Compilation and Interpretation

COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

Based in part on slides and notes by S. Olivier, A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts.

Sunday, January 17, 2010
Executing High-Level Languages

A processor can only execute machine code.

- Some can execute several “dialects” (e.g., ARM).

Thus, high-level languages must be translated for execution.

- Ahead of execution: \textit{compilation}.
- Piece-wise during execution: \textit{interpretation}.
Compilation

Ahead of time translation.
- From (high-level) source language to (lower-level) target language.
- Deep inspection of source program as a whole.
- Compiler is unaware of subsequent input.
Compilation

Ahead of time translation:

- From (high-level) source language to (lower-level) target language.
- Deep inspection of source program as a whole.
- Compiler is unaware of subsequent input.

Translation occurs only once, but program is executed many times.
Compilation

Advantages.
- No translation cost at runtime: efficient execution.
- Translation cost amortized over many runs.
- Can distribute program without revealing either source or compiler (commercial software distribution).
- Extensive (and slow) optimizations possible.

Disadvantages.
- Runtime errors hard(er) to diagnose.
- Slow edit-compile-test cycle (large systems can take minutes or hours to compile).
- Source may get lost (de-compilation/reverse engineering is difficult and lossy).
- Good compilers are difficult to build.
- Only limited checks possible at compile time.
Target Language

Target language.
- Often assembly or machine code.
- Can be any language.

Generating C code.
- C code generation is a lot easier.
- C compilers often perform many optimizations.
- Since C is portable, this makes the higher-language portable “for free.”

Examples.
- cfront (first C++ compiler) produced C code.
- ghc (Glasgow Haskell Compiler) can produce either assembly or C code.
Compilation vs. Assembly

What is the fundamental difference between an assembler and a compiler?

Compiler.
- Deep **inspection** of program **semantics**.
- May **reject** syntactically correct **programs** for many reasons.
- E.g., type checking.
- E.g., “return missing”
- Transforms code.
- Optimization.
- **Complex code generation**.
- **Never produces invalid machine code** (only generates code for valid programs).

Assembler.
- **Little/no checks** beyond basic syntax correctness.
- Syntactically correct programs are **not rejected**.
- No transformation (beyond macro expansion).
- Simple translation (**table lookup** of instruction encoding).
- Can produce invalid machine code (if fed bad input).
Translation during execution.

- **Each run** requires on-the-fly translation.
- Interpreter operates on **two inputs**: program and actual input.
- Source program is “configuration” for interpreter to transform actual input.
- Often line/function/instruction interpreted **individually on demand**.
Interpretation

Advantages.
- **Excellent debugging facilities:** source code known when error occurs.
- **Excellent checking:** both input and source are known.
- **Easy to implement.**
- Quick feedback due to rapid edit-test cycle.
- Can be embedded into other applications (for scripting purposes).
- Can generate and evaluate new code at runtime (eval).

Disadvantages.
- Translation occurs many times (redundant work).
- Translation cost occur at runtime: inefficient.
- Protecting intellectual property requires source code obfuscation (which can be unreliable).
- Reasonably fast interpreters are hard to implement.
- Errors in seldom-executed branches may go unnoticed.

Translation during execution.
- Interpretation: on-the-fly translation.
- Requires **two inputs:** program and actual input.
- Source program is "configuration" for interpreter to transform actual input.
- Often line/function/instruction interpreted individually on demand.

**Source Program**

Input $\xrightarrow{}$ **Interpreter** $\xrightarrow{}$ Output
Comparison

Compilation

- Resulting program executes much **faster** than if interpreted.
- Requires code generation and **detailed platform knowledge**.

```
Source Program

Compiler

Target Program

Output
```

Interpretation

- Programming language can be much more **flexible**.
- Can be **portable**.
- **Inefficient**.

```
Source Program

Interpreter

Output
```
Mixing Compilation and Interpretation

Interpreting high-level languages is usually slow.
- First **compile** high-level to low-level byte code.
- **Interpret** much simpler byte code.

Implicit compilation.
- Tool appears as interpreter to user.
- Compilation occurs “behind the scenes.”
- Compilation **only required once** if byte code is cached (e.g., Python).

Explicit compilation.
- **Separate compilation step.**
- User is aware of byte code (e.g., Java).
Mixing Compilation and Interpretation

Advantages.
- Enables “compile once, run everywhere.”
- Low-level interpreter (virtual machine) easier to optimize.
- Optimization during compilation possible.
- Checks like a compiler.
- Implicit: Flexibility like an interpreter.
- Explicit: Source code not revealed.

Disadvantages.
- If byte code is interpreted not as fast as machine code. (Will talk about “just-in-time” compilation when we cover runtime systems.)
- Implicit: Program startup slower due to compilation step.
- Explicit: Byte code is easier to decompile.
The source program is spread out across several files.
Separate Compilation + Linking

Each file is \textit{compiled independently} into “object code” (partial programs).

\begin{itemize}
\item Source File 1
\item \hspace{2cm} Compiler
\item \hspace{2cm} Object Code
\end{itemize}

\begin{itemize}
\item Source File 2
\item \hspace{2cm} Compiler
\item \hspace{2cm} Object Code
\end{itemize}

\begin{itemize}
\item Source File N
\item \hspace{2cm} Compiler
\item \hspace{2cm} Object Code
\end{itemize}

Input \xrightarrow[]{} Target Program \xrightarrow[]{} Output
Some functionality may be provided as an **object code library** (e.g., mathematical functions, system calls).
The linker is used to **merge all program fragments** and **library routines** into the final, executable target program.
Separate Compilation

Advantages.
- Enables collaboration: teams can work on different files in parallel.
- Enables code reuse.
- Enables library/module/unit systems.

Disadvantages.
- Requires intricate build systems for larger projects (e.g., Makefiles, ant, industry employs specialized build engineers).
- Non-trivial bugs can be created if assumptions diverge across compilations (e.g., compiler version, constant definitions).
Separate Compilation + Linking

Source File 1 → Compiler → Object Code
Source File 2 → Compiler → Object Code
... → Compiler → Object Code
Source File N → Compiler → Object Code

Library → Linker

Input → Target Program → Output
Separate Compilation + Interpretation

Source File 1
Compiler
Byte Code
Library
Input
Virtual Machine
Output

Source File 2
Compiler
Byte Code

Source File N
Compiler
Byte Code

Approach can also be combined with **virtual machines** (e.g., see Java).
Time of Error

Terminology: When is an error reported?

*compile-time*

Source File 2

Compiler

Object Code

Library

Target Program

Run-time error

Linker

Link-time error

Also applies to **optimization**, e.g., LLVM supports “link-time optimization.”

Input

Output

Sunday, January 17, 2010
Preprocessing

Source-to-source transformations.

- **Modify source code** before it is passed to the actual compiler or interpreter.
- Macro expansion.
- **Code generation**.
- Remove comments.
- **Conditional compilation** (#ifdef).

Examples.

- Text-based: e.g., `sed`, `perl` (not recommended!)
- External tool: e.g., `m4`.
- Integrated: e.g., C preprocessor.
C: Preprocessing Example

```c
#ifdef ENABLE_INVARIANT_CHECKING
#define INARIANT(x) \  
  if (!x) {fprintf(stderr, “%s failed!
”, #x); exit(1);}  
#else  
#define INARIANT(x) /* nothing to do */
#endif
```

Conditional invariant checking.

- Programmer can specify invariants: e.g., `INARIANT(foo >= 0)`.
- If `ENABLE_INVARIANT_CHECKING` is defined at compile time (using the `-D` switch in gcc), the preprocessor will replace all invariants with if-statements that verify that the assumption holds.
- Otherwise, the preprocessor will remove all invariants from the code before passing the code to the compiler.

Advantages.

- Assumptions made explicit.
- **Simplifies debugging**: turn on all checking with one change.
- **No performance penalty in final release**: checking can be turned off.
C: Preprocessing Example

But keep in mind:

“Finally, it is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his lifejacket when training on dry land, but takes it off as soon as he goes to sea?”


Conditional invariant checking.

➡ Programmer can specify invariants: e.g.,

\[
\text{INVARIANT}(\text{foo} \geq 0)
\]

➡ If `ENABLE_INVARIANT_CHECKING` is defined at compile time (using the `-D` switch in `gcc`), the preprocessor will replace all invariants with if-statements that verify that the assumption holds.

➡ Otherwise, the preprocessor will remove all invariants from the code before passing the code to the compiler.

Advantages.

➡ Assumptions made **explicit**.

➡ **Simplifies debugging**: turn on all checking with one change.

➡ **No performance penalty in final release**: checking can be turned off.
Compilation vs. Preprocessing

The first C++ compiler was called "cfront" and compiled C++ to C.

Since C is (mostly) a subset of C++, should we consider it to be a preprocessor?

No!
Compilation vs. Preprocessing

Why is a pre-processor not the same as a source-to-source compiler?

- Preprocessor: no inspection of semantical correctness.
- A correct compiler does not generate incorrect code.
- Given bad input, most preprocessors will produce code that later fails compilation.
- A preprocessor performs mostly only simple substitutions, without (deeper) understanding of the underlying programming language.

*The C++ compiler `cfront` performs type checking and only generates C programs for C++ programs that pass all semantic tests.*
“Compiled” vs. “Interpreted” Languages

Not a well-defined concept!

Any language can be interpreted.
- Even machine language (e.g., Qemu, virtualization).
- For example, the Tiny C Compiler (tcc) can be used as an interpreter.

Trivial compilation is always possible.
- Include source program as string constant when compiling interpreter.
- Similarly: package byte code and virtual machine together.

However, languages differ in amount of checking that can be done ahead of runtime.
- A language is compilable if “most” checks can be done at compile time.
- This requires careful language design and some restrictions.
- Most languages were designed with either compilation or interpretation in mind.
- Some languages support both (e.g., Lisp, Haskell).
Bootstrapping and Cross-Compilation

Building the first compiler for a new platform.

Many compilers are written in the language that they implement.

- Called a “self-hosting” compiler.
- Virtually all C compilers are written in C.
- The Glasgow Haskell Compiler (ghc) is written in Haskell.
- Lisp dialects are commonly implemented in Lisp.
- This creates a “chicken and egg” problem.

Given a new hardware platform, how do you obtain a compiler?

- From scratch: bootstrapping.
- If you already have another working platform: cross-compilation.
Bootstrapping

Starting from the spec.

First step.
- Write a slow, “quick-n-dirty” interpreter for a subset of the language (as simple as possible) using machine code, assembly, or some low-level language.
- Using the chosen subset, write compiler prototype (version 0) for the chosen subset.
- Use interpreter to run the version-0 compiler for the purpose of compiling itself: we now have a (very limited) compiler that is self-hosting.

Iterative improvements: given a version-N compiler…
- Implement a version-(N+1) compiler using only language features supported by the version-N compiler.
- Use version-N compiler to compile version-(N+1) compiler.
- Repeat, until full language support is complete.
Cross-Compilation

Starting from a host machine.

Notation: ("runs on"\rightarrow"generates machine code for")

On host machine, given a (host → host) compiler.
- Write portable source code for a (any → target) compiler.
- Use (host → host) compiler to compile the (any → target) compiler, which yields a (host → target) cross compiler.
- Use the (host → target) cross compiler to compile the (any → target) a second time.
- This builds a (target → target) self-hosting compiler.
- Copy (target → target) compiler to target machine.
- We now have a self-hosting compiler on the target machine.
Example: Cross-Compilation

Going from Intel x86 to Sun’s SPARC V9.

x86 → x86
V9 → V9
Example: Going from Intel x86 to Sun’s SPARC V9.

Step 1: Write a portable compiler for V9 in C: (any ➜ V9). Name this compiler cv9.
Example: Cross-Compilation

**Step 2:** Given **Gnu C Compiler (gcc)** on our Intel machine, a \((x86 \rightarrow x86)\) compiler, …
Example: Cross-Compilation

Going from Intel x86 to Sun’s SPARC V9.

... use gcc to compile cv9.
Example: Cross-Compilation

Going from Intel x86 to Sun’s SPARC V9.

This yields a (x86 → V9) cross compiler.
Step 3: Use the (x86 \rightarrow V9) cross compiler to compile the (any \rightarrow V9) source code again…
Example: Cross-Compilation

Going from **Intel x86** to Sun’s **SPARC V9**.

- **x86 ➔ x86**
- **V9 ➔ V9**

```

Source

cv9: any ➔ V9

gcc: x86 ➔ x86

cv9: x86 ➔ V9

Target Compiler

cv9: V9 ➔ V9

Cross Compiler

Host Compiler

Input

... this time, we obtain a (V9 ➔ V9) self-hosting compiler!

```
Example: Cross-Compilation

Going from Intel x86 to Sun’s SPARC V9.

\[ x86 \rightarrow x86 \quad V9 \rightarrow V9 \]

Writing a new \( \text{(any } \rightarrow \text{target)} \) compiler/backend for every target can be \textit{prohibitively expensive}. This can be circumvented by using a \textit{virtual machine + bootstrapping}.

In this case, only one \( \text{(any } \rightarrow \text{virtual machine)} \) backend is required, but a (much simpler) \textit{virtual machine} must be \textit{translated by hand}.

See Pascal P-Code example on page 21 in the textbook.
Compilation Phases

Character Stream → Scanner (lexical analysis) → Symbol Table

Token Stream → Parser (syntax analysis) → Symbol Table

Parse Tree → Semantic analysis & intermediate code gen. → Symbol Table

Abstract syntax tree → Machine-independent optimization (optional) → Symbol Table

Modified intermediate form → Target code generation. → Symbol Table

Target (machine) language → Machine-specific optimization (optional) → Symbol Table

Modified target language
Compilation Phases

Front end

- Character Stream
- Token Stream
- Parse Tree
- Scanner (lexical analysis)
- Parser (syntax analysis)
- Semantic analysis & intermediate code gen.

Back end

- Symbol Table
- Abstract syntax tree
- Modified intermediate form
- Target (machine) language
- Modified target language
- Machine-independent optimization (optional)
- Target code generation
- Machine-specific optimization (optional)
Example Program GCD

program gcd(input, output);
var i, j: integer;
begin
  read(i,j); // get i & j from read
  while i<>j do
    if i>j then i := i-j
    else j := j-1;
  writeln(i)
end.
Lexical Analysis

- Recognizes consecutive characters that form a unit and groups them into tokens.

\[
\text{program gcd(input, output)};
\]

- The purpose of the scanner is to simplify the parser by reducing the size of the input.
Token: atomic semantical unit; the smallest unit of input with individual meaning.

- Recognizes consecutive characters that form a unit and groups them into tokens.

The purpose of the scanner is to simplify the parser by reducing the size of the input.
Syntax Analysis

- Parsing discovers the **structure** in the token stream based on a **context-free grammar** and yields a **syntax tree**.

```plaintext
program gcd ( input , output ) ;
```

Token stream:

```
program gcd ( input , output ) ;... 
```

```
token stream:
‣ Parsing discovers the structure in the token stream based on a context-free grammar and yields a syntax tree.
```

```
program
• id(GCD) ( id(INPUT) more_ids ) ; block
  , id(OUTPUT) more_ids
  , empty

Rest of code
```

```
Parser (syntax analysis)
```

Sunday, January 17, 2010
Syntax Analysis

- Parsing discovers the **structure** in the token stream based on a **context-free grammar** and yields a **syntax tree**.

The syntax analysis rejects **all malformed statements**.

### Token stream:
```
program gcd ( input , output ) ; ...
```

### Syntax Tree:
```
program
   └─ gcd
      └─ ( input , output ) ;
```

- The syntax analysis rejects **all malformed statements**.

---

Parser (syntax analysis)
Syntax Analysis

- Parsing discovers the **structure** in the token stream based on a **context-free grammar** and yields a **syntax tree**.

The parse tree is sometimes called a **concrete syntax tree** because it contains all tokens...
Syntax Analysis

- Parsing discovers the **structure** in the token stream based on a **context-free grammar** and yields a **syntax tree**.

...however, much of this information is **extraneous** for the “meaning” of the code (e.g., the only purpose of “;” is to **end a statement**).
Semantic Analysis

- Semantic analysis discovers the **meaning of a program** by creating an **abstract syntax tree** that removes “extraneous” tokens.
- To do this, the analyzer builds & maintains a **symbol table** to map identifiers to information known about it. (i.e., scope, type, internal structure, etc...)
- By using the symbol table, the semantic analyzer can **catch problems not caught by the parser**. For example, it can enforce that
  - identifiers are declared before use, and that
  - subroutine calls provide correct number and type of arguments.
Semantic Analysis

from concrete to abstract syntax tree

<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTEGER</td>
<td>type</td>
</tr>
<tr>
<td>2</td>
<td>TEXTFILE</td>
<td>type</td>
</tr>
<tr>
<td>3</td>
<td>INPUT</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>OUTPUT</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>GCD</td>
<td>program</td>
</tr>
<tr>
<td>6</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>J</td>
<td>1</td>
</tr>
</tbody>
</table>
Semantic Analysis

Not all semantic rules can be checked at compile time.

- Those that can are called **static** semantics of the language.
- Those that cannot are called **dynamic** semantics of the language. For example,
  - Arithmetic operations **do not overflow**.
  - Array subscripts expressions **lie within the bounds of the array**.
Intermediate Code Generation

Intermediate form (IF) generation is done after semantic analysis (if the program passes all checks)

- IFs are often chosen for **machine independence**, **ease of optimization**, or **compactness** (these are somewhat contradictory)
- They often resemble **machine code for some imaginary idealized machine**; e.g. a stack machine, or a machine with arbitrarily many registers
- Many compilers actually move the code through more than one IF.
Target code generation

- Code generation takes the abstract syntax tree and the symbol table to produce machine code.
- **Simple** code follows directly from the abstract syntax tree and symbol table.
- Follows **basic pattern**:
  - Load operands into registers (from **memory**).
  - Compute basic function (e.g., add, div, sub).
  - Store results (to **memory**).
- Other patterns: conditional jumps, subroutine calls.
Optimization

The process so far will produce correct code, but it may not be fast.

- Optimization will transform the code to improve performance without changing its semantics.
- In theory... in practice, compiler bugs often lurk in the optimizer.
- It is easy to overlook corner cases when coming up with optimizations.
- Proper program transformations require rigorous proof of the claimed equivalences.

First aid in case of compiler trouble:
Remove all intermediate files (make clean),
turn off all optimizations (-O0), and try again.
Machine-Independent Optimization

Examples.

➡ Loop unrolling.
   ‣ Enables **hardware parallelism**.
   ‣ Reduces number of times that abort condition is evaluated.

➡ Inlining of (short) subroutines.
   ‣ E.g., getter/setter methods.
   ‣ Reduces **subroutine call overhead**.

➡ Store-load pair elimination.
   ‣ Reduces **unnecessary memory accesses**.

➡ **Jump-coalescing**.
   ‣ Avoid jump to a jump to a jump…

➡ **Escape analysis**.
   ‣ Determine which variables are only updated locally.
Common theme: these overheads are bad on any machine.

Examples:

- Loop unrolling.
  - Enables hardware parallelism.
  - Reduces number of times that abort condition is evaluated.

- Inlining of (short) subroutines.
  - E.g., getter/setter methods.
  - Reduces subroutine call overhead.

- Store-load pair elimination.
  - Reduces unnecessary memory accesses.

- Jump-coalescing.
  - Avoid jump to a jump to a jump…

- Escape analysis.
  - Determine which variables are only updated locally.
Machine-Specific Optimizations

Examples.
- Instruction **scheduling**
  ‣ Overlay **memory latency** with computation.
- Branch-prediction-friendly code layout.
  ‣ **Move failure cases** out of “hot path.”
- Instruction **selection**.
  ‣ Either for **speed** or **size**.
  ‣ xorl %eax, %eax vs. movl $0, %eax.
- **Clever register allocation**.
  ‣ Avoid spill code (minimize storeLoads).
  ‣ This sub-problem by itself is **NP-complete**.
  ‣ Uses graph coloring algorithms.
Machine-Specific Optimizations

Examples.

- Instruction scheduling
  - Overlay memory latency with computation.

- Branch-prediction-friendly code layout.
  - Move failure cases out of “hot path.”

- Instruction selection
  - Either for speed or size.
  - For example, using xorl %eax, %eax instead of movl $0, %eax.

- Clever register allocation
  - Avoid spill code (minimize store/loads).
  - This sub-problem by itself is NP-complete.
  - Uses graph coloring algorithms.

These are all quite complicated to do well…

…and can be completely avoided by compiling to C instead of assembly.

(Unless you are writing a C compiler, that is.)
Summary: Compilation Phases

- **Character Stream**
- **Token Stream**
- **Parser (syntax analysis)**
- **Semantic analysis & intermediate code gen.**
- **Machine-independent optimization (optional)**
- **Target code generation.**
- **Machine-specific optimization (optional)**
- **Symbol Table**

Production compilers tend to be a whole lot more *messy*...
Summary: Compilation and Interpretation

Two fundamental approaches.

- **Compilation.**
  - Resulting program can be efficient.

- **Interpretation.**
  - Can be very flexible.

Implementation approaches.

- **Preprocessing.**
  - Macro expansion and code filtering.

- **Separate compilation.**
  - Divide and conquer…

- **Virtual machines**
  - Simple interpreters are faster.