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**

INTRODUCTION TO PROGRAMMING LANGUAGES

Programming languages

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

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), ...

Language characteristics

- **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.
- **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.
- ...

Evaluation

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

Influences on Language design

1. Computer Architecture

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**.
     - *Object-oriented* (OOP).

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

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.

LANGUAGE IMPLEMENTATION

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**.

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.
2. *Pure Interpretation*: Programs are translated and executed line-by-line.
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.

Compilers

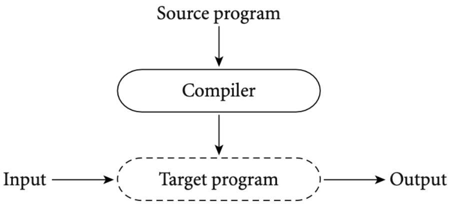


Figure: A compiler

Compilers

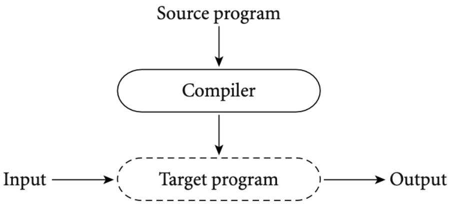


Figure: A compiler

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.

Pure interpreters

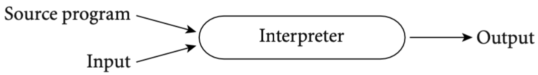


Figure: A pure interpreter

Pure interpreters

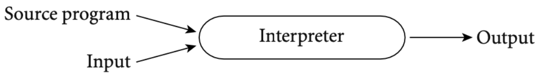


Figure: A pure interpreter

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

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.

Hybrid systems

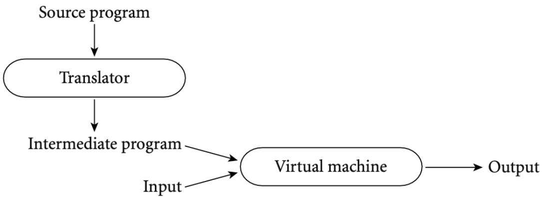


Figure: A hybrid system

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

Just-in-time (JIT) compilers

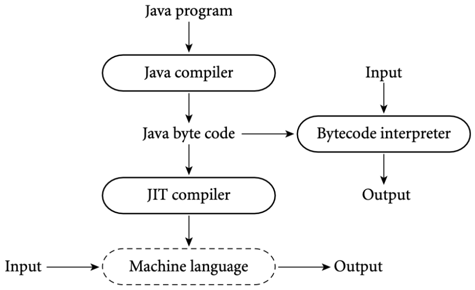


Figure: A just-in-time (JIT) compiler: Case study with Java

Just-in-time (JIT) compilers

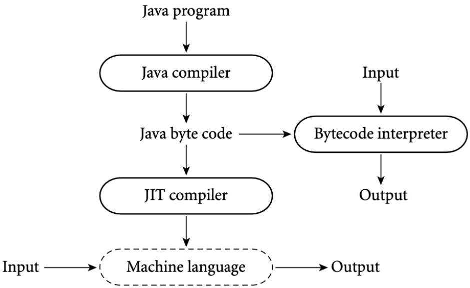


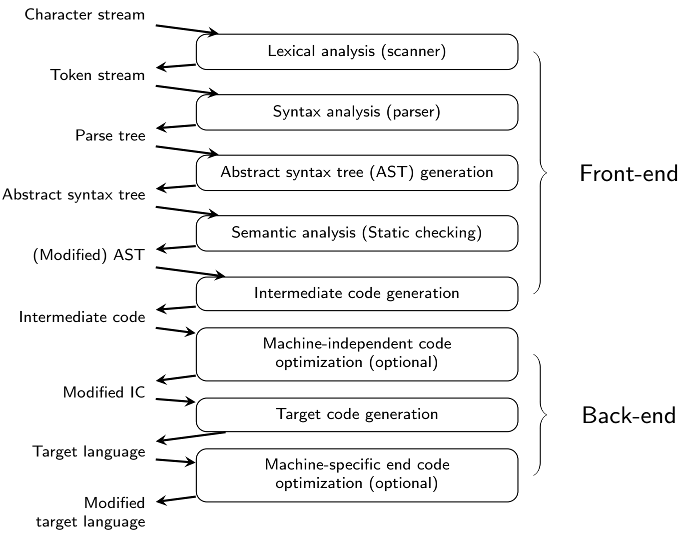
Figure: A just-in-time (JIT) compiler: Case study with Java

- 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.

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, ";")`

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.

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.

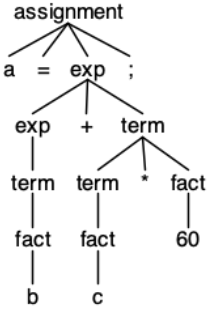


Figure: An example of parse tree

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).

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).

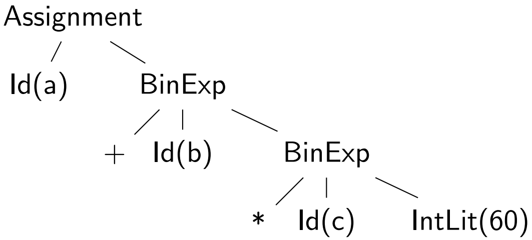


Figure: An example of abstract syntax tree (AST)

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.

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).

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.

THANK YOU.