CS42/413
Introduction to Compilers
Radu Rugina

Lecture 14: Intermediate Representation
20 Feb 02

Semantic Analysis
- Check errors not detected by lexical or syntax analysis
- Scope errors:
  - Variables not defined
  - Multiple declarations
- Type errors:
  - Assignment of values of different types
  - Invocation of functions with different number of parameters or parameters of incorrect type
  - Incorrect use of return statements

Intermediate Code
- IR = Intermediate Representation
- Allows language-independent, machine-independent optimizations and transformations

What Makes a Good IR?
- Easy to translate from AST
- Easy to translate to assembly
- Narrow interface: small number of node types (instructions)
  - Easy to optimize
  - Easy to retarget
  - AST (40 node types)
  - IR (13 node types)
  - Pentium (200 opcodes)
**Multiple IRs**
- Some optimizations require high-level structure
- Others more appropriate on low-level code

```
AST → IR → Pentium
         → Java bytecode
```

**Machine Optimizations**
- ... some other optimizations take advantage of the features of the target machine
- Machine-specific optimizations

```
AST → HIR → LIR
         → Java bytecode
```

**Next Lectures**
- Next few lectures: intermediate representation
- Optimizations covered later

```
AST → HIR → LIR
         → Java bytecode
```

**Multiple IRs**
- Usually two IRs:
  - High-level IR
    - Language-independent (but closer to language)
  - Low-level IR
    - Machine-independent (but closer to machine)

```
C → HIR → LIR → Pentium
Fortran → HIR → LIR → Java bytecode
Pascal → HIR → LIR → Alpha
```

**Multiple IRs**
- Another benefit: a significant part of the translation from high-level to low-level is
  - Language-independent
  - Machine-independent

```
C → HIR → LIR → Pentium
Fortran → HIR → LIR → Java bytecode
Pascal → HIR → LIR → Alpha
```
High-Level IR

- High-level intermediate representation is essentially the AST
  - Must be expressive for all input languages
- Preserves high-level language constructs
  - Structured control flow: if, while, for, switch, etc.
  - Variables, methods
- Allows high-level optimizations based on properties of source language (e.g. inlining)

Low-Level IR

- Low-level representation is essentially an abstract machine
- Has low-level constructs
  - Unstructured jumps, registers, memory locations
- Allows optimizations specific to these constructs (e.g. register allocation, branch prediction)

Low-Level IR

- Alternatives for low-level IR:
  - Three-address code or quadruples:
    \( a = b \text{ OP } c \)
  - Tree representation (Tiger Book)
  - Stack machine (like Java bytecode)
- Advantages:
  - Three-address code: easier dataflow analysis
  - Tree IR: easier instruction selection
  - Stack machine: easier to generate

Three-Address Code

- In this class: three-address code
  \( a = b \text{ OP } c \)
- Also named quadruples because can be represented as: \( (a, b, c, \text{ OP}) \)
- Has at most three addresses (may have fewer)
- Example:
  \[ a = (b+c)^2; \quad t1 = b + c \]
  \[ t2 = -e \]
  \[ a = t1 * t2 \]

IR Instructions

- Assignment instructions:
  - \( a = b \text{ OP } c \): binary operation
  - Arithmetic: ADD, SUB, MUL, DIV, MOD
  - Logical: AND, OR, XOR
  - Comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - \( a = \text{ OP } b \): unary operation
  - Arithmetic: MINUS, logic NEG
  - \( a = b \): copy instruction
  - \( a = \text{ bad } b \): load instruction
  - \( a = [b] \): store instruction
  - \( [a] = b \): symbolic address

IR Instructions (Ctd)

- Flow of control instructions:
  - label \( L \): label instruction
  - jump \( L \): unconditional jump
  - jump a \( L \): conditional jump
- Function call
  - call \( (a_1, ..., a_n) \)
  - \( a = \text{ call } (a_1, ..., a_n) \)
  - Is an extension to quads
- ... IR describes the Instruction Set of an abstract machine
Temporary Variables

- The operands in the quadruples can be:
  - Program variables
  - Integer constants
  - Temporary variables
- Temporary variables = new locations
  - Use temporary variables to store intermediate values

Arithmetic / Logic Instructions

- Abstract machine supports a variety of different operations
  \[ a = b \text{ OP } c \quad \text{or} \quad a = \text{ OP } b \]
- Arithmetic operations: ADD, SUB, DIV, MUL
- Logic operations: AND, OR, XOR
- Comparisons: EQ, NEQ, LE, LTF, GE, GEQ
- Unary operations: MINUS, NEG

Data Movement

- Copy instruction
  \[ a = b \]
- Models a load/store abstract machine
  \[ a = [b] \quad \text{or} \quad [a] = b \]
- Take symbolic addresses of variables:
  \[ a = \text{addr } b \]

Branch Instructions

- Unconditional jump: go to statement after label L
  \[ \text{jump } L \]
- Conditional jump:
  - Test condition variable a
  - If true, jump to label L
  \[ \text{cjump } a \text{ L} \]

Call Instruction

- Supports function call statements
  \[ \text{call } f(a_1, \ldots, a_n) \]
- ... and function call assignments:
  \[ a = \text{call } f(a_1, \ldots, a_n) \]
- No explicit representation of argument passing, stack frame setup, etc.

Example

\[
\begin{align*}
n &= 0; \\
\text{while } (n < 10) \{ \\
\quad & n = n + 1 \\
\}
\end{align*}
\]

\[
\begin{align*}
n &= 0 \\
\text{label test} \\
t_2 &= n < 10 \\
t_3 &= \text{not } t_2 \\
\text{cjump } & t_3 \text{ end label body} \\
\quad & n = n + 1 \\
\text{jump test} \\
\text{label end}
\end{align*}
\]
Another Example

```c
m = 0;
if (c == 0) {
    m = m + n*n;
} else {
    m = m + n;
}
```

---

How To Translate?

- May have nested language constructs
  - Nested if and while statements
- Need an algorithmic way to translate
- Solution:
  - Start from the AST representation
  - Define translation for each node in the AST
  - Recursively translate nodes in the AST