Chapter 1: Compilation Phases

Aarne Ranta

Slides for the book "Implementing Programming Languages. An Introduction to Compilers and Interpreters", College Publications, 2012.

Compilation Phases

Phases on the way from source code to machine code

Concepts and terminology for later discussions

Compilers vs. interpreters

Low vs. high level languages

Data structures and algorithms in language implementation

From language to binary

Machines manipulate **bits**: 0's and 1's.

Bit sequences used in **binary encoding**.

Information = bit sequences

Binary encoding of integers:

```
0 = 0
1 = 1
2 = 10
3 = 11
4 = 100
```

Binary encoding of letters, via ASCII encoding:

```
A = 65 = 1000001
B = 66 = 1000010
C = 67 = 1000011
```

Thus all **data** manipulated by computers can be expressed by 0's and 1's.

But what about **programs**?

Binary encoding of instructions

E.g. JVM machine language (**Java Virtual Machine**)

Programs are sequences of **bytes** - groups of eight 0's or 1's (there are 256 of them)

A byte can encode a numeric value, but also an **instruction**

Examples: addition and multiplication (of integers)

```
+ =  96 = 0110 0000 = 60
* = 104 = 0110 1000 = 68
```

(The last figure is a **hexadecimal**, where each half-byte is encoded by a base-16 digit that ranges from 0 to F, with A=10, B=11,. . . ,F=15.)

Arithmetic formulas

Simple-minded **infix**:

```
5 + 6 = 0000 0101     0110 0000     0000 0110
```

Actual JVM uses **postfix**

$$5\ +\ 6 \Longrightarrow 5\ 6\ +$$

No need of parentheses:

$$(5\ +\ 6)\ *\ 7 \Longrightarrow 5\ 6\ +\ 7\ *$$

$$5\ +\ (6\ *\ 7) \Longrightarrow 5\ 6\ 7\ *\ +$$

Stack machines

JVM manipulates expressions with a **stack** - the working memory of the machine

Values (byte sequences - 4 bytes in a 32-bit machine) are **pushed** on the stack

The one last pushed is the **top** of the stack

An arithmetic operation such as + (usually called "add") takes (**pops**) the two top-most elements and pushes their sum

Example: compute 5 + 6 (instructions on left, stack on right)

```
bipush 5          ;  5
bipush 6          ;  5 6
iadd              ;  11
```

The instructions are shown as **assembly code**, human-readable names for byte code.

A more complex example: the computation of 5 + (6 * 7)

```
bipush 5  ; 5
bipush 6  ; 5 6
bipush 7  ; 5 6 7
imul      ; 5 42
iadd      ; 47
```

In the end, there's always just one value on the stack

Separating values from instructions

To make it clear that a byte stands for a numeric value, it is prefixed with the instruction `bipush`

`5 + 6` $\Longrightarrow$ `bipush 5 bipush 6 iadd`

To convert this all into binary, we only need the code for the push instruction,

`bipush = 16 = 0001 0000`

Now we can express the entire arithmetic expression as binary:

`5 + 6 = 0001 0000  0000 0101  0001 0000  0000 0110  0110 0000`

Why compilers work

Both data and programs can be expressed as binary code, i.e. by 0's and 1's.

There is a systematic **.translation** from conventional (" user-friendly") expressions to binary code.

Of course we will need more instructions to represent variables, assignments, loops, functions, and other constructs found in programming languages, but the principles are the same as in the simple example above.

How compilers work

1. **Syntactic analysis**: Analyse the expression into an operator F and its operands X and Y.

2. **Syntax-directed translation**: Compile the code for X, followed by the code for Y, followed by the code for F.

Both use **recursion**: they are functions that call themselves on parts of the expression.

Levels of languages

A compiler may be more or less demanding. This depends on the distance of the languages it translates between. (Cf. English to French is easier than English to Japanese.)

In computer languages,

- **High level**: closer to human thought, more difficult to compile
- **Low level**: closer to the machine, easier to compile

This is no value judgement, since low level languages are indispensable!

_____ human

human language

ML Haskell

Lisp Prolog

C++ Java

C

assembler

machine language

_____ machine

Some programming languages from the highest to the lowest level.

Both humans and machines are needed to make computers work in the way we are used to.

Some people might claim that only the lowest level of binary code is necessary, because humans can be trained to write it.

But humans could never write very sophisticated programs by using machine code only - they could just not keep the millions of bytes needed in their heads.

Therefore, it is usually much more productive to write high-level code and let a compiler produce the binary.

The history of programming languages shows progress from lower to higher levels.

Programmers can be more productive when writing in high-level languages.

However, raising the level implies a challenge to compiler writers.

Thus the evolution of programming languages goes hand in hand with developments in compiler technology.

It has of course also helped that the machines have become more powerful:

- the computers of the 1960's could not have run the compilers of the 2010's
- it is harder to write compilers that produce efficient code than ones that waste some

A rough history of programming languages

- 1940's: connecting wires to represent 0's and 1's
- 1950's: assemblers, macro assemblers, **Fortran**, **COBOL**, **Lisp**
- 1960's: **ALGOL**, **BCPL** ($\rightarrow$ **B** $\rightarrow$ **C**), **SIMULA**
- 1970's: **Smalltalk**, **Prolog**, **ML**
- 1980's: **C++**, **Perl**, **Python**
- 1990's: **Haskell**, **Java**

A compiler reverses the history of programming languages: from a "1960's" source language:

```
5 + 6 * 7
```

to a "1950's" assembly language

```
bipush 5 bipush 6 bipush 7 imul iadd
```

to a "1940's" machine language

```
0001 0000  0000 0101  0001 0000  0000 0110
0001 0000  0000 0111  0110 1000  0110 0000
```

The second step is very easy: look up the binary codes for each assembly instruction and put them together in the same order.

The level of **assembly** is often regarded as separate from compilation proper.

Compilation vs. interpretation

A compiler is a program that **translates** code to some other code. It does execute the program.

An **interpreter** does not translate, but it **executes** the program.

A source language expression,

```
5 + 6 * 7
```

is by an interpreter turned to its value,

```
47
```

Combinations

- **C** is usually compiled to machine code by GCC.
- **Java** is usually compiled to JVM bytecode by Javac, and this bytecode is usually interpreted, although parts of it can be compiled to machine code by **JIT** (**just in time compilation**).
- **JavaScript** is interpreted in web browsers.
- Unix shell scripts are interpreted by the shell.
- **Haskell** programs are either compiled to machine code using GHC, or to bytecode interpreted in Hugs or GHCI.

Notice: Java is not an "interpreted language" - but JVM is!

Trade-offs

Advantages of interpretation:

- faster to get going
- easier to implement
- portable to different machines

Advantages of compilation:

- if to machine code: the resulting code is faster to execute
- if to machine-independent target code: the resulting code is easier to interpret than the source code

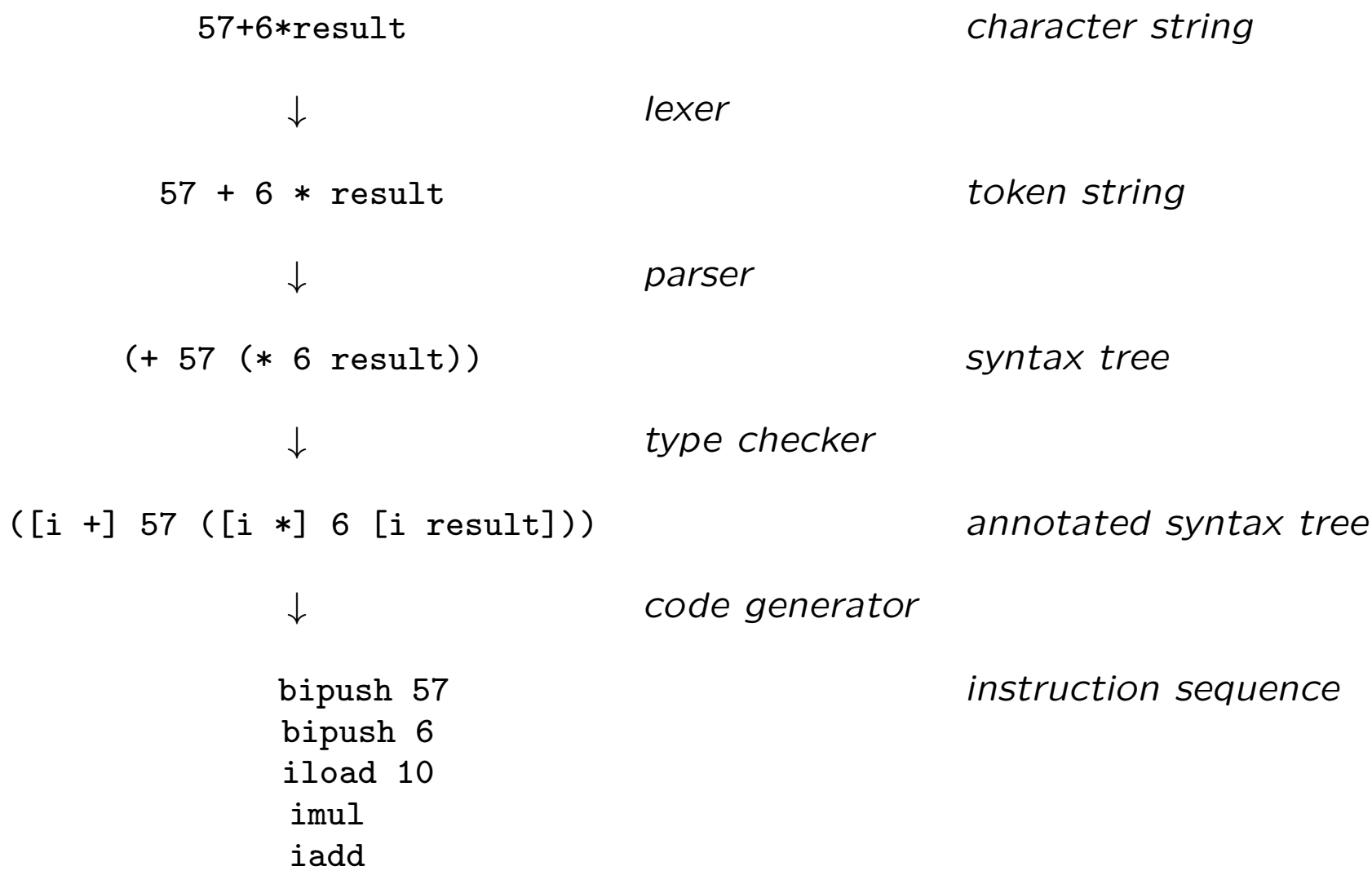
JIT is blurring the distinction, and so do virtual machines with actual machine language instruction sets, such as VMWare and Parallels.

Compilation phases

A compiler is a complex program, which should be divided to smaller components.

These components typically address different **compilation phases** - parts of a pipeline, which transform the code from one format to another.

The following diagram shows the main compiler phases and how a piece of source code travels through them.



Compilation phases from Java source code to JVM assembly code

- The **lexer** reads a string of **characters** and chops it into **tokens**.
- The **parser** reads a string of tokens and groups it into a **syntax tree**.
- The **type checker** finds out the **type** of each part of the syntax tree and returns an **annotated syntax tree**.
- The **code generator** converts the annotated syntax tree into a list of target code instructions.

The difference between compilers and interpreters is just in the last phase: interpreters don't generate new code, but execute the old code.

Compilation errors

Each compiler phase can fail with a characteristic errors:

- **Lexer errors**, e.g. unclosed quote,

  ```
  "hello
  ```

- **Parse errors**, e.g. mismatched parentheses,

  ```
  (4 * (y + 5) - 12))
  ```

- **Type errors**, e.g. the application of a function to an argument of wrong kind,

  ```
  sort(45)
  ```

Front end and back end

Front end: **analysis**, i.e. inspects the program: lexer, parser, type checker.

Back end: **synthesis**, i.e. constructs something new: code generator.

Errors on later phases than type checking are usually not supported; cf. Robin Milner (the creator of **ML**): "well-typed programs cannot go wrong".

Compile time vs. run time

Compilers can only find **compile time** errors.

Error detection at **run time** needs **debugging**.

As compilation is automatic and debugging is manual, efforts are made to find more errors at compile time.

Examples of run-time errors

Array index out of bounds, if the index is a variable that gets its value at run time.

Binding analysis of variables:

```
int main () {
  int x ;
  if (readInt()) x = 1 ;
  printf("%d",x) ;
}
```

It is not decidable at compile time if `x` has a value.

More compilation phases

Desugaring/normalization: remove **syntactic sugar**,

`int i, j ;` $\Longrightarrow$ `int i ; int j ;`

This can be done early (which can result to worse error messages).

Optimizations: improve the code in some respect. This can be done

- **source code optimization**, precomputing values known at compile time:
  `i = 5 + 6 * 7 ;` $\Longrightarrow$ `i = 47 ;`
- **target code optimization**, replacing instructions with cheaper ones:
  `bipush 31 bipush 31` $\Longrightarrow$ `bipush 31 dup`

(The gain is that the `dup` instruction is just one byte, whereas `bipush 31` is two bytes.)

Modern compilers may have dozens of phases, often performed on the level of **intermediate code**, so that the work can be reused for different source and target languges..

Theory and practice

phase	**theory**
lexer	finite automata
parser	context-free grammars
type checker	type systems
interpreter	operational semantics
code generator	compilation schemes

A theory

- provides **declarative notations** that support different implementations
- enables **reasoning**, e.g. checking if every program can be compiled in a unique way

Syntax-directed translation is a common name for the techniques used in type checkers, interpreters, and code generators alike.