tạo ra các chức năng của chương trình.

Có nhiều ngôn ngữ OOP khác nhau:

- Thuần OOP (pure oop): Smalltalk (cái gì cũng là đối tượng, kể cả đoạn code).
- Phát triển từ thủ tục: C++, Ada.
- Không hỗ trợ các cách lập trình khác OOP nhưng phát triển từ lập trình thủ tục: Java, C# (bên trong method vẫn có lập trình thủ tục).
- Hỗ trợ lập trình hàm: Scala.

Một số đặc trưng cơ bản quan trọng của OOP:

- Suy diễn kiểu (type inference).
- ***Trừu tượng dữ liệu (data abstraction).***
- ***Đa hình (Polymorphism)*** → ràng buộc động (dynamic binding)
- Hàm bậc cao (higher – order functions).
- ***Phân cấp lớp (class hierachy).***
- ***Thừa kế (Inheritance).***
- Tính toán lười (lazy evaluation).

Data abstraction: dùng tính bao đóng (encapsulation) và vấn đề che dấu thông tin (information hiding).

- Tính bao đóng: bao tất cả thuộc tính và phương thức vào một đơn vị văn phạm. Những đơn vị văn phạm có thể dịch riêng biệt → Một đơn vị phần mềm trở thành một đơn vị độc lập, có khả năng hoạt động riêng (có chức năng riêng của nó, không bị tác động bởi bên ngoài).
- Che giấu thông tin: chặn tác động từ bên ngoài. Việc truy suất từ bên ngoài vào thành phần bên trong class sẽ được điều khiển. Từ đó tăng sự tin cậy.

Class hierachy:

- Một lớp có thể có nhiều lớp con.
- Một lớp có thể có một hay nhiều lớp cha.
- Lớp cha sẽ trừu tượng hơn, lớp con là một cái gì đó cụ thể của lớp cha của nó → hình thành tính thừ kế.

Inheritance: gia tăng tính tái sử dụng của OOP. Có thể có 1 (đơn kế thừa) và nhiều (đa kế thừa) lớp cha.

- Lớp con có thể thừa kế các phần tử không phải private của lớp cha.
- Lớp con có thể override lại các phần tử được thừa kế.

Diamond Problem in Multiple Inheritance: A có con là B và C, D là con của cả B và C. Nếu A có 1 method, method đó bị override ở B, còn C vẫn giữ nguyên. Khi D thừa kế thì sẽ có 2 version của cùng 1 method.

Cách giải quyết khác nhau với các nguôn ngữ khác nhau:

- khi đa kế thừa, class nào được viết trước sẽ có tính ưu tiên cao hơn.
- khi có vấn đề sẽ raise lỗi.

→ Khi sử dụng ngôn ngữ cần biết rõ ngôn ngữ sẽ giải quyết vấn đề theo cách nào.

Class vs Object:

- Class: bảng mẫu mô tả các đặc tính, hành vi của 1 lớp đối tượng.
- Object: một thực thể cụ thể của bảng mô tả đó.

Method vs Message:

- Method: bảng mô tả hành vi mà đối tượng có thể thực hiện được.
- Message: quá trình method được gọi.

Polymorphism: những đối tượng khác nhau đáp ứng cùng message theo các cách khác nhau.

- Dùng override để tạo các version khác nhau của cùng method.

→ Dùng kỹ thuật Dynamic binding (ràng buộc động) để thực hiện tính đa hình.

Các loại thuộc tính (variable):

- Class variable: tác động lên tất cả đối tượng của class.
- Instance variable: thuộc tính của riêng đối tượng.

Các loại phương thức:

- Class method: chấp nhận message tới class.
- Instance method: chỉ nhận message của đối tượng.

→ Khi gọi phải gọi cho đúng.

Những dạng thuộc tính (field/attribute) trong một class:

- Thuộc tính lớp (class attribute).
- Thuộc tính toàn cục (global attribute).
- Thuộc tính cục bộ (local attribute).

Một instance message có thể tương ứng với nhiều method khác nhau tuỳ theo đối tượng nhận message thuộc lớp nào.

Một class message chỉ tương ứng duy nhất một method.

Khi A là cha của B, A có biến a, B có biến b. a = b đúng b = a sai.

Scala:

- Do Martin Ordesky phát minh.
- Tương tự Java, chạy trên JVM (Java Virtual Machine) nên cả 2 bên kết hợp với nhau rất dễ dàng.
- OOP + FP.
- Giúp tạo lexer và parser.

Class:

- class.
- abstract class: lớp trừu tượng không thể tạo đối tượng.
- trait: dạng giống interface, có thể đặt các phương thức, 1 class sẽ thừa kế và implement trait, trait không thể tạo đối tượng cụ thể → thuận tiện cho việc kế thừa, stackable modification (thay vì phải kế thừa lại 1 class để override 1 method, ta dùng trait override method vào gán lại vào class đó).
- case class: một class không cần dùng lệnh new để tạo đối tượng. Giúp bớt đi chữ new, khi viết các cấu trúc phức tạp thì sẽ tự nhiên hơn.

Object:

- new <class name>
- <case class name>: dùng cho case classs
- object <class name>: có thể xem là 1 phần của class. Các method và attribute khai báo trong class sẽ là instance, còn được khai báo bên trong object sẽ là static.

Stackable modification:

```
abstract class A {

        def put (x: Int)}

class B extends A {

        private val buf = new ArrayBuffer [Int]

        def put (x: Int) {buf += x}}

trait Doubling extends A {

        abstract override def put (x: Int) {super.put (2*x)}}

trait Incrementing extends A {

        abstract override def put (x: Int) {super.put (x+1)}}

val C = new B with Incrementing with Doubling

queue.put (10)
```

- Đầu tiên x = 10 sẽ được Doubling lên 20, sau đó được Incrementing lên 21. Kế đó, 21 mới truyền vào put của class B để buf tăng lên 21.

Access Modifiers:

- public
- protected
- private
- protected[<name>]
- privated[<name>]

Các lớp sẽ có 1 package. Nếu name là 1 ID nào đó, nó sẽ có thể truy cập đó trong cả package. Nếu name là this thì nó chỉ có thể truy cập trong chính đối tượng chứa nó.

Functional Programming:

- Các hàm sẽ được xem như là các giá trị, chúng không có tên mà chỉ định nghĩa như các giá trị thông thường.
- Có thể gán hàm cho một biến, truyền 1 hàm vào 1 hàm khác như truyền giá trị và nhận 1 hàm được trả ra như nhận giá trị được trả ra.
- Các hàm có thể tác động với nhau để thực thi (như có 3 + 4 thì ta có f o g (từ thông số truyền vào tính toán hàm g trước, sau đó dùng kết quả tính toán hàm f)).

Về lý thuyết nền tản:

- Ngôn ngữ lập trình hướng thủ thục → dùng kiến trúc Von Neumann: hiệu quả.
- Ngôn ngữ lập trình hàm dùng tính toán lambda (Lambda Calculus): nền tản lý thuyết khá vững chắc, sức mạnh tương đương mô hình Von Neumann. Nền tản toán rất thuận lợi với người dùng do cách ghi phép tính lambda khá tương tự cách viết tính toán các biểu thức thông thường.
- Tuy nhiên kiến trúc Von Neumann không phù hợp nên độ hiểu quả khi chạy sẽ không bằng các ngôn ngữ hướng thủ tục.
- Nhưng nhờ sự phát triển của lĩnh vực vi sử lý, tốc độ xử lý của máy tính hiện nay rất là lớn nên sự không hiệu quả của lập trình hàm đã được giảm đi rất nhiều nên lập trình hàm đã phát triển trở lại. Các ngôn ngữ mới đã kết hợp lập trình hàm vào.

Hàm toán học:

- Là một ánh xạ các phần tử của một tập hợp đến các phần tử của một tập hợp khác.
- Với biểu thức lambda ta sẽ viết λ(x) x * x * x, nó tương đương cube(x) = x * x * x.
- Biểu thức lambda không có tên hàm.
- Để truyền thông số ta làm như sau (λ(x) x * x * x)(2).

Higher – order Functions: Hàm bậc cao: là sức mạnh của lập trình hàm vì nhờ các hàm bậc cao, ta có thể kết hợp các tác vụ lại với nhau theo nhiều cách thức rất đa dạng, làm cho chương trình viết rất ngắn gọn nhưng có đủ sức mạnh như viết lập trình thủ tục.

- Là những hàm thoả ít nhất một trong 2 đặc điểm: nhận các hàm khác vào như là các thông số; có thể trả về các hàm như kết quả của nó
- Các ví dụ: hàm hợp (Function composition), Apply – to – all, Forall/Exists, Insert – left/Insert – right, Functions as parameters, Closures.

Function Composition: nhận 2 hàm vào như là thông số và trả về 1 hàm. Ví dụ f o g giá trị truyền vào là g và x, tính g theo x và truyền giá trị vào f.

Scala:

```
val f = (x:Double) => x + 2

val g = (x:Double) => x * x

val h = f compose g                    Nghĩa là f o g (tính g truyền vào f)

h(3)
```

val k = f andThen g *Nghĩ là g o f (tính f truyền cho g)*

k(3)

Apply – to – all: nhận 1 hàm là thông số truyền vào và 1 danh sách các giá trị. Hàm trả về một danh sách mà các phần tử là gán hàm f vào từng phần tử trong danh sách.

$$\alpha f:\langle x_1, x_2, \ldots, x_n\rangle = \langle f : x_1, f : x_2, \ldots, f : x_n\rangle$$

Ví dụ: h(x) = x * x => αh:(1,2,3) nhận được (1,4,9)

Scala:

List(2,3,4).map((x:Int) => x * x)

def inc (x:Int) = x + 1

List(4,5,6).map(inc)

Forall/Exists: Vị từ for all (Với mọi) và exist (tồn tại). Nhận vào 1 hàm ***vị từ*** như là một thông số (hàm có kết quả là true hoặc false) và 1 list các giá trị.

- Forall: True nếu tất cả phần tử trong list là true với hàm vị từ.
- Exists: True nếu tồn tại 1 phần tử trong list là true với hàm vị từ.

$$\forall f :< x_1, x_2, \ldots, x_n >= \bigcap f : x_i$$
$$\exists f :< x_1, x_2, \ldots, x_n >= \bigcup f : x_i$$

Scala:

def isEqualToThree(x:Int) = x == 3

List(2,3,4).forall(isEqualToThree)

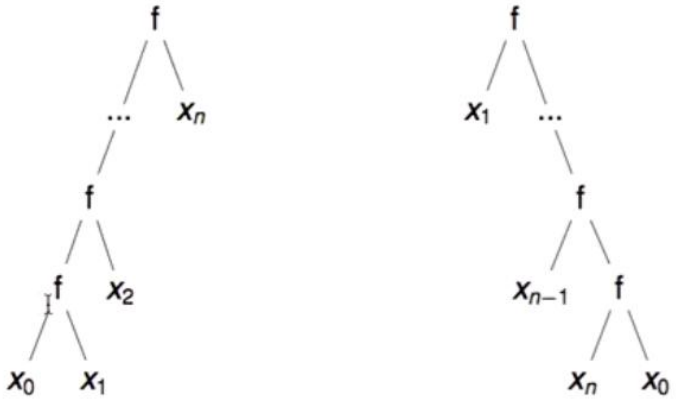
List(2,3,4).exists(isEqualToThree)

Insert Left/Insert Right: dùng để tích luỹ quá trình tính toán, từ một kết quả, lại đùng để tính toán bước kế tiếp.

Insert Left: $/f : \langle x_0\rangle, \langle x_1, x_2, \ldots, x_n\rangle$

Insert Right: $¥f : \langle x_0\rangle, \langle x_1, x_2, \ldots, x_n\rangle$

$/f :< x_0 >, < x_1, x_2, ..., x_n >$ $\backslash f :< x_0 >, < x_1, x_2, ..., x_n >$



Hàm sẽ dùng hàm f tính với 2 đối số là x_0 và số đầu tiên (left) hoặc số cuối cùng (right), và dùng kết quả để tính tiếp với đối số tiếp theo theo thứ tự.

Scala:

List(2,3,4).foldLeft(0)((a,b) => a+b) ***9***

List(2,3,4).foldLeft(1)((a,b) => a*b) ***24***

List(2,3,4).foldLeft("A")((a,b) => a+b) ***"A234"***

List(2,3,4).foldRight("A")((a,b) => a+b) ***"234A"***

Cho biết kết quả của biểu thức sau viết bằng Scala:

List(3,5,9,10).foldLeft(0)((x,y)=>x+y)

27

Cho filter là hàm bậc cao nhận vào một vị từ (predicate - hàm có kết quả kiểu luận lý) và trả về một danh sách con gồm các phần tử thoả vị từ đó và cho phép toán % là phép toán tính modulo. Hãy viết biểu thức để trả về một danh sách con chỉ gồm các giá trị chẵn của danh sách vào? List(2,3,4,5,6)._____ => List(2,4,6)

filter(x => x % 2 == 0)

Cho một danh sách List(4,2,6,9), hãy cho biết dùng hàm bậc cao nào để tạo được một danh sách là các giá trị bình phương của các phần tử trong danh sách được cho, tức List(16,4,36,81)

map (khi danh sách kết quả có cùng số phần tử với danh sách vào và mỗi phần tử kết quả là từ mỗi phần tử của danh sách vào thì map là hàm bậc cao thích hợp nhất)

Cho reverse là hàm đảo ngược thứ tự các phần tử của một danh sách (ví dụ List(1,4,2).reverse => List(2,4,1)) và :: dùng để nối 1 phần tử vào đầu một danh sách (a :: List(b,c,d) => List(a,b,c,d)). Với lst đang chứa một danh sách các giá trị nguyên, cho biết biểu thức nào dưới đây có kết quả tương tự reverse?

lst.foldLeft(List[Int]())((x,y) => y::x)

Functions as Parameters:

Scala cho phép định nghĩa các hàm bậc cao.

Truyền thông số là một hàm vào một hàm: kiểu hàm cũng là một kiểu dữ liệu như các kiểu khác.

```
def apply (x:Int) (f:Int => Int) = f(x)

val incl = (x: Int) => x + 1

val sq = (x: Int) => x * x

val fl = List(inc1,sq)

fl.map(apply(3))
```

Đối với lập trình hàm, ta có rất nhiều các khác nhau để kết hợp các tác vụ cơ bản lại, để tạo các tác vụ mạnh hơn. Đó là sức mạnh của lập trình hàm.

Closure: là một hàm kết hợp các dữ liệu.

```
def power (exp: Double) = (x: Double) => math.pow(x, exp)
```

một hàm trả về 1 hàm

```
val square = power(2)

square(4)
```

Hãy viết một hàm (forAllExist) nhận vào 3 thông số gồm 1 danh sách các số nguyên và 2 predicate, chỉ trả về kết quả true nếu danh sách có tất cả phần tử thoả predicate thứ nhứt và có ít nhất 1 phần tử thoả predicate thứ hai.

```
def forAllExist(lst:List[Int],f1:Int=>Boolean,f2:Int=>Boolean) = lst.forall(f1) && lst.exists(f2)
```

Hãy viết một hàm (doubleCheck) nhận vào 3 thông số gồm 1 danh sách các số nguyên và 2 predicate, chỉ trả về kết quả true nếu danh sách có ít nhất một phần tử thoả predicate thứ nhứt và có ít nhất một phần tử thoả predicate thứ hai.

```
def doubleCheck(lst:List[Int],f1:Int=>Boolean,f2:Int=>Boolean) = lst.exists(f1) && lst.exists(f2))
```

Cho một hàm được định nghĩa như sau:

```
def increaseClosures(n:Int)(x:Float) = x + n
```

Một lênh khai báo biến inc3 được viết như sau:

```
val inc3 = increaseClosures(3) _
```

sẽ làm cho inc3 được cất giữ giá trị gì?

Một hàm có kiểu là Float => Float (hàm này là (x:Float) => x + 3)

Currying functions: hàm chỉ có 1 thông số, trả về 1 cái hàm có 1 thông số.

f: X_1 x X_2 x … x X_n → Y

curry: f : X_1 → X_2 → … → X_n → Y

Scala:

def add(x: Int, y: Int) = x + y

add(1), add(1,3) sẽ lỗi

def plus (x: Int) (y: Int) = x + y *hàm curry*

plus(1)(3)

val inc1 = plus(1) _

inc1(3)

val addCurried = (add _).curried biến 1 hàm thành curry

val plusUncurried = Function.uncurried(plus _) biến 1 hàm curry thành không curry

Cho định nghĩa hàm như sau:

def foo(x:Int)(y:Float) = x * y

Khi định nghĩa x như sau: val x = foo(3) _ thì x sẽ có kiểu là gì

Kiểu hàm: Float => Float (vì hàm foo có kiểu Int => Float => Float nên khi gọi hàm foo với chỉ 1 thông số foo thì sẽ nhận kiểu trả về là Float => Float)

Cho hàm foo được định nghĩa như sau:

def foo(x:Boolean,y:Float)(z:Float)= if (x) y else z

và m được định nghĩa như sau: val m = foo(true) _ _

Cho biết kiểu của m?

Không có kiểu gì cả vì lệnh gọi foo chưa đúng

Immutability: không thể thay đổi.

Lập trình hàm không có các trạng thái → không có sự thay đổi trên các biến. Các biến từu đầu đến cuối chỉ có 1 giá trị duy nhất.

Java có chuỗi là immutable. Khi gọi "Hello".toUpper() không thay đổi chuỗi Hello ban đầu mà chỉ trả về 1 chuỗi mới là HELLO.

Scala, val là immutable.

val num = 12

num = 10 *gây lỗi*

Pure functional programming: không có sự thay đổi.

Không thay đổi → luôn trả về giá trị là một giá trị mới.

Hàm không làm thay đổi trạng thái thì có thể song song hoá được mà không ảnh hưởng lẫn nhau.

Expression: Biểu thức.

- Mọi thứ trong lập trình hàm đều là biểu thức, không phải phát biểu. Biểu thức trong lập trình hàm sẽ tín toán trên các thông số đưa vào và trả ra 1 kết quả chứ không làm thay đổi gì hết. Trong khi đó, trong lập trình thủ tục, 1 phát biểu sẽ không trả về gì hết nhưng làm thay đổi giá trị của biến (phát biểu gán sẽ thay đổi phát biểu bên vế trái, phát biểu if, for sẽ thay đổi bên trong thân của nó).

- Nếu có nhiều biểu thức, giá trị cuối cùng sẽ được trả về, do đó không cần return.

Scala:

```
def fact(x: Int) : Int =
        if (x == 0) 1 else x * fact(x – 1)

val s = for (x <- 1 to 25 if x * x > 50) yield 2 * x
```

*for sẽ tạo tập hợp từ 1 đến 25, cho x chạy trên tập đó mà thoả x * x > 50 thì chạy lệnh 2 * x.*

vòng for không làm thay đổi giá trị x mà x chỉ là đại diện cho các phần tử của tập hợp 1 đến 25.

Cho listA là một danh sách có 3 phần tử {a, b, c} và listB là một danh sách có 3 phần tử là {d,e,f}, nếu tác vụ listA.append(listB) sẽ làm cho listA trở nên có 6 phần tử gồm 3 phần tử ban đầu của listA và 3 phần tử của listB thì tác vụ append là?

Mutable(một tác vụ immutable không làm thay đổi các thông số của nó. Ở đây, append làm listA bị thay đổi (từ 3 phần tử trở thành 6 phần tử) nên append không phải là một tác vụ Immutable)

Cho listA là một danh sách có 3 phần tử {a, b, c} và listB là một danh sách có 3 phần tử là {d,e,f}, nếu tác vụ listA.append(listB) trả về một danh sách mới có 6 phần tử gồm 3 phần tử của listA và 3 phần tử của listB, trong khi listA và listB không đổi, thì tác vụ append là?

Immutable

Giả sử Scala là một ngôn ngữ lập trình hàm tinh khiết (pure functional programming language) và giả sử Scala có phát biểu while như sau:

while (a < 1) { ... }

Thân vòng lặp là một hộp đen, không rõ về nội dung. Bạn có thể suy luận gì về lệnh lặp trên?

Hoặc không thực thi hoặc lặp vô hạn (trên ngôn ngữ lập trình hàm thuần khiết, các biến đại diện cho một giá trị không đổi, do đó, biểu thức a < 1 hoặc là luôn luôn sai hoặc là luôn luôn đúng nên lệnh while sẽ hoặc là không được thực thi (khi a < 1 sai) hoặc là lặp vô hạn (khi a < 1 đúng). Vì vậy trên các ngôn ngữ lập trình hàm thuần khiết, không có lệnh lặp dựa vào biểu thức điều kiện như while, do while.)

Pattern Matching: So trùng mẫu: Giống như switch nhưng mạnh hơn.

<exp> match {

case <pattern 1> => <exp1>

case <pattern 2> => <exp2>

…

}

Đều tiên sẽ tính toán exp, sau đó so sánh nếu bằng pattern i thì trả về exp i. Nếu không match case nào thì sẽ thoát.

Scala:

```
def matchTest (x: Int): String = x match {
        case 1 => "one"
        case 2 => "two"
        case _ => "many"                _ là bất kỳ giá trị nào
}
```

Recursion: Đệ quy. Sử dụng rất phổ biến. Được dùng để giải quyết các vấn đề liên quan vòng lặp trong lập trình thông thường.

```
def fact(n : Int) : Int = if (x == 0) 1 else n * fact(n - 1)
```

Cần dùng def bởi vì tên được sử dụng bên phải.

Scala:

```
def mem (x: Int, lst: List[Int]) : Boolean = lst match {
        case List() => false
        case head :: tail => if (x == head) true
                             else mem(x, tail)
}
```

Trên Scala, phép toán :: dùng để nối 1 phần tử vào 1 danh sách (3::List(2,1,5) sẽ tạo thành danh sách List(3,2,1,5)). Giả sử biến sl đang chứa giá trị List(3,5,6,2), hãy cho biết phép match sau có thành công không và nếu có thì giá trị của h và t là bao nhiêu?

```
sl match {

    case h :: tl => ....
```

Thành công, h là 3 và t là List(5,6,2)

Trên Scala, kiểu tuple là kiểu kết hợp nhiều giá trị với nhau, ví dụ val m = (3,4.3, True) là một giá trị kiểu tuple kết hợp 3 kiểu Int, Double và Boolean. Cho biết với m định nghĩa trong ví dụ trên, trong phép match sau thì thành công ở case nào?

```
m match {

  case (3,1.0,True) => ...

  case (3,4.3,False) => ...

  case (_,_,_) =>

}
```

Thành công ở case cuối(vì _ có thể trùng với bất kỳ giá trị nào)

Lazy Evalution: Tính toán lười:

- Thông thường các biểu thức sẽ được tính toán eagerly (trước), khi chúng được xuất hiện → eagerly evaluated.
- Lazy evalutation nghĩa là biểu thức chỉ được tính khi chỉ khi mà chúng ta cần dùng.

lazy val x = 1 + y

Biểu thức 1 + y chỉ được tính khi x lần đầu tiên được sử dụng.

Scala:

def foo(b: Boolean, x:=>Int, y:=>Int) = if (b) x else y

foo(a==0,1,b/a) *khi a == 0 nó ko trả về 1 được mà gây ra lỗi b/0 → dùng tính toán lười*

Trong scala cho phép truyền thông số bằng tên (pass – by – name parameter) để tính toán lười.

<tên thông số> :=> <kiểu dữ liệu>

Các thông số này sẽ chỉ được tính toán khi được sử dụng bên trong thân hàm.

Type Inference: Suy biến kiểu:

- Scala là ngôn ngữ kiểm tra kiểu rất chặt. Mọi dữ liệu trên scala đều phải có kiểu, kiểu sẽ được kiểm tra rất chặt chẽ.
- Khi ta tạo 1 dữ liệu, không cần gán kiểu. Nhưng Scala sẽ tự suy diễn kiểu dữ liệu và từ đây về sau, biến đó chỉ có kiểu dữ liệu đó.

- Có thể thay đổi (override) kiểu bằng supertype.

val greeting: Any = “Hello”

greeting = 42 *lệnh được phép do greeting kế thừa kiểu Any là lớp gốc*

- Các parameter của hàm phải định nghĩa kiểu.

def mistery (x) = 42 * x *Error*

def mistery (x: Int) = 42 * x *Ok*

- Kiểu trả về của hàm sẽ được tự suy diễn. Nếu x là Int → 42 * x là Int.
- Biểu thức và hàm đệ quy cần khai báo kiểu trả về.
- Nếu trong định nghĩa hàm, hàm truyền vào đã được khai báo kiểu trả về, ta có thể không cần định nghĩa lại kiểu trả về nữa khi truyền tham số vào.

def twice (f: (Int)=>Int, x:Int) = f(f(x))

twice (x=>42*x, 3) không cần định nghĩa kiểu của x=>42*x do suy diễn từ ngữ cảnh

- Điều này hữu dụng khi dùng thư viện.

List(1,2,3).filter(x=>x%2==0)

List[A].filter(p:(A)=>Boolean) : List[A]

A là Int từ list(1,2,3) là List[Int]

➔ p: phải là (Int) => Boolean
➔ x là Int

- Ta có thể rút gọn thông số với các thông số chỉ suất hiện 1 lần và theo đúng thứ tự truyền vào.
  List(1,2,3).filter(x => x%2==0)
  List(1,2,3).sortWith((x,y)=>x>y)
  Có thể viết lại sử dụng _ thành:
  List(1,2,3).filter(_%2==0)
  List(1,2,3).sortWith(_>_)

Cho biết kiểu trả về của hàm sau:

def foo(a:Boolean) = if (a) 1 else List()

Any (trên cây phân lớp của Scala, Any là lớp gốc nên nó là lớp tổ tiên của cả lớp Int và List)

Chọn khai báo hàm đúng và ngắn nhất trong các khai báo sau:
def foo(x:Int):Int = if (x == 0) 1 else x*foo(x-1)(hàm đệ qui nên cần phải khai báo kiểu trả về)

List(4,5,6).forall(_ > 3)

Abstract Syntract Tree (AST): Cây cú pháp trừu tượng:

- Là một biểu diễn cây cho mã nguồn.
- Khác với cây parse tree là bỏ đi rất nhiều các chi tiết, nhờ vậy cây cú pháp trừu tượng rất ít node và cây nhỏ hơn rất nhiều.
- Nhờ cây nhỏ hơn nên ở các bước tiếp theo trong quá trình dịch sẽ thao tác nhanh hơn.
- Rút gọn các node đã có giá trị cố định, chỉ giữ các node có giá trị thay đổi được.

Generate AST for ANTLR:

ANTLR Parse Tree:

- Khi phân tích một chương trình nguồn, sẽ tạo ra một cây parse tree.
- Khi đọc grammar sẽ sinh ra một class với tên là tên của grammar (văn phạm).
- Một ký hiệu không kết thúc sẽ sinh ra inner class (class lồng bên trong). Tên giống nhưng ký tự đầu là chữ hoa và thên Context vào cuối. Ví dụ: ký hiệu không kết thúc assign có class là AssignContext. Các đối tượng của class là node trên cây parse tree.
- Các ký hiệu kết thúc là terminal node, giá trị của các node này là các token.
- Các ký hiệu không kết thúc là các đối tượng của các inner class tương ứng.

Traversal on Parse Tree: quá trình duyệt cây:

- Mỗi ký hiệu bên vế phải sẽ có 1 hoặc 2 phương thức nằm trong inner class.

```
assign: ID ASSIGN exp SEMI;

class AssignContext {
   TerminalNode ID () {…}
   TerminalNode ASSIGN () {…}
   ExpContext exp () {…}
   TerminalNode SEMI () {…}
}
```

Some useful methods: Cây phân tích cú pháp có số node con không biết trước. ANTLR cung cấp phương thức xác định node con thuộc về vế phải nào.

- *RuleContext getChild (int i):* lấy ra node con thứ i (đếm từ 0). RuleContext là lớp cha của mọi lớp.

```
stmt: assign | ifstmt;

Nếu không dùng getChild:

if (assign() != null)
   // do something on assign()
else
   // do same thing on ifstmt()
```

Nếu mà cả 2 cùng làm 1 việc giống nhau, ta sẽ dùng getChild (0) cho thuận tiện.

```
    // do something on getChild (0)
```

- *int getChildCount ():* cho biết số lượng node con ở thời điểm hiện tại.

```
term: ID | INTLIT | LP exp RP;
if (getChildCount() == 3)
    // do something on exp()
else
    // do something on getChild(0)
```

Multiple Appearances of symbol on RHS: vế phải có nhiều sự xuất hiện của một ký hiệu. Một ký hiệu xuất hiện nhiều lần trong vế phải → có 2 phương thức để phân biệt.

```
ifstmt: IF exp THEN stmt ELSE stmt;
class IfstmtContext {
    List<StmtContext> stmt() {…}      Trả về list chứa các StmtContext
    StmtContext stmt(int i) {…}       Trả về StmtContext tuỳ theo thông số truyền vào
                                      (trường hợp này là 0 và 1, nếu sai trả về None)
    ExpContext exp() {…}
    …
}
prog: stmt+;
class ProgContext {
    List<StmtContext> stmt() {…}
    StmtContext stmt(int i) {…}
}
```

Cho luật sinh của một văn phạm như sau:

assign → ID ASSIGN exp SEMI

Giả sử biến **ctx** đang cất giữ nút ứng với assign trên cây phân tích cú pháp sinh ra bởi ANTLR. Hãy viết biểu thức để truy xuất nút ứng với **exp** bên vế phải?

ctx.exp

Cho luật sinh của một văn phạm như sau:

ifstmt → IF exp THEN stmt ELSE stmt

Giả sử biến **ctx** đang cất giữ nút ứng với **ifstmt** trên cây phân tích cú pháp (parse tree). Hãy viết biểu thức để truy xuất nút ứng với **stmt** của vế else?

ctx.stmt(1)

Cho luật sinh của văn phạm như sau:

prog → stmt+

Giả sử ctx là biến đang giữ nút ứng với prog. Hãy viết biểu thức để nhận được một danh sách tất cả các nút con **stmt** của nút ứng với **prog**?

ctx.stmt

Chuyển đổi Parse Tree sang AST:

- Cây phân tích cú pháp có rất là nhiều những loại node khác nhau. Mỗi node ứng với một ký hiệu không kết thúc. Có rất nhiều loại ký hiệu không kết thúc nên có rất nhiều loại node khác nhau.
- Mỗi node có số node con khác nhau.
- Dùng visitor pattern để duyệt cây.

Visitor Generated by ANTLR:

- Để sinh từ ANTLR cần dùng option -visitor.
- Phương thức accept được thêm vào mỗi inner class.
- Tạo ra thêm 2 class <grammar name>+Visitor.java và <grammar name>+BaseVisitor.java.
- Mỗi ký hiệu không kết thúc abc sẽ sinh ra phương thức visitAbc(ctx: AbcContext)

VD có grammar MC, ta có 2 class MCVisitor và MCBaseVisitor.

- MCVisitor là interface chứa các phương thức visit của các Node. Ta cần implement lại chúng.
- MCBaseVisitor là implement của MCVisitor nhưng các method chỉ gồm 1 dòng return visitChildren (ctx). Nó sẽ không làm gì hết và bắt đầu thực hiện với con của nó.
- Trong trường hợp trong các loại Node, không cần phải duyệt toàn bộ, ta sẽ viết class là lớp con của MCBaseVisitor.
- Trong trường hợp ta biết chắc phải duyệt hết tất cả các loại Node, ta sẽ viết class là implement của MCVisitor.

AST Generation:

trait AST

case class Prog(sl:Stmt) extends AST

trait Stmt extends AST

case class Assign(id:String,e:Exp) extends Stmt

case class IfStmt(e:Exp,s1:Stmt,s2:Stmt) extends Stmt

trait Exp extends AST

case class BinOp(op:String,e1:Exp,e2:Exp) extends Exp

```
case class Id(id:String) extends Exp
case class Intlit(lit:Int) extends Exp
```

VD term: ID | INTLIT | LP exp RP

Scala:

```
class ASTGen extends MCVisitor [AST] {
   override def visitTerm (ctx: TermContext) =
      if (ctx.getChildCount() == 3)
         ctx.exp().accept(this)
      else if (ctx.ID != null)    Id(ctx.ID.getText)
      else Intlit(ctx.INT.getText.toint)
}
```

ANTLR sẽ sinh ra một phương thức visit cho mỗi ký hiệu không kết thúc

Cho văn phạm của ngôn ngữ ABC có luật sinh sau:

exp → exp ADD term | term

Prototype của phương thức visit sinh ra bởi ANTLR cho luật sinh này là gì?

T visitExp(ABCParser.ExpContext ctx)

Nếu có import ABCParser thì prototype có thể viết ngắn hơn: T visitExp(ExpContext). Chú ý prototype này là prototype của Java

Name (Tên):

- Một chuỗi các ký tự dùng để biểu diễn một vấn đề khác.
- Dùng các ký hiệu thay cho địa chỉ để tham khảo đến entity.
- Trừu tượng hơn, gần gũi người người lập trình, dễ nhớ hơn.

Binding (Sự ràng buộc):

- Sự kết hợp của 2 thực thể với nhau.
- Binding time (Thời gian xảy ra ràng buộc): thời gian từ khi ràng buộc xảy ra đến kết thúc.
- Vài vấn đề:
  - Early binding (ràng buộc xảy ra sớm, từ khi thiết kế ngôn ngữ) vs Late binding (ràng buộc xảy ra trễ, xảy ra khi máy đưa chương trình vào thực thi).
  - Static binding (ràng buộc tĩnh, xảy ra trước khi đưa vào thực thi) vs Dynamic binding (ràng buộc động, xảy ra vào thời gian chạy).
  - Polymorphism (đa hình): Tên ràng buộc với nhiều thực thể.
  - Alias (Bí danh): Nhiều tên ràng buộc cho một thực thể.

Binding Time:

- Language design time (Thời gian thiết kế ngôn ngữ): ràng buộc xảy ra do người thiết kế ngôn ngữ đưa ra.
- Language implementation time (Thời gian hiện thực ngôn ngữ): ràng buộc xảy ra viết biên dịch.
- Programming time (Thời gian lập trình): ràng buộc xảy do người lập trình quyết định.
- Compilation time (Thời gian dịch): ràng buộc được thực hiện khi dịch chương trình. Trong thời gian dịch, chương trình dịch chưa xác định được địa chỉ tuyệt đối của biến toàn cục.
- Linking time: ràng buộc xảy ra khi liên kết các file object thành một file để chạy. Thời gian kết nối là thời gian Link editor nối các file object lại thành một file thực thi. Trong thời gian này, chỉ mới xác định được địa chỉ tương đối của biến toàn cục trong vùng bộ nhớ sẽ được cấp phát tĩnh chứ chưa xác định được địa chỉ tuyệt đối
- Load time: thời gian nạp từ đĩa vào bộ nhớ trong để thực thi. Thời gian nạp là lúc loader nạp chương trình thực thi từ bộ nhớ ngoài vào bộ nhớ trong. Tuỳ theo thực tế của bộ nhớ trong lúc chương trình được nạp mà chương trình sẽ được cấp phát vào một vùng nhớ phù hợp. Sau khi cấp phát thì các biến toàn cục mới có địa chỉ tuyệt đối và khi đó, ràng buộc mới xảy ra,
- Run time: khi chương trình chạy mới xảy ra ràng buộc. Một chương trình khi thực thi sẽ có 3 vùng nhớ: vùng nhớ tĩnh, stack và heap. Các biến toàn cục được cấp phát trong vùng nhớ tĩnh nên địa chỉ tuyệt đối của chúng sẽ được xác định trước khi thực thi.

Object Lifetime: là thực thể trong chương trình. Là bất cứ thứ gì (đoạn code, …).

- Mỗi thực thể có thời gian sống: thời gian từ khi được tạo ra đến khi bị huỷ bỏ.
- Binding Lifetime (Thời gian sống của một ràng buộc): thời gian từ khi ràng buộc được tạo ra đến khi bị huỷ bỏ.
- Dangling reference (Tham chiếu treo): trường hợp ràng buộc với đối tượng còn nhưng đối tượng lại bị huỷ (2 biến cùng trỏ 1 đối tượng, xoá đối tượng qua 1 biến thì biến còn lại vẫn còn ràng buộc)
- Leak memory – Garbage: đối tượng vẫn còn nhưng ràng buộc bị huỷ. (1 biến trỏ đến 1 đối tượng, gán biến bằng null → đối tượng chưa bị xoá nhưng không thể truy cập gây rác trong bộ nhớ)
- Một trong các cách giảm thiểu lỗi: không cho phép lệnh delete → giải quyết được Dangling reference → xuất hiện các Garbage → Dùng Garbage Collection giải quyết.
- Garbage Collection: khi bộ nhớ có dấu hiệu sắp đầy, tiến hình kiểm tra bộ nhớ, nếu có đối tượng rác, sẽ tiến hành xoá.

Object Allocation: cấp phát đối tượng.

- Static: Đối tượng được cấp phát trong vùng nhớ tĩnh (vùng nhớ được tính toán ngay từ lúc dịch), thời gian sống là từ khi chương trình bắt đầu đến khi chương trình kết thúc. Cấp phát cho các biến toàn cục, các hằng, chuỗi. Dành cho các phần không thay đổi được.

- Stack Dynamic: Cái nào tạo trước, huỷ bỏ sau. Quản lý dễ. Chỉ phù hợp các đối tượng cấp phát, huỷ bỏ đúng quy luật tạo trước huỷ sau → dùng cho các biến cục bộ.
- Heap Dynamic: Dùng cho các đối tượng được tạo ra và huỷ bỏ không theo quy luật.
  - Explicit Heap Dynamic: Có lệnh cụ thể để tạo ra đối tượng.
  - Implicit Heap Dynamic: Không có lệnh cụ thể để tạo ra nhưng theo cách viết, chương trình sẽ ngầm định tạo ra đối tượng.

Block: là một vùng văn bản chứa đựng các khai báo cho vùng văn bản đó.

Scope (tầm vực): tầm vực của một ràng buộc là vùng văn bản của chương trình mà ràng buộc có hiệu lực. Một ngôn ngữ sẽ thuộc tầm vực tĩnh hoặc tầm vực động.

- Static scope (tầm vực tĩnh): những khai báo quyết định vào thời gian dịch. Khi viết hay dịch chương trình, chúng ta đã quyết định được các khai báo có hiệu lực ở vùng nào.
- Dynamic scope (tầm vực động): hiệu lực những khai báo được quyết định vào thời gian thực thi.
- Ngôn ngữ có tầm vực tĩnh, ngay từ khi dịch chương trình, ta đã xác định được những khai báo có hiệu lực trong vùng nào của chương trình.
- Nguôn ngữ tầm vực động, khi chạy chương trình mới biết được một khai báo được dùng ở đâu.

Static Scope Rules for Blocks:

- Một tham khảo đến một ID luôn ràng buộc với khai báo local nhất. (các khai báo trong block là khai báo local, nếu có nhiều block lồng vào nhau đều có khai báo 1 ID, khai báo local nhất là ID trong block hiện đang xét)
- Một khai báo không thể được thấy bên ngoài block.
- Khai báo ở bên ngoài thì có thể được thấy từ block bên trong trừ khi được khai báo lại bên trong.
- Một block có thể được đặt tên, nếu block có tên thì tên sẽ thuộc khai báo local của block bên ngoài.

Referencing Environment (môi trường tham khảo): là hàm ngược của tầm vực.

- Môi trường tham khảo của một phát biểu (statement) là tập hợp những khai báo mà phát biểu đó có thể sử dụng đc. Tầm vực của một khai báo là tập hợp những phát biểu mà dùng được khai báo đó.
- Ngôn ngữ tầm vực tĩnh, môi trường tham khảo của một phát biểu là những biến local và những tên có thể nhìn thấy được của các tầm vực bao lấy tầm vực đó.
- Ngôn ngữ tầm vực động, môi trường tham khảo là những ràng buộc local và những ràng buộc có thể thấy được trong các chương trình con đang hoạt động.

Data Types:

- Tập hợp các giá trị đồng nhất với nhau về hình thức.
- Tập hợp các tác vụ có thể thao tác được trên dữ liệu đó.

Sử dụng hệ thống kiểu:

- Tổ chức chương trình dễ hiểu hơn.
- Kiểm tra lỗi.
- Trong quá trình hiện thực.

Hệ thống kiểu: Bao gồm:

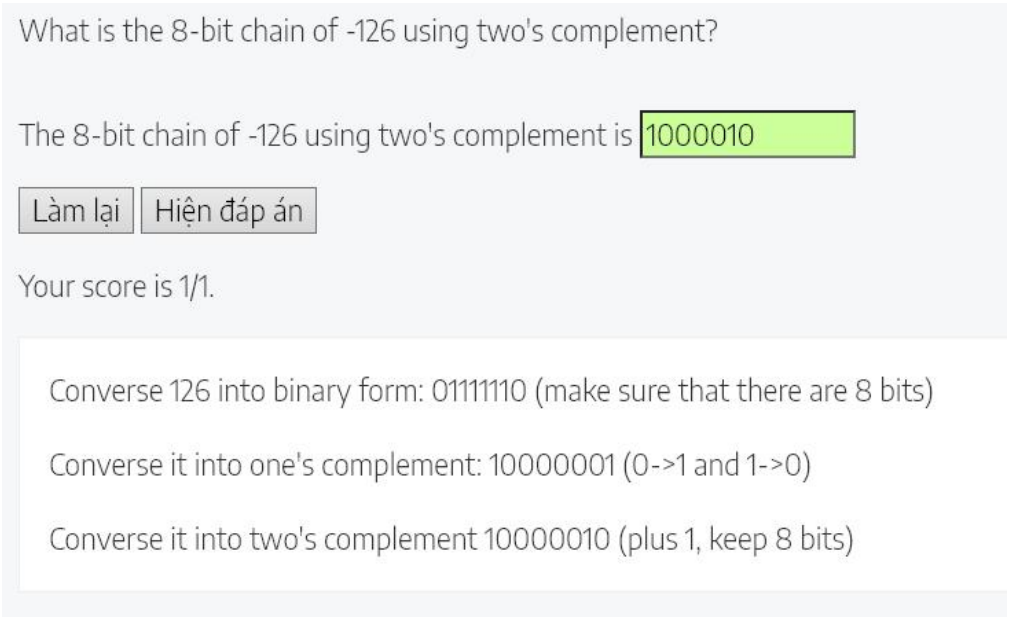
- Tập hợp các kiểu đã được định nghĩa trước.
- Cơ chế định nghĩa kiểu dữ liệu mới.
- Cơ chế điều khiển các kiểu: tương đương kiểu, tương thích kiểu, suy biến kiểu.
- Đặc tả các ràng buộc về kiểu được kiểm tra lúc tĩnh (kiểm tra lúc dịch chương trình) hay lúc động (kiểm tra khi thực thi).

Scalar Types:

- Kiểu dữ liệu nguyên tố, không thể tách thành các thành phần để thao tác.
- Dùng tổng hợp để tạo ra các kiểu dữ liệu khác lớn hơn.
- Đôi khi hỗ trợ trực tiếp bởi phần cứng.
- boolean, characters, integers, floating-point, fixed-point, complex, void, enmerations, intervals…

Integer:

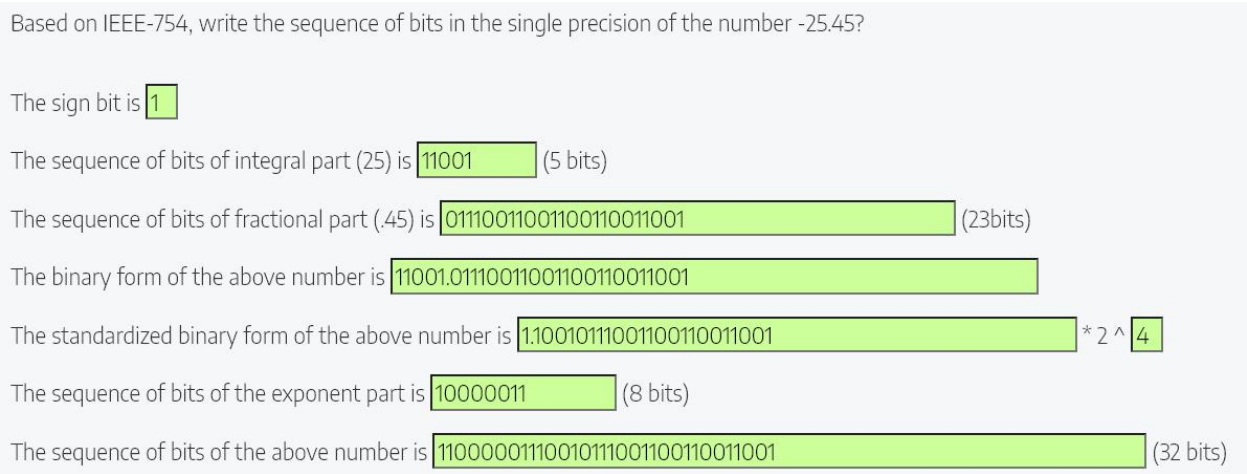
- Ngôn ngữ hỗ trợ nhiều kích thước của số nguyên (Java có byte, short, int, long).
- Một số nguyên ngữ bao gồm unsigned integers.
- Hỗ trợ trực tiếp bởi phần cứng: chuỗi bits.
- Lưu trữ giá trị âm ở dạng: số bù 2.



Floating-Point:

- Mô hình hoá giá trị thực nhưng chỉ xấp xỉ.
- Hỗ trợ các chương trình tính toán khoa học.
- Khác nhau ở tính chính xác và range (giá trị nhỏ nhất đến lớn nhất có thể lưu trữ được).
- Lưu trữ theo chuẩn IEEE 754.

- Single precision (Chính xác đơn): 1 Sign bit; 8 Exponent (số mũ); 23 Fraction (chuỗi bit lưu trữ giá trị).
- Double precision (Chính xác kép): 1 Sign bit; 11 Exponent (số mũ); 52 Fraction (chuỗi bit lưu trữ giá trị).
- Cách tính: Cho số thực x.
  - Đổi sang dạng nhị phân.
  - Dời dấu chấm động về dạng: 1.xxxxx * 2^n
  - Phần mũ: lấy n + số thiên vị (127 với single, 1023 với double) và đổi sang nhị phân.
  - Ta có Dấu | Phần mũ | Fraction bỏ đi số 1 trước dấu chấm và giữ đủ theo quy định.



Decimal:

- Dùng trong business (tiền): số lẻ không nhiều nhưng cần chính xác, không kiểm sấp xỉ được.
- Lưu trữ các con số cố định.
- Ưu điểm: chính xác.
- Nược điểm: giới hạn range, lãng phí bộ nhớ.

Boolean:

- Đơn giản nhất.
- Chỉ có 2 giá trị true và false.
- Có thể hiện thực dạng bit, nhưng có thể dạng byte.

Character:

- Lưu trữ dạng mã hoá.
- Hầu hết hỡ trợ ASCII (mã hoá 8 bits).
- Sau này, các ngôn ngữ hỗ trợ cho Unicode (mã hoá 16 bit). Mã cho phép đưa ngôn ngữ tự nhiên vào.

User – Defined Ordinal Types: kiểu dữ liệu có thứ tự do người dùng tự định nghĩa.

- Miền các giá trị có thể có, có thể kết hợp với tập hợp các giá trị nguyên dương.

Enumeration Type: kiểu liệt kê.

- Bao gồm các giá trị có thể có, là những hằng có tên.
- Vấn đề thiết kế:
    - Danh hiệu được xuất hiện trong nhiều kiểu được định nghĩa hay không.
    - Giá trị liệt kê có được ép kiểu về giá trị nguyên không.
    - Các kiểu giá trị khác có thể ép về kiểu enumeration hay không.
- Ưu điểm:
    - Dễ đọc (readability): không cần mã hoá màu như 1 con số.
    - Tin cậy (reliability): Không thực hiện các tác vụ sai với các tác vụ liệt kê được. Đảm bảo biến không nằm ngoài giá trị cho phép. Hỗ trợ tốt hơn C++, không cho ép kiểu int về kiểu liệt kê.
    - Hiện thực như là int.

Subrange Type: kiểu miền con. Kiểu dữ liệu cho phép định nghĩa 1 cái miền: type pos = 0 .. MAXINT;

Composite Types: kiểu dữ liệu tổng hợp.

- Có nhiều thành phần, có thể truy cập đến các thành phần con.
- Các kiểu của các thành phần có thể giống nhau hoặc khác nhau.
- Số lượng thành phần cố định hoặc thay đổi được.
- Có các tác vụ trên toàn bộ đối tượng hoặc các thành phần.
- Có thể tác vụ thêm/xoá thành phần trong kiểu có số lượng thay đổi được.
- Có tác vụ tạo/huỷ để tiết kiệm bộ nhớ.

Array Types:

- Các kiểu dữ liệu thành phần giống nhau.
- Truy suất thành phần bằng index. Index hợp lệ trong Pascal, Ada là bất kỳ loại nào, trong ngôn ngữ khác là miền con của int. Index có nằm trong miền cho phép hay không có thể được kiểm tra (Java, ML, C#) hoặc không.
- Tuỳ theo các ghi sẽ cấp phát khác nhau:
    - static int x[10] hoặc int x[10] bên ngoài function (***static***)
    - int x[10] bên trong function (***fixed stack - dynamic***)
    - int x[n] (***stack - dynamic***)
    - int[] x = new int[10] (***fix heap - dynamic***)
    - int[] x = new int[n] (***heap - dynamic***)
- Một vài ngôn ngữ cho phép khởi tạo array tại thời gian cấp phát.
- Jagged array: mảng 2 chiều có kích thước từng hàng khác nhau.
- Rectangular array: mảng 2 chiều có kích thước các hàng bằng nhau.
- Một vài ngôn ngữ cho truy suất một slice (một bộ phận của dãy). (python arr[3:6])

Implementation of Arrays: hiện thực của dãy.

- Dãy 1 chiều: các phần tử xếp tăng dần theo thứ tự trong bộ nhớ.
- Truy cập: addres (k) = address (0) + (k * size_1_element)
- Dãy 2 chiều: có 2 cách xếp:
    - Row – major order: xếp các phần tử theo hàng từ [0][x] xong đến [1][x] thành 1 hàng trên bộ nhớ.

    - Column – major order: xếp các phần tử theo cột từ [x][0] xong đến [x][1] thành 1 hàng trên bộ nhớ.
- Truy cập:
    - Row – major order: address (i, j) = address (0,0) + (i * num_ele_1_row + j) * size_1_ele
    - Column – major order: address (i, j) = address (0,0) + (j * num_ele_1_col + i) * size_1_ele

Associative Array: dãy kết hợp, đánh chỉ số bằng key.

- dt = [(“name”,”John”);(“age”,”28”)]
- Có kiểu Map trong Scala.

String Types: giá trị là chuỗi các ký tự.

- Vấn đề thiết kế: Có cần định nghĩa kiểu string hoặc sử dụng array của char. Chiều dài chuỗi xác định lúc dịch nên cố định (static) hay xác định lúc thực thi và có thể thay đổi được (dynamic).
- Các tác vụ: gán, so sánh, nối 2 chuỗi, trích 1 chuỗi con, tìm kiếm một chuỗi con trong một chuỗi.
- Chiều dài của chuỗi:
    - ***Static (Python, Java):*** chiều dài cố định tại thời gian dịch, chỗ trống sẽ điền khoảng trắng vào, cần compile – time descriptor để mô tả lúc dịch, sau không cần nữa.
    - ***Limited Dynamic (C, C++):*** thay đổi nhưng có giới hạn, cần thêm run – time descriptor để ghi nhận tại 1 thời điểm, chiều dài chuỗi là bao nhiêu nhưng không vượt quá chiều dài tối đa của compile – time descriptor.
    - ***Dynamic (Perl, JavaScript):*** chiều dài không giới hạn, không có compile – time descriptor, có run – time descriptor để xác định chiều dài hiện tại của chuỗi là bao nhiều hoặc lưu trữ dưới dạng linked list.
    - Ada hỗ trợ cả 3 loại.

Descriptor:

- Compile – time: ghi nhận static string, string length, address (offset so với một điểm nào đó).
- Run – time: gồm limited dynamic string, maximum length, current length, addres.

Record Types: tương tự array.

- Cho phép kết hợp nhiều thành phần khác nhau.
- Truy suất dựa vào tên các thành phần.
- Phổ biến trên nhiều ngôn ngữ lập trình, object trong OOP để thay thế record.
- Vấn đề thiết kế: Làm sao truy cập một thành phần của record. Có cho phép rút gọn việc truy suất hay không.
- Cách truy suất: dùng dấu chấm (a.b.c); dùng keyword – based (c OF b OF a).
- Rút gọn hay không:
    - Fully qualified references: ghi đầy đủ.
    - Elliptical references: có thể bỏ đi một số thành phần miễn không bị nhập nhằng với việc truy cập 1 phần tử khác. Ví dụ nếu chỉ có 1 c, ta có thể dùng c, c OF a, C OF b.

Operations in Records: tác vụ trên record:

- Tác vụ gán cùng loại thành phần, copy vùng nhớ từ record này sang record khác.

- Ada cho phép so sánh.
- Tác vụ khởi tạo.
- COBOL cung cấp MOVE CORRESPONDING cho phép gán các thành phần khác nhau, nó sẽ copy các thành phần có tên giống nhau.

Đánh giá record:

- Kiểu rất an toàn.
- So sánh với array: record có các thành phần khác nhau; dãy truy suất cùng 1 cách, record truy suất nhiều cách khác nhau; record truy suất thành phần tĩnh, array truy suất động, cần biết index mới truy suất được.

Hiện thực record:

- Record – Name1 – Type1 – Offset1 - … - NameN – TypeN – OffsetN – Address.
- Data Alignment: b – byte align thì giá trị sẽ đặt tại vị trí là bội số của b.
- Data struture Padding: vùng bộ nhớ đưa vào để lắp đầy phần không dùng. Tại cuối của struct, cần padding thêm cho size của struct bằng với b – byte của phần tử lớn nhất.

Union Types:

- Các biến được phép lưu trữ các phần tử khác nhau có kiểu khác nhau vào các thời điểm khác nhau trong lúc thực thi.
- Cấu trúc lưu trữ của Ada: Các vùng khác nhau sẽ tạo số vùng nhớ theo loại lớn nhất rồi dựa theo các thời điểm để gán vào vùng nhớ.
- Vấn đề thiết kế:
    - Có type checking (bỏ vô type này lấy ra type khác) không. Gồm:
        - Free Union: không hỗ trợ type checking.
        - Discriminated: có thành phần ghi nhận đã đưa vô type gì để biết lấy ra type gì.
    - Có được nhúng vào record không.
- Java và C# không hỡ trợ union để đề cao tính an toàn.

Set Types: kiểu tập hợp.

- Ngôn ngữ lập trình cung cấp tác vụ: membership (xác định một phần tử có thuộc tập hợp không), union (hợp), intersection (giao), different (hiệu), …
- Hiện thực bằng chuỗi bit hoặc bảng băm.

Pointer Types:

- Nhận giá trị là địa chỉ bộ nhớ hoặc giá trị đặc biệt là nil.
- Dùng truy suất một cách gián tiếp hoặc để quản lý các bộ nhớ cấp phát động.
- Tác vụ gán: lấy địa chỉ của một biến con trỏ này bỏ vào một biến con trỏ khác.
- Tác vụ dereferencing: dựa vào giá trị địa chỉ con trỏ để tìm vị trí con trỏ đang trỏ đến để lấy hoặc sửa giá trị.
- Dangling pointer: con trỏ trỏ đến đối tượng đã bị huỷ.
- Lost heap – dynamic variable: đối tượng còn nhưng không truy cập được (garbage).

Pointer trong C và C++:

- Dùng con trỏ void* trỏ đến bất kỳ đối tượng nào, sau đó ép kiểu lúc cần thiết.
- Khi trỏ đến struct có thể truy suất bằng (*p).a (explicit) hoặc p->a (implicit).

Vấn đề thiết kế của Pointer:

- Scope và thời gian sống của một biến pointer.
- Thời gian sống của một đối tượng cấp phát động.
- Pointer có bị giới hạn trỏ đến type của một giá trị nào đó hay không.
- Pointer dùng để quản lý cấp phát động, truy suất gián tiếp hay cả 2.
- Ngôn ngữ có hỗ trợ pointer type và reference type hay cả 2.
- Reference Type: Pointer trỏ tới địa chỉ, reference trỏ tới đối tượng hoặc giá trị.

Tổng kết pointer:

- Linh hoạt nhưng dễ gây lỗi.
- Giống goto – làm chương trình rối rắm.
- Cần thiết như để viết driver, giúp dễ truy cập đến bất kỳ vùng nhớ nào.
- Reference cũng linh hoạt như pointer, nhưng không có huỷ bỏ, tránh gây lỗi như pointer. Tuy nhiên gặp vấn đề về Alias, khiến sửa 1 bên thì bên khác bị thay đổi, dễ dẫn đến lỗi.

Biểu diễn pointer: tuỳ theo cách lưu địa chỉ của các thiết bị khác nhau, pointer sẽ lưu trữ khác nhau.

Vấn đề dangling pointer (con trỏ trỏ đến đối tượng bị huỷ): Tombstone, Locks – and – keys.

Recursive Type: khi khai báo kiểu, nó lại nhắc đến chính kiểu đó (trong link list, node chứa node tiếp theo).

Type Expression: biểu thức kiểu

- Kiểu cơ bản (basic type), tên kiểu (type name), biến kiểu (type variable) là một biểu thức kiểu.
- Type constructor kết hợp các biểu thức kiểu thành một biểu thức kiểu.
    - Array: array(index type, ele type)
    - Product: T1 x T2
    - Record: record((name1 x T1) x (name2 x T2) x …)
    - Pointer: pointer(T)
    - Function: T1 → T2

array(1..10,record((a*array(5..10,integer))*(b*record((c*real)*(d*array(1..3,real))))))

Type Checking: kiểm tra kiểu: hành vi để đảm bảo chương trình tuân theo tất cả các luật đã quy định liên quan hệ thống kiểu.

- ***Static type checking:*** kiểm tra thời gian dịch (compiling time). Ngôn ngữ buộc khai báo kiểu hoặc ràng buộc về kiểu có thể kiểm tra được ở thời gian dịch.
- ***Dynamic type checking:*** kiểm tra thời gian thực thi (running time). Vận dụng trong các ngôn ngữ ràng buộc kiểu xảy ra trong thời gian thực thi (dynamic type binding). Sử dụng trên static type binding để kiểm tra các phần không thể kiểm tra trong thời gian dịch mà phải kiểm tra trong thời gian thực thi.

Type Inference: suy biến kiểu: không cần khai báo đầy đủ kiểu, chương trình sẽ tự suy diễn kiểu cho các phần tử không khai báo. Khi biến được gán tự động cho kiểu nào rồi sẽ vĩnh viễn là kiểu đó, không thay đổi được.

- Gán kiểu cho các node lá trên AST.
- Tạo ràng buộc kiểu cho các node bên trong của AST.
- Giải ràng buộc kiểu để suy ra các kiểu chưa có.

Type Equivalence: tương đương kiểu. Kiểm tra đối tượng suất hiện tại vị trí, có kiểu thích hợp hay không.

- Một toán hạn của một kiểu có thể thay thế bằng toán hạn khác mà không cần ép kiểu.
- Tương đương theo tên: 2 kiểu cùng tên (a = Float và b = Float → a tương đương b dù cả 2 khác kiểu).
- Cấu trúc tương đương: tên khác nhau nhưng cấu trúc bên trong giống nhau.

Type Compatibility: tương thích kiểu. T tương thích với S nếu bất kỳ chỗ nào cho phép S, T có thể để vào. T tương thích với S khi:

- T là tương đương với S.
- Giá trị của T là tập con giá trị của S.
- Tất cả tác vụ trên S thì T thực hiện được.
- Giá trị của T thì tương ứng theo một cách nào đó với các giá trị của S.
- Giá trị của T có thể chuyển đổi sang giá trị của S.

Type Conversion: chuyển đổi kiểu. Khi chuyển đổi giá trị từ kiểu này sang kiểu khác.

- Implicit conversion – coersion: ép kiểu tự động (bên dưới tự động đổi kiểu để gán vào).

- Explicit conversion – cast: lập trình viên tự ép đối tượng sang kiểu mình muốn.

Polymorphism: tính đa hình trên kiểu:

- Monomorphic: một đối tượng chỉ có 1 kiểu.
- Polymorphic: một đối tượng có thể có nhiều hơn 1 kiểu.
- Ad hoc polymorphism – Overloading.
- Universal polymorphism: Parameter polymorphism (thông số đa hình), Subtyping polymorphism (đa hình kiểu con).

JVM and Jasmin

Compiler: từ Source code

- Scanner (phân tích từ vựng), sinh ra các token.
- Parser (nhận lỗi văn phạm), sinh ra AST.
- Semantic analyzer (phân tích ngữ nghĩa, kiểm tra lỗi tầm vực, kiểu)
- Code generation (sinh ra mã máy), tạo ra mã Jasmin (file ".j"), là mã assembly của ngôn ngữ java bytecode.
- Java Bytecode Assembler: chuyển Jasmin code thành Java Byte code (file ".class").
- Java Virtual Machine: máy ảo của Java để chạy file Java Byte code.

Java Programming Environment:

- Code trên các file ".java"
- Dùng Java compiler chuyển của file ".java" sang file ".class".
- Đưa file ".class" lên máy ảo Java Virtual Machine, kết hợp với các file ".class" của thư viện có sẵn để thực thi các file ".class".

Mã Jasmin:

- Là mã máy, tương ứng 1:1 với các lệnh trên máy ảo Java.
- Có thể đọc được.

Java Virtual Machine

JVM = stack – based machine: chạy dựa trên stack. Các giá trị các phép toán dựa trên stack.

- Một stack cho mỗi phương thức.
- Stack được sử dụng lưu trữ các phép toán và kết quả của biểu thức.
- Dùng stack để truyền thông số và nhận giá trị trả về.
- Code generation cho stack – based machine thì dễ hơn sử dụng register – based machine.

Kiến trúc bên trong của JVM:

- Bộ "class loader subsystem": nạp các class file lên để bỏ vào các vùng lưu dữ liệu.

- Các vùng lưu dữ liệu gồm: method area, heap, Java stacks, pc register, native method stacks (stack lưu các phương thức được viết bằng mã máy chứ không phải mã Java Bytecode).
- "Execution engine": đọc các dữ liệu trong các vùng để thực thi.

Các vùng lưu trữ dữ liệu:

- Method area: chứa class data (dữ liệu tĩnh về các class).
- Heap: chứa các object.
- Java stacks: có nhiều stack, mỗi stack là một thread chạy song song. Những thread là stack chứa các stack frame (bảng ghi hoạt động). Chương trình con mới sẽ ở đầu stack.
- Pc register: mỗi thread sẽ có 1 cái program counter để xác định vị trí đang chạy.
- Native method stacks: stack chứa các phương thức viết dưới dạng mã máy.

Data Type:

Type	Range	Description
boolean	{0,1}	Z
byte	-2^7 to 2^7 - 1, inclusive	B
short	-2^{15} to 2^{15} – 1, inclusive	S
int	-2^{31} to 2^{31} – 1, inclusive	I
long	-2^{63} to 2^{63} – 1, inclusive	L
char	16 bit usigned Unicode (0 to 2^{16} -1)	C
float	32-bit IEEE 754 single-precision float	F
double	64-bit IEEE 754 double-precision float	D
returnAddress	address of an opcode within the same method	
class reference		Lclass-name;
interface reference		Linter-name;
array reference		[[..[component-type;
void		V

Operand stack: để cất các toán hạng lên, mỗi chương trình con khi thực thi sẽ có 1 operand stack. Truy cập bằng cách push và pop giá trị.

- Lưu trữ toán hạng và nhận kết quả phép toán.
- Truyền tham số và nhận kết quả các phương thức.

Local Variable Array:

- Vùng cấp phát cho các dữ liệu cục bộ, được tạp ra trong stack frame.
- Đánh chỉ số bắt đầu từ 0.
- Đối với phương thức instance: vị trí số 0 là this, các phần khác tính từ 1.
- Đối với phương thức class: các phần tính từ 0 do không có this.
- 1 slot là dữ liệu dạng boolean, byte, char, short, int, float, reference, returnAddress.
- Đối với long và double sẽ là 2 slot.
- Chỉ số slot chỉ tính theo thứ tự biến tồn tại từ trên xuống trong quá trình thực thi.

Instructions: gồm 10 nhóm:

- Arithmetic Instructions.
- Load and store Instructions.
- Control transfer Instructions.
- Type conversion Instructions.
- Operand stack management Instructions.
- Object creation and manipulation.
- Method invocation Instructions.
- Throwing Instructions (không dùng).
- Implementing ***finally*** (không dùng).
- Synchronisation (không dùng).

Arithmetic Instructions: lệnh số học.

- Add: *iadd*, *ladd*, *fadd*, *dadd*.
- Subtract: *isub*, *lsub*, *fsub*, *dsub*.
- Multiply: *imul*, *lmul*, *fmul*, *dmul*.
- Divide: *idiv*, *ldiv*, *fdiv*, *ddiv*.
- Remainder: *irem*, *lrem*, *frem*, *drem*.
- Negate: *ineg*, *lneg*, *fneg*, *dneg*.
- Shift: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.
- Bitwise OR: *ior*, *lor*.
- Bitwise AND: *iand*, *land*.
- Bitwise exclusive OR: *ixor*, *lxor*.
- Local variable increment: *iinc*.
- Comparison: *dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*, *lcmp*.

Load and Store: nạp và lấy dữ liệu từ stack để cất vào local variable array.

- Load a local variable onto the operand stack:
  iload, *iload_<n>*, ⇒ n:0..3, used for int, boolean, byte, char or short
  lload, *lload_<n>*, ⇒ n:0..3, used for long
  fload, *fload_<n>*, ⇒ n:0..3, used for float
  dload, *dload_<n>*, ⇒ n:0..3, used for double
  aload, *aload_<n>*, ⇒ n:0..3, used for a reference
  Taload. ⇒ T:b,s,i,l,f,d,c,a

- Store a value from the operand stack into a local variable:
  istore, *istore_<n>*, ⇒ n:0..3, used for int, boolean, byte, char or short
  lstore, *lstore_<n>*, ⇒ n:0..3, used for long
  fstore, *fstore_<n>*, ⇒ n:0..3, used for float
  dstore, *dstore_<n>*, ⇒ n:0..3, used for double
  astore, *astore_<n>*, ⇒ n:0..3, used for a reference and returnAddress
  Tastore. ⇒ T:b,s,i,l,f,d,c,a

  - Taload và Tastore để đọc và lưu các phần tử trên dãy. T là kiểu dữ liệu (b, s, I, l, f, d, c, a)

- Load a constant onto the operand stack:
  bipush, ⇒ for an integer constant from -2^7 to $2^7 - 1$
  sipush, ⇒ for an integer constant from -2^{15} to $2^{15} - 1$
  ldc, ⇒ for a constant that is an integer, float or a quoted string
  ldc_w,
  ldc2_w, ⇒ for a constant that is a long or a double
  aconst_null, ⇒ for a null
  iconst_m1, ⇒ for -1
  iconst_<i>, ⇒ for 0,…,5
  lconst_<l>, ⇒ for 0,1
  fconst_<f>, ⇒ for 0.0,1.0 and 2.0
  dconst_<d>. ⇒ for 0.0,1.0

Control Tranfer Instruction: nhóm định chuyển điều khiển.

- Unconditional branch:
  goto, *goto_w*, *jsr*, *jsr_w*, *ret*.
- Conditional branch:
  ifeq, *iflt*, *ifle*, *ifne*, *ifgt*, *ifge*, ⇒ compare an integer to zero
  ifnull, *ifnonnull*, ⇒ compare a reference to null
  if_icmpeq, *if_icmpne*, *if_icmplt*, *if_icmpgt*, *if_icmple*, *if_icmpge*, ⇒ compare two integers
  if_acmpeq, *if_acmpne*. ⇒ compare two references
- Compound conditional branch:
  tableswitch, *lookupswitch*.

Let a and b be declared as an one-dimension arrays of shorts whose index is 2 and 3, respectively. What is the Jasmin code of expression a[10] = b[3] * 4.

aload_2
bipush_10
aload_3
iconst_3
saload
iconst_4
imul
sastore

Let a and b be an int variables whose index is 1 and 2, respectively. What is the Jasmin code of the following Java code?

```
if (a < 10) then b = 100; else b = 1000;
```

iload_1
bipush_10
if_icmpge Label0
bipush_100
istore_2
goto Label1
Label0:
sipush_1000
istore_2
Label1:

Đối với số thực muốn so sánh, dùng fcmpg và fcmpl:

value_2
value_1
fcmpg

- Nếu value 1 > value 2: push 1.
- Nếu value 1 == value 2: push 0.
- Nếu value 1 < value 2: push -1.
- Cả 2 lệnh đều giống nhau. Nhưng nếu trong trường hợp một trong 2 số không phải là số, fcmpg trả về 1, fcmpl trả về -1.

```
int foo(float a) { int c; if (a < 10.0f) c = 22; else c = 400; return c;}
```

fload_1
ldc_10.0
fcmpl
ifge Label0
bipush_22
istore_2

```
goto Label1
Label0:
sipush_400
istore_2
Label1:
```

Object Creation and Manipulation:

- Tạo ra đối tượng: new.
- Tạo ra mảng: newarray, anewarray, multianewarray.
- Truy suất field của lớp và field của đối tượng: getfield, putfield, getstatic, putstatic.
  Syntax: <lệnh> <field_spec (class_name + '.' + field_name)> <descriptor (field_type)>

```
float x[] =new float [100];
bipush_100
newarray float
astore_3
```

```
public class VD {
   static float x[] = new float[25];
...
What are the Jasmin code of the declaration of field x?
```

```
bipush_25
newarray float
putstatic VD.a [F
```

Method Invocation Instructions: gọi method.

- Đối với invokestatic (gọi các phương của class), invokevirtual (gọi các phương thức của đối tượng), invokespecial (gọi constructor <init>, private method, method trong lớp cha). Thêm <method_spec (class_name + '/' + method_name + method_type)>.
- Đối với invokeinterface. Thêm <method_spec> <num_args>.

invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

Method Return: đối với các loại khác "return", yêu cần còn 1 giá trị trên đỉnh stack để trả về.

- return → void
- ireturn → int, short, char, boolean, byte
- freturn → float
- lreturn → long
- dreturn → double
- areturn → reference

Jasmin Directive: các chỉ dẫn để sinh mã.

- .source <source.java>: tên của file java.
- .class <the current class>: tên file class.
- .super <the super class>: tên của lớp cha lớp hiện tại.
- .limit: limit stack hoặc limit local.
- .method <the method description>: tên method.
- .field <the field description>: tên field.
- .end: kết thúc một method.
- .var <the variable description>: mô tả về biến.
- .line <the line number in source code>: line trên source code.

Code Generation Framework:

Gồm 4 thành phần:

- ***Machine – Dependent Code Generation:*** phụ thuộc hoàn toàn mã cần sinh. Là một class để sinh mã cần sinh.
- ***Intermediate Code Generation:*** vừa phụ thuộc mã nguồn, vừa phụ thuộc mã máy. Giúp việc sinh mã dễ dàng hơn.
- ***Frame:*** quản lý các nhãn (địa chỉ của các lệnh để nhảy) giữa các method, operand stack của các method, tập các biến cục bộ của các method.
- ***Machine – Independent Code Generation:*** (phải làm) dùng các hàm, phương thức đã được cung cấp ở Frame và Intermedia Code Generation để hiện thực sinh mã.

Machine – Dependent Code Generation:

- Được hiện thực trong các module là machine code.
- Để sinh ra mã Jasmin thì trong đó sẽ có 1 class JasminCode.
- Có các phương thức, mỗi phương thức sẽ ứng với lệnh của Jasmin.

Intermediate Code Generation:

- Phụ thuộc cả ngôn ngữ và máy.
- Giúp lựa chọn lệnh dễ dàng.
- Giúp lựa chọn đối tượng dữ liệu.
- Mô phỏng việc thực thi của máy. Trong Intermediate Code Generation sẽ có 1 cái stack, giả sử khi gọi emitICONST, nó sẽ push() và, khi gọi emitISTORE sẽ pop() ra. Sau khi xong, gọi để nhất kích thước stack lớn nhất trong các lần và được limit stack.

Directives Generation APIs:

- emitVAR(self, index, varName, inType, fromLabel, toLabel)

  Sinh ra: .var ***index*** is ***varName inType***; from ***fromLabel*** to ***toLabel***

- emitATRIBUTE(self, lexeme (tên field), inType, isFinal (const?), value = None)

  Sinh ra: .field public static writer Ljava/io/Writer;

- emitMETHOD(self, lexeme, inType, isStatic)

  Sinh ra: .method public foo(I)I

- emitENDMETHOD(self, frame)

  Sinh ra 3 dòng: .limit stack 1(dựa vào frame)

  .limit locals 1(dựa vào frame kiểm tra có bao nhiêu biến local)

  .end method

- emitPROLOG(self, name, parent)

  Sinh ra 3 dòng: .source io.java

  .class public io

  .super java/lang/Object

- emitEPILOG(self) (dùng để ghi tất cả các lệnh đã ghi ở buffer xuống file)

Type:

- class IntType(Type)
- class FloatType(Type)
- class StringType(Type)
- class VoidType(Type)
- class BoolType(Type)
- class ClassType(Type): # cname:str
- class ArrayType(Type): #eleType:Type,dimen:List[int]
- class Mtype(Type): #partype:List[Type],rettype:Type

Operation Generation APIs:

- emitADDOP(self, lexeme(‘+’|’-‘), inType(‘int’|’float’), frame) → iadd, fadd, isub, fsub
- emitMULOP(self, lexeme(‘*’|’/‘), inType(‘int’|’float’), frame) → imul, fmul, idiv, fdiv
- emitDIV(self,frame) → idiv
- emitMOD(self,frame) → irem
- emitANDOP(self,frame) → and
- emitOROP(self,frame) → or

- emitREOP(self,op,inType,frame) → mã cho >, <, >=,<=, !=, ==. Kết quả trên stack sẽ là 1 hoặc 0.
- emitRELOP(self,op,inType,trueLabel,falseLabel,frame) → mã cho điều kiện trong ifstmt. Sẽ tự nhảy đến vị trí cần thiết khi biểu thức đúng hoặc sai, không cần thêm 0 hoặc 1 trên stack.

Read/Write Variables APIs:

- emitREADVAR(self,name,inType,index,frame) → [ifa]load
- emitALOAD(self,inType,frame) → [ifa]aload
- emitWRITEVAR(self,name,inType,index,frame) → [ifa]store
- emitASTORE(self,inType,frame) → [ifa]astore
- emitGETSTATIC(self,lexeme,inType,frame) → getstatic
- emitGETFIELD(self,lexeme,inType,frame) → getfield
- emitPUTSTATIC(self,lexeme,inType,frame) → putstatic
- emitPUTFIELD(self,lexeme,inType,frame) → putfield

Other APIs:

- emitPUSHICONST(self,input(hằng nguyên hoặc chuỗi true/false),frame) → iconst, bipush, sipush, ldc
- emitPUSHFCONST(self,input(phải là str),frame) → fconst, lds
- emitINVOKESTATIC(self,lexeme,inType(phải là Mtype),frame) → invokestatic
- emitINVOKESPECIAL(self,frame,lexeme=None,inType=None) → invokespecial
- emitINVOKEVIRTUAL(self,lexeme,inType,frame)→invokevirtual
- emitIFTRUE(self,label(phải là giá trị nguyên),frame) → ifgt
- emitIFFALSE(self,label(phải là giá trị nguyên),frame) → ifle
- emitDUP(self,frame) → dup
- emitPOP(self,frame) → pop
- emitI2F(self,frame) → i2f
- emitRETURN(self,intType,frame) → return, ireturn
- emitLABEL(self,label(phải là giá trị nguyên),frame) → Label
- emitGOTO(self,label(phải là giá trị nguyên),frame) → goto

Frame: công cụ dùng để quản lý các thông tin liên quan đến một phương thức, hàm trong quá trình sinh mã cho nó. Khi qua một phương thức, hàm khác, tất cả sẽ được khởi động lại.

- ***Label:*** chỉ hợp lệ trong thân một phương thức, qua phương thức khác sẽ bị reset.
  getNewLabel(): trả về 1 giá trị nguyên ứng với 1 label.
  getStartLabel(): trả về label đầu tiên của scope.
  getEndLabel(): trả về label cuối cùng của scope.
  getContinueLabel(): trả về label lệnh continue nhảy đến.
  getBreakLabel(): trả về label lệnh break nhảy đến.
  enterScope(): sinh ra StartLabel và EndLabel.
  exitScope(): loại bỏ StartLabel và EndLabel.
  enterLoop(): sinh ra ContinueLabel và BreakLabel.

exitLoop(): loại bỏ ContinueLabel và BreakLabel.

- ***Local variable array:*** tạo ra index cho các biến cục bộ.
  getNewIndex(): trả về index mới cho biến.
  getMaxIndex(): trả về size của local variable array.
- ***Operand stack:*** xác định giá trị max của operand stack để cấp phát cho mỗi frame.
  push(): mô phỏng push thực thi.
  pop(): mô phỏng pop thực thi.
  getMaxOpStackSize(): trả về max size của operand stack.
- Hiện thực trong class Frame.

Machine – Independent Code Generation:

- Dựa trên ngôn ngữ nguồn.
- Sử dụng phương thức trong Frame và Intermediate Code Generation (Emitter).

Sequence Control: Điều khiển dòng thực thi của chương trình.

- ***Expression:*** Điều khiển trình tự tính toán trong biểu thức.
- ***Statement:*** Điều khiển trình tự tính toán trong phát biểu.
- ***Program Unit:*** Điều khiển việc chuyển điều khiển từ chương trình này đến chương trình khác.

Expression: quá trình tính toán tạo ra 1 giá trị, nếu gặp lỗi sẽ ra undefined.

- Dùng cơ chế functional composite nature (chồng trực hàm): tính biểu thức con, chồng lên tính toán biểu thức lớn.
- Các hình thức: Infix (trung tố), Prefix (tiền tố), Postfix (hậu tố).

❖ ***Infix:*** (a + b) * (c - d)

- Tốt cho phép toán 2 ngôi.
- Dùng nhiều cho imperative programming language (ngôn ngữ lập trình thủ tục).
- Gặp vấn đề khi có nhiều hơn 2 toán hạng.
- Gặp vấn đề về độ ưu tiên nếu không có dấu ngoặc.
- Gặp vấn đề về tính kết hợp khi tính 2 hay nhiều phép toán có cùng độ ưu tiên nên tính trái trước hay phải trước. Thường tính từ trái quá phải, phép luỹ thừa từ phải qua trái. Có thể thay đổi trình tự tính toán nếu giúp tối ưu và không thay đổi kết quả.
- Để giải quyết độ ưu tiên và tính kết hợp → dùng dấu ngoặc để thay đổi độ ưu tiên, độ kết hợp → đơn giản nhưng khó viết và khó đọc.

❖ ***Prefix:*** 3 loại:

- Polish Prefex: * + a b – c d. Số toán hạng phải là cố định.
- Cambridge Polish Prefix: (* (+ a b) (- c d)). Cho phép số toán hạng thay đổi được.
- Normal Prefix: *(+(a,b),-(c,d)). Dạng hàm f(x,y).
- Prefix không cần tính ưu tiên.

❖ ***Postfix:*** 3 loại:

- Polish Postfix: a b + c d - *. Số toán hạng phải là cố định.
- Cambridge Postfix Prefix: ((a b +) (c d -) *). Cho phép số toán hạng thay đổi được.
- Normal Postfix: ((a,b)+,(c,d)-)*. Dạng hàm f(x,y).
- Postfix không cần tính ưu tiên.

❖ ***Cơ chế tính toán:***

- Eager evaluation: Tính toán tất cả toán hạng trước, sau đó tính các phép toán → có thể gặp vài lỗi như chia cho 0.
- Lazy evaluation (tính toán lười): truyền toán hạng vào phép toán, phép toán tự quyết định tính toán hạng nào → chi phí cao hơn eager.
- Dùng lazy cho phép toán điều kiện, eager cho các phép còn lại.

❖ ***Short – Circuit Evaluation:*** rút ngắn quá trình tính toán. Thường xảy trong trong phép toán AND và OR. Cần quy định tính toán số hạng bên nào trước. Giúp tránh một số lỗi như chia cho 0. Một số ngôn ngữ cung cấp cả 2 do non short – circuit chi phí thực thi thấp hơn short – curcuit.

Statement: đơn vị văn phạm thực thi không trả về giá trị nhưng làm thay đổi trạng thái của hệ thống.

❖ ***Assignment Stmt:*** leftExpr AssignOp rightExpr. Bên phải tính ra giá trị, bên trái tính ra địa chỉ để chứa giá trị bên phải vào. Tính toán bên nào trước phụ thuộc lúc hiện thực.
- Trên một số ngôn ngữ, xem phép gán như 1 biểu thức, sẽ trả về 1 giá trị là giá trị của rightExpr.
- Có thêm các phép kết hợp vừa gán vừa tính toán (+=, ++, …)

❖ ***Control Structure:*** Cho phép lựa chọn các nhánh điều khiển khác nhau, thực thi một nhóm các phát biểu. Cấu trúc điều khiển là phát biểu điều khiển và tập hợp các phát biểu được nó điều khiển.

- ***Two – way Selection:*** cấu trúc điều khiển 2 nhánh (If Stmt). Là phát biểu căn bản và cần thiết trên hầu hết ngôn ngữ lập trình.
  - Nếu cung cấp cả loại if có elsse và không có else sẽ có thể gây dangling else. Tuỳ theo ngôn ngữ lập trình sẽ xử lý khác nhau (gán else cho if gần nhất, dùng block để xác định, …).
  - Một vài ngôn ngữ không gặp lỗi này nhờ dùng ký hiệu đặc biệt để kết thúc stmt (Fortan 95, Ada, Ruby), dùng cơ chế indentation (chia theo cột ngang hàng).
- ***Multiple – Selection:*** cho phép lựa chọn 1 trong nhiều nhóm mệnh đề khác nhau, Perl và Python không có. Các vấn đề: kiểu biểu thức lựa chọn, làm sao lựa chọn, thực thi 1 hay nhiều nhóm, làm sao mô tả giá trị mỗi trường hợp, nếu giá trị không phù hợp với tất cả lựa chọn.
- ***Iterative Statement:*** cho phép lặp đi lặp lại 1 hay nhiều phát biểu 0, 1 hay nhiều lần. Là sức mạnh của máy tính.
  - Làm sao điều khiển số lần lặp: logic hoặc phép đếm.
  - Điều khiển sẽ kiểm tra ở đâu trong vòng lặp: pretest, posttest.
  - Điều khiển vòng lặp bằng biến đếm cần có: biến đếm, giá trị bắt đầu và kết thúc, bước tăng.
  - Điều khiển bằng biểu thức luận lý: tổng quát hơn lặp dựa trên biến đếm. Cần quan tâm là pretest hay posttest, liệu vòng lặp đó là 1 dạng riêng của lặp trên biến đếm hay dùng 1 phát biểu khác.
- ***User – Located Loop Control:*** Người lập trình chọn vị trí để kiểm tra điều kiện lặp thay vì đầu hay cuối vòng lằm. Cách hiện thức đơn giản là vòng lặp vô hạn kết hợp break và continue. Cần giới hạn các phát biểu goto.
- I***teration Based on Data Structures:*** dựa trên số lượng dữ liệu trong 1 cấu trúc dữ liệu nào đó.
  - Dùng Iterator: gọi ở bắt đầu mỗi vòng lặp, trả về phần tử mỗi lần lặp.

- Dùng pre – defined iterator (Iterator được định nghĩa trước) hoặc user – defined.

- ***Unconditional Branching:*** phát biểu nhảy không điều kiện (goto). Là một lệnh mạnh nhưng làm chương trình khó đọc, khó bảo trì, không tin cậy → không dùng goto. Java, Python, Ruby không có goto.

Control Abstraction: điều khiển tuần tự ở cấp đơn vị chương trình con.

Subprogram definition (định nghĩa chương trình con): bao gồm:

- Đặc tả (specification): tên, thông số (input, output, thứ tự thông số, kiểu, cơ chế truyền thông số), hành vi chương trình con.
- Hiện thực (implementation): dữ liệu cục bộ, tập hợp các phát biểu để tạo thành thân chương trình con.

Subprogram Activation (bảng hoạt động của chương trình con):

- Được tạo ra khi chương trình con được thực thi.
- Được huỷ khi chương trình con hoàn tất.

Gồm:

- Static part (phần tĩnh): mã của chương trình con (code được dịch sang mã).
- Dynamic part (phần động): bảng ghi hoạt động (activation record) lưu trữ các thông số hình thức, dữ liệu cục bộ, địa chỉ trả về, những đường link khác.'

Subprogram Mechanisms (cơ chế chương trình con):

- Simple Call – Return (gọi trả về đơn giản).
- Recursive Call (gọi đệ quy).
- Exception Processing Handler (biến cố và xử lý biến cố).
- Coroutines (cộng hành).
- Scheduled Subprograms (định thời).
- Tasks (các nhiệm vụ).

Simple Call – Return: cơ chế đơn giản nhất.

- Không cho gọi đệ quy.
- Gọi đơn giản: lệnh gọi tường mình, rõ ràng.
- Chỉ có 1 điểm vào: chỉ có 1 điểm bắt đầu chương trình.
- Truyền điều khiển ngay lập tức khi được truyền.
- Thực thi một chương trình trong 1 thời điểm.

Recursive Call:

- Gồm: đệ quy trực tiếp (gọi lại chính nó), đệ quy gián tiếp (gọi qua chương trình khác).
- 4 đặc tính còn lại như Simple Call – Return.

Exception Processing Handler:

- Có thể không có lệnh gọi tường minh: không có lệnh gọi nhưng vẫn chuyển điều khiển được.
- Sử dụng trong:

- Lập trình hướng sự kiện (Event – Driven Programming)
- Xử lý biến cố (Error Handler – gặp lỗi ném ra biến cố).

- Cơ chế biến cố:
  - Những loại biến cố sẽ xử lý.
  - Làm sao chúng được định nghĩa: Java (con của Throwable), Ada (dữ liệu của loại đặc biệt), C++ (bất kỳ dữ liệu nào).
  - Làm thế nào để raise một biến cố.
    - Thao tác của người dùng: click, di chuyển chuột, thao tác, … (trình điều khiển theo dõi thao tác của người dùng để ném ra các biến cố).
    - Do hệ điều hành.
    - Bằng một đối tượng trong chương trình (Timer).
    - Do lập trình viên (throw).
  - Làm thế nào một biến cố được xử lý.
    - Định nghĩa protected block để quan sát có biến không không rồi xử lý.
    - Có những ngữ nghĩa kết thúc khác nhau:
      - ***Non – resumable*** (xử lý xong không quay lại chỗ xảy ra biến cố) kết hợp ***stack unwinding*** (các bảng hoạt động của các chương trình đặt lên stack, khi gặp biến cố sẽ bay về bảng thấp hơn và xoá bảng hiện tại, nó sẽ làm liên tục đến khi gặp bảng hoạt động có phần xử lý biến cố).
      - ***Resumable***: xử lý xong sẽ quay lại vị trí gây biến cố để chạy lại hoặc quay lại sau phát biểu gây ra lỗi để chạy tiếp.

Coroutines: trình cộng hành:

- Có nhiều điểm để vào trong chương trình.
- 2 hay nhiều chương trình thay phiên nhau chạy, chương trình này qua chương trình kia, chương trình kia chạy 1 phần xong trở lại điểm kết của chương trình ban đầu để chạy tiếp.
- Thay phiên nhau chạy.
- Vị trí chuyển sẽ do người lập trình quyết định.

Tasks: trình công tác:

- Có thể thực thi đồng thời với những task khác.
- Chạy trên máy có nhiều bộ xử lý.
- Trên máy có một bộ xử lý, có thể chạy bằng time sharing (tương tự Coroutines nhưng vị trí chuyển task do máy quyết định). Có thể gây ra vấn đề về đồng bộ (race condition, deadlock) hoặc vấn đề trao đổi dữ liệu giữa các task.

Scheduled Subprograms: cơ chế gọi định thời.

- Khi chuyển đổi giữa các chương trình, không nhất thiết thực hiện ngay, có thể trì hoãn theo thời gian hoặc theo độ ưu tiên.
- Điều khiển bởi scheduler.

Parameter Passing (truyền thông số):

- Thông số hình thức: thông số khi khái báo. Gồm các thông số truyền vào và các thông số trả về. Chỉ là tên đơn, giống khai báo biến, thường có thêm ký hiệu để chỉ phương pháp truyền thông số.
- Thông số thức: thông số truyền vào khi gọi hàm. Nó là một biểu thức.

- Kết hợp thông số hình thức và thông số thức: có tương ứng theo vị trí (thông số thực đứng trước đưa và thông số hình thức trước, cần đảm bảo đủ thông số và nhớ vị trí) và tương ứng theo tên (cần nhớ tên thông số hình thức để truyền).
- Các phương pháp truyền thông số: Input – Output, Input Only, Output Only.

Input – Output: thông số vừa truyền vào vừa lấy ra.

- ***Truyền bằng trị - kết quả.*** Giá trị được truyền vào và tính toán, khi chương trình kết thúc, giá trị sẽ được chép ngược trở lại cho thông số truyền vào. Thông số truyền vào sẽ bị thay đổi chỉ khi chương trình kết thúc.
- ***Truyền bằng tham khảo (reference).*** Thông số hình thức bị thay đổi bên trong hàm, thông số thực truyền vào cũng thay đổi theo. Dấu & trong C.
- ***Truyền bằng tên.*** Truyền trực tiếp tên vào thông số hình thức. Thông số hình thức sẽ lưu cả biểu thức của thông số thực.

Input Only: chỉ là thông số truyền vào để sử dụng.

- ***Truyền bằng trị.*** Truyền vào giá trị của biểu thức. Chương trình kết thức không truyền ngược ra lại.
- ***Truyền bằng tham khảo hằng (constant reference).*** Cũng truyền địa chỉ vào, nhưng thông số hình thức sẽ xem là 1 hằng, nên không thể thay đổi. Dùng const và & trong C. Nên dùng cho thông số thực là dữ liệu có kích thước lớn do copy địa chỉ thì chi phí thấp hơn copy toàn bộ dữ liệu vào.

Output Only: thông số chỉ để nhận dữ liệu chương trình con trả ra.

- ***Truyền bằng kết quả.*** Khi truyền thông số thực vào, sẽ không chép qua thông số hình thức. Thay vào đó, khi kết thúc, giá trị của thông số hình thức sẽ được chép ngược trở ra thông số thực truyền vào.
- ***Truyền dạng kết quả của một hàm.*** Dùng return trả ra thông số là kiểu của hàm → không có thông số thực.

High – order functions:

Môi trường tham khảo không cục bộ (non – local environment):

- Deep binding (ràng buộc sâu), dùng cho ngôn ngữ tầm vực động. Khi truyền hàm vào hàm, truyền kèm theo môi trường hiện tại.
- Shallow binding (ràng buộc cạn), dùng cho ngôn ngữ tầm vực động. Khi gọi hàm được truyền vào trong hàm, môi trường của hàm đó sẽ là môi trường hiện tại.

Hàm là giá trị trả về:

- Cái gì trả về như là hàm: địa chỉ đến code của hàm, môi trường tham khảo của chương trình.