Introduction to Programming Languages and Compilers

Principles of Programming Languages

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy
Department of Computer Science
Faculty of Computer Science and Engineering
Ho Chi Minh University of Technology, VNU-HCM

Overview

1. **Programming languages**
   - Language evaluation
   - Influences on Language design
2. **Language implementation**
3. **The Structure of a Compiler**

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

INTRODUCTION TO PROGRAMMING LANGUAGES

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Programming languages

Programming languages are notations for describing computations to people and to machines.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Programming languages

Programming languages are notations for describing computations to people and to machines.

Programming domains

- Scientific Applications: Fortran, ALGOL 60, Julia, MATLAB.
- Business Applications: COBOL, DATABUS, PL/B, ...
- Artificial Intelligence: LISP, Prolog, ...
- Systems Programming: PL/S, BLISS, Extended ALGOL, C, ...
- Web Software: XHTML, JavaScript, PHP, Java, C# (ASP.NET), ...

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Language characteristics

Clarity: ++a, a++, a+=1... -> mất tính rõ ràng
"abc" + "def"-> mất tính đơn giản

- **Clarity, simplicity, and unity**: provides both a framework for thinking about algorithms and a means of expressing those algorithms.
- **Orthogonality**: every combination of features is meaningful. Independent functions should be controlled by independent mechanisms. Mỗi tính năng đều độc lập + có nghĩa, giảm ngoại lệ
- **Support of abstraction (Control, Data)**: program data reflects problem being solved, program structure reflects the logical structure of algorithm.
- **Safety**: The ease of which errors can be found and corrected and new features added.
- ... a[4], truy cập a[5] không báo lỗi (trị rác) -> không safety

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

1. Clarity, simplicity, and unity
Clarity (rõ ràng): Ngôn ngữ phải dễ đọc, dễ hiểu, giúp lập trình viên tư duy rõ ràng hơn khi xây dựng thuật toán.
Simplicity (đơn giản): Cung cấp cú pháp và chức năng đơn giản, tránh sự phức tạp không cần thiết.
Unity (tính nhất quán): Các khái niệm và cơ chế phải được thiết kế chặt chẽ, logic, và thống nhất trong toàn bộ ngôn ngữ.
Ý nghĩa: Ngôn ngữ cần vừa là công cụ để lập trình viên diễn đạt thuật toán, vừa là khung để tư duy về cách giải quyết bài toán.
2. Orthogonality
Khái niệm: Tính orthogonality (tính trực giao) nghĩa là các tính năng của ngôn ngữ phải kết hợp với nhau một cách độc lập và hợp lý.
Mỗi tính năng hoạt động độc lập, không phụ thuộc hay ảnh hưởng đến tính năng khác.
Mỗi sự kết hợp giữa các tính năng đều có ý nghĩa, không tạo ra những hành vi mơ hồ.
Ví dụ: Trong ngôn ngữ trực giao, bạn có thể kết hợp kiểu dữ liệu (data types) với các thao tác (operations) mà không gặp lỗi khó hiểu.
Nếu bạn có kiểu dữ liệu int và float, bạn nên có thể cộng chúng (int + float) một cách hợp lý.
Ý nghĩa: Tránh tình trạng chồng chéo hoặc không nhất quán, giúp lập trình viên dễ hiểu và kiểm soát mã nguồn.

3. Support of abstraction (Control, Data)
Abstraction (trừu tượng hóa): Ngôn ngữ cần hỗ trợ:
Control abstraction: Cho phép bạn định nghĩa các cấu trúc điều khiển (như hàm, vòng lặp) để phản ánh logic của thuật toán.
Data abstraction: Cho phép bạn làm việc với dữ liệu ở mức trừu tượng (như lớp, kiểu dữ liệu tùy chỉnh), phản ánh bài toán thay vì chi tiết mức thấp.
Ý nghĩa:
Dữ liệu và cấu trúc chương trình phải trực quan, phản ánh logic của bài toán và thuật toán.
Giúp chương trình dễ đọc, dễ hiểu, và có cấu trúc tốt.
4. Safety
Khái niệm: Ngôn ngữ nên giúp lập trình viên:
Tìm lỗi dễ dàng: Có các công cụ phát hiện lỗi (như trình biên dịch mạnh mẽ hoặc kiểm tra runtime).
Sửa lỗi dễ dàng: Khi có lỗi, ngôn ngữ cần cung cấp thông tin rõ ràng để sửa chữa.
Thêm tính năng mới một cách an toàn: Cho phép mở rộng chương trình mà không gây ra các lỗi không mong muốn.
Ý nghĩa: Cải thiện khả năng bảo trì (maintainability) và độ tin cậy của chương trình.

Evaluation

1. Readability
2. Writability
3. Reliability
4. Cost

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Influences on Language design

1. Computer Architecture

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Influences on Language design

1. Computer Architecture
2. Programming methodologies:
   - Declarative: on what the computer is to do.
     - Functional: Lisp, ML, Haskell
     - Logic or constraint-based: Prolog
     - Data-flow: Verilog, Simulink
     - Ontology
     - Query-based: SQL, GraphQL.
   - Imperative: on how the computer should do it.
     - Procedural-based: Fortran, ALGOL, COBOL, PL/I, BASIC, **Pascal and C**. tuần tự từ trên xuống
     - *Object-oriented* (OOP).

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Slide này giải thích các yếu tố ảnh hưởng đến thiết kế ngôn ngữ lập trình, cụ thể là từ kiến trúc máy tính và các phương pháp lập trình khác nhau. Dưới đây là diễn giải chi tiết:

1. Computer Architecture

Thiết kế ngôn ngữ lập trình bị ảnh hưởng mạnh mẽ bởi kiến trúc máy tính mà nó sẽ chạy trên.

Ví dụ:

Ngôn ngữ như C được thiết kế để tương thích chặt chẽ với kiến trúc phần cứng của máy tính (như hệ thống thanh ghi, bộ nhớ, CPU).

Assembly language gần như ánh xạ trực tiếp đến các lệnh phần cứng.

Ý nghĩa: Ngôn ngữ thường được tối ưu hóa để tận dụng hiệu quả tài nguyên của phần cứng, đặc biệt là tốc độ và bộ nhớ.

2. Programming Methodologies

Phương pháp luận lập trình (programming paradigms) định hình cách ngôn ngữ được thiết kế và sử dụng.

Các phương pháp này tập trung vào cách diễn đạt chương trình, từ mô tả cái gì cần làm (declarative) đến cách thực hiện (imperative).

Declarative (Khai báo)

Tập trung vào cái gì máy tính cần làm, thay vì cách thực hiện.

Các nhánh chính:

Functional programming (Lập trình hàm):

Xử lý dựa trên các hàm toán học.

Không có trạng thái thay đổi, tập trung vào tính bất biến.

Ví dụ: Lisp, ML, Haskell.

Logic or constraint-based (Lập trình logic/điều kiện):

Xác định các ràng buộc hoặc quy tắc để giải bài toán.

Ví dụ: Prolog (ngôn ngữ logic), thường dùng cho AI.

Data-flow programming (Dòng dữ liệu):

Xây dựng chương trình như một tập hợp các nút xử lý dữ liệu.

Ví dụ: Verilog (mô tả phần cứng), Simulink (mô phỏng).

Query-based programming (Truy vấn dữ liệu):
Lập trình dựa trên việc truy vấn và thao tác dữ liệu.
Ví dụ: SQL (truy vấn cơ sở dữ liệu), GraphQL (truy vấn API).
Imperative (Mệnh lệnh)
Tập trung vào cách để máy tính thực hiện các bước, với các lệnh cụ thể thay đổi trạng thái của chương trình.
Các nhánh chính:
Procedural-based programming (Dựa trên thủ tục):
Chương trình được tổ chức thành các thủ tục (hoặc hàm), với trạng thái được thay đổi qua các lệnh tuần tự.
Ví dụ: Fortran, ALGOL, COBOL, PL/I, BASIC, Pascal, C.
Object-oriented programming (OOP - Lập trình hướng đối tượng):
Tổ chức chương trình thành các đối tượng chứa cả dữ liệu và hành vi (method).
Khuyến khích tái sử dụng mã và mô hình hóa thế giới thực.
Ví dụ: Java, Python, C++, Ruby.
Ý nghĩa của các phương pháp luận lập trình
Declarative: Đơn giản hóa việc diễn đạt các bài toán phức tạp, đặc biệt là các bài toán không yêu cầu mô tả chi tiết cách thực hiện (như AI, truy vấn dữ liệu).
Imperative: Phù hợp với bài toán yêu cầu kiểm soát chi tiết cách thực thi, như tối ưu hóa hiệu năng hoặc xử lý phần cứng.

Language design: Computer Architecture

- Well-known computer architecture: von Neumann.
- Imperative languages, most dominant, because of von Neumann computers.
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
  - Variables model memory cells
  - Assignment statements model writing to memory cell
  - Iteration is efficient

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Language design: Programming methodologies

- 1950s and early 1960s: Simple applications; worry about machine efficiency.
- Late 1960s: Efficiency became important; readability, better control structures.
  - Structured programming.
  - Top-down design and step-wise refinement.
- Late 1970s: Process-oriented to data-oriented
  - data abstraction
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Language Design Trade-Offs

1. Reliability vs. cost of execution
   E.g. Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs.
2. Readability vs. writability
   E.g. APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability.
3. Writability (flexibility) vs. reliability
   E.g. C++ pointers are powerful and very flexible but not reliably used.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

LANGUAGE IMPLEMENTATION

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Implementation methods

Translators

Before a program can be run, it first must be translated into a form in which it can be executed by a computer. This software systems that do this translation are called **translators**.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Implementation methods

Translators

Before a program can be run, it first must be translated into a form in which it can be executed by a computer. This software systems that do this translation are called **translators**.

1. *Compilation*: Programs are entirely translated into machine language and then executed. nhanh
2. *Pure Interpretation*: Programs are translated and executed line-by-line. chạy từng dòng
3. *Hybrid Implementation Systems*: A compromise between compilers and pure interpreters.
4. *Just-in-time Compiler*: A compiler inside an interpreter compiles just hot methods. vòng lặp -> dịch sang mã máy luôn thay vì đi như bình thường

nâng cấp của hybrid

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

1. Compilation (Biên dịch)

Cách hoạt động:
Chương trình được dịch hoàn toàn từ mã nguồn (source code) sang ngôn ngữ máy (machine language) trước khi thực thi.
Kết quả là một file nhị phân (binary), có thể được chạy trực tiếp bởi phần cứng.
Ưu điểm:
Chạy nhanh vì mã đã được tối ưu hóa và dịch hoàn toàn trước khi chạy.
Không cần dịch lại mỗi lần chạy.
Nhược điểm:
Thời gian biên dịch ban đầu lâu.
Nếu có lỗi, cần phải biên dịch lại sau khi sửa lỗi.
Ví dụ: C, C++, Fortran.

2. Pure Interpretation (Diễn giải thuần túy)

Cách hoạt động:
Chương trình không được dịch toàn bộ trước khi chạy.
Mỗi dòng mã được dịch và thực thi từng dòng một, trực tiếp tại thời điểm chạy.
Ưu điểm:
Dễ dàng kiểm tra và sửa lỗi (do bạn có thể nhìn thấy kết quả ngay lập tức).
Phù hợp với các ngôn ngữ scripting hoặc môi trường tương tác.
Nhược điểm:
Hiệu năng chậm, vì mã phải được dịch liên tục trong quá trình thực thi.
Ví dụ: Python (ở chế độ thông dịch), Ruby, JavaScript.

3. Hybrid Implementation Systems (Hệ thống triển khai lai)

Cách hoạt động:
Kết hợp giữa biên dịch và thông dịch.
Chương trình được dịch sang mã trung gian (intermediate code), không phải mã máy.
Sau đó, mã trung gian này được diễn giải để thực thi.
Ưu điểm:
Hiệu quả hơn so với thông dịch thuần túy, vì mã trung gian thường dễ dàng tối ưu hóa.
Dễ dàng port (di chuyển) sang các nền tảng khác nhau.
Nhược điểm:
Không nhanh bằng biên dịch hoàn toàn.
Ví dụ: Java (dịch sang bytecode và thực thi qua JVM), Python (dịch sang bytecode .pyc).

4. Just-in-time (JIT) Compiler

Cách hoạt động:
Đây là một phương pháp nâng cao của thông dịch, trong đó các phần quan trọng của chương trình (thường được gọi là hot methods) được biên dịch tại thời điểm chạy (runtime).
Các phần ít sử dụng vẫn được thông dịch.
Ưu điểm:
Hiệu năng cao vì các phần mã thường xuyên chạy được tối ưu hóa ngay trong runtime.
Linh hoạt hơn biên dịch truyền thống.
Nhược điểm:
Tiêu tốn tài nguyên (CPU và bộ nhớ) trong runtime để biên dịch.
Ví dụ:
Java (HotSpot JVM).
JavaScript (trình duyệt hiện đại như V8 engine của Chrome).

Compilers

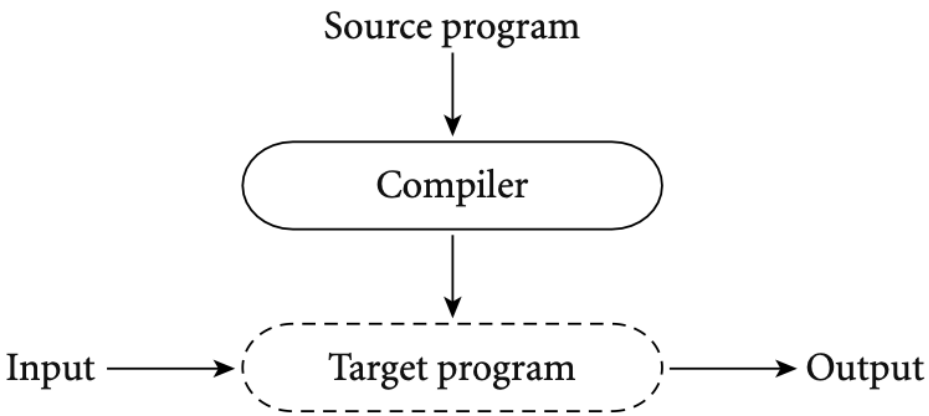


Figure: A compiler

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Compilers

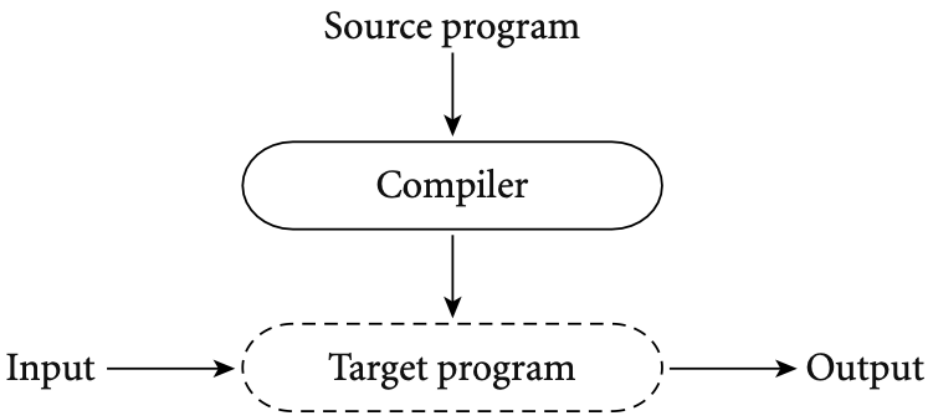


Figure: A compiler

Definition

A compiler is a program that can read a program in one language - the *source* language - and translate it into an equivalent program in another language - the *target* language. If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Pure interpreters

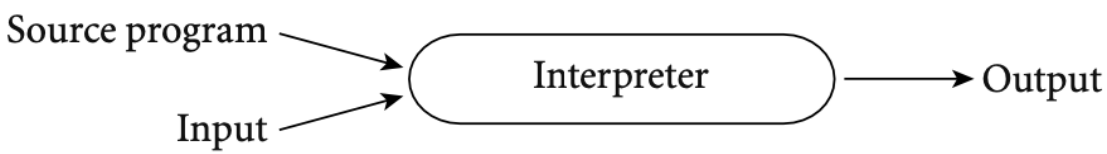


Figure: A pure interpreter

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Pure interpreters

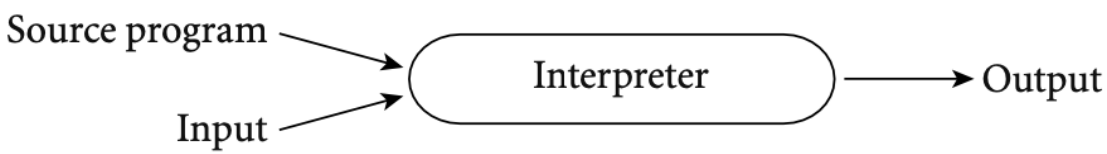


Figure: A pure interpreter

Definition

A pure interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Hybrid systems

- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.
- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Hybrid systems

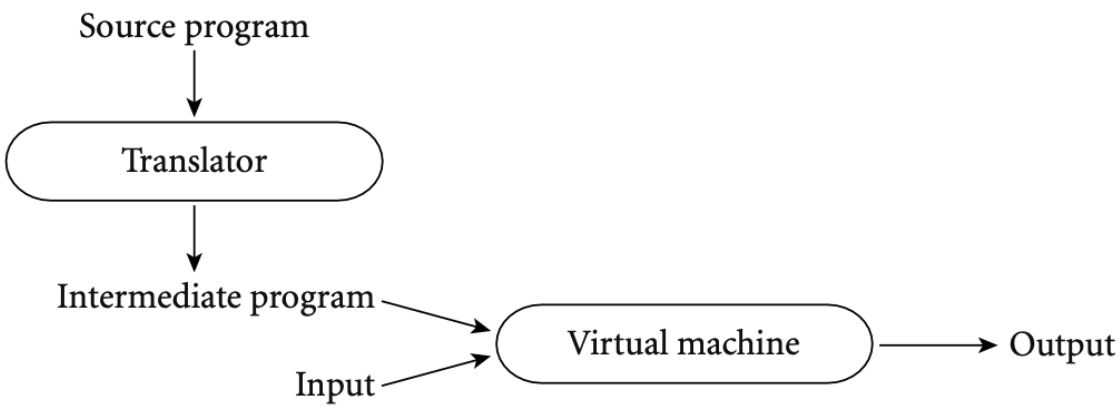


Figure: A hybrid system

Intermediate code compiled on one machine can be *interpreted* on another machine (virtual machine).

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Just-in-time (JIT) compilers

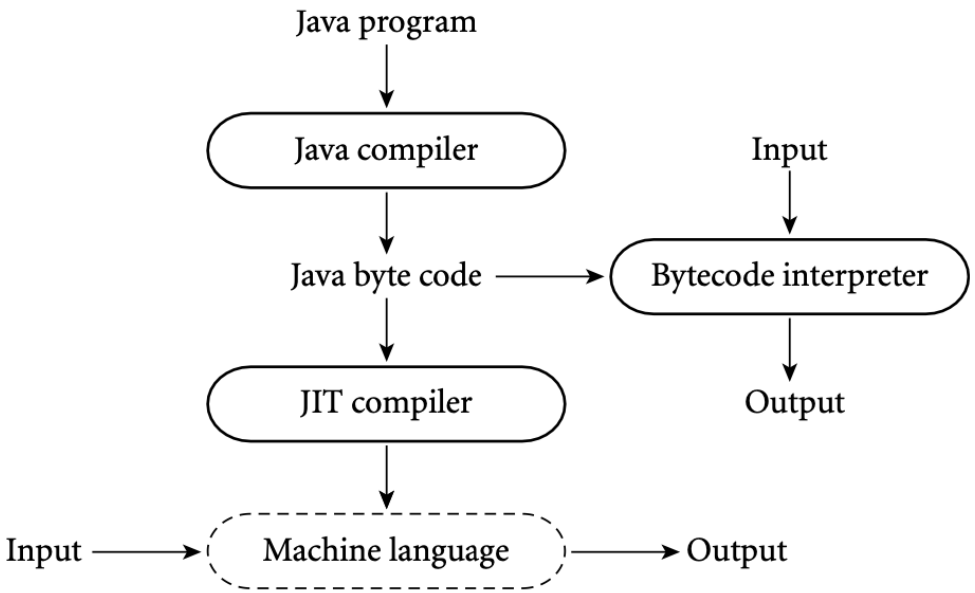


Figure: A just-in-time (JIT) compiler: Case study with Java

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Just-in-time (JIT) compilers

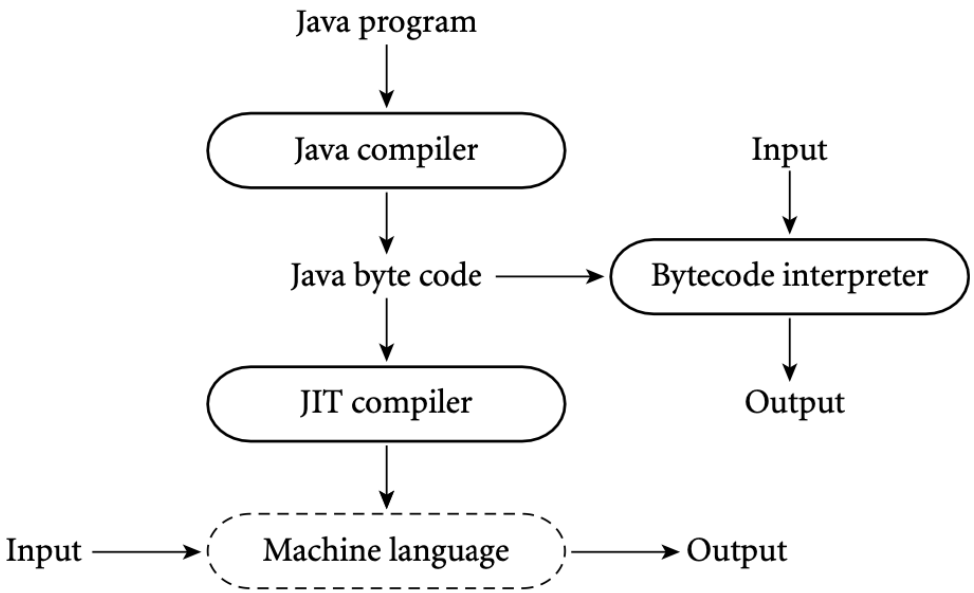


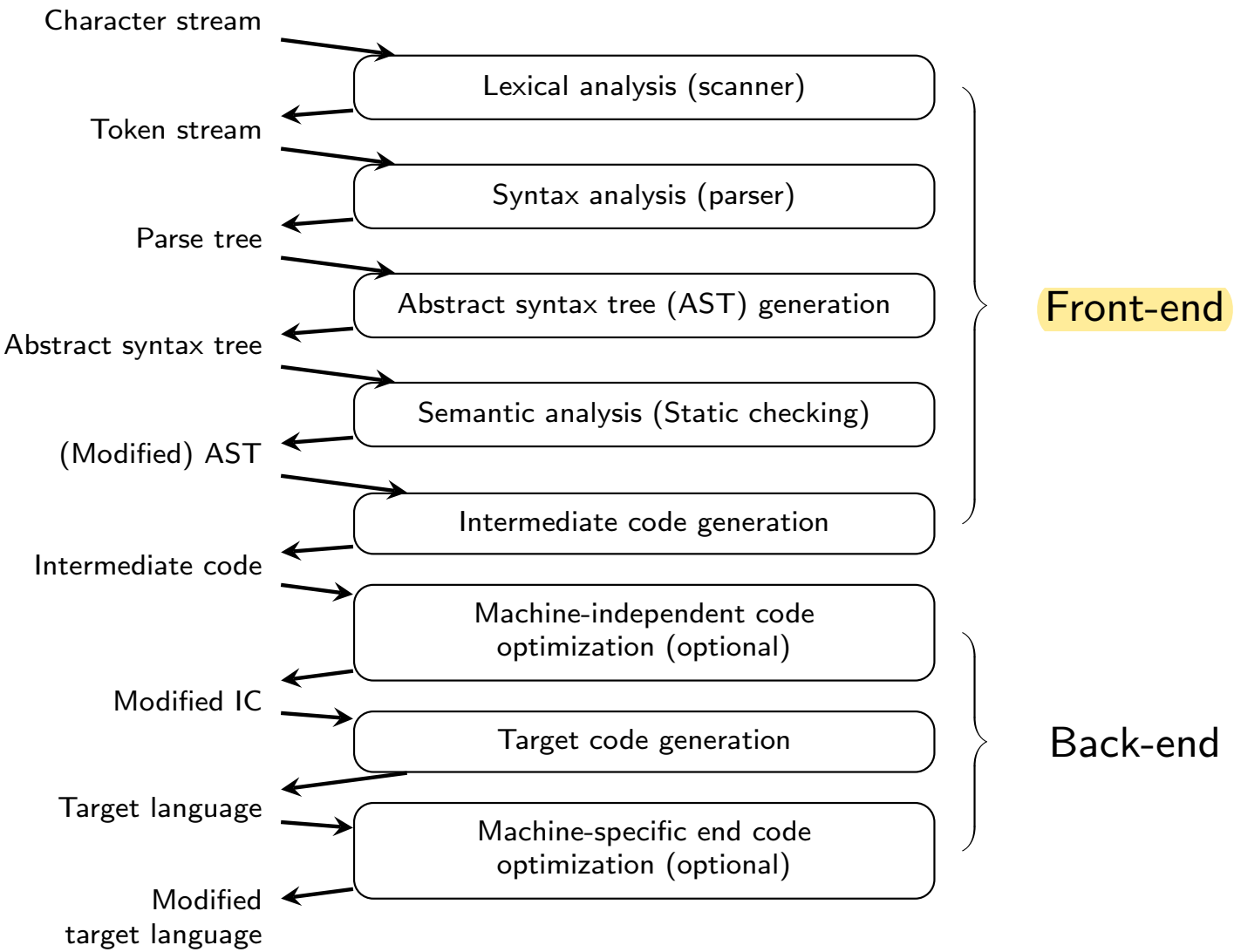
Figure: A just-in-time (JIT) compiler: Case study with Java

- Code starts executing interpreted with no delay.
- Methods that are found commonly executed (hot) are JIT compiled.
- Once compiled code is available, the execution switches to it.

THE STRUCTURE OF A COMPILER

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

An overview of compilation



Lexical Analysis

- The first phase of a compiler is called **lexical analysis** or **scanning**.
- Reading the stream of characters making up the source program and groups the characters into meaningful sequences called **lexemes**.
- Removing extraneous characters like white space.
- Typically, removing comments and tags tokens with line and column numbers.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Lexical Analysis

Example

- Input: `a = b + c * 60;`
- Output:
  1. `(ID, "a")`
  2. `(EQUAL, "=")`
  3. `(ID, "b")`
  4. `(PLUS, "+")`
  5. `(ID, "c")`
  6. `(MUL, "*")`
  7. `(INT, "60")`
  8. `(SEMI, ";")`

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Syntax Analysis

- The second phase of the compiler is **syntax analysis** or **parsing**.
- Using the first components of the tokens produced by the scanner to create a tree-like intermediate representation (*syntax tree*) that depicts the grammatical structure of the token stream.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Syntax Analysis

- The second phase of the compiler is **syntax analysis** or **parsing**.
- Using the first components of the tokens produced by the scanner to create a tree-like intermediate representation (*syntax tree*) that depicts the grammatical structure of the token stream.

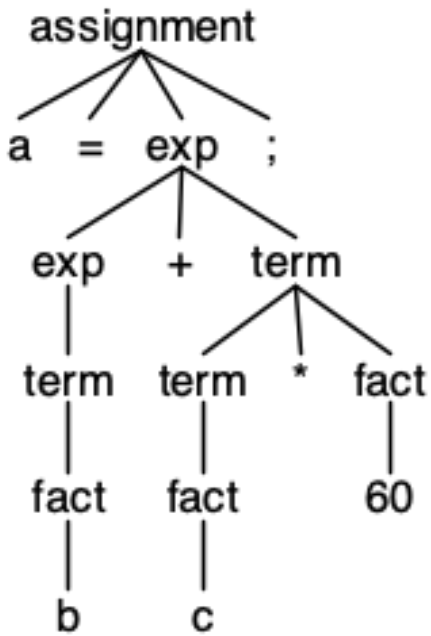


Figure: An example of parse tree

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Abstract syntax tree (AST) generation

- Parse tree demonstrates completely and concretely, how a particular sequence of tokens can be derived under the rules of the grammar. Much of the information in the parse tree is irrelevant to further phases of compilation.
- Removing most of the "artificial" nodes in the tree's interior and returning an **abstract syntax tree** (AST).

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Abstract syntax tree (AST) generation

- Parse tree demonstrates completely and concretely, how a particular sequence of tokens can be derived under the rules of the grammar. Much of the information in the parse tree is irrelevant to further phases of compilation.
- Removing most of the "artificial" nodes in the tree's interior and returning an **abstract syntax tree** (AST).

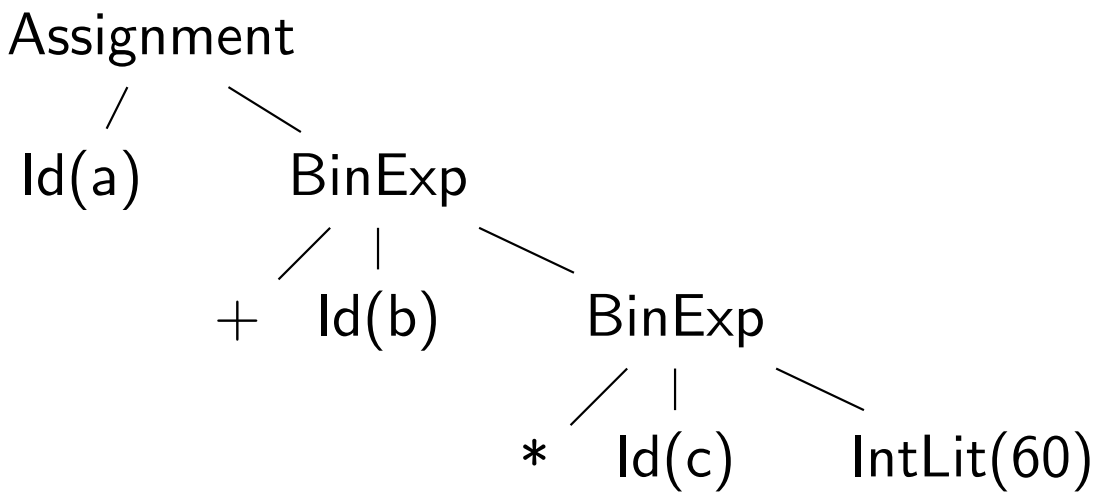


Figure: An example of abstract syntax tree (AST)

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Semantic analysis (Static checking)

Static checks are consistency checks that are done during compilation. Not only do they assure that a program can be compiled successfully, but they also have the potential for catching programming errors early, before a program is run.

- **Syntactic Checking**: There is more to syntax than grammars.

  For example, constraints such as an identifier being declared at most once in a scope, or that a break statement must have an enclosing loop or switch statement, are syntactic, although they are not encoded in, or enforced by, a grammar used for parsing.

- **Type Checking**: The type rules of a language assure that an operator or function is applied to the right number and type of operands.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Intermediate code generation

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which can think of as a program for an abstract machine.
- This intermediate representation should have two important properties:
  1. easy to produce.
  2. easy to translate into the target machine.

Example

Example of intermediate representation

- Three-address code.
- Java bytecode.
- Common Intermediate Language (CIL).

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Applications of Compiler technology

1. Implementation of High-Level Programming Languages.
2. Optimizations for Computer Architectures.
3. Design of New Computer Architectures.
4. Program Translations.
5. Software Productivity Tools.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

Related program

- **Preprocessor**: processes its input data to produce output that is used as input in another program.
- **Assembler**: changes computer instructions into machine code.
- **Linker**: takes one or more object files (generated by a compiler or an assembler) and combines them into a file.
- **Loader**: a major component of an operating system that ensures all necessary programs and libraries are loaded
- **Debugger**: test and debug other programs.
- **Editor**: enable the user to create and edit (source) files.

Dr. Nguyen Hua Phung, MEng. Tran Ngoc Bao Duy

1. Lexical Structure
Lexical structure là các quy tắc cơ bản về cách chương trình được "chia nhỏ" thành các phần tử nhỏ hơn, gọi là tokens. Tokens chính là các đơn vị cơ bản để xây dựng lên chương trình.

2. Tokens là gì?
Tokens là các khối xây dựng cơ bản của chương trình. Mỗi token là một chuỗi ký tự nhỏ nhất có ý nghĩa riêng. Chúng giống như nguyên liệu làm bánh mì: bột, hương vị, nước... Mỗi nguyên liệu có chức năng riêng để tạo nên cái bánh.

Các loại tokens phổ biến:

Keywords: Các từ khóa được định nghĩa sẵn trong ngôn ngữ lập trình, như if, else, while,...
Operators: Các toán tử, như +, -, *, /, ==.
Separators: Các ký hiệu phân cách, như ;, ,, {}, ().
Identifiers: Tên biến, hàm, class... (do người lập trình tự đặt).
Literals: Các giá trị trực tiếp, như số (123), chuỗi ("hello"), ký tự ('a').

3. Lexical Errors
Khi một chuỗi ký tự không hợp lệ, nó sẽ dẫn đến lexical errors. Có 3 loại lỗi thường gặp:

Unclosed string: Chuỗi không được đóng bằng dấu ", ví dụ: "hello. Trong VSCode, phần còn lại của chương trình thường bị highlight màu khác (như màu xanh lá).
Illegal escape in string: Ký tự escape không hợp lệ, ví dụ: "hello\q". (\q không hợp lệ trong hầu hết các ngôn ngữ).
Error token: Không có token nào phù hợp với chuỗi.

4. Quy tắc bắt Tokens

Tokens bắt chuỗi dài nhất có thể. Ví dụ, trong chuỗi 123abc, token 123 (số) được bắt trước, sau đó đến abc.
Nếu hai tokens có độ dài bằng nhau, token nào xuất hiện trước sẽ được ưu tiên.
Cách viết hoa/thường: Nhiều ngôn ngữ có quy tắc về viết hoa, ví dụ: Python yêu cầu các tên class thường bắt đầu bằng chữ cái viết hoa.

5. Cú pháp trong ANTLR

ANTLR là công cụ mạnh để xây dựng lexer và parser. Dưới đây là giải thích từng ký hiệu thường dùng trong ANTLR:

'ký tự'
Khớp chính xác chuỗi ký tự bên trong dấu nháy đơn.
Ví dụ: 'a' chỉ khớp với ký tự a. 'xyz' chỉ khớp với chuỗi xyz.

A B
Yêu cầu A phải xuất hiện trước, ngay sau đó là B.
Ví dụ: 'a' 'b' khớp với ab.

A | B
Chọn một trong hai, A hoặc B.
Ví dụ: 'a' | 'b' khớp với a hoặc b.

'text'
Tương tự 'ký tự', nhưng dành cho chuỗi dài hơn.
Ví dụ: 'Hello' chỉ khớp với Hello.

A?
A có thể xuất hiện hoặc không.
Ví dụ: 'a'? khớp với chuỗi rỗng ("") hoặc a.
A*

A có thể xuất hiện 0 hoặc nhiều lần.
Ví dụ: 'a'* khớp với "", a, aa, aaa,...

A+
A phải xuất hiện ít nhất 1 lần.
Ví dụ: 'a'+ khớp với a, aa, aaa,...

[a-z]
Chọn một ký tự trong khoảng từ a đến z.
Ví dụ: [a-z] khớp với a, b, ..., z.

[A-C]
Chọn một ký tự trong khoảng A đến C.
Ví dụ: [A-C] khớp với A, B, hoặc C.

[0-9]
Chọn một chữ số trong khoảng 0 đến 9.
Ví dụ: [0-9] khớp với 0, 1, ..., 9.

[a-z0-9]
Chọn một ký tự từ a-z hoặc 0-9.
Ví dụ: [a-z0-9] khớp với a, 1,...

[a-zA-Z0-9]

Chọn ký tự từ a-z, A-Z hoặc 0-9.
\n

Ký tự xuống dòng.
\f, \r, \

Ký tự sang trang (\f), quay về đầu dòng (\r), hoặc dấu gạch chéo ngược (\\).
. (dấu chấm)

Khớp với mọi ký tự (trừ xuống dòng trong một số chế độ).
~ [0-9]

Chọn mọi ký tự trừ các chữ số 0-9.
[a] -> skip

Bỏ qua ký tự a. Ví dụ: 'abc' sẽ chỉ phân tích bc.
fragment INT: [0-9]+;

fragment là một đoạn định nghĩa chỉ dùng trong lexer rule khác. Ví dụ, INT mô tả một số nguyên và có thể dùng trong một rule như NUMBER: INT ('.' INT)?;.
{ self.text = self.text[1:-1]; }

Đoạn code Python tùy biến xử lý chuỗi. Ví dụ, self.text[1:-1] sẽ bỏ ký tự đầu và cuối của chuỗi.