

CS134b: Compilers

- Course outline
 - Intro
 - OCaml tutorial
 - Compiling
 - Front-end (lexing, parsing, AST)
 - Middle-end (intermediate languages, program analysis and optimization)
 - Back-end (code generation)



Administrivia

- Instructor: Jason Hickey, jyh@cs.caltech.edu
 - Office hours: MW 3-4pm, 260JRG
- TAs:
 - Justin Smith, justins@chaos2.org
 - Adam Granicz, granicz@cs.caltech.edu
 - Suleyman Gokyigit, sg@cs.caltech.edu
- Course times
 - Lectures: MWF 2-3pm, 74JRG
 - Recitations: TBD



Online info

- Course web site
 - <http://www.cs.caltech.edu/cs134/cs134b/>
- Mailing lists (wait a few days until these are online)
 - Q&A bulletin board (note these are at metaprl.org)
 - cs134-labs@metaprl.org
 - CVS update messages
 - cs134-cvs@metaprl.org
 - Private correspondence
 - cs134-admin@metaprl.org



Course info

- Course books
 - Appel, "Modern Compiler Implementation in ML"
 - Hickey, "An Introduction to OCaml" (rough draft on the web site)
- Web site
 - Bulletin board
 - OCaml 3.04 documentation, *toploop*, *compiler*
 - Intel x86 architecture manual
- Prerequisites
 - CS20
 - CS134a is not required



Grading

- No exams!
 - 5 labs:
 - OCaml exercise: 10%
 - Front-end: 20%
 - Semantic analysis: 20%
 - Middle-end: 20%
 - Code generation: 30%
 - Extra credit, optimizations (fractions of 1 lab)
 - Dead-code elimination: 10%
 - Inlining: 10%
 - Strength reduction: 10%
 - Alias analysis: 20%
 - Hoisting (code motion): 20%



Where we left off

- Modern OS, like Spin, Vino, which are *extensible* kernels
 - Spin extensions are written in Modula-3; all programs in Modula-3 are safe
 - Safety of Vino extensions is guaranteed by software fault isolation
 - Monolithic, general purpose kernels do not satisfy the need for new features and performance
 - Special-purpose kernels need help from the compiler
 - Safety
 - Performance
- Distribution of resources (Amoeba)



Need for new languages is ubiquitous

- Distributed systems: can we make resource distribution transparent and fault-tolerant?
- Graphics: can the compiler help build *models*?
- Simulation: can the compiler help guarantee *predictive simulation*?
- Aerospace: can the compiler help guarantee the *correctness* of the code?
- Devices: can we squeeze Linux onto a watch?



Programming languages and compilers

- A *programming language* defines a *syntax* and *semantics* of valid programs
- An *interpreter* defines a program *evaluator*
- A *compiler* defines a translation from one programming language to another, usually so that the program can be efficiently interpreted
 - C -> machine code
 - Java -> Java byte code
 - Java -> machine code (this is what we'll do)
 - Modula-3 -> C
 - ...



Programming Languages

- Webster: "language, *n.* Any means of expressing or communicating, as gestures, signs, animal sounds, etc."
- What are programming languages?
- How many do you know?



The study of programming languages

- Syntax: the words, sentences, and programs in a language
- Semantics: the *meaning* of the words, sentences, and programs
- Elegance: are programs clear and concise?
- What *properties* hold for all valid programs?

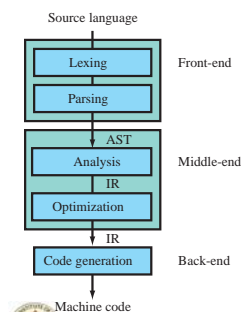


Compilers

- Why do we study compilers (isn't C/C++/Java enough)?
 - You should know how to do it
 - You will encounter many new languages
 - You will probably design new languages
 - The language should fit the task



Traditional compilers



- Front-end produces "abstract syntax tree"
- Middle-end produces "intermediate representation"
- Back-end performs code generation



Languages

- The compiler compiles a *source* language, and produces code in a *target* language
- The compiler may, or may not, be written in the *source* language (and evaluated by the *target* interpreter)
- For this class, we will be compiling a subset of Java
- The compiler language will be OCaml
- The next few lectures will be about OCaml



OCaml

- OCaml is a “semi-functional” language
 - *Most code is functional*
 - *Side-effects are permitted*
- Programs are *strongly-typed*
 - *The compiler rejects programs that are ill-typed*
 - *Well-typed programs are safe (they will never seg-fault)*
- Interpreter: *ocaml*
- Compiler: *ocamlc*
- Debugger: *ocamldebug*



Basic types

- int, float, string, bool, unit

