



Compilers

Stanford | Online



Alex Aiken

Contents

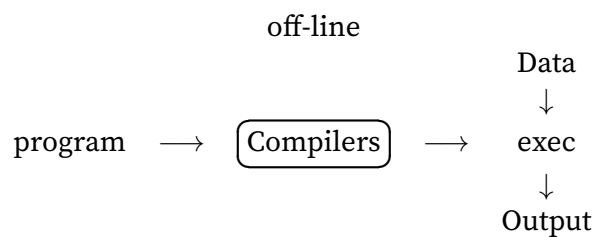
1	Introduction	2
1.1	Introduction	2
1.2	Structure of Compiler	2
1.3	The Economy of Programming Languages	3
2	The Cool Programming Language	4
2.1	Cool Overview	4
2.2	Cool Examples	4
3	Lexical Analysis	5
3.1	Lexical Analysis	5
3.2	Lexical Analysis Examples	6
3.3	Regular Languages	6

CHAPTER 1

Introduction

1.1 Introduction

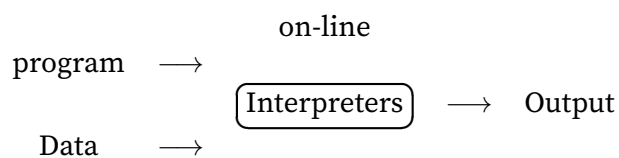
- Compilers



1954 IBM develops the 704
software > hardware
"Speedcoding"

- 10-20x slower
- 300 bytes = 30% memory

- Interpreters



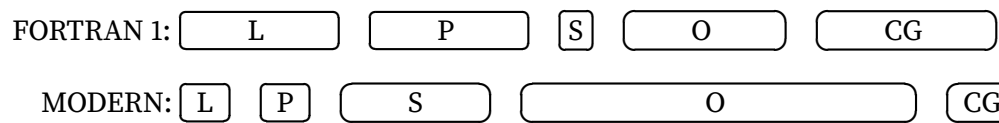
FORTRAN 1(Formulas Translated)
1954-1957
1958 50% program in FORTRAN 1

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 semantic analysis to catch inconsistencies.
→ Programming Languages define strict rules to avoid such ambiguities.
4. **Optimization**: Automatically modify programs so that they
 - Run faster
 - Use less space
 - Reduce power consumption...
5. **Code Generation (Code Gen)**
 - Produces assembly code. (usually)
 - A translation into another language. (Analogous to human translation)



1.3 The Economy of Programming Languages

Question

1. Why are there so many Programming Languages?

Application domains 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 Time Constraints	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 look like old languages because of the Claim
→ Reducing programming training, like Java vs C++.

3. What is a good programming language?

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 \rightarrow MIPS(spim) \rightarrow 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 o: ");  
  
    let input: Int <- in_int() in  
    if input < o then  
      out_string("ERROR: Number must be greater-than or equal-to o\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 = o then 1 else num * factorial(num - 1) fi  
  };  
};
```

CHAPTER 3

Lexical Analysis

3.1 Lexical Analysis

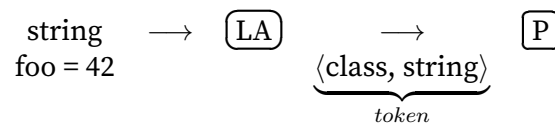
Token class

Token classes correspond to sets of strings.

- **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.
- **Operator**: like "=", "<" or ">" in cpp.
- **single character token(punctuatin mark)**: "(", ")", ";", "=".

Classify program substrings according to role(token class).

Communicate tokens to the parser



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

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

3.2 Lexical Analysis Examples

1. FORTRAN rule: white space is insignificant.

(a) `VAR1` is the same as `VA R1` .

(b) `DO 5 I = 1,25`

This is a loop and `I` is a variable from 1 to 25. The number 5 means execute the following 5 lines of the codes in the loop.

(c) `DO 5 I = 1.25`

It's exactly the same as `DO5I = 1.25` .

(a) The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time.

(b) "Lookahead" may be required to decide where one token ends and the next token begins.

2. PL/1(Programming language 1) keywords are not reserved

(a) `DECLARE(ARG 1, ..., ARG N)`

Is `DECLARE` is a keyword or an array reference? Unbounded lookahead.

3. • C++ template syntax

`Foo<Bar>`

• C++ template syntax

`cin >> var;`

`Foo<Bar<Bazz>>`

The Problem is how the compiler will deal with the code `>>` .

Summary

- The goal of lexical analysis is to
 1. Partition the input string into lexemes.
 2. Identify the token of each lexeme.
- Left-to-right scan \Rightarrow lookahead sometimes required.

3.3 Regular Languages

The **Lexical structure** equals to **token class**. We must say what set of strings is in a token class \rightarrow Use regular languages. Then, we need to define the regular expression.

Define. Regular Expression

- Single Character

`'c' = {"c"}`

- Epsilon

$\epsilon = \{""\} \neq \phi$

- Union

$$A + B = \{a \mid a \in A\} \cup \{b \mid b \in B\}$$

- Concatenation

$$AB = \{ab \mid a \in A \wedge b \in B\}$$

- Iteration

$$A^* = \bigcup_{i \geq 0} A^i$$

$$\text{e.g. } A^0 = \epsilon, A^i = \underbrace{A \dots A}_{i \text{ times}}$$

E.g. $\Sigma = 0, 1$

1. $\underbrace{1^*}_{=1^*+1} = \bigcup_{i \geq 0} 1^i = "" + 1 + 11 + \underbrace{11 \dots 11}_{i \text{ times}} = \text{all strings of 1's}$
2. $\underbrace{(1+0)1}_{=11+01} = ab \mid a \in 1+0 \wedge b \in 1 = 11, 01$
3. $0^* + 1^* = 0^i \mid i \geq 0 \cup 1^i \mid i \geq 0$
4. $(0+1)^* = \bigcup_{i \geq 0} (0+1)^i = "", (0+1), (0+1)(0+1), \underbrace{(0+1) \dots (0+1)}_{i \text{ times}}$
 = all strings of 0's and 1's
 = Σ^*

Question

$$(0+1)^*1(0+1)^*, \Sigma = 0, 1 \Rightarrow (0+1) \dots (0+1)1(0+1) \dots (0+1)$$

$$\rightarrow (01+11)^*(0+1)^* \quad \times$$

No guarantee of the first " $(0+1)^*$ ", like "10" is not allowed by " $(01+11)^*$ ".

$$\rightarrow (0+1)^*(10+11+1)(0+1)^* \quad \checkmark$$

$(10+11+1)$ guarantees the "1", $\Rightarrow 10+11 \Leftrightarrow "1" + "0"$ or $"1" + "1" = 1(1+0)^*$.

$$\rightarrow (1+0)^*1(1+0)^* \quad \checkmark$$

It's the same as $(0+1)^*1(0+1)^*$.

$$\rightarrow (0+1)^*(0+1)(0+1)^* \quad \times$$

It's the same as $(0+1)^*$, and no guarantee the middle "1".

Summary

1. **Regular expression**(syntax) specify **regular languages**(set of things)
2. Five constructs
 - (a) Two base cases empty and 1-character strings
 - (b) Three compound expressions union, concatenation, iteration