CS322 Languages and Compiler Design II
Winter 2011
Lecture 1

**COMPILER BACK-END TASKS**

Starting from type-checked abstract syntax for source language...
- Simplify expressions and statements into flat goto/label form
- Fix location of variables & temps in memory & registers
- Generate machine instructions
- Manage machine resources
- Interact with O/S, runtime system

...generate machine code for specific target architecture. Or, build interpreter for higher-level language features.

**Topics**
- Semantics
- Interpreters
- Runtime Organization
- Intermediate Code Generation
- Machine Code Generation
- Optimization

**Project**
- Build **fab** Interpreter
- Complete **fab** Compiler for X86-64

**Themes**
- Mapping from high-level to low-level
- Implementing resource management
- Integration with OS and hardware environment
- Syntax-directed techniques

---

**Translator Structure**

Source code

- Lexical Analysis
  - Tokens
  - Parsing
  - Abstract syntax

- Type-checking
  - Abstract syntax
  - Intermediate Code Generation

- Intermediate Code
  - Optimization
  - Machine Code Generation

- Machine code

**Language Design Issues**

- Syntax
- Data Types
- Control Structures
  - machine-independent
  - machine-dependent
- Runtime Models

---

CS322 Languages and Compiler Design II
Winter 2011
Lecture 1
**Three-address code** - A typical linear IR

Generate list of “instructions”

- Each has an operator, up to 2 args, and up to 1 result
- Instructions can be labeled
- Operands are names for locations in some abstract memory (e.g., symbol table entries)

Examples of instructions:

- **A := B** copy
- **A = B op C** binary ops
- **A = op B** unary ops
- **goto L** jumps
- **if A relop B goto L** conditional jumps
- **param A** procedure call setup
- **call P,N** procedure call
- **return N** procedure return
- **A[I]** array dereference

**Intermediate code generation**

Emit intermediate code/representation (“IR”) from abstract syntax or directly from parser.

Advantages:

- Keeps more of compiler machine-independent
- Facilitates some optimizations

Typical examples:

- Postfix stack code
- Trees or directed acyclic graphs (DAGs)
- Three-address code (quadruples, triples, etc.)

Abstracts key features of machine architectures

- e.g., sequential execution, explicit jumps
- but hides details
- e.g., # of registers, style of conditionals, etc.

Many possible levels of IR; some compilers use several
Machine Code Generation

"Read" IR and generate assembly language (symbol or binary).
Must cooperate with IR to define and "enforce" runtime environment.
Must deal with idiosyncrasies of target machine,
• e.g., instruction selection
and perform resource management,
• e.g., register assignment.
Lots of case analysis, especially for complex target architectures.
Can do by hand, but hard.
Tools limited but sometimes useful; mainly based on pattern matching.

Optimization

Improve (don't perfect) code by removing inefficiencies:
• in original program
• introduced by compiler itself
Can operate on source, IR, object code.
Local Improvements
• Example: changing
  \[
  \text{if } c > 7 \text{ goto } L1 \\
  \text{goto } L2
  \]
  \[
  L1: \ldots \\
  L2: \ldots
  \]
to
  \[
  \text{if } c \leq 7 \text{ goto } L2 \\
  L1: \ldots \\
  L2: \ldots
  \]

Optimization (continued)

"Global" Improvements
• Example: changing
  \[
  \text{for (i := 0; i < 1000; i++)} \\
  a[i] := b*c + i;
  \]
to
  \[
  t1 := b \ast c; \\
  \text{for (i = 0; i < 1000; i++)} \\
  a[i] = t1 + i;
  \]
Interprocedural improvements
• Example: Inlining a function
Most of a modern compiler is devoted to optimization.
INTERPRETATION

Simulate execution of program (source, AST, or other IR) on an abstract machine.
Implement abstract machine on a real machine.
Inputs to interpreter are
- Program to be interpreted
- Input to that program
Simpler than compiling and takes no time up front, but interpreted code runs (~10X) more slowly than compiled code.
Much more portable than real machine code (as for Java).
Helps with semantic definition.