Compilers

Thanks: Almost all of the slides and assignments, as well as the general layout of this course are modeled after Professor Alex Aiken's CS 143 compilers course at Stanford University. Much thanks to him for graciously providing all of the material from his course for use in this one.

Administrivia

- Syllabus is on-line, of course
  - Assignment dates will not change
  - Please note midterm and final dates!
- Communication
  - email, phone, office hours, Skype

Text

- The Purple Dragon Book
- Aho, Lam, Sethi & Ullman
- A useful and required resource!

Course Structure

- Course has theoretical and practical aspects

- Need both in compilers!

- Written assignments = theory
- Programming assignments = practice
- Electronic hand-in for both

Academic Honesty

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

The Course Project

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

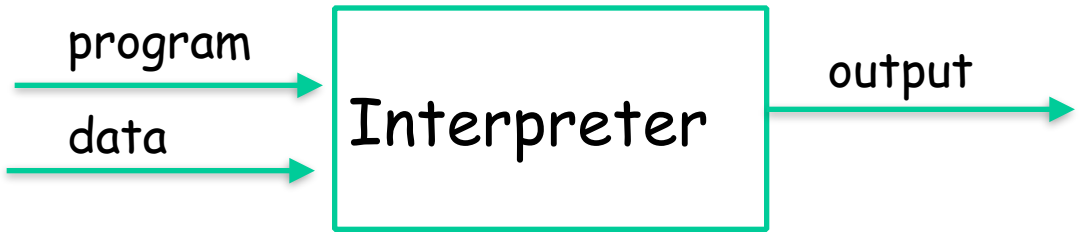
How are Languages Implemented?

- Two major strategies:
  - Interpreters (older)
  - Compilers (newer)

- Interpreters run programs "as is"
  - Little or no preprocessing of the program before running

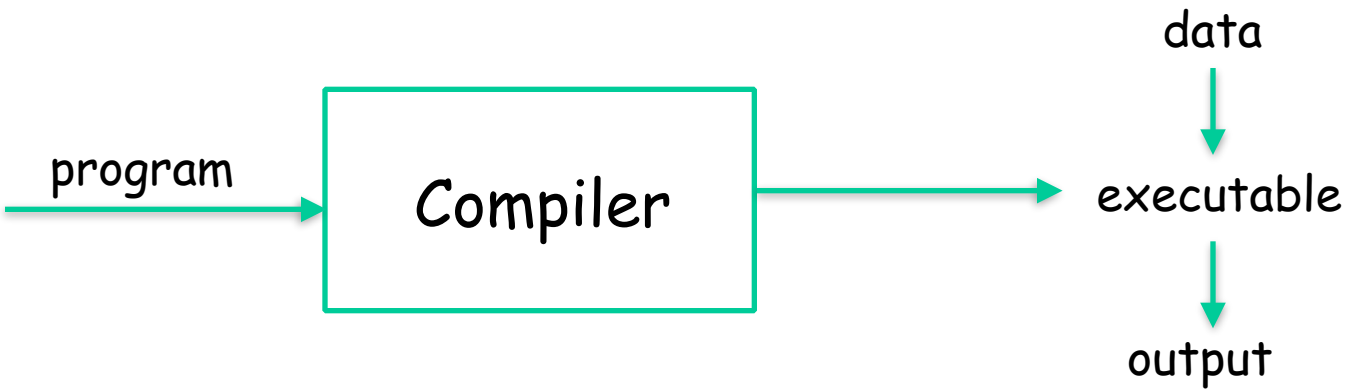
- Compilers do extensive preprocessing

Interpreters



» Work done is "online" in sense that the work occurs while program is running

Compiler



» Compiler is "offline" in the sense that the program is processed before begin run.

Language Implementations

- Batch compilation systems dominate
  - gcc
- Some languages are primarily interpreted
  - Java bytecode
- Some environments (Lisp) provide both
  - Interpreter for development
  - Compiler for production

History of High-Level Languages

- 1954 IBM develops the 704
  - Successor to the 701
  - First commercially successful computer

- Problem
  - Surprise! Software costs exceeded hardware costs (by a LOT)!
  - And hardware was expensive. It cost far more in relative terms than it ever would again

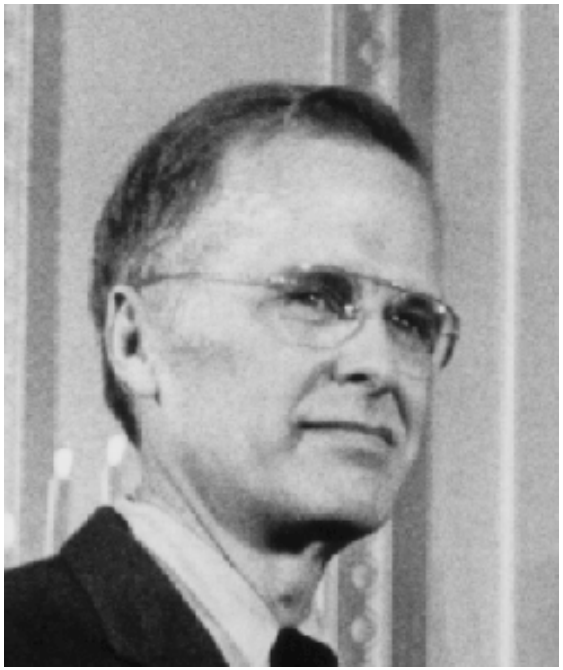
- All programming done in assembly

So Naturally: How to Code Faster?

- Enter "Speedcoding" (John Backus, 1953)
  - Really an interpreter
  - Programming became much faster, but…
  - Ran 10-20 times slower than hand-written assembly (also true of interpreted code today)
  - Also speed code interpreter took up 300 bytes of data!
    - Which was in fact 30% of the memory in those days! (so space was another concern)

FORTRAN I (Formula Translation Project)

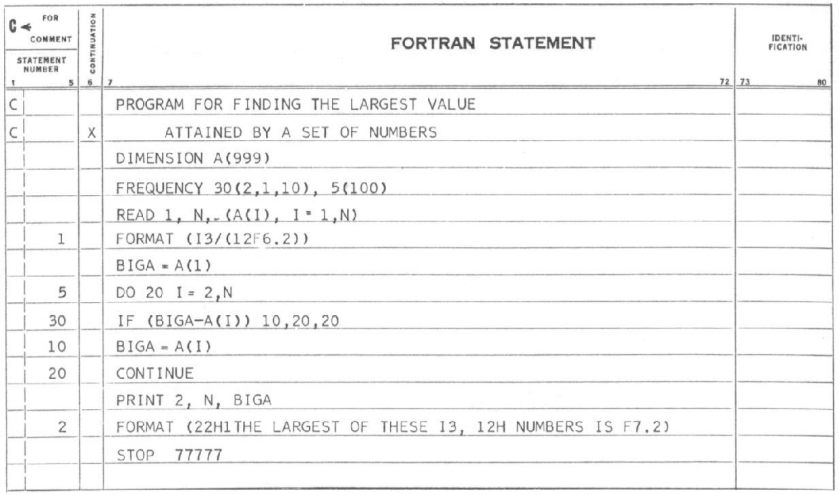
- Enter John Backus



- Idea
  - Translate high-level code to assembly
  - High-level to allow scientists to write code closer in form to equations they used
  - 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 (only one year after compiler developed!)
- Development time halved
  - Allowed far better use of machines



C ← FOR COMMENT / STATEMENT NUMBER (1–5)	CONTINUATION (6)	FORTRAN STATEMENT (7–72)	IDENTIFICATION (73–80)
C		PROGRAM FOR FINDING THE LARGEST VALUE	
C	X	ATTAINED BY A SET OF NUMBERS	
		DIMENSION A(999)	
		FREQUENCY 30(2,1,10), 5(100)	
		READ 1, N, (A(I), I = 1,N)	
1		FORMAT (I3/(12F6.2))	
		BIGA = A(1)	
5		DO 20 I = 2,N	
30		IF (BIGA-A(I)) 10,20,20	
10		BIGA = A(I)	
20		CONTINUE	
		PRINT 2, N, BIGA	
2		FORMAT (22H1THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)	
		STOP 77777	

FORTRAN I

- The first compiler
  - Huge impact on computer science

- Led to an enormous body of theoretical work
  - And requires a good amount of engineering as well
  - Compiler design and programming languages combines systems work with subtleties of theory

- Modern compilers preserve the outlines of FORTRAN I
- Compiler design one of the great historical successes of computer science research

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.

Clarification

- Optimization: Need not be just about making program run faster. Can also mean using less power and/or memory.
- Code generation: Target need not be assembly language. Might be byte code for a virtual machine, or another high level language

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.

Lexical Analysis

This is a sentence.

- Work is being done, though it may seem automatic
  - You immediately recognize that there are four words: 'this', 'is', 'a' and 'sentence'
  - You have to recognize the separators, namely the blanks.
  - And the punctuation, things like the periods
  - And clues like capital letters
  - And these help you to divide up this group of letters into a bunch of words that you can understand

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:

And More Lexical Analysis

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

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

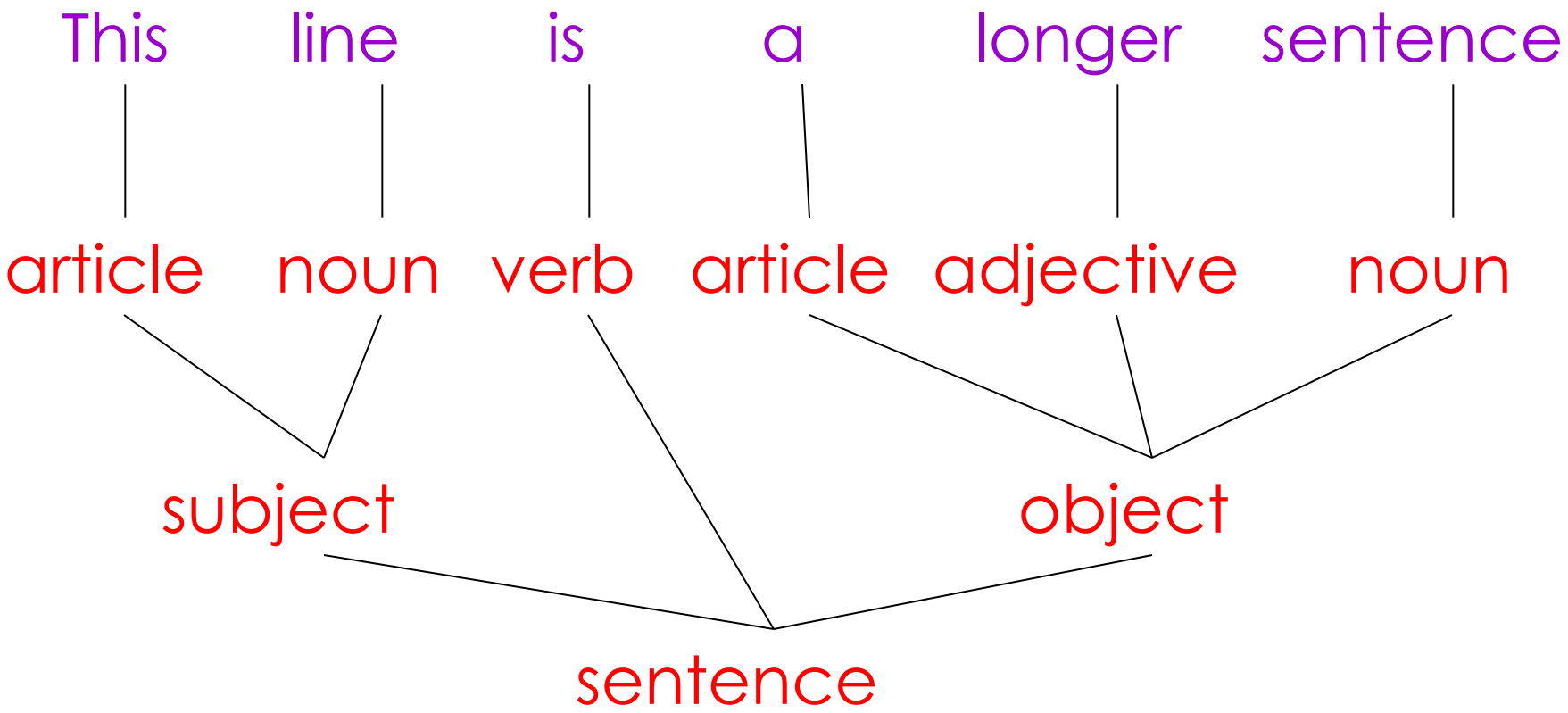
- Units:
  - Keywords
  - Variable Names
  - Constants
  - Operators: '==' and '='
    - How do we know == isn't just two =
  - Punctuation and spaces

Parsing

- Once words are understood, the next step is to understand sentence structure
  - I.e., group words together into higher constructs

- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence



Parsing

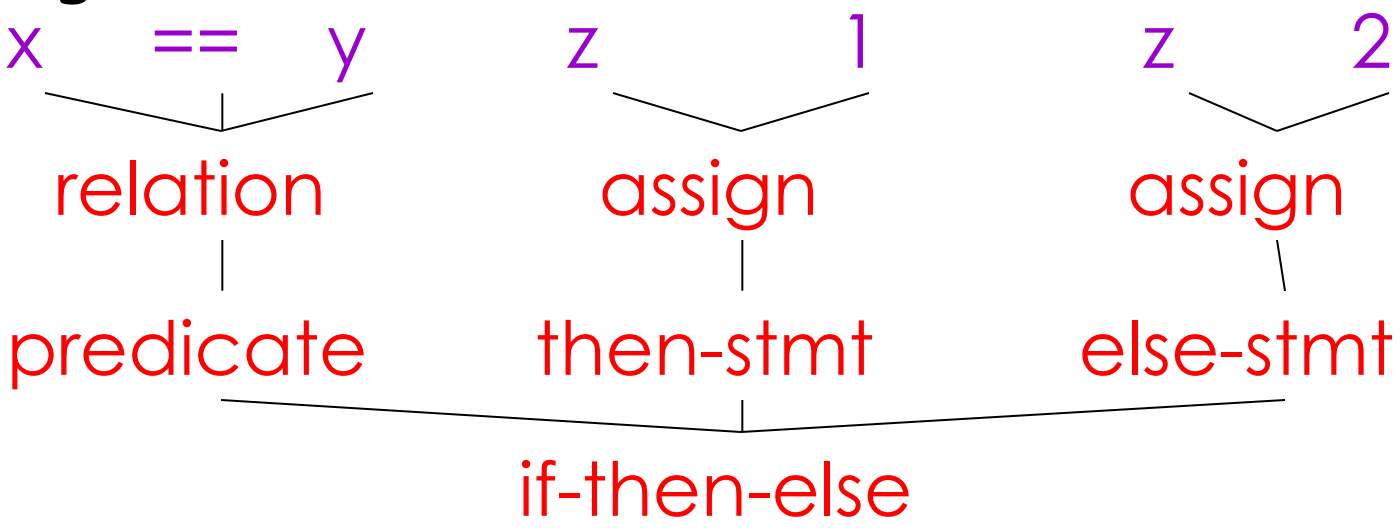
» Analogy between parsing English text and parsing program text is very strong. In fact they are exactly the same thing.

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
  - And we don't even really understand how it happens in humans

- Compilers perform limited semantic analysis to catch inconsistencies
  - E.g., if a program is self-inconsistent or there are ambiguities, compiler can catch this and report error
  - But compiler has no real understanding of "meaning"

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? Possibilities?

  Which one left the assignment?

Analogy in programming languages is variable bindings.

Semantic Analysis in Programming

- Programming languages define strict rules to avoid such ambiguities

- This C++ code prints "4"; the inner definition is used (standard rule for many lexically scoped languages)

```
{
    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
  - This kind of analysis is analogous to type checking

Optimization

- No strong counterpart in English, but akin to editing
  - E.g., removing words to make a page limit, but keeping meaning the same
- Automatically modify programs so that they
  - Run faster
  - Use less memory
  - Reduce number of database accesses or network packets
  - In general, conserve some resource
- The project has no optimization component

Optimization Example

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

Real improvement: make a multiplication and assignment into just an assignment

Optimization Example

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

Real improvement: make a multiplication and assignment into just an assignment

Except that it's not always correct…

Optimization Example

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

For integers it works, but not for floating point.

Special number in IEEE standard called NaN.
And NaN * 0 is not 0. It's NaN.

Optimization Example

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

So if, for example, X and Y are plotting point numbers, you can't do this, since it breaks some important algorithms that depend on correct propagation of NaN.

Optimization Example

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

This is one of the important aspects of compiling optimization: it's not always obvious when it's legal to do certain optimizations.

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