Compilers

Stanford | Online

Alex Aiken

LaTeXed by Simon Hua

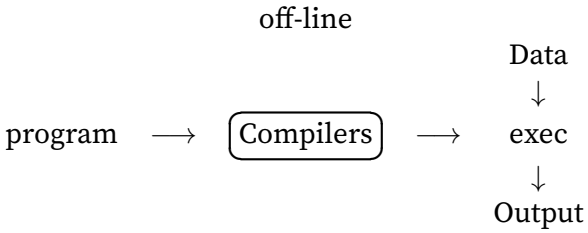
Contents

CHAPTER 1

Introduction

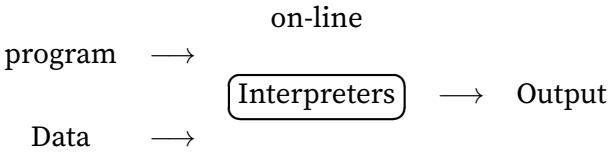
1.1 Introduction

- Compilers



> 1954 IBM develops the 704
> software > hardware
> "Speedcoding"
>
> - 10-20x slower
> - 300 bytes = 30% memory

- Interpreters



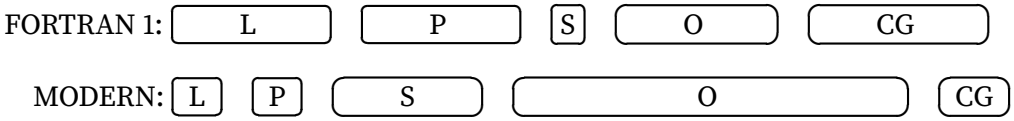
> FORTRAN 1(Formulas Translated)
> 1954-1957
> 1958 50% program in FORTRAN 1

1.2 Structure of Compiler

5 phases

1. Lexical Analysis: divides program text into "words" or "tokens".

2. Parsing: diagramming sentences.
3. Semantic Analysis: try to understand "meaning". (hard)
   Compilers perform limited senmantic analysis to catch inconsistencies.
   → Programming Languages define strict rules to avoid such ambiguities.
4. Optimization: Antomatically modify prgrams so that they
   → Run faster
   → Use less space
   → Reduce power consumption...
5. Code Generation(Code Gen)
   → Produces assembly code.(usually)
   → A translation int another language.(Analgous to human translation)



1.3 The Economy of Programming Languages

Question

1. Why are there so many Programming Languages?
   Application domians have distinctive / conflicting needs.

Scientific Computing	→ Good Float Points → Good Arrays → Parallelism	FORTRAN
Business Application	→ Persistence → Report Generation → Data Analysis	SQL
Scientific Computing	→ Control of Resources → Real TimeConstraints	C/C++

2. Why are there new programming languages?
   Claim: **Programmer training** is the dominant cost for a Programming Languages

   (a) widely-used Languages are slow to change.
   (b) Easy to start a new language. ⟶ Productivity > Training Cost
   (c) Languages adopted to fill a void.

   New languages tend to looks like old languages because of the Claim
   → Reducing programming training, like Java vs C++.

3. What is a good programming languages?
   There is no universally accepted metric for language design.

CHAPTER 2

The Cool Programming Language

2.1 Cool Overview

COOL (Classroom Object Oriented Language)
Designed to be implemented in a short time and small enough for a one term project.
Cool $\longrightarrow$ MIPS(spim) $\longrightarrow$ Assembly Language

2.2 Cool Examples

1. example 1

```
class Main inherits IO {
    main() : Object {
        out_string("Hello, world!\n")
    };
};
```

2. exmaple 2

```
class Main inherits IO {
    main(): Object {{
        out_string("Enter an integer greater-than or equal-to 0: ");

        let input: Int <- in_int() in
            if input < 0 then
                out_string("ERROR: Number must be greater-than or equal-to 0\n")
            else {
                out_string("The factorial of ").out_int(input);
                out_string(" is ").out_int(factorial(input));
                out_string("\n");
            }
            fi;
    }};

    factorial(num: Int): Int {
      if num = 0 then 1 else num * factorial(num - 1) fi
    };
};
```

CHAPTER 3

Lexical Analysis

3.1 Lexical Analysis Part 1

Token class
Token classes correspond to sets of strings. Classify program substrings according to role(token class).

- Identifier: strings of letters or digits, starting with a letter.
- Integer/Number: a non-empty string of digits.
- Keyword: "else" or "if" or "begin" or ...
- Whitespace: a non-empty sequence of blanks, newlines, and tabs.
- Operater: like "==", "<" or ">" in cpp.
- single character token(punctuatin mark): "(", ")", ";", "=".

Communicate tokens to the parser

$$\begin{array}{c}\text{string}\\ \text{foo = 42}\end{array} \longrightarrow \boxed{\text{LA}} \longrightarrow \underbrace{\langle \text{class, string} \rangle}_{token} \quad \boxed{\text{P}}$$

For exmaple:

$$\langle \text{Id, "foo"} \rangle \quad \langle \text{Op, "="} \rangle \quad \langle \text{Int, "42"} \rangle$$

> An implementation must do two things
>
> 1. Recognize substrings corresponding to tokens
>   → The lexemes
> 2. Identify the token class of each lexeme

$$\underbrace{\langle \text{token class, lexeme} \rangle}_{\text{token}}$$