• HW5 is due on Dean’s date, 5pm.
• After this week, drop in office hours 3-4pm on Wednesday, appointments on WASE.
• Final exam
  • Released 9am on May 14, must be submitted by 5pm on May 15th.
  • No time limit. Expected time is 2 hours
  • Open book, open notes, computer allowed. No collaboration.
  • Some programming questions – make sure to have a working OCaml installation.
  • Ask questions using private posts on Piazza
  • Mostly material since the midterm. Topics:
    • Type systems (be comfortable reading inference rules, writing proof trees)
    • Data flow analysis (translate a global specification into local constraints)
    • Register allocation (graph coloring, coalescing)
    • Control flow analysis (dominators, loops, SSA conversion)
Review
Software engineering

• Compilers are large software projects
  • Decompose the problem into lots of small phases, each of which accomplishes one thing
  • E.g., the optimization phase is also a large piece of software – it too is composed of lots of small individual phases

• Many problems do not have a “right” answer: pick a *convention*, document it well, and adhere to it.
  • E.g., calling conventions, pass environment as first argument to a closure, ...
Intermediate representations

- An IR breaks code generation up into two phases. Simpler & easier to implement
- IRs (such as SSA) can drastically simplify optimization
- Makes compiler back-end re-usable
Lexing and parsing

- The lexing phase of a compiler breaks a stream of characters (source text) into a stream of tokens.
- The parsing phase of a compiler takes in a stream of tokens (produced by a lexer), and builds an abstract syntax tree (AST).
Lexing and parsing

- The **lexing** phase of a compiler breaks a stream of characters (source text) into a stream of **tokens**
- The **parsing** phase of a compiler takes in a stream of tokens (produced by a lexer), and builds an abstract syntax tree (AST).
- Lexing and parsing are based on **automata**
  - Lexing: finite automata (DFAs, NFAs)
  - Parsing: (deterministic) pushdown automata
Lexing and parsing

- The **lexing** phase of a compiler breaks a stream of characters (source text) into a stream of **tokens**
- The **parsing** phase of a compiler takes in a stream of tokens (produced by a lexer), and builds an abstract syntax tree (AST).
- Lexing and parsing are based on **automata**
  - Lexing: finite automata (DFAs, NFAs)
  - Parsing: (deterministic) pushdown automata
- Useful tool to have in your toolbox!
  - Parsing useful for programming languages, domain specific languages, custom data formats, ...
  - Lexer generators: lex, flex, ocamllex, jflex
  - Parser generators: Yacc, Bison, ANTLR, menhir
Type Systems

- Specified by *inference rules*

\[
\frac{J_1}{J} \quad \frac{J_2}{J} \quad \cdots \quad \frac{J_n}{J} \quad \text{SIDE-CONDITION}
\]

- **Succinct** way to communicate a *precise* specification
- Pervasive in formal logic and programming language theory. Can be used to specify
  - the semantics of programming languages
  - logics for reasoning about programs
  - program analyses
  - ...

Type theory is a large subject and an active area of research. Close ties to logic (Curry-Howard correspondence: formulas are types, programs are proofs). More in COS 510.
Type Systems

• Specified by inference rules

\[
\frac{J_1 \quad J_2 \quad \cdots \quad J_n}{J} \quad \text{SIDE-CONDITION}
\]

• **Succinct** way to communicate a **precise** specification
• Pervasive in formal logic and programming language theory. Can be used to specify
  • the semantics of programming languages
  • logics for reasoning about programs
  • program analyses
  • ...

• Type theory is a large subject and an active area of research
  • Close ties to logic (Curry-Howard correspondence: formulas are types, programs are proofs)
  • More in COS 510
Dataflow analysis

• Dataflow analysis is an approach to program analysis that unifies the presentation and implementation of many different analyses.
Dataflow analysis

- Dataflow analysis is an approach to program analysis that unifies the presentation and implementation of many different analyses
  - Define a system of inequations \( \{X_i \supseteq R_i\}_{i \in I} \), where “unknowns” \( X_i \) are values in some partially ordered set, and right-hand-sides are monotone expressions over unknowns
Dataflow analysis

Dataflow analysis is an approach to program analysis that unifies the presentation and implementation of many different analyses

- Define a system of inequations \( \{ X_i \supseteq R_i \}_{i \in I} \), where “unknowns” \( X_i \) are values in some partially ordered set, and right-hand-sides are monotone expressions over unknowns
- Solve the system by repeatedly:
  1. Choosing a constraint \( X_j \supseteq R_j \) that is not satisfied
  2. Increasing \( X_j \) so that the constraint is satisfied

until all constraints are satisfied
Dataflow analysis

• Dataflow analysis is an approach to program analysis that unifies the presentation and implementation of many different analyses
  • Define a system of inequations \( \{ X_i \sqsupseteq R_i \}_{i \in I} \), where “unknowns” \( X_i \) are values in some partially ordered set, and right-hand-sides are monotone expressions over unknowns
  • Solve the system by repeatedly:
    1. Choosing a constraint \( X_j \sqsupseteq R_j \) that is not satisfied
    2. Increasing \( X_j \) so that the constraint is satisfied
   until all constraints are satisfied

• Idea: can sometimes transform a global specification into a system of local constraints, which can be solved iteratively
LL parsing revisited

- LL(1) parser can be constructed from nullable, first, and follow, which have the following global specifications
  - Fix a grammar $G = (N, \Sigma, R, S)$
  - For any word $\gamma \in (N \cup \Sigma)^*$, define $\text{first}(\gamma) = \{ a \in \Sigma : \gamma \Rightarrow^* aw \}$
  - For any word $\gamma \in (N \cup \Sigma)^*$, say that $\gamma$ is nullable if $\gamma \Rightarrow^* \epsilon$
  - For any non-terminal $A$, define $\text{follow}(A) = \{ a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma A a \gamma' \}$
LL parsing revisited

LL(1) parser can be constructed from **nullable**, **first**, and **follow**, which have the following global specifications:

- Fix a grammar $G = (N, \Sigma, R, S)$
- For any word $\gamma \in (N \cup \Sigma)^*$, define $\text{first}(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^{*} aw\}$
- For any word $\gamma \in (N \cup \Sigma)^*$, say that $\gamma$ is **nullable** if $\gamma \Rightarrow^{*} \epsilon$
- For any non-terminal $A$, define $\text{follow}(A) = \{a \in \Sigma : \exists \gamma, \gamma' : \exists S \Rightarrow \gamma A a \gamma'\}$

**nullable** : $N \rightarrow \{\text{true, false}\}$ (w/ false $\sqsubseteq$ true) is the least function such that

- For each rule $A ::= \gamma_1...\gamma_n$, $\text{nullable}(A) \sqsupseteq \text{nullable}(\gamma_1) \land \cdots \land \text{nullable}(\gamma_1)$
LL parsing revisited

- LL(1) parser can be constructed from nullable, first, and follow, which have the following global specifications
  - Fix a grammar $G = (N, \Sigma, R, S)$
  - For any word $\gamma \in (N \cup \Sigma)^*$, define $\text{first}(\gamma) = \{ a \in \Sigma : \gamma \Rightarrow^* aw \}$
  - For any word $\gamma \in (N \cup \Sigma)^*$, say that $\gamma$ is nullable if $\gamma \Rightarrow^* \epsilon$
  - For any non-terminal $A$, define $\text{follow}(A) = \{ a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma A a \gamma' \}$
- **nullable** : $N \rightarrow \{\text{true}, \text{false}\}$ (w/ false $\sqsubseteq$ true) is the least function such that
  - For each rule $A ::= \gamma_1...\gamma_n$, $\text{nullable}(A) \sqsupseteq \text{nullable}(\gamma_1) \land \cdots \land \text{nullable}(\gamma_1)$
- **first** is the smallest function such that
  - For each $a \in \Sigma$, $\text{first}(a) = \{a\}$
  - For each $A ::= \gamma_1...\gamma_i...\gamma_n \in R$, with $\gamma_1, ..., \gamma_{i-1}$ nullable, $\text{first}(A) \supseteq \text{first}(\gamma_i)$
LL parsing revisited

• LL(1) parser can be constructed from nullable, first, and follow, which have the following global specifications
  • Fix a grammar $G = (N, \Sigma, R, S)$
  • For any word $\gamma \in (N \cup \Sigma)^*$, define $\text{first}(\gamma) = \{a \in \Sigma : \gamma \Rightarrow^* aw\}$
  • For any word $\gamma \in (N \cup \Sigma)^*$, say that $\gamma$ is nullable if $\gamma \Rightarrow^* \epsilon$
  • For any non-terminal $A$, define $\text{follow}(A) = \{a \in \Sigma : \exists \gamma, \gamma'. S \Rightarrow \gamma Aa\gamma'\}$

**nullable** : $N \rightarrow \{\text{true, false}\}$ (w/ $\text{false} \sqsubseteq \text{true}$) is the least function such that
  • For each rule $A ::= \gamma_1...\gamma_n$, $\text{nullable}(A) \sqsupseteq \text{nullable}(\gamma_1) \land \cdots \land \text{nullable}(\gamma_1)$

**first** is the smallest function such that
  • For each $a \in \Sigma$, $\text{first}(a) = \{a\}$
  • For each $A ::= \gamma_1...\gamma_i...\gamma_n \in R$, with $\gamma_1, ..., \gamma_{i-1}$ nullable, $\text{first}(A) \supseteq \text{first}(\gamma_i)$

**follow** is the smallest function such that
  • For each $A ::= \gamma_1...\gamma_i...\gamma_n \in R$, with $\gamma_{i+1}, ..., \gamma_n$ nullable, $\text{follow}(\gamma_i) \supseteq \text{follow}(A)$
  • For each $A ::= \gamma_1...\gamma_i...\gamma_j...\gamma_n \in R$, with $\gamma_{i+1}, ..., \gamma_{j-1}$ nullable, $\text{follow}(\gamma_i) \supseteq \text{first}(A)$
Current research
Conferences

- Programming Language Design and Implementation (PLDI)
- Principles of Programming Languages (POPL)
- Object Oriented Programming Systems, Languages & Applications (OOPSLA)
- Principles and Practice of Parallel Programming (PPoPP)
- Code Generation and Optimization (CGO)
- Compiler Construction (CC)
- International Conference on Functional Programming (ICFP)
- European Symposium on Programming (ESOP)
- Architectural Support for Programming Languages and Operating Systems (ASPLOS)
The job of a compiler is to translate from the syntax of one language to another, but preserve the semantics.
The job of a compiler is to translate from the syntax of one language to another, but preserve the semantics.

- Compiler correctness is critical
  - Trustworthiness of every component built in a compiled language depends on trustworthiness of the compiler
The job of a compiler is to translate from the syntax of one language to another, but preserve the **semantics**.

- Compiler correctness is **critical**
  - Trustworthiness of every component built in a compiled language depends on trustworthiness of the compiler
- Compilers tend to be well-engineered and well-tested, but that does not mean bug-free
Bug-finding in compilers

- CSmith\(^1\): randomized differential testing of C compilers
  - Randomly generate a C program *without undefined behavior*
    - Integrates program analysis to find interesting test cases
  - Compile with several different compilers
  - Compare the results
- Over 3 years found several real bugs
  - 79 bugs in GCC (25 maximum-priority/release-blocking)
  - 202 bugs in LLVM

\(^1\)Yang et al. Finding and Understanding Bugs in C Compilers, PLDI 2011
Verified compilation

- **CompCert**: (Xavier Leroy, primary developer of OCaml)
  - Optimizing C compiler, implemented and **proved correct** in the Coq proof assistant
  - Coq proof assistant an (essentially) implementation of a sophisticated type system (CoIC)

The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent

Verified compilation

- **CompCert**: (Xavier Leroy, primary developer of OCaml)
  - Optimizing C compiler, implemented and **proved correct** in the Coq proof assistant
  - Coq proof assistant an (essentially) implementation of a sophisticated type system (CoIC)
  
  *The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent*
  
  - Yang et al. *Finding and Understanding Bugs in C Compilers, 2011*

- At Princeton: **CertiCoq** (Andrew Appel)
  - CompCert is implemented the proof assistant Coq... but why should we trust the Coq compiler?
  - CertiCoq is an optimizing compiler for Coq, implemented and verified in Coq.
Automatic parallelization

- Moore's law: processor advances double speed every 18 months
Automatic parallelization

- Moore's law: processor advances double speed every 18 months
- Moore's law ended in 2006 for single-threaded applications
  - Started to hit fundamental limits in how small transistors can be
- Processor manufacturers shifted to *multi-core* processors
Automatic parallelization

- Moore’s law: processor advances double speed every 18 months
- Moore’s law ended in 2006 for single-threaded applications
  - Started to hit fundamental limits in how small transistors can be
- Processor manufacturers shifted to multi-core processors
- Need new compiler technology to take advantage of multi-core – automatically find and exploit opportunities for parallel execution
- At Princeton: David August’s parallelization project
Program synthesis

- **Verification**: Given a program and a specification, prove that the program satisfies the specification

- **Synthesis**: Given a specification, find a program that satisfies the specification

- **Superoptimization**: find the least costly sequence of instructions that is equivalent to a given sequence

- Specification is a program, but used as a black box

- Solved by exhaustive search

- Symbolic search (SAT, SMT), stochastic search (Markov-Chain Monte Carlo sampling)

- At Princeton: Synthesizing Lenses (David Walker), synthesis via logical games (Z)
Program synthesis

• **Verification**: Given a program and a specification, prove that the program satisfies the specification
• **Synthesis**: Given a specification, find a program that satisfies the specification
Program synthesis

- **Verification**: Given a program and a specification, prove that the program satisfies the specification
- **Synthesis**: Given a specification, find a program that satisfies the specification
- **Superoptimization**: Find the least costly sequence of instructions that is equivalent to a given sequence
  - Specification is a program, but used as a black box
  - Solved by exhaustive search
  - Symbolic search (SAT, SMT), stochastic search (Markov-Chain Monte Carlo sampling)
Program synthesis

- **Verification**: Given a program and a specification, prove that the program satisfies the specification
- **Synthesis**: Given a specification, find a program that satisfies the specification
- **Superoptimization**: Find the least costly sequence of instructions that is equivalent to a given sequence
  - Specification is a program, but used as a black box
  - Solved by exhaustive search
  - Symbolic search (SAT, SMT), stochastic search (Markov-Chain Monte Carlo sampling)
- At Princeton: Synthesizing Lenses (David Walker), synthesis via logical games (Z)
Program analysis

• The goal of a program analysis is to answer questions about the run-time behavior of software
  • In compilers: data flow analysis, control flow analysis
  • Typical goal: determine whether an optimization is safe
Program analysis

- The goal of a program analysis is to answer questions about the run-time behavior of software
  - In compilers: data flow analysis, control flow analysis
  - Typical goal: determine whether an optimization is safe
- Research in program analysis has shifted to more sophisticated properties
  - Numerical analyses – e.g., find geometric regions that contain reachable values for integer variables. Can be used to verify absence of buffer overflows, divide-by-zero, ....
  - Shape analyses – determine whether a data structure in the heap is a list, a tree, a graph, ... Can be used to verify memory safety.
  - Resource analyses – e.g., find a conservative upper bound on the run-time complexity of a loop. Can be used to find timing side-channel attacks.

Industrial program analysis

- Static Driver Verifier (Microsoft): finds bugs in device driver code
- Infer (Facebook): proves memory safety & finds race conditions
- Astrée (AbsInt): static analyzer for safety-critical embedded code (e.g., automotive & aerospace applications)
- Several commercial static analyzers: Codesonar, Coverity, PVS-Studio, Fortify, ...
Program analysis

- The goal of a program analysis is to answer questions about the run-time behavior of software
  - In compilers: data flow analysis, control flow analysis
  - Typical goal: determine whether an optimization is safe
- Research in program analysis has shifted to more sophisticated properties
  - Numerical analyses – e.g., find geometric regions that contain reachable values for integer variables. Can be used to verify absence of buffer overflows, divide-by-zero, ....
  - Shape analyses – determine whether a data structure in the heap is a list, a tree, a graph, ... Can be used to verify memory safety.
  - Resource analyses – e.g., find a conservative upper bound on the run-time complexity of a loop. Can be used to find timing side-channel attacks.
- Industrial program analysis
  - Static Driver Verifier (Microsoft): finds bugs in device driver code
  - Infer (Facebook): proves memory safety & finds race conditions
  - Astrée (AbsInt): static analyzer for safety-critical embedded code (e.g., automotive & aerospace applications)
  - Several commercial static analyzers: Codesonar, Coverity, PVS-Studio, Fortify, ...
Program analysis at Princeton

- Synthesis, Learning, and Verification project (Aarti Gupta)
  - Idea: learn program invariants, termination arguments, etc from data

- Compositional program analysis
  - Program analyses typically work by propagating information forwards through a program
  - Compositional analysis: Analyze the program by breaking it into parts, analyzing each part, and then combining the results

- Understandable program analysis
  - Program analyzers rely on heuristics for predicting program behavior. Can be brittle and have unexpected behavior.
  - Software developers should be able to predict how a change in their program affects the results of an analysis
Program analysis at Princeton

• Synthesis, Learning, and Verification project (Aarti Gupta)
  • Idea: learn program invariants, termination arguments, etc from data

• My work:
  • *Compositional* program analysis
    • Program analyses typically work by propagating information forwards through a program
    • Compositional analysis: Analyze the program by breaking it into parts, analyzing each part, and then combining the results
  • *Understandable* program analysis
    • Program analyzers rely on *heuristics* for predicting program behavior. Can be brittle and have unexpected behavior.
    • Software developers should be able to predict how a change in their program affects the results of an analysis
What next?

- COS 375: Computer Architecture and Organization
- COS 326: Functional Programming
- COS 510: Programming Languages
- COS 516: Automated Reasoning about Software
Thanks!