Compilers

CS143
10:30-11:50TT
Gates B01

Administrivia

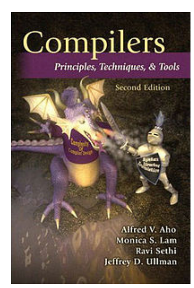
- Syllabus is on-line, of course
  - cs143.stanford.edu
  - Assignment dates will not change
  - Midterm
    - Thursday, 5/3
    - in class
  - Final
    - Monday, 6/11
    - 12:15-3:15pm
- Communication
  - Use discussion forum, email, phone, office hours

Staff

- Instructor
  - Alex Aiken
- TAs
  - Todd Warszawski
  - Samir Menon
  - Ivan Robles

Text

- The Purple Dragon Book
- Aho, Lam, Sethi & Ullman
- Not required
  - But a useful reference



Course Structure

- Course has theoretical and practical aspects
- Need both in programming languages!
- Written assignments = theory
- Programming assignments = practice

Academic Honesty

- Don't use work from uncited sources
- We use plagiarism detection software
  - many cases in past offerings

The Course Project

- A big project
- … in 4 easy parts
- Start early!

How are Languages Implemented?

- Two major strategies:
  - Interpreters (slightly older)
  - Compilers (slightly newer)
- Interpreters run programs "as is"
  - Little or no preprocessing
- Compilers do extensive preprocessing

Language Implementations

- Batch compilation systems dominate "low level" languages
  - C, C++
- "Higher level" languages are often interpreted
  - python
- Some (Java) provide both
  - Interpreter + "Just in Time (JIT)" compiler

History of High-Level Languages

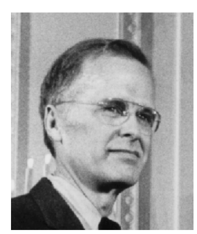
- 1954: IBM develops the 704
  - Successor to the 701
- Problem
  - Software costs exceeded hardware costs!
- All programming done in assembly

The Solution

- Enter "Speedcoding"
- An interpreter
- Ran 10-20 times slower than hand-written assembly

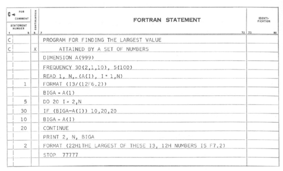
FORTRAN I

- Enter John Backus
- Idea
  - Translate high-level code to assembly
  - Many thought this impossible
  - Had already failed in other projects



FORTRAN I (Cont.)

- 1954-7
  - FORTRAN I project
- 1958
  - >50% of all software is in FORTRAN
- Development time halved



FORTRAN I

- The first compiler
  - Huge impact on computer science
- Led to an enormous body of theoretical work
- Modern compilers preserve the outlines of FORTRAN I

The Structure of a Compiler

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

The first 3, at least, can be understood by analogy to how humans comprehend English.

Lexical Analysis

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

More Lexical Analysis

- Lexical analysis is not trivial. Consider:

  ist his ase nte nce

And More Lexical Analysis

- Lexical analyzer divides program text into "words" or "tokens"

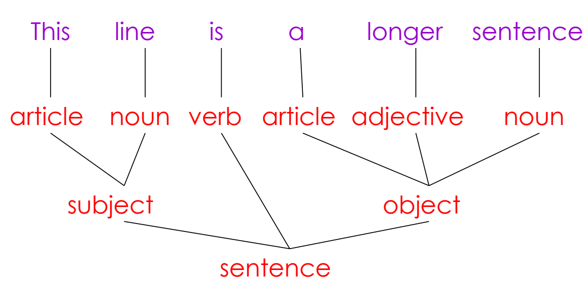
  If x == y then z = 1; else z = 2;

- Units:

Parsing

- Once words are understood, the next step is to understand sentence structure
- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence

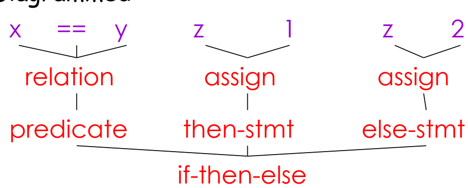


Parsing Programs

- Parsing program expressions is the same
- Consider:

  If x == y then z = 1; else z = 2;

- Diagrammed:



Semantic Analysis

- Once sentence structure is understood, we can try to understand "meaning"
  - But meaning is too hard for compilers

- Compilers perform limited analysis to catch inconsistencies

Semantic Analysis in English

- Example:

  Jack said Jerry left his assignment at home.

  What does "his" refer to? Jack or Jerry?

- Even worse:

  Jack said Jack left his assignment at home?

  How many Jacks are there?

  Which one left the assignment?

Semantic Analysis in Programming

- Programming languages define strict rules to avoid such ambiguities

- This C++ code prints "4"; the inner definition is used

```
{
  int Jack = 3;
  {
     int Jack = 4;
     cout << Jack;
  }
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings
- Example:

  Jack left her homework at home.

- A "type mismatch" between her and Jack; we know they are different people
  - Presumably Jack is male

Optimization

- No strong counterpart in English, but akin to editing
- Automatically modify programs so that they
  - Run faster
  - Use less memory
  - In general, conserve some resource
- The project has no optimization component

Optimization Example

X = Y * 0 is the same as X = 0

Code Generation

- Produces assembly code (usually)
- A translation into another language
  - Analogous to human translation

Intermediate Languages

- Many compilers perform translations between successive intermediate forms
  - All but first and last are *intermediate languages* internal to the compiler
  - Typically there is 1 IL
- IL's generally ordered in descending level of abstraction
  - Highest is source
  - Lowest is assembly

Intermediate Languages (Cont.)

- IL's are useful because lower levels expose features hidden by higher levels
  - registers
  - memory layout
  - etc.
- But lower levels obscure high-level meaning

Issues

- Compiling is almost this simple, but there are many pitfalls.
- Example: How are erroneous programs handled?
- Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - Course theme: many trade-offs in language design

Compilers Today

- The overall structure of almost every compiler adheres to our outline
- The proportions have changed since FORTRAN
  - Early: lexing, parsing most complex, expensive
  - Today: optimization dominates all other phases, lexing and parsing are cheap