Where we are
- Source code (character stream)
  - Lexical analysis (regular expressions)
  - Syntactic Analysis (grammars)
  - Semantic Analysis (static semantics)
  - Intermediate Code Generation (syntax-directed translation)

Intermediate Code
- Abstract machine code - simpler
- Allows machine-independent code generation, optimization
  - AST
  - Java bytecode
  - Alpha

Intermediate Code
- Abstract machine code
- Allows machine-independent code generation, optimization

Optimizing compilers
- Goal: get program closer to machine code without losing information needed to do useful optimizations
- Need multiple IR stages
  - AST → HIR → LIR → Java bytecode

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)
- Translation ideal for visitor impl.

Medium-level IR (MIR)
- Intermediate between AST and assembly
- Appel’s IR: tree structured IR (triples)
- other MIRs exist
  - quadruples: $a = b \text{ OP } c$ ("a" is explicit, not arc)
  - UCODE: stack machine based (like Java bytecode)
- advantage of tree IR: easy to generate, easier to do reasonable instruction selection
- advantage of quadruples: easier optimization
- Unstructured jumps, registers, memory loc’ns
- Convenient for translation to high-quality machine code

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

MIR tree
- Intermediate Representation (or IL) 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
- Iota statement ≠ IR statement!
- Expression nodes return a value, children are executed non-deterministically
  - e.g. ADD, SUB
- non-determinism gives flexibility for optimization

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

CONST
- $\text{CONST}$ node represents an integer constant $i$
- $\text{CONST}(i)$
- Value of node is $i$
**TEMP**

- TEMP node is one of the infinite number of registers (temporaries)
- For brevity, FP = TEMP(FP)
- Value of node is the current content of the named register at the time of evaluation

\[ \text{TEMP}(t) \]

**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
- Order of evaluation of \( e_1 \) and \( e_2 \) is non-deterministic

**MEM**

- MEM 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 creation, etc.
- Value of node is result of call

**NAME**

- 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)

\[ \text{NAME}(n) \]

**ESEQ**

- Evaluates an expression \( e \) after completion of a statement \( s \) that might affect result of \( e \)
- Result of node is result of \( e \)

\[ \text{ESEQ}(s, e) \]
**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, discard result
- **SEQ(s1, ..., sn)**: execute each stmt si in order
- **JUMP(e)**: jump to address e
- **CJUMP(e, l1, l2)**: jump to statement named l1 or l2 depending on whether e is true or false
- **LABEL(n)**: a labeled statement (may be used in NAME, CJUMP)

**Example**

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

**Structure of IR tree**

- Top of tree is a statement
- Expressions are under some statement
- 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?

**Syntax-directed Translation**

- Technique: syntax-directed translation
- Abstract syntax tree ⇒ IR tree
- Each subtree of AST translated to subtree in IR tree (typically)
- Implemented as recursive traversal
  - like type checking, but makes a new tree
  - visitor impl. just complicates traversal unless split into several passes
Translation Code

- Like type-checking: add method to AST nodes that does the translation

abstract class Node {
    IRNode translate(SymTab A) { … }
}

- Next: how to express these translations precisely

Variables

- AST expression node translated to IR expression node that has same value
- Local variable v located at offset k -- reference to v in AST becomes IR expression MEM(PLUS(FP, k)) or TEMP(v)
  \[ \begin{array}{c}
    v \\
    \Rightarrow \\
    MEM \\
    FP \\
    CONST(k)
  \end{array} \]

Operators

- AST node corresponding to arithmetic becomes corresponding IR node
  \[ \begin{array}{c}
    e_1 \\
    \Rightarrow \\
    ADD \\
    e_2 \\
    e_1 \\
    e_2
  \end{array} \]

- Use E[e] to represent result of translating AST expression tree e to an IR expression tree

Assignment

- Assignment v = e translates to a MOVE(dest, e) node, where e is the translation of expression E, and dest is the location of v.
  \[ x = 2 \]

Assignment rule

- General rule:
  \[ v = e \]

- Problem: generates statement node that has no value; what about \( x = (y = 2) \)?

Eliminating extra v

- As expression:
  \[ v = e \]
  \[ v = e \]
  \[ v = e \]

- As statement:
  \[ v = e \]
  \[ v = e \]
Statements

- A sequence of statements translates to a SEQ node:
- If $s_1$ translates to IR tree $\llbracket s_1 \rrbracket$ and $s_2$ to $\llbracket s_2 \rrbracket$
- Then $s_1 ; s_2$ translates to $\text{SEQ}(\llbracket s_1 \rrbracket, \llbracket s_2 \rrbracket)$

$$s_1 ; s_2 \rightarrow \text{SEQ}(\llbracket s_1 \rrbracket, \llbracket s_2 \rrbracket)$$

Example again

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

```plaintext
MOVE\ CONST(0)\ TEMP(n)
CJUMP\ LT\ TEMP(n)\ TEMP(10)
LABEL\ (END)
NAME\ (BODY)
LABEL\ (HEAD)
SEQ
JUMP\ NAME\ (HEAD)
LABEL\ (END)
MOVE\ TEMP(n)\ TEMP(n)\ ADD\ TEMP(n)\ TEMP(n)\ CONST(1)
ADD\ TEMP(n)\ TEMP(n)\ CONST(1)
```