Use Cases for Compile-Time Analysis (1/2)

- Traditional compilation (C, C++, Fortran)
  - Analysis in the compiler for correctness & performance

- Modern compilation (Java w/ bytecode, C# w/ CIL)
  - Analysis in the translator (e.g., javac)
  - Lightweight analysis in the just-in-time (JIT) compiler inside the virtual machine
Use Cases for Compile-Time Analysis (2/2)

• Software development environments
  – E.g., in Eclipse: finds code smells and potential defects; performs code refactoring

• Software verification/checking tools
  – Prove the absence of certain categories of defects

• Testing tools
  – E.g., for regression testing – which tests do not need to be rerun after some changes to the program?

• Also: comprehension tools, debugging tools (after failure), performance analysis tools, etc.

• More generally, static analysis (vs. dynamic analysis)
Inside a Traditional Compiler: Front End

• Lexical analyzer (aka scanner)
  – Converts ASCII or Unicode to a stream of tokens

• Syntax analyzer (aka parser)
  – Creates a parse tree from the token stream

• Semantic analyzer
  – Type checking and conversions; other semantic checks
  – Some compile-time analyses done here, on the AST

• Generator of intermediate code
  – A parse tree is too high-level for code generation & optimization
  – Create lower-level intermediate representation (IR): e.g., three-address code
Inside the Compiler: Middle Part

• Compile-time analysis of intermediate code
  – Additional IRs: control-flow graph (CFG), static single-assignment form (SSA), def-use graph, etc.
  – Control-flow analysis, data-flow analysis, pointer analysis, side-effect analysis, polyhedral analysis, ...

• Machine-independent optimization of intermediate code: better three-address code
  – Copy propagation, dead code elimination, code motion, constant propagation, redundancy elimination, parallelization, data locality optimizations, ...

• Currently, this is where most of compiler research is focused
Three-Address Code

- ASTs are high-level IRs
  - Close to the source language
  - Suitable for tasks such as type checking
- Three-address code is a lower-level IR
  - Closer to the target language (i.e., assembly code)
  - Suitable for tasks such as code generation/optimization
- Basic ideas
  - A small number of simple instructions: e.g. \( x = y \ op \ z \)
  - A number of compiler-generated temporary variables
    \( a = b + c + d; \) in source code \( \rightarrow t = b + c; a = t + d; \)
  - Simple flow of control – conditional and unconditional jumps to labeled statements
Important Note

• The choice of the program representation on which to perform analysis is critical
  – E.g. if you are writing an Eclipse plug-in, you have access to the AST, but not to a lower-level IR
    • Plus, the results of the analysis are useful for ASTs (e.g., code smells reported to the programmer)

• In a compiler, we usually prefer to have access to a lower-level IR, since the analyses and transformations are easier
  – In this course, we will focus on this scenario
Addresses and Instructions

• “Address”: a program variable, a constant, or a compiler-generated temporary variable

• Instructions
  – \( x = y \text{ op } z \): binary operator \( op \); \( y \) and \( z \) are variables, temporaries, or constants; \( x \) is a variable or a temporary
  – \( x = \text{ op } y \): unary operator \( op \); \( y \) is a variable, a temporary, or a constant; \( x \) is a variable or a temporary
  – \( x = y \): copy instruction; \( y \) is a variable, a temporary, or a constant; \( x \) is a variable or a temporary
  – Arrays, flow-of-control
  – Each instruction contains at most three “addresses”
    • Thus, three-address code
Simple Examples

\( x = y \) produces one three-address instruction
  Left: a pointer to the symbol table entry for \( x \)
  Right: a pointer to the symbol table entry for \( y \)
  For convenience, we will write this as \( x = y \)

\( x = -y \) produces \( t1 = -y; x = t1; \)
\( x = y + z \) produces \( t1 = y + z; x = t1; \)
\( x = y + z + w \) produces \( t1 = y + z; t2 = t1 + w; x = t2; \)
\( x = y + -z \) produces \( t1 = -z; t2 = y + t1; x = t2; \)
Flow of Control

• Three-address instructions
  – goto L: unconditional jump to the three-address instruction with label L
  – if (x relop y) goto L: x and y are variables, temporaries, or constants; relop ∈ { <, <=, ==, !=, >, >= }  

• The labels are symbolic names
More Examples

- Possible three-address code: two versions
  - Example: if (x < 100 || x > 200 && x != y) x = 0;
    if (x < 100) goto L2;
goto L3;
L3: if (x > 200) goto L4;
goto L1;
L4: if (x != y) goto L2;
goto L1;
L2: x = 0;
L1: ...

- Example: if (x < 100) goto L2;
goto L3;
if (x <= 200) goto L1;
if (x == y) goto L1;
L2: x = 0;
L1: ...;
Main Topics

• **Control-flow analysis**: what sequences of instructions could be executed at run time?
  – Infinite number of sequences \( \rightarrow \) need finite static representation (control-flow graph)

• **Data-flow analysis**: what are the effects of these instruction sequences on the state of the program?
  – Infinite (or very large) sets of possible states \( \rightarrow \) need finite/small abstractions, often with loss of precision
  – Key technical challenges: abstractions must be
    • **correct** (depending on the client)
    • **precise and efficient-to-compute**

• **Code transformations**: enabled by analysis