CSE 401 Introduction to Compiler Construction

Ruth Anderson & Mark Roberts
Winter 2008

1

Today's Outline

- Administrative Info
- Overview of the Course

2

CSE 401: Intro to Compiler Construction

Goals
- Learn principles and practice of language translation
  - Bring together theory and pragmatics of previous classes
  - Understand compile-time vs run-time processing
- Study interactions among
  - Language features
  - Implementation efficiency
  - Compiler complexity
  - Architectural features
- Gain more experience with OO design
- Gain more experience with working in a team
- Gain experience working with SW someone else wrote

3

Course Info

- **Prerequisites**: 303, 322, 326, 341, 378
- **Text**: *Engineering a Compiler*, Cooper and Torczon, Morgan-Kaufmann 2004
- **Course Web** is the place to look for materials:
  - Lecture Slides
  - Archive of course mailing list
  - Message Board
  - Homework and Project assignments

4

Staff

- Instructors
  - Ruth Anderson (rea@cs.washington.edu)
  - Mark Roberts (markro@cs.washington.edu)
- Teaching Assistant
  - Jonathan Beall (jibb@cs.washington.edu)

5

CSE 401 E-mail List

- Used for important announcements from **instructors** and **TA.**
- You are responsible for anything sent here.
- If you are registered for the course you will be automatically added to the list.
- Emails will be sent to your @u.washington.edu address.
- Emails will also be archived on the course web page.

6

CSE 401 Discussion Board

- The course will have a Catalyst GoPost message board.
- Students and Instructors can post and reply to posts.
- Please use this!!
- Use:
  - General discussion of class contents
  - Hints and ideas about assignments (but **not** detailed code or solutions)
  - Other topics related to the course

7

Evaluation

- **Grading:**
  - Compiler Project 40%
  - Written Homework 15%
  - Midterm Exam 15%
  - Final Exam 25%
  - Class Participation 5%
- **Late policy:**
  - Each student has **three** late days to use over the course of the quarter.
  - Beyond that, 25% penalty for each calendar day it is late.
  - Assignments are due **at the start of class**, unless otherwise noted.

8

Academic Conduct

- **Written Homework**: to be done *individually*
- **Compiler Project**: to be done with a partner

- Things that are academic **<u>mis</u>**-conduct: (cheating)
  - Sharing solutions, doing work for others, accepting work from others
  - Searching for solutions on the web
  - Consulting or copying solutions to assignments or projects from previous offerings of this or other courses

9

Policy on collaboration

- “Gilligan’s Island” rule:
  - You may discuss problems with your classmates to your heart's content.
  - After you have solved a problem, *discard all written notes* about the solution.
  - Go watch TV for a ½ hour (or more). Preferably *Gilligan's Island*.
  - *Then* write your solution.

10

Homework for Today!!

1) **Reading for this week:** (in Cooper & Torczon) Chapter 1 (all), 2.1-2.4
2) **Information Sheet**: Bring to lecture by Friday (1/11)
3) **Homework #1 (Due 1/16):** See course web page.
4) **Compiler Project:** See course web page.
   1) **Read Project Overview**
   2) **Read Project #1 Description**
   3) **Project Partners (Due 1/16)**

11

UNIVERSITY of VIRGINIA

Ruth Anderson

- Grad Student at UW (Programming Languages, Compilers, Parallel Computing)
- Taught Computer Science at the University of Virginia for 5 years
- Grad Student at UW (Educational Technology, Pen Computing)
- Defended my PhD last fall

12

Mark Roberts

- BS Math at UW
- MS Computer Science at UCLA
- Worked over 30 years building compilers and related development tools
- Last 19 years at Microsoft in a variety of positions:
  - Development manager of compiler backend team
  - Development manager of Visual Basic for Applications (VBA)
  - Manager of Binary Optimization Group
- Card carrying member of ACM and ACE

Bring to Class by Friday:

- Name
- Email address
- Year (1,2,3,4)
- Major
- Hometown
- Interesting Fact or what I did over summer/winter break.

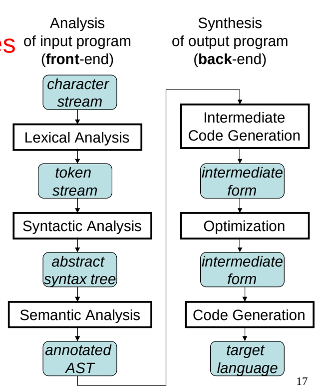
Course Overview

Course Project

- Start with a MiniJava complier in Java … improve it
  - Add:
    - Comments
    - Floating-point values
    - Arrays
    - Static (class) variables
    - For loops
    - Break Statements
    - … And more
  - Completed in stages over the term
  - Strongly encouraged: Work in teams, but only if joint work, not divided work

Grading Basis
•Correctness
•Clarity of design/impl
•Quality of test cases

Compiler Passes



Example Compilation

Sample (extended) MiniJava program: `Factorial.java`

```
// Computes 10! and prints it out
class Factorial {
    public static void main(String[] a) {
         System.out.println(
               new Fac().ComputeFac(10));
     }
}
class Fac {
   // the recursive helper function
   public int ComputeFac(int num) {
      int numAux;
      if (num < 1)
         numAux = 1;
      else numAux = num * this.ComputeFac(num-1);
      return numAux;
   }
}
```

First Step: Lexical Analysis

“Scanning”, “tokenizing”

Read in characters, clump into tokens

– strip out whitespace & comments in the process

19

Specifying tokens: Regular Expressions

Example:

Ident ::= Letter AlphaNum*
Integer ::= Digit+
AlphaNum ::= Letter | Digit
Letter ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
Digit ::= '0' | ... | '9'

20

Second Step: Syntactic Analysis

“Parsing” -- Read in tokens, turn into a tree based on syntactic structure

– report any errors in syntax

21

Specifying Syntax: Context-free Grammars

EBNF is a popular notation for CFG’s

Example:

```
Stmt ::= if (Expr ) Stmt [else Stmt]
    | while (Expr ) Stmt
    | ID = Expr;
    | ...
Expr ::= Expr + Expr | Expr < Expr | ...
    | ! Expr
    | Expr . ID ( [Expr {, Expr}] )
    | ID
    | Integer
    | (Expr)
    | ...
```

EBNF specifies *concrete syntax* of language; parser constructs tree of the *abstract syntax* of the language

22

Third Step: Semantic Analysis

“Name resolution and type checking”

- Given AST:
  - figure out what declaration each name refers to
  - perform type checking and other static consistency checks
- Key data structure: symbol table
  - maps names to info about name derived from declaration
  - tree of symbol tables corresponding to nesting of scopes
- Semantic analysis steps:
  1. Process each scope, top down
  2. Process declarations in each scope into symbol table for scope
  3. Process body of each scope in context of symbol table

23

Fourth Step: Intermediate Code Gen

- Given annotated AST & symbol tables, translate into lower-level intermediate code
- Intermediate code is a separate language
  - Source-language independent
  - Target-machine independent
- Intermediate code is simple and regular
  - Good representation for doing optimizations

Might be a reasonable target language itself, e.g. Java bytecode

24

Example

```
Int Fac.ComputeFac(*? this, int num) {
  int t1, numAux, t8, t3, t7, t2, t6, t0;
  t0 := 1;
  t1 := num < t0;
  ifnonzero t1 goto L0;
  t2 := 1;
  t3 := num - t2;
  t6 := Fac.ComputeFac(this, t3);
  t7 := num * t6;
  numAux := t7;
  goto L2;
label L0;
  t8 := 1;
  numAux := t8
label L2;
  return numAux
}
```

25

Fifth Step: Optimization

Identify inefficiencies in intermediate or target code

Replace with equivalent but better sequences

- equivalent => "has the same externally visible behavior"

Target-independent optimizations best done on IL code

Target-dependent optimizations best done on target code

"Optimize" overly optimistic

- Optimize => "usually improve"

Scope of study for optimizations:

- Peephole, local, global (intraprocedural) and interprocedural
- Larger scope => better optimization but more cost and complexity

26

Sixth Step: Target Machine Code Gen

Translate intermediate code into target code

- Need to do:
  - Instruction selection: choose target instructions for (subsequences) of IR instructions
  - Register allocation: allocate IR code variables to registers, spilling to memory when necessary
  - Compute layout of each procedures stack frames and other runtime data structures
  - Emit target code

27