Course Overview

CS 132: Compiler Design
January 17, 2001

Compilers and Interpreters

• Compilers translate code from one language (source language) to another (target language).
  - Ideally, the translation preserves meaning
  - Target often assembly or machine language, but could be bytecodes, or C, ...
• Interpreters execute code as given
• In practice, the boundary is fuzzier.
To Interpret or Compile?

• It is true that...
  – some languages are easier to compile than others.
  – some languages have been historically compiled, and others have been historically interpreted.
  – some languages require an interactive system

• But compilation or interpretation is not an intrinsic property of the language!

Bad Example

“...it's clear that a well-written program in Java could never run as fast as a well-written program in C or C++. That's because the Java bytecode is interpreted, not compiled. Programs written in C are compiled into binaries which can be executed by a specific computer processor. Programs written in Java require one more step – they must be interpreted by the Java `virtual machine’ before running on a particular computer architecture. As a result, a computer running a Java program has to execute more machine-language instructions to do the same amount of work than a computer running an equivalent program written in C.”

Simson Garfinkel, Salon Magazine, 1/8/2001
Why Compilers are Interesting

• Use ideas from both theory and systems
  - Finite automata, context-free grammars, graph algorithms, lattices, formal semantics, ...
  - Assembly/machine language, data layout, memory management, register usage, calling conventions, cache behavior, cycle counts, CPU pipelines ...

Challenges

• Perfect information about code usually undecidable.
  - Is this function ever called at run-time?
  - Can these two pointers ever alias?
  - Is this array access guaranteed to be in-bounds?
  - Is this loop always executed at least once?
  - Does this function always look at its argument?
  - Which functions can be bound to this variable?

• Accurate approximations can be expensive or require source of entire program.
Challenges

- Optimization tasks frequently NP-complete
  - Put variables into registers so as to minimize the number of loads and stores
  - Order a list of instructions, respecting dependencies, so as to minimize the number of cycles required.
  - Lay out data in memory so as to minimize cache conflicts/memory contention

Optimality

- Define a fully-optimizing compiler to be one which computes the smallest-possible code for all programs.

Theorem: For a nontrivial language there is no fully-optimizing compiler.
Is the Task Hopeless?

- Of course not!

- Compilers just use a lot of heuristics and coarse approximations
  - Ideally, safe approximations
  - Code isn’t perfect, but is usually “good enough”
  - Over time, these improve (smarter techniques, faster machines)

Theorem [Full Employment for Compiler Writers]:

For any optimizing compiler there exists a better one.
Generic Compiler

Preprocessor → **Program Text**
              ↓ Lexer
              ↓ Token stream
              ↓ Parser
              ↓ Syntax Tree
              ↓ Semantic Analyzer
Optimizer → **Intermediate Representation**
            ↓ Code Generator
            ↓ Assembly Code

Books

- **Required Text:**
  - Modern Compiler Implementation in ML by Andrew Appel.

- **“Optional” Texts** [No assigned readings, but you need to know the material]:
  - SPARC Architecture, Assembly Language Programming, and C (2nd Ed.) by Richard Paul.
  - Introduction to Programming using SML by Michael Hansen and Hans Richel.
Structure of This Course

• Core material:
  – First half of Appel’s text
  – Covers phases of a compiler for the Tiger language (to be described momentarily)
• Plus:
  – Discussion of other topics (from Appel and lecture notes) as time permits
  – E.g., Code optimizations, compiling functional or object-oriented languages, ...

Homeworks

• All from Appel’s text
  – Read the first 12 chapters
  – Implement the code described in the “Program” sections of these chapters.
  – Programming assignments assigned approximately weekly
• The course requires a fair bit of coding
  – After the first assignment, you will work in pairs.
  – Randomly assigned for each assignment.
**Why Use Standard ML?**

- Strengths of SML match up well with demands of compiler writing:
  - Symbolic manipulation
    - Pattern-matching and datatypes
  - Modularity and interfaces
    - Structures and signatures
  - Plus safety, garbage collection, etc.

**Tiger Language**

- “Simple but nontrivial language of the Algol family”
  - This family includes Pascal and C
- Includes
  - Assignment, for and while loops, conditionals, strings, printing, ...
  - Arrays
  - Nested function definition
  - Heap-allocated records
    - Can program linked lists
Sample Tiger Code

/* A program to solve the 8-queens problem*/
let
var N := 8

fun printboard() =
  for i := 0 to N-1 do
    for j := 0 to N-1 do
      print(if col[i]=j then " O" else " .");
    print("\n")
  print("\n")
End

fun try(c:int) =
  if c=N then
    printboard()
  else
    for r := 0 to N-1 do
      if (row[r]=0 &
          diag1[r+c]=0 &
          diag2[r+7-c]=0) then
        (row[r]:=1;
          diag1[r+c]:=1;
          diag2[r+7-c]:=1;
          col[c]:=r;
          try(c+1);
          row[r]:=0;
          diag1[r+c]:=0;
          diag2[r+7-c]:=0)
      in
    try(0)
End

Grading

• Grade is determined as follows:
  80% : Result of weekly assignments
  20% : Class participation

• No final or midterm.
Assignment 1

• Read Chapter 1, do the PROGRAM part.
• Turn in before next Wednesday’s class.
• More details on Assignments web page.