Where we are

Source code
(character stream)

Token stream

Lexical analysis

regular expressions

Syntactic Analysis

grammars

Abstract syntax tree

Semantic Analysis

static semantics

Abstract syntax tree

Intermediate Code

+ type objects, symbol tables

Intermediate Code Generation

Intermediate Code

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)

Intermediate Code

• Abstract machine code - simpler
• Allows machine-independent code generation, optimization

Pentium

Java bytecode

Alpha

Optimizing compilers

• Goal: get program closer to machine code without losing information needed to do useful optimizations
• Need multiple IR stages

Pentium (LIR)

Java bytecode (LIR)

Itanium (LIR)

optimize

optimize

optimize

optimize

AST → HIR → MIR
High-level IR (HIR)
- AST + new node types not generated by parser
- Preserves high-level language constructs
  - structured flow, variables, methods
- Allows high-level optimizations based on properties of source language (e.g., inlining, reuse of constant variables)
- More passes: ideal for visitors

Medium-level IR (MIR)
- Intermediate between AST and assembly
- Appel’s IR: tree structured IR (triples)
- Unstructured jumps, registers, memory locations
- Convenient for translation to high-quality machine code
- Other MIRs:
  - quadruples: \( a = b \cdot \text{OP} c \)
  - UCODE: stack-machine based (like Java bytecode)
  - advantage of tree IR: easier instruction selection
  - advantage of quadruples: easier dataflow analysis, optimization
  - advantage of UCODE: slightly easier to generate

Low-level IR (LIR)
- Assembly code + extra pseudo-instructions (+ infinite registers)
- Machine-dependent
- Translation to assembly code is trivial
- Allows optimization of code for low-level considerations: scheduling, memory layout

MIR tree
- Intermediate Representation is a tree of nodes representing abstract-machine instructions: can be interpreted
- IR almost the same as Appel’s (except CJUMP)
- Statement nodes return no value, are executed in a particular order
  - e.g., MOVE, SEQ, CJUMP
- CubeX statement ≠ IR statement!
- Expression nodes return a value, children are executed in no particular order
  - e.g., ADD, SUB
- non-determinism gives flexibility for optimization

IR expressions (lecture only)
- \( \text{CONST}(i) \): the integer constant \( i \)
- \( \text{TEMP}(t) \): a temporary register \( t \). The abstract machine has an infinite number of registers
- \( \text{OP}(e_1, e_2) \): one of the following operations
  - arithmetic: \( \text{ADD}, \text{SUB}, \text{MUL}, \text{DIV}, \text{MOD} \)
  - bit logic: \( \text{AND}, \text{OR}, \text{XOR}, \text{LSHIFT}, \text{RSHIFT}, \text{ARSHIFT} \)
  - comparisons: \( \text{EQ}, \text{NEQ}, \text{LT}, \text{GT}, \text{LEQ}, \text{GEQ} \)
- \( \text{MEM}(e) \): contents of memory location w/ address \( e \)
- \( \text{CALL}(f, a_0, a_1, \ldots) \): result of function \( f \) applied to arguments \( a_0 \)
- \( \text{NAME}(n) \): address of the statement or global data location labeled \( n \) (TBD)
- \( \text{ESEQ}(s, e) \): result of \( e \) after statement \( s \) is executed

CONST
- \( \text{CONST} \) node represents an integer constant \( i \)
  - \( \text{CONST}(i) \)
TEMP

- TEMP node is one of the infinite number of registers (temporaries)
- Used for local variables and temporaries
- Value of node is the current content of the named register at the time of evaluation

OP

- Abstract machine supports a variety of different operations
  \[ \text{OP}(e_1, e_2) \]
- Evaluates \( e_1 \) and \( e_2 \) and then applies operation to their results
- \( e_1 \) and \( e_2 \) must be expression nodes
- Any order of evaluation of \( e_1 \) and \( e_2 \) is allowed

MEM

- MEM(\( e \)) node is a memory location
  \[ \text{MEM}(e) \]
- Computes value of \( e \) and looks up contents of memory at that address

CALL

- CALL node represents a function call
  \[ \text{CALL}(e_f, e_0, e_1, e_2, \ldots) \]
- No explicit representation of argument passing, stack frame setup, etc.
- Value of node is result of call

NAME

- NAME(\( n \))
  \[ \text{NAME}(n) \]
- Address of memory location named \( n \)
- Two kinds of named locations
  - labeled statements in program (from LABEL statement)
  - global data definitions (not represented in IR)

ESEQ

- ESEQ(\( s, e \))
  \[ \text{ESEQ}(s, e) \]
- Evaluates an expression \( e \) after completion of a statement \( s \) that might affect result of \( e \)
- Result of node is result of \( e \)
IR statements

- **MOVE(dest, e)**: move result of \( e \) into \( dest \)
  - \( dest = TEMP(t) \): assign to temporary \( t \)
  - \( dest = MEM(e) \): assign to memory locn \( e \)
- **EXP(e)**: evaluate \( e \) for side-effects, discard result
- **SEQ(s_1, ..., s_n)**: execute each stmt \( s_i \) in order
- **JUMP(e)**: jump to address \( e \)
- **CJUMP(e, l_1, l_2)**: jump to statement named \( l_1 \) or \( l_2 \) depending on whether \( e \) is true or false
- **LABEL(n)**: labels a statement (for use in NAME)

Example

```plaintext
n = 0;
while (n < 10) {
    n = n + 1
}
```

Structure of IR tree

- Top of tree is a statement
- Expressions are under some statements
- Statements under expressions only if there is an ESEQ node

Executing the IR

- IR tree is a program representation; can be executed directly by an interpreter
- Execution is tree traversal (exc. jumps)

How to translate?

- How do we translate an AST/High-level IR into this IR representation?
- Next: syntax-directed translation