Important facts:

Name: Dr. Tony Hosking
Email: hosking@cs.purdue.edu
Office: LWSN 3154L

Basis for grades:
20% midterm
30% final
40% project
10% homeworks
Things to do

- read Appel chapter 1
- make sure you have a working account
- start brushing up on Java
- review Java development tools
- find http://www.cs.purdue.edu/homes/hosking/352
What is a compiler?

- a program that translates an *executable* program in one language into an *executable* program in another language
- we expect the program produced by the compiler to be better, in some way, than the original

What is an interpreter?

- a program that reads an *executable* program and produces the results of running that program
- usually, this involves executing the source program in some fashion

This course deals mainly with *compilers*

Many of the same issues arise in *interpreters*
Motivation

Why study compiler construction?

Why build compilers?

Why attend class?
Compiler construction is a microcosm of computer science

**artificial intelligence**
  greedy algorithms, learning algorithms

**algorithms**
  graph algorithms, union-find, dynamic programming

**theory**
  DFAs for scanning, parser generators, lattice theory

**systems**
  allocation and naming, locality, synchronization

**architecture**
  pipeline management, hierarchy management, instruction set use

Inside a compiler, all these things come together
Isn’t it a solved problem?

*Machines are constantly changing*

Changes in architecture ⇒ changes in compilers

- new features pose new problems
- changing costs lead to different concerns
- old solutions need re-engineering

*Changes in compilers should prompt changes in architecture*

- New languages and features
Intrinsic Merit

*Compiler construction is challenging and fun*

- interesting problems
- primary responsibility for performance (blame)
- new architectures $\Rightarrow$ new challenges
- *real* results
- extremely complex interactions

*Compilers have an impact on how computers are used*

Some of the most interesting problems in computing
Experience

You have used several compilers

What qualities are important in a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for separate compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimization

Each of these shapes your expectations about this course
Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- agreement on format for object (or assembly) code

*Big step up from assembler — higher level notations*
Traditional two pass compiler

Implications:

- intermediate representation (IR)
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes ⇒ better code
Can we build \( n \times m \) compilers with \( n + m \) components?

- must encode \textit{all} the knowledge in each front end
- must represent \textit{all} the features in one IR
- must handle \textit{all} the features in each back end

\textit{Limited success with low-level IRs}
Front end

Responsibilities:

- recognize legal procedure
- report errors
- produce IR
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated
Front end

Scanner:

- maps characters into tokens – the basic unit of syntax
  
  \[ x = x + y; \]
  
  becomes
  
  \[ <\text{id}, x> = <\text{id}, x> + <\text{id}, y> ; \]

- character string value for a token is a lexeme

- typical tokens: number, id, +, -, *, /, do, end

- eliminates white space (tabs, blanks, comments)

- a key issue is speed
  
  ⇒ use specialized recognizer (as opposed to lex)
Front end

Parser:

- recognize context-free syntax
- guide context-sensitive analysis
- construct IR(s)
- produce meaningful error messages
- attempt error correction

*Parser generators mechanize much of the work*
Context-free syntax is specified with a grammar

\[
\text{<sheep noise>} \; ::= \; \text{baa} \\
\quad \mid \; \text{baa} \; \text{<sheep noise>}
\]

The noises sheep make under normal circumstances

This format is called Backus-Naur form (BNF)

Formally, a grammar \( G = (S, N, T, P) \) where

\( S \) is the start symbol

\( N \) is a set of non-terminal symbols

\( T \) is a set of terminal symbols

\( P \) is a set of productions or rewrite rules

\((P : N \rightarrow N \cup T)\)
Context free syntax can be put to better use

1 \<\text{goal}\> ::= \<\text{expr}\>

2 \<\text{expr}\> ::= \<\text{expr}\> \<\text{op}\> \<\text{term}\>

3 \hspace{1em} | \hspace{1em} \<\text{term}\>

4 \<\text{term}\> ::= \text{number}

5 \hspace{1em} | \hspace{1em} \text{id}

6 \<\text{op}\> ::= +

7 \hspace{1em} | \hspace{1em} -

Simple expressions with addition and subtraction over tokens \text{id} and \text{number}

\[S = \<\text{goal}\>\]
\[T = \text{number}, \text{id}, +, -\]
\[N = \<\text{goal}\>, \<\text{expr}\>, \<\text{term}\>, \<\text{op}\>\]
\[P = 1, 2, 3, 4, 5, 6, 7\]
Given a grammar, valid sentences can be derived by repeated substitution.

<table>
<thead>
<tr>
<th>Prod’n.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;goal&gt;</td>
</tr>
<tr>
<td>1</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; &lt;op&gt; &lt;term&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;expr&gt; &lt;op&gt; y</td>
</tr>
<tr>
<td>7</td>
<td>&lt;expr&gt; − y</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; &lt;op&gt; &lt;term&gt; − y</td>
</tr>
<tr>
<td>4</td>
<td>&lt;expr&gt; &lt;op&gt; 2 − y</td>
</tr>
<tr>
<td>6</td>
<td>&lt;expr&gt; + 2 − y</td>
</tr>
<tr>
<td>3</td>
<td>&lt;term&gt; + 2 − y</td>
</tr>
<tr>
<td>5</td>
<td>x + 2 − y</td>
</tr>
</tbody>
</table>

To recognize a valid sentence in some CFG, we reverse this process and build up a parse.
A parse can be represented by a parse, or syntax, tree.

Obviously, this contains a lot of unnecessary information.
So, compilers often use an *abstract syntax tree*

```
+   <id:y>
  / 
<id:x> <num:2>
```

This is much more concise

Abstract syntax trees (ASTs) are often used as an IR between front end and back end
Back end

Responsibilities

- translate IR into target machine code
- choose instructions for each IR operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

*Automation has been less successful here*
Instruction selection:

- produce compact, fast code
- use available addressing modes
- pattern matching problem
  - ad hoc techniques
  - tree pattern matching
  - string pattern matching
  - dynamic programming
Register Allocation:

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult

*Modern allocators often use an analogy to graph coloring*
Traditional three pass compiler

source code \[\rightarrow\] front end \[\rightarrow\] IR \[\rightarrow\] middle end \[\rightarrow\] IR \[\rightarrow\] back end \[\rightarrow\] machine code

errors

Code Improvement

- analyzes and changes IR
- goal is to reduce runtime
- must preserve values
Modern optimizers are usually built as a set of passes

Typical passes

- constant propagation and folding
- code motion
- reduction of operator strength
- common subexpression elimination
- redundant store elimination
- dead code elimination
The MiniJava compiler

Source Program

Lex Tokens Parse Reductions

Abstract Syntax

Tables

Environments

Parse Actions Semantic Analysis Translate IR Trees

Frame

Frame Layout

Control Flow Analysis Flow Graph

Data Flow Analysis

Register Allocation

Interference Graph

Register Assignment

Code Emission

Assembler

Assembly Language

Relocatable Object Code

Linker

Machine Language

Pass 1

Pass 2

Pass 3

Pass 4

Pass 5

Pass 6

Pass 7

Pass 8

Pass 9

Pass 10
# The MiniJava compiler phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lex</td>
<td>Break source file into individual words, or <em>tokens</em></td>
</tr>
<tr>
<td>Parse</td>
<td>Analyse the phrase structure of program</td>
</tr>
<tr>
<td>Parsing Actions</td>
<td>Build a piece of <em>abstract syntax tree</em> for each phrase</td>
</tr>
<tr>
<td>Semantic Analysis</td>
<td>Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase</td>
</tr>
<tr>
<td>Frame Layout</td>
<td>Place variables, function parameters, etc., into activation records (stack frames) in a machine-dependent way</td>
</tr>
<tr>
<td>Translate</td>
<td>Produce <em>intermediate representation trees</em> (IR trees), a notation that is not tied to any particular source language or target machine</td>
</tr>
<tr>
<td>Canonicalize</td>
<td>Hoist side effects out of expressions, and clean up conditional branches, for convenience of later phases</td>
</tr>
<tr>
<td>Instruction Selection</td>
<td>Group IR-tree nodes into clumps that correspond to actions of target-machine instructions</td>
</tr>
<tr>
<td>Control Flow Analysis</td>
<td>Analyse sequence of instructions into <em>control flow graph</em> showing all possible flows of control program might follow when it runs</td>
</tr>
<tr>
<td>Data Flow Analysis</td>
<td>Gather information about flow of data through variables of program; e.g., <em>liveness analysis</em> calculates places where each variable holds a still-needed <em>(live)</em> value</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>Choose registers for variables and temporary values; variables not simultaneously live can share same register</td>
</tr>
<tr>
<td>Code Emission</td>
<td>Replace temporary names in each machine instruction with registers</td>
</tr>
</tbody>
</table>
A straight-line programming language

\[ Stm \rightarrow Stm ; Stm \quad \text{CompoundStm} \]
\[ Stm \rightarrow \text{id} := Exp \quad \text{AssignStm} \]
\[ Stm \rightarrow \text{print} ( ExpList ) \quad \text{PrintStm} \]
\[ Exp \rightarrow \text{id} \quad \text{IdExp} \]
\[ Exp \rightarrow \text{num} \quad \text{NumExp} \]
\[ Exp \rightarrow Exp \ Binop \ Exp \quad \text{OpExp} \]
\[ Exp \rightarrow ( Stm , Exp ) \quad \text{EseqExp} \]
\[ ExpList \rightarrow Exp , ExpList \quad \text{PairExpList} \]
\[ ExpList \rightarrow Exp \quad \text{LastExpList} \]
\[ Binop \rightarrow + \quad \text{Plus} \]
\[ Binop \rightarrow − \quad \text{Minus} \]
\[ Binop \rightarrow \times \quad \text{Times} \]
\[ Binop \rightarrow / \quad \text{Div} \]

An example straight-line program:

\[ a := 5 + 3; \ b := \text{print}(a, a - 1), 10 \times a); \text{print}(b) \]

prints:

\[ 8 \ 7 \]
\[ 80 \]
a := 5 + 3; b := (print(a, a - 1), 10 × a); print(b)

This is a convenient internal representation for a compiler to use.
abstract class Stm {}
class CompoundStm extends Stm
    Stm stm1, stm2;
    CompoundStm(Stm s1, Stm s2)
    { stm1=s1; stm2=s2; }
}
class AssignStm extends Stm
{
    String id; Exp exp;
    AssignStm(String i, Exp e)
    { id=i; exp=e; }
}
class PrintStm extends Stm {
    ExpList exps;
    PrintStm(ExpList e)
    { exps=e; }
}

abstract class Exp {}
class IdExp extends Exp {
    String id;
    IdExp(String i) {id=i;}
}
class NumExp extends Exp {
    int num;
    NumExp(int n) {num=n;}
}
class OpExp extends Exp {
    Exp left, right; int oper;
    final static int
        Plus=1,Minus=2,Times=3,Div=4;
    OpExp(Exp l, int o, Exp r)
    { left=l; oper=o; right=r; }
}
class EseqExp extends Exp {
    Stm stm; Exp exp;
    EseqExp(Stm s, Exp e)
    { stm=s; exp=e; }
}
abstract class ExpList {}
class PairExpList extends ExpList {
    Exp head; ExpList tail;
    public PairExpList(Exp h, ExpList t)
    { head=h; tail=t; }
}
class LastExpList extends ExpList {
    Exp head;
    public LastExpList(Exp h) {head=h;}
}