Bridging the semantic gap

- MiniJava to MIPS assembly

Intermediate Representations
- why?
  - characteristics

3-address code

Tiger book expression Trees

Tiger book Assem representation

Intermediate Program Representations

AST
- usually language dependent

Intermediate Representation (IR)
- Usually a language independent and target independent representation
- Examples
  - 3-address code
  - RTL used in GCC (like 3-address code)
  - LLVM used in the LLVM compiler (like 3-address code but typed)
  - Microsoft’s Common Intermediate Language (CIL)
  - Java byte code
  - Tree data structure in the MiniJava Compiler (a little different)

AST => IR => target code

A Low-Level IR: 3-address code

3-address code
- Linear representation
- Typically language-independent
- Nearly corresponds to machine instructions

Example operations
- Assignment: \( x = y \)
- Unary op: \( x = \text{op} \ y \)
- Binary op: \( x = y \ 	ext{op} \ z \)
- Address of: \( p = &y \)
- Load: \( x = \text{*p} \)
- Store: \( p = y \)
- Pass param: \( \text{param} \ x, \text{L} \)
- Call: \( y = \text{call} \ p, \ L \)
- Branch: \( \text{goto} \ L \)
- Cbranch: \( \text{if (x==3) goto} \ L \)
**IR Code Generation**

**Goal**
- Transforms AST into low-level *intermediate representation* (IR)

**Simplifies the IR**
- Removes high-level control structures: `for`, `while`, `do`, `switch`
- Removes high-level data structures: arrays, structs, unions, enums

**Results in assembly-like code**
- Semantic lowering
- Control-flow expressed in terms of “gotos”
- Each expression is very simple (three-address code)
  
  *e.g.*  
  
  \[
  x = a \ast b \ast c \quad t = a \ast b \quad x = t \ast c
  \]

---

**Compiling Control Flow**

**Switch statements**
- Convert `switch` into low-level IR
  
  *e.g.*  
  
  \[
  \text{switch} \: (c) \: \{ \\
  \quad \text{case} \: 0: \: f(); \\
  \quad \quad \text{break}; \\
  \quad \text{case} \: 1: \: g(); \\
  \quad \quad \text{break}; \\
  \quad \text{case} \: 2: \: h(); \\
  \quad \quad \text{break}; \\
  \}
  \]
  
- Optimizations (depending on size and density of cases)
  - Create a jump table (store branch targets in table)
  - Use binary search

---

**Compiling Arrays**

**Array declaration**
- Store name, size, and type in symbol table

**Array allocation**
- Call `malloc()` or create space on the runtime stack

**Array referencing** (*C* is source code)
- *e.g.*  
  
  \[
  A[i] \quad \rightarrow \quad *(A + i \ast \text{sizeof}(A\_elem))
  \]
  
  \[
  t2 = \text{sizeof}(A\_elem) \\
  t3 = i \ast t2 \\
  t4 = A + t3 \\
  *t4
  \]
Missed Opportunities

3-address code is low level
- many architectures have CISC-like instructions
- the low-level IR might preclude using certain instructions in the ISA
- 6800 example

Desired characteristics of IRs
- should be easy to translate to
- should be easy to translate from to all target machines
- each piece should have simple semantics
- should be able to efficiently and effectively apply program optimizations

MiniJava Compiler Tree Language (Array Example)

x[2] = 42;