<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>2</td>
</tr>
<tr>
<td>2. Lexical analysis</td>
<td>31</td>
</tr>
<tr>
<td>3. LL parsing</td>
<td>58</td>
</tr>
<tr>
<td>4. LR parsing</td>
<td>110</td>
</tr>
<tr>
<td>5. JavaCC and JTB</td>
<td>127</td>
</tr>
<tr>
<td>6. Semantic analysis</td>
<td>150</td>
</tr>
<tr>
<td>7. Translation and simplification</td>
<td>165</td>
</tr>
<tr>
<td>8. Liveness analysis and register allocation</td>
<td>185</td>
</tr>
<tr>
<td>9. Activation Records</td>
<td>216</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction
Things to do

- make sure you have a working SEAS account
- start brushing up on Java
- review Java development tools
- check out the discussion forum on the course webpage
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

<table>
<thead>
<tr>
<th>artificial intelligence</th>
<th>greedy algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>learning algorithms</td>
</tr>
<tr>
<td>algorithms</td>
<td>graph algorithms</td>
</tr>
<tr>
<td></td>
<td>union-find</td>
</tr>
<tr>
<td></td>
<td>dynamic programming</td>
</tr>
<tr>
<td>theory</td>
<td>DFAs for scanning</td>
</tr>
<tr>
<td></td>
<td>parser generators</td>
</tr>
<tr>
<td></td>
<td>lattice theory for analysis</td>
</tr>
<tr>
<td>systems</td>
<td>allocation and naming</td>
</tr>
<tr>
<td></td>
<td>locality</td>
</tr>
<tr>
<td></td>
<td>synchronization</td>
</tr>
<tr>
<td>architecture</td>
<td>pipeline management</td>
</tr>
<tr>
<td></td>
<td>hierarchy management</td>
</tr>
<tr>
<td></td>
<td>instruction set use</td>
</tr>
</tbody>
</table>

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

*Machines are constantly changing*

Changes in architecture \(\Rightarrow\) 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

Compiler construction poses 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 feelings about the correct contents of 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 $\Rightarrow$ better code
A fallacy

Can we build $n \times m$ compilers with $n + m$ components?

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

Limited success with low-level IRs
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 \textit{tokens} – the basic unit of syntax
  \[ x = x + y; \]
  becomes
  \[ <\text{id}, x> = <\text{id}, x> + <\text{id}, y> ; \]
- character string value for a \textit{token} is a \textit{lexeme}
- typical tokens: \textit{number, id, +, -, *, /, do, end}
- eliminates white space (\textit{tabs, blanks, comments})
- a key issue is speed
  \[ \Rightarrow \text{use specialized recognizer (as opposed to lex)} \]
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

\[ \text{<sheep noise>} ::= \text{ baa} \]
\[ \text{ | baa } \text{<sheep noise>} \]

This grammar defines the set of noises that a sheep makes under normal circumstances.

The format is called Backus-Naur form (BNF).

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

- \( S \) 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  | <goal> ::= <expr>
2  | <expr> ::= <expr> <op> <term>
3  |     | <term>
4  | <term> ::= number
5  |     | id
6  | <op> ::= +
7  |     | -
```

This grammar defines simple expressions with addition and subtraction over the tokens id and number

\[
S = <goal> \\
T = \text{number}, \text{id}, +, - \\
N = <goal>, <expr>, <term>, <op> \\
P = 1, 2, 3, 4, 5, 6, 7
\]
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>1</td>
<td>&lt;goal&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;expr&gt; &lt;op&gt; &lt;term&gt;</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.
Front end

A parse can be represented by a tree called a parse or syntax tree

```
<id: x>

<id: y>

<term + <num: 2> - <expr expr op term>

<expr expr op term>

expr

goal
```

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

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 → front end → IR → middle end → IR → back end → 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
Compiler example
## Compiler phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lex</strong></td>
<td>Break source file into individual words, or <em>tokens</em></td>
</tr>
<tr>
<td><strong>Parse</strong></td>
<td>Analyse the phrase structure of program</td>
</tr>
<tr>
<td><strong>Parsing Actions</strong></td>
<td>Build a piece of <em>abstract syntax tree</em> for each phrase</td>
</tr>
<tr>
<td><strong>Semantic Analysis</strong></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><strong>Frame Layout</strong></td>
<td>Place variables, function parameters, etc., into activation records (stack frames) in a machine-dependent way</td>
</tr>
<tr>
<td><strong>Translate</strong></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><strong>Canonicalize</strong></td>
<td>Hoist side effects out of expressions, and clean up conditional branches, for convenience of later phases</td>
</tr>
<tr>
<td><strong>Instruction Selection</strong></td>
<td>Group IR-tree nodes into clumps that correspond to actions of target-machine instructions</td>
</tr>
<tr>
<td><strong>Control Flow Analysis</strong></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><strong>Data Flow Analysis</strong></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><strong>Register Allocation</strong></td>
<td>Choose registers for variables and temporary values; variables not simultaneously live can share same register</td>
</tr>
<tr>
<td><strong>Code Emission</strong></td>
<td>Replace temporary names in each machine instruction with registers</td>
</tr>
</tbody>
</table>
A straight-line programming language

- A straight-line programming language (no loops or conditionals):

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

- e.g.,

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

  prints:
  
  \[8 \quad 7\]
  \[80\]
This is a convenient internal representation for a compiler to use.
Java classes for trees

abstract class Stm {
}
class CompoundStmt extends Stm
Stm stm1, stm2;
CompoundStmt(Stm s1, Stm s2)
{ stm1=s1; stm2=s2; }
}
class AssignStmt extends Stm {
String id; Exp exp;
AssignStmt(String i, Exp e)
{ id=i; exp=e; }
}
class PrintStmt extends Stm {
ExpList exprs;
PrintStmt(ExpList e)
{ exprs=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 stmt; Exp exp;
EseqExp(Stm s, Exp e)
{ stmt=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;}
}