Introduction to Programming Languages and Compilers

CS164

1-2:30 MW

306 Soda Hall

Aministrivia

- Course home page:
  https://sites.google.com/site/cs164fall2015/
- If you are on the **waiting list**, follow the normal procedures (see class web page)
  - The course staff is not involved !
  - If you are **enrolled**, you don't need to do anything
- No discussion sections this week
- Pick up class accounts
  - At the end of lecture today, and from Ben afterwards

Course Structure

- Course has theoretical and practical aspects
- Need both in programming languages!
- Written assignments = theory
  - Class hand-in, right before lecture
- Programming assignments = practice
  - Electronic hand-in
- Strict deadlines

Academic Honesty

- Don't use work from uncited sources
  - Including old code
- We use plagiarism detection software
  - 6 cases in last few semesters

The Course Project

- A big project
- … in 5 easy parts
- Start early!

How are Languages Implemented?

How are Languages Implemented?

- Two major strategies:
  - Interpreters (older, less studied)
  - Compilers (newer, more extensively studied)

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

- Compilers do extensive preprocessing
  - Most implementations use compilers

(Short) History of High-Level Languages

- 1953 IBM develops the 701
- All programming done in assembly
- Problem: Software costs exceeded hardware costs!
- John Backus: "Speedcoding"
  - An interpreter
  - Ran 10-20 times slower than hand-written assembly

FORTRAN I

- 1954 IBM develops the 704: 12K FPS
- John Backus
  - Idea: translate high-level code to assembly
  - Many thought this impossible

- 1954-7 FORTRAN I project
- By 1958, >50% of all software is in FORTRAN
- Cut development time dramatically
  - (2 wks $\rightarrow$ 2 hrs)

FORTRAN I

- The first compiler
  - Produced code almost as good as hand-written
  - 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

The Structure of a Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Intermediate Code Generation
5. Machine Independent Optimization
6. Code Generation
7. Machine-Dependent Optimization

Lexical Analysis

- Reads a stream of characters
- Groups characters into sequences: lexemes
- For each lexeme, outputs a token
- Sequence of tokens is passed to syntax analyzer

position = initial + rate * 60

Lexical Analysis

position = initial + rate * 60

Lexical Analysis

- Reads a stream of characters
- Groups characters into sequences: lexemes
- For each lexeme, outputs a token
- Sequence of tokens is passed to syntax analyzer

```
*p->f+=-.12345e-5
```

Parsing

- Parsing = create a tree-like intermediate representation from a sequence of tokens: syntax tree
- Interior node = operation
- Children of a node = operands

Parsing

<ID,1> <=> <ID,2> <+> <ID,3> <*> <NUM,60>

Semantic Analysis

- Uses syntax tree
- Checks for semantic consistency
  - Type checking: each operator has matching operands
  - Array index cannot be a floating point number
  - Cannot apply modulo (%) operator on a string
  - Permit some type conversions: coercion
    - Integer to float

Parsing

<ID,1> <=> <ID,2> <+> <ID,3> <*> <NUM,60>

Intermediate Code Generation

- Generate a low-level machine-like intermediate representation from syntax tree
  - Easy to produce
  - Easy to translate to the target machine language
- Three-address code
  - Three or fewer operands per instruction
  - Assignment instruction has at most one operator

Intermediate Code Generation

position = initial + rate * 60

Code Optimization

- Improve intermediate code
  - Faster code
  - Uses fewer resources
  - May use less power

Code Generation

- Produces assembly code (usually)
  - which is then assembled into executables by an assembler
- Registers and memory locations are selected for each variables used
- Judicious assignment of registers to hold variables

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

Trends in Compilation

- Optimization for speed is less interesting. But:
  - scientific programs
  - advanced processors (Digital Signal Processors, advanced speculative architectures)
  - Small devices where speed = longer battery life
- Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting concurrency errors (data races)
  - ...