Introduction to Compilers

ECE 351
Mon/Fri, 10 – 11:20 AM
QNC 1502

Staff and Website

- Instructor
  - Vijay Ganesh
- TAs
  - Matthew Ma, Riyad Parvez, Reza Babaee
- Lab Instructor
  - Tiuley Alguindigue
- Course Website
  - https://ece.uwaterloo.ca/~vganesh/TEACHING/S2014/ECE351/index.html

Textbook and course material

- "Crafting a Compiler" by Fischer, Cytron and LeBlanc
- I will also be using material from research papers (by myself and others), notes and slides by other professors (with permission)
- All slides will be made available before the respective lectures on the course website.
- Towards the end of the course I may talk about my research as well, relating to software engineering and security

Course Structure: lectures

- The course has 5 modules (sets of 4-5 lectures) + labs + assignments + exams
- The lecture modules are
  - Stages of compilation, regular and context-free languages, context-free grammars
  - Lexical analysis and parsing
  - Semantic analysis, type systems, ASTs, and IR
  - Code optimization
  - Code generation and runtime support
- Lectures are on Mondays/Fridays from 10-11:20 AM
- There are extra lecture slots on Fridays from 12:30 – 1:20 PM. Also Tutorials from 4:30 – 5:20 PM on Tuesdays
- I will use some of these extra slots and tutorials. Announcements will be made 1 week in advance.
- Please sign up on Piazza (linked off the course website)

Course Structure: labs, assignments, exams

- The Labs
  - 12 labs in total (lab0 is not graded)
  - 1 lab a week, due every Friday midnight
  - Writing a compiler for a hardware description language
  - To be done in groups of at most 2 persons

- 3 assignments to be done individually

- Mid-term exam (closed book)

- Final exam (closed book)

Grade Distribution

- Assignments 9% (3% per assignment)
- Mid-term 8%
- Labs 33% (3% per lab. First lab is not graded)
- Final Exam 50% of total grade
- The course is heavy. You will learn a lot

What this course is about

- The course is about compiler construction
- You will learn the basics of
  - Formal language theory (regular and context-free languages)
  - Basic knowledge about types of programming languages
  - Stages of modern compilers
    - lexical analysis
    - context-free grammars and parsing
    - type checking
    - static analysis
    - code optimization, code generation
    - memory management and runtime support

What is a programming language and how are they implemented?

- Programming languages are powerful mathematical constructs that enable one to program a computer

- Two aspects of such constructs are:
  - Syntax
    - Language constructs, e.g., if-else, while,…
  - Semantics
    - What do these constructs mean
    - Assumes or enables a model of computation, e.g., Turing machines or mathematical functions
    - Denotational, operational and axiomatic

Types of programming languages

- Declarative: You specify the "what" of the computation. The system will figure out the "how". E.g. SQL, Matlab,…
  - Pure Functional and Logic-based languages are also declarative. E.g., of such declarative languages include Prolog and Haskell
- Imperative: You specify both the "what" and the "how" of the computation. C/C++/Java
- Scripting. E.g., Unix bash, Awk.
- There are many other categories of languages. However, most languages can be categorized as declarative or imperative or mixed.
- You can add features like object orientation to any of the above category of languages.

Programming language design

- Not easy

- Languages must
  - Make it easier to program
  - Increase programmer productivity
  - Automate routine tasks
  - Enable programmers to produce efficient (time, space, power,…) code
  - Enable secure and correct construction

How are Languages Implemented?

- Major strategies:
  - Interpreters (older)
  - Compilers (newer)
  - Mixed: JIT compilation

- Interpreters run programs “as is”
  - Little or no preprocessing

- Compilers do extensive preprocessing and offline optimizations

What is a compiler?

- Compilers are computer programs that translate programs typically written in high-level language to a lower-level language
  - In the process of translation, the compiler checks the input program for correctness (up to a point)
  - Optimizes the code (up to a point)
  - Links to code libraries
  - Generates what is called a binary or executable
- Lower level languages typically expose a lot about the underlying hardware to the programmer
- Higher-level languages hide these details through appropriate abstractions

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

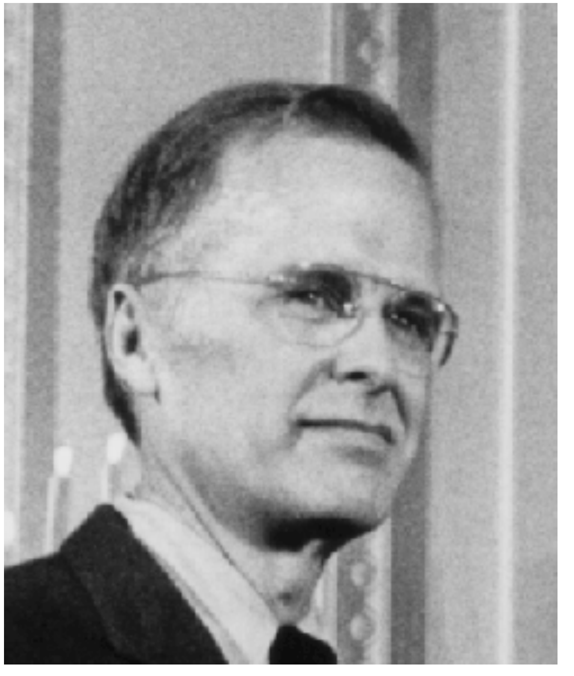
- 1954 IBM develops the 704
  - Successor to the 701
- Problem
  - Software costs exceeded hardware costs!
- All programming done in assembly

The Solution

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

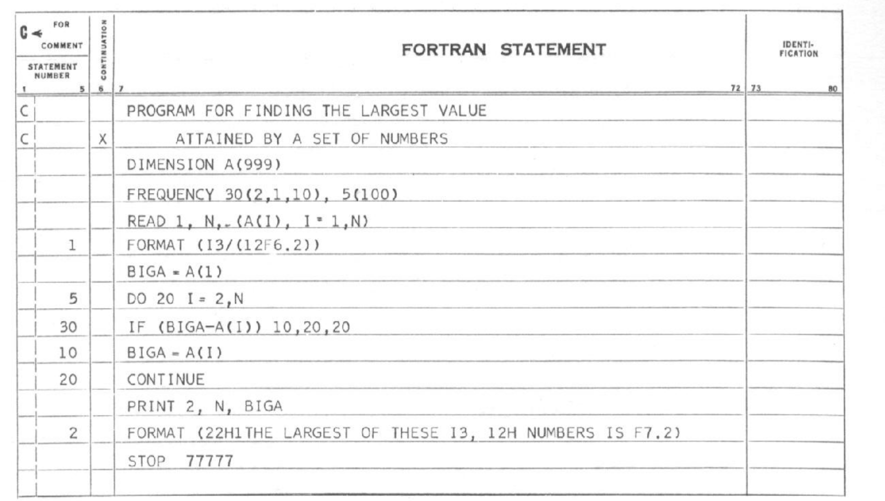
FORTRAN I

- 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



C ← FOR COMMENT | STATEMENT NUMBER | CONTINUATION | FORTRAN STATEMENT | IDENTI-FICATION

```
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

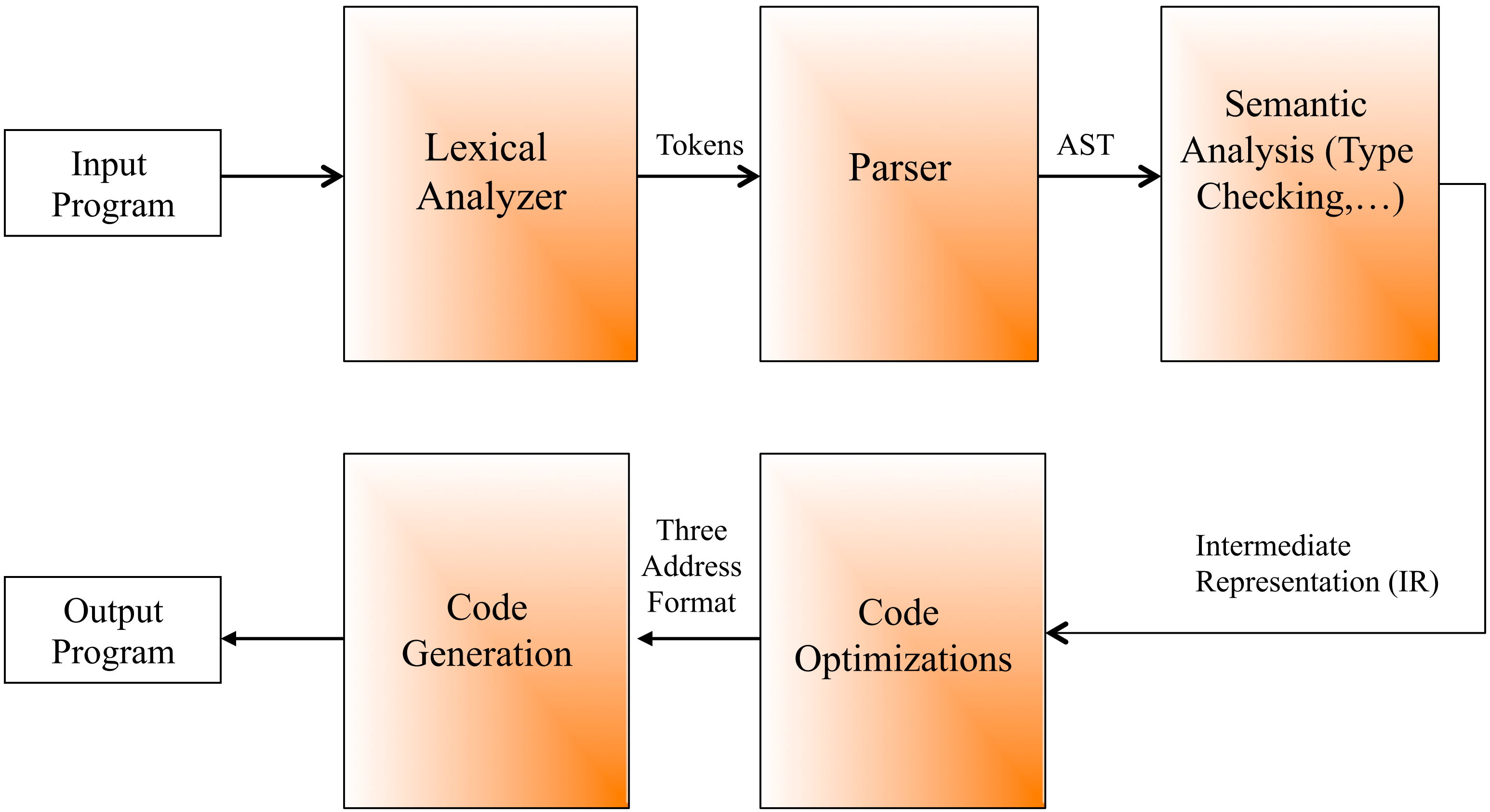
- The first compiler
  - Huge impact on computer science
- Led to an enormous body of theoretical work
- Modern compilers preserve the outlines of FORTRAN I

The Structure (Phases) 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.

The Structure (Phases) of a Compiler



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"

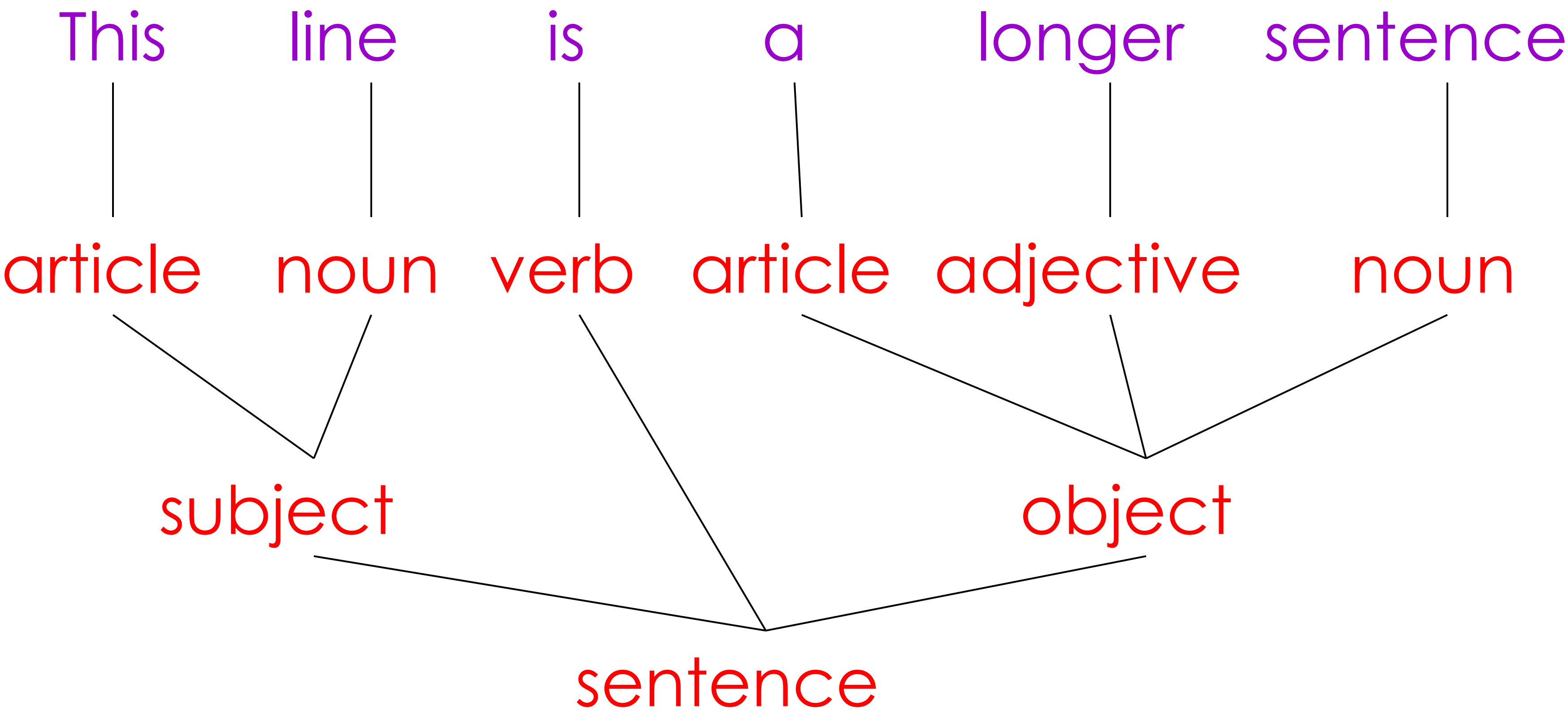
  If x == y then z = 1; else z = 2;

- 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

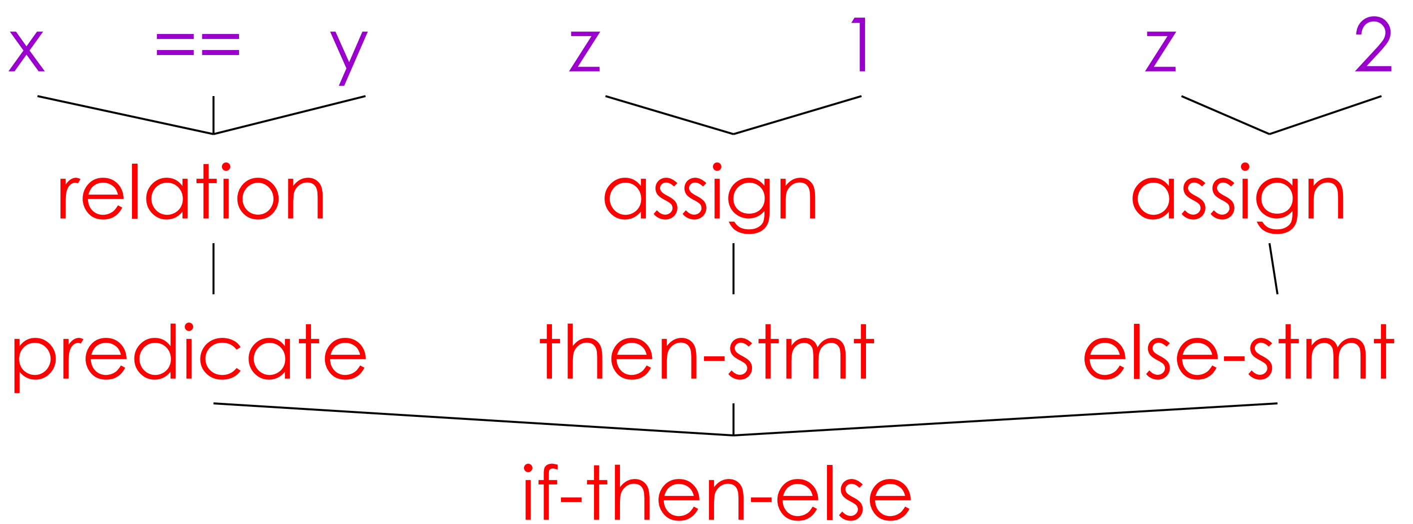


Parsing Programs

- 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

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 only one IL (aka IR)

- 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

Next Lecture: Formal Language Theory, Basics of Computability and Complexity