Why study compilers?

• After all, few developers build compilers ...
  BUT ...
  – Many (good) developers build translators
    A key tool in the solutions toolbox
  – And besides, compiler construction is
    • A case study of successful interplay between CS
      theory and systems
    • A case study in the development of a standard
      application software architecture

Two Myths of CS Research

• The pipeline myth:
  First there was theory, and it was good and then lesser creatures learned to apply it

• The vacuum myth:
  Systems researchers developed XXX on their own while theoreticians wrote irrelevant papers
In compiler construction, theory followed practice but utterly transformed it

Language Paleontology

• The history of compiler techniques is readable in the bones of old languages
  – Fortran IV, BASIC: Lexical analysis mixed with parsing. Algol 60 separates them.
    • DO 20 I = 30, 5 is an assignment
    • DO 20 I = 30, 5 is a loop head
  – K&R C: Primitive register allocation, programmer hints (“register” declarations) necessary for decent code;
  – Java: Garbage collection finally reaches the big time, after 30 years of experience and improvement
Course Goals

- Understand (a bit of) the relevant theory
  - Lexical analysis, parsing, attribution & analysis, code improvement and generation
- Understand how compilers work
  - Standard architecture & basic techniques, including run-time organization
- Learn to use compiler construction tools
  - At least lexer & parser generators

In 10 weeks? How?

- Tight schedule for theory + project, but a complete “end-to-end” project is essential
- We can cope with:
  - Small, frequent assignments
  - Heavy use of tools (at least for parsing)
  - Light touch on theory
  - Immediate start
  - Optional: 2-3 person teams

Project Stages

- (1) Lexical analysis (using JFlex)
- (2) Parser (using CUP)
- (3) Abstract syntax & symbol table
- (4) Static semantics (type checker)
- (5) Code generator
  - Your choice: JVM byte code, assembly language, Ansi C, or your own interpreter

Standard Compiler Architecture

1. Text stream
2. Lexical analysis (“scanner”)
3. Syntax analysis (“parser”)
4. Attribution (type checking, static semantics)
5. Code generation

- i = i + 1;
- if (i > lim) {
- text stream
- token stream
- object code
Ground Rules

• Collaboration on projects (only)
  – Within team: Unlimited, but documented
  – Between teams:
    • Discussion is ok, code sharing is not
    • If you aren’t sure where the line is, ask me
• Project Grading: Mainly by test cases
  – Cases you provide: show us what works
  – Cases we provide in advance
  – Additional cases we make up

Cheating Policy

• Undocumented borrowing of ideas or code is plagiarism
  – Your best defense is to ask in advance and to carefully document what is taken from elsewhere.
• Cheating will be reported and punished
  – Failure, or lower grade, or ...
  – Following UO student conduct code guidelines

Course Grading

• 50% project total, in phases [+/- 10%]
  – with special weight on final phase: How much of the Cool language is correctly implemented
• 50% midterm and final exams [+/- 10%]
• 461 versus 561
  – Grads may have different or extra exam questions
• No fixed distribution of grades

Standard Compiler Architecture

1. Text stream
   \[ i = i + 1; \]
   \[ \text{if} \ (i > \text{lim}) \{ \]

2. Token stream
   \[ i = i + 1; \]
   \[ ; \text{if} \ (i > \]

3. Syntax analysis
   \[ \text{attribute (type checking, static semantics)} \]

4. Code generation
   \[ \text{ld } R0,28(R1); \]
   \[ i \text{ldc } R1,1 \]
   \[ \text{add } R0,R1,R0 \]
   \[ \text{st } R0,28(R12) \]
Front / Back Ends

- Division into front/back standard since 70s
  - with further division into lexical, syntactic analysis and application of theory-based tools
- Back end organization not yet standard
  - theory available since 80s, tools appearing since 90s, but far from standard (as witness our textbook)
- Really three parts: Front (analysis), Middle (analysis & transformation), Back (transformation and interpretation)

Interpreter Structure (typical)

Lexing & Parsing

- Lexical analyzer reads text, produces token stream
  - Theory base: regular languages, finite automata
- Parser analyzes grammatical structure
  - Theory base: context-free languages, LL or LR parsing
Historical Note: Parsing

• Modern organization of lexer, parser depends on language design
  • Requires division of language into lexical and syntactic structure, as pioneered in Algol
  • FORTRAN, BASIC defy lexical analysis; C requires hacks for context-sensitive lexical analysis

• Theory => Parsing tools => Language design
  – Modern languages are designed to be parsed!

“Static semantic” analysis

• Type checking, overload analysis, attribution
  – Syntactic analyses that are not (efficiently) expressible in a context-free grammar

• Theory: Type systems (inference rules), attribute grammars, data flow analysis
  – Tools exist but are little used. Most implementations are hand-coded and loosely follow theory (as will yours).

Back End Tasks (Translation)

• Assignment of run-time structures
  Frame layout, register allocation
  – Intimately tied to libraries (e.g., garbage collector)

• Instruction choice and emission

• Theory: Many bits and pieces
  – Register allocation by graph coloring
  – Tree-rewriting for code generation

Summary: Theory and Tools

• Front end: stable theory, robust tools
  – Lexical analyzer generators (regular languages)
  – Parser generators (context-free languages)

• Middle and back end: maturing theory, tools in flux
  – Attribute grammars
  – Data flow analysis / abstract interpretation
  – Register allocation by graph coloring
  – Code generation by rewriting (tree grammars)

• Run-time structures
  – Lots of knowledge, but no theory-supported tools