A Tour of C++

ksolomon

June 5, 2024

Abstract

Notes on Compilers Course

1 Introduction

1.1 Introduction

Compilers make generic executable which can run offline and process variable inputs to generate outputs. Interpreters run on data and translation unit in parallel to generate the output. Interpreters are much slower and any change to either data or program requires reinterpretation. As a result interpreters have lost favor. Fortran (Formula translation @ John Backus) was first major breakthrough in coding language (1950s) which inspired more research into computer science and became first compiler. Modern compilers still have similar data flow to Fortran.

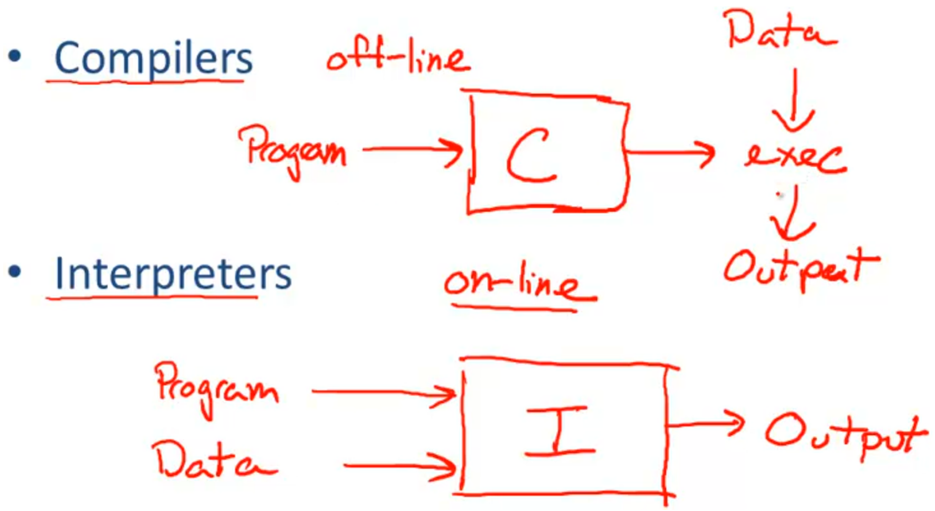


Figure 1: Compilers vs. Interpreters

Steps of the Compiler:

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

1.2 Structure of Compiler

Lexical Analysis: is the process of breaking a program down into a series of tokens that are recognizable by the compiler. definition

Parsing: is the process of converting the tokens into an abstract syntax tree.

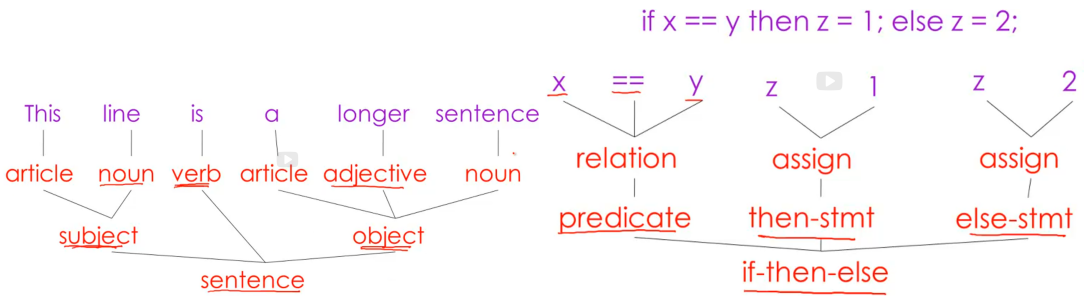


Figure 2: Example of parsing English vs Code

Semantic Analysis: is the process of checking the correctness of the program. In an ideal world this would fully understand the meaning of the program, however this is currently too hard for modern compilers. Ambiguity is a big issue in semantic analysis, so programming languages define strict syntactical rules to compile.

Optimization: is the process of reducing the size of the program to make it run faster, use less memory/power/networking/api calls, etc.

Code Generation: is the process of generating the actual machine code from the abstract syntax tree.

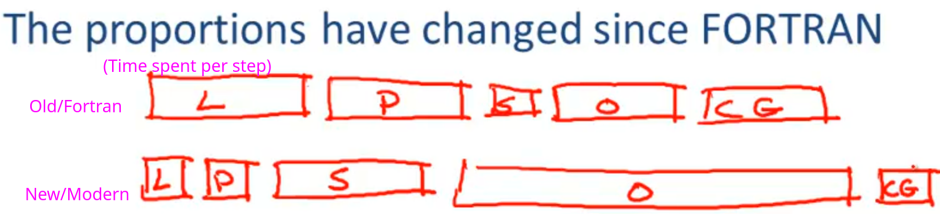


Figure 3: Ratio of time spent in each step of compilers. In modern: small means well-understood/developed tools.

1.3 Economy of Programming Languages

Why so many languages? Application domains have different/conflicting needs. E.g. Scientific computing: good FP precision, array/matrix operations, parallelism. vs Business: persistence, reporting, data analysis.

Why new languages? Programmer training is the dominant cost for a programming language. Widely-used languages are slow to change. Cost to start a new language is low (since no responsibility to existing users). If productivity of new language is higher than maintenance of existing languages and training cost is low then they're incentivized to switch. New languages arise to meet gap in existing, more widely adopted, languages. New languages tend to look like old languages, since that can reduce the training cost (e.g. Java vs C++).

What makes a good language? There is no universally accepted metric to judge a language. Can't optimize what isn't defined.

2 Cool

2.1 Overview

COOL: Classroom Object-Oriented Language. Probably the only language with more compilers written than programs. Designed to be implemented quickly. Gives a taste of:

- Abstraction
- Static Typing
- Inheritance
- Memory Management

Compile Cool into MIPS asm.

In 5 assignments:

1. Write a Cool Program
2. Lexical Analysis
3. Parse
4. Semantic Analysis
5. Code generation
6. Optional: Optimization

3 Lexical Analysis

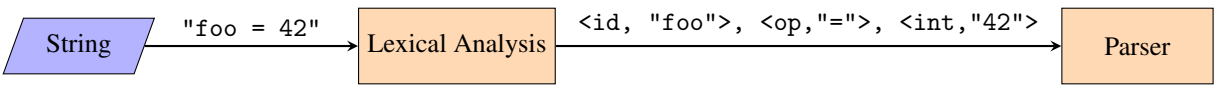
3.1 Intro

Goal

- Partition input string into lexemes
- Identify token of each lexeme

Token Class Correspond to sets of strings.

- English: Noun, Verb, Adjective, etc.
- Programming Token Classes
  - Identifiers: `abc123`
  - Integers: `123`
  - Keywords: `if, else`
  - Whitespace: `space, newline, tab, vertical tab, formfeed, etc.`
  - Punctuation:
    * `(`
    * `)`
    * `;`
    * `=`
  - Operator: `+ - * / ==`



Each identified token is a ”lexeme”.

3.2 Examples

Fortran Whitespace is ignored. So `VAR1` is the same thing as `VA R1`. This rule was implemented because in punchcard days it was very easy to accidentally add whitespace. Only way to distinguish between below two examples is to implement "lookahead" for punctuation. Lookahead creates complication in compiler logic. It is always needed, but it is a good idea to define upper bounds on complexity.

```
DO 5 I = 1,25 // do-loop where loop extends from DO down to statement 5
DO 5 I = 1.25 // assignment of DO5I=1.25
```

PL/1 Keywords are not reserved. Example legal code: `IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN` This made lexical analysis much harder.

```
DECLARE (ARG1, ..., ARGN) // DECLARE can be keyword or array reference in given context.
    This requires unbounded lookahead since you'd have to scan ahead of ARGN in order to
    actually know


Foo<Bar>; // Template Syntax
cin >> var; // Stream Syntax
```

This creates a conflict with nested templates (`Foo<Bar<Buzz>>`) and older compliers would often require extra whitespace between last two `> >` in order to not treat as syntax error.

3.3 Regular Languages

Regular Languages (syntax) specify regular languages (set of legal strings).

Regular Expression Constructs

- single character: (A)
- ε: (the empty string "")
- union: (A+B)
- concatenation: (AB)
- iteration: (A*) where $(A^0 = \varepsilon)$

Given a language $(\Sigma) = 0, 1$ this can be represented as the following sets of strings:

- $1^* = \bigcup_{i>=0} 1^i$ = 1, 11, 111, ...
- $(1+0)1 = \{ab | a \in 1+0 \wedge b \in 1\} = \{11, 01\}$
- $0^* + 1^* = \{0^i | i >= 0\} \cup \{1^i | i >= 0\}$
- $(0+1)^* = \bigcup_{i>=0}(0+1)^i = \Sigma^*$ (all strings of 0's and 1's aka the superset of the language)

3.4 Formal Languages

Play a large role in theoretical computer science.

Formal Language Let Σ be a set of characters (alphabet). Then the language L is a formal language over Σ if it is a subset of $\Sigma*$.

- Alphabet = English characters, Language = English sentences (not exactly formal since not all sentences are agreed upon).
- Alphabet = ASCII, Language = C programs (this IS a formal language).

Meaning Function Let L be a formal language. Then the meaning function $M : L \rightarrow \Sigma^*$ is a function that maps each string in L to a string in Σ^*. Aka maps syntax to semantics (meaning).

4 Lexical Specifications

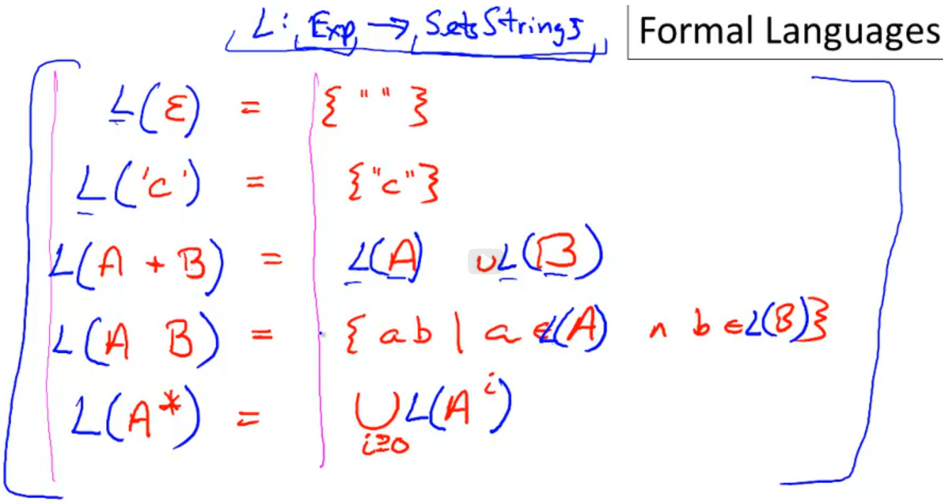


Figure 4: L (Meaning Function) is a map from Expressions to Sets of Strings

Purpose of using a Meaning function is to:

- Decouple Syntax from Semantics
- change syntax without changing semantics (optimize)
- Expressions do not have a 1:1 relationship with meanings. Generally N:1 (multiple ways to say/write the same thing)

Regular Expressions

- Keyword: ”if” or ”else” or ”then” or ... $'if' + 'else' + 'then'$
- Integer: non-empty string of digits: $'0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'$, however can't be empty string so instead of digit $digit^*$ we can write this as $digit^+$
- Identifier: strings of letters or digits starting with a letter, where letter = [a-zA-Z]. $letter(letter + digit)^*$
- Whitespace: non-empty sequence of $(blanks/newlines/tabs)^+$

anyone1@cs.stanford.edu: $letter(letter + digit)^* + @ + letter^+ + . + letter^+ + . + letter^+$

PASCAL

- digits = $digit^+$
- optFraction = $('.'digits) + \varepsilon$ where ε means empty set is valid
  - ('.'digits)?
- optExponent = $('E'('+' + '-' + \varepsilon)digits) + \varepsilon$
  - ('E'('+'+'-')?digits)?
- number = $digits + optFraction + optExponent$

Question mark (?) means the aforementioned component is optional

Given string s and regular expression r, is it in the language (L) of a regular expression? $s \in \mathscr{L}(r)$?

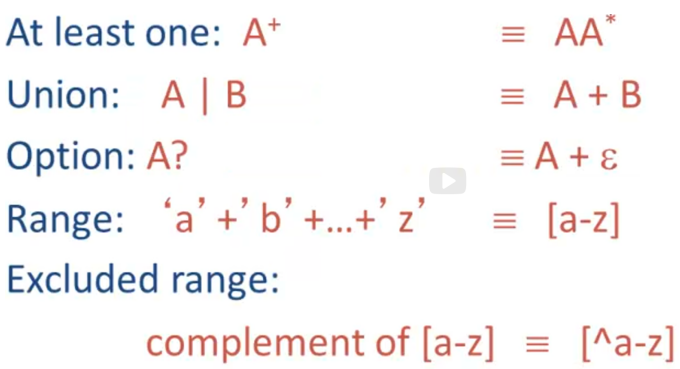


Figure 5: Regular Expression Notation

Lexical Specification:

1. Write a rexp for the lexemes of each token class
   - Number: $digit^+$
   - Keyword: $if|else|...$
   - Identifier: $letter(letter + digit)^*$
   - OpenPar: (
   - ...
2. Construct R, matching all lexemes for all tokens i.e $R = Keyword + Identifier + Number + OpenPar + ...$
3. Let input be $x_1...x_n$. For $1 <= i <= n$ find $x_1...x_n \in \mathscr{L}(R)$?
4. If success, then we know that $x_1...x_n \in \mathscr{L}(R_j)$ for some j
5. Remove $x_1...x_i$ from input and jump to step 3

Maximal Munch: when faced with a choice of two different prefixes for input of different length, always take the longer one

Ambiguous Class? When there are multiple possible classes for a given token, use priority queue to determine which one to use.

No Rule Matches? Match error (as base case when other things don't match) " all strings not in the lexicon"

What makes a good algorithm? Requires only single pass over input, Few operations per character (table lookup)

4.1 Finite Automata

A good implementation model for regular expressions where regular expressions are the specification and finite automata are the implementation.

Finite Automata:
- input alphabet Σ
- set of states S
- state start state n
- set of accepting states $F \subseteq S$
- set of transitions $state \rightarrow^{input} state$

4.2 RegEx to NFAs

4.3 NFA to DFA

4.4 Implementing Finite Automata

5 Assignment 1