Intermediate Representation (IR)

IR encodes all knowledge the compiler has derived about source program.

Simple compiler structure

More typical compiler structure

Components and Design Goals for an IR

Components of IR
- Code representation: actual statements or instructions
- Symbol table with links to/from code
- Analysis information with mapping to/from code
- Constants table: strings, initializers, ...
- Storage map: stack frame layout, register assignments

Design Goals for an IR?
There is no universally good IR. Many forms of IR have been used. The right choice depends strongly on the goals of the compiler system.

Common Code and Analysis Representations

Code representations
- Usually have only one at a time
- Common alternatives:
  - Abstract Syntax Tree (AST)
  - SSA form + CFG
  - 3-address code [+ CFG]
  - Stack code
- Influences:
  - semantic information
  - types of optimizations
  - ease of transformations
  - speed of code generation
  - size

Analysis representations
- May have several at a time
- Common choices:
  - Control Flow Graph (CFG)
  - Symbolic expression DAGs
  - Data dependence graph (DDG)
  - SSA form
  - Points-to graph / Alias sets
- Call graph
- Influences:
  - analysis capabilities
  - optimization capabilities

Categories of IRs By Structure

Graphical IRs
- trees, directed graphs, DAGs
- node / edge data structures tend to be large
- harder to rearrange
- Examples: AST, CFG, SSA, DDG, Expression DAG, Points-to graph

Linear IRs
- pseudo-code for abstract machine
- many possible semantic levels
- simple, compact data structures
- easier to rearrange
- Examples: 3-address, 2-address, accumulator, or stack code

Hybrid IRs as the Code Representation
- CFG + 3-address code (SSA or non-SSA)
- CFG + 3-address code + expression DAG
- AST (for control flow) + 3-address code (for basic blocks)
- AST (for control flow) + expression DAG (for basic blocks)
Abstract syntax tree

An Abstract Syntax Tree (AST) is a simplified parse tree. It retains syntactic structure of code.

- Well-suited for source code
- Widely used in source-source translators
- Captures both control flow constructs and straight-line code explicitly
- Traversal and transformations are both relatively expensive
- Transformations are memory-allocation-intensive

Directed acyclic graph

A Directed Acyclic Graph (DAG) is similar to an AST but with a unique node for each value.

**Advantages**

- Sharing of values is explicit
- Exposes redundancy (value computed twice)
- Powerful representation for symbolic expressions

**Disadvantages**

- Difficult to transform (e.g., delete a stmt)
- Not useful for showing control flow structure
- Better for analysis than transformation

Control Flow Graph: CFG

**Definitions**

- **Basic Block** = a consecutive sequence of statements (or instructions) \( S_1 \ldots S_n \) such that (a) the flow of control must enter the block at \( S_1 \), and (b) if \( S_1 \) is executed, then \( S_2 \ldots S_n \) are all executed in that order (unless one of the statements causes the program to halt).
- **Leader** = the first statement of a basic block
- **Maximal Basic Block** = a maximal-length basic block
- **CFG** = a directed graph (usually for a single procedure) in which:
  - Each node is a basic block
  - There is an edge \( b_1 \rightarrow b_2 \) if control may flow from last stmt of \( b_1 \) to first stmt of \( b_2 \) in some execution

**NOTE:** A CFG is a conservative approximation of the control flow! Why?
Examples 1 - Conditional Control Flow

**Conditional branch in C:**
```
stmtlist_0
if (x == y)
  stmtlist_1
else
  stmtlist_2
stmtlist_3
```

**“switch” statement in C:**
```
stmtlist_0
switch (V) {
  case 1: stmtlist_1
  case 2: stmtlist_2
  ...
  case n: stmtlist_n
  default: stmtlist_n
}
stmtlist_{n+1}
```

Examples 2 - Loops

**“while” loop in C:**
```
stmtlist_0
while (x < k)
  stmtlist_1
stmtlist_2
```

**“do-while” loop in C:**
```
stmtlist_0
do
  stmtlist_1
while (x < k);
stmtlist_2
```

Examples 3 - Exceptions

**“try-catch-finally” in Java:**
```
stmtlist_0
try {
  S_0; // may throw
  S_1; // may throw
} catch (etype, e1) {
  S_3; // simple statement
} catch (etype, e2) {
  S_3; // simple statement
} finally {
  S_4; // simple statement
}
stmtlist_1
```

Dominance in Control Flow Graphs

**Domination** \( B_1 \) dominates \( B_2 \) iff all paths from entry node to \( B_2 \) include \( B_1 \).

Intuitively, \( B_1 \) is always executed before executing \( B_2 \) (or \( B_1 = B_2 \)).

**Which assignments dominate \((X+Y)\)?**
```
X = 1;
if (...) {  // \( Y = 4 \) if true
  Y = 4;
}  // \( \ldots = X + Y \) if true
... = X + Y;
```

**Which assignments dominate \((X\cdot Y)\)?**
```
X = 1;
if (...) {  // \( Y = 4 \) if true
  Y = 4;
}  // \( \ldots = X \cdot Y \) if true
```
Informally, a program can be converted into SSA form as follows:
- Each assignment to a variable is given a unique name.
- All of the uses reached by that assignment are renamed.

Easy for straight-line code:

\[
V_0 \leftarrow 4
\]
\[
V + 5 \leftarrow V_0 + 5
\]
\[
V \leftarrow 6
\]
\[
2V_0 - 6
\]
\[
V + 7 \leftarrow 2 - V_1 + 7
\]

What about flow of control?
Introduce \( \phi \)-functions!

\[
\begin{align*}
X_0 & = 5; \\
X_1 & = 3; \\
Y & = X;
\end{align*}
\]

\[
\begin{align*}
X & = \phi(X_0, X_1); \\
Y_0 & = X_2; \\
Y_1 & = X_3;
\end{align*}
\]

Definition of SSA Form

\[
\begin{align*}
\phi \text{ is not an executable operation.} \\
\phi \text{ has exactly as many arguments as the number of incoming BB edges.} \\
\text{Think about } \phi \text{ argument } V_i \text{ as being evaluated on CFG edge from predecessor } P_j \text{ to } B
\end{align*}
\]

Definition (SSA form):
A program is in SSA form if:
1. each variable is assigned a value in exactly one statement
2. each use of a variable is dominated by the definition

Static Single Assignment (SSA) Form

Static Single Assignment with Control Flow

The SSA Graph

Definition (SSA Graph):
The SSA Graph is a directed graph in which:
- **Nodes** = All definitions and uses of SSA variables
- **Edges** = \( \{(d, u) : u \text{ uses the SSA variable defined in } d\} \)

Examples
Draw the SSA graphs for the examples with control flow
So Where Do We Need Phi Functions?

Choices (for each variable X):
- At every merge point in the CFG?
- At every merge point after a write to X?
- At every merge point (after a write to X) that reaches a read of X?
- At some proper subset of the above merge points?

Informal Conditions:
If basic block B contains an assignment to a variable V, then a \( \phi \) must be inserted in each basic block Z such that all of these are true:
1. there is a non-empty path \( B \rightarrow^+ Z \);
2. there is a path from ENTRY to Z that does not go through B;
3. Z is the first node on the path \( B \rightarrow^+ Z \) that satisfies (2).

These conditions must be reapplied for every \( \Phi \) inserted in the code!

Intuition for Placement Conditions:
(1) \( \Rightarrow \) the value of \( V \) computed in B reaches Z
(2) \( \Rightarrow \) there is a path that does not go through B, so some other value of \( V \) reaches Z along that path (ignore bugs due to uses of uninitialized variables). So, two values must be merged at B with a \( \phi \).
(3) \( \Rightarrow \) The \( \phi \) for the value coming from B itself has not been placed in some earlier node on the path \( B \rightarrow^+ Z \).

A constructive description
\[
\text{PhiFunctionPlacement} \ 
\begin{aligned}
\text{Worklist} & \leftarrow \text{all assignments to scalars} \\
\text{while (Worklist is not empty)} { & \\
\text{Remove one assignment, } S, \text{ from Worklist;} \\
\text{B} & \leftarrow \text{the basic block containing } S; \\
\text{for (every basic block, } Z, \text{ such that } \\
\text{B} & \text{dominates some predecessor of } Z, \text{ and } \\
\text{B} & \text{is not a proper dominator of } Z) \{ \\
\text{Place a Phi assignment at the start of block } Z; \\
\text{Add this Phi assignment to Worklist;} \\
\} \\
\}
\end{aligned}
\]

Does the inner (for) loop above compute exactly the set of nodes satisfying the Informal Conditions on the previous slide?

Tradeoffs of SSA form

**Strengths:**
1. Each use is reached by a single definition
   - Can sometimes use simpler analyses (flow-insensitive instead of flow-sensitive)
2. Def-use pairs are explicit: compact dataflow information
3. No write-after-read and write-after-write dependences
4. Can be directly transformed during optimizations

(1-3) \( \Rightarrow \) Many dataflow optimizations are much faster

**Weaknesses:**
1. Space requirement: many variables, many \( \phi \) functions
2. Limited to scalar values; an array is treated as one big scalar
3. When target is low-level machine code, limited to “virtual registers” (memory is not in SSA form)
4. Copies introduced when converting back to real code
Stack machine code

Used in compilers for stack architectures: B5500, B1700, P-code, BCPL. Popular again for bytecode languages: JVM, MSIL.

**Advantages**
- compact form
- introduced names are implicit, not explicit
- simple to generate & execute code

**Disadvantages**
- does not match current architectures
- many spurious dependences due to stack:
  - difficult to do reordering transformations
- cannot "reuse" expressions easily (must store and re-load):
  - difficult to express optimized code

Example

\[ x - 2 \times y - 2 \times z \]

Stack machine code:
- push \( x \)
- push 2
- push \( y \)
- multiply
- push 2
- push \( z \)
- multiply
- add
- subtract

Three address code

A term used to describe many different representations. Each statement = single operator + at most three operands.

**Advantages**
- compact and very uniform
- makes intermediates values explicit
- suitable for many levels (high, mid, low):
  - high-level: e.g., array refs, min / max ops
  - mid-level: e.g., virtual regs, simple ops
  - low-level: close to assembly code

**Disadvantages**
- Large name space (due to temporaries)
- Loses syntactic structure of source

Example

\[ (x > y) \]

\[ z = x - 2 \times y \]

3-address code:
- \( t_1 \) - load \( x \)
- \( t_2 \) - load \( y \)
- \( t_3 \) - load \( 2 \)
- \( t_4 \) - load \( z \)
- \( t_5 \) - load \( t_2 \)
- \( t_6 \) - load \( t_1 \)
- \( t_7 \) - load \( t_3 \)
- \( t_8 \) - load \( t_4 \)
- \( t_9 \) - store \( t_5 \)

Storage Formats for Three Address Code

**Quadriples**

- Size vs. Ease of Reordering vs. Locality
- Quadruples
- \( x - 2 \times y \)
- load \( t_1 \) \( y \) \( - \)
- loadi \( t_2 \) \( 2 \) \( - \)
- mult \( t_3 \) \( f_2 \) \( f_1 \)
- load \( t_4 \) \( x \) \( - \)
- sub \( f_2 \) \( f_1 \) \( f_3 \)

- table of \( k \times 4 \) small integers (indexes into symbol table)
- not very easy to reorder
- fast to traverse
- all names are explicit

**Indirect Triples**

- Size vs. Ease of Reordering vs. Locality
- Indirect Triples
- \( x - 2 \times y \)
- load \( t_1 \) \( y \) \( - \)
- loadi \( t_2 \) \( 2 \) \( - \)
- mult \( t_3 \) \( f_2 \) \( f_1 \)
- load \( t_4 \) \( x \) \( - \)
- sub \( f_2 \) \( f_1 \) \( f_3 \)

- implicit name
- easier to reorder
- more expensive to traverse

**Linked I**

- Size vs. Ease of Reordering vs. Locality
- Linked I
- \( x - 2 \times y \)
- (103) \( (100) \) \( (101) \)
- load \( y \)
- load \( 2 \)
- mult \( (100) \)
- load \( x \)
- sub \( (103) \)

- explicit name
- easy to reorder
- costly to traverse

Compilation Strategies

**High-level Model**
- Retain high-level data types: Structs, Arrays, Pointers, Classes
- Retain high-level control constructs (AST) on 3-address code
- Generally operate directly on program variables (i.e., no registers)

**Mid-level Model**
- Retain some high-level data types: Structs, Arrays, Pointers
- Linear 3-address code + CFG
- Distinguish virtual regs from memory
- Expose all low-level architectural details: Addressing modes, stack frame, calling conventions, data layout

**Low-level Model**
- Linear memory model (no high-level data types)
- Distinguish virtual registers from memory
- Low-level 3-address code + CFG
- Explicit addressing arithmetic
- Expose all low-level architectural details: Addressing modes, stack frame, calling conventions, data layout
Some Examples of Real Systems

Example 1: Sun Compilers for SPARC (C, C++, Fortran, Pascal)

**Code**
- 2 different IRs

**Analysis info**
- CFG + dependence graph + ???

**High-level IR**: linked-list of triples

**Low-level IR**: SPARC-assembly-like operations

Example 2: IBM Compilers for Power, PowerPC (Same as Sun + PL/8)

**Code**
- Low-level IR (+ optional high-level IR with SSA)

**Analysis info**
- CFG + "intervals" + value graph + dataflow graphs

**Low-level IR**: indirect list of variable-length instructions

Examples of Real Systems (continued)

Example 3: LLVM Compiler (C, C++, ...)

**Code**
- CFG + Mostly 3-address IR in SSA form

**Analysis info**
- Value Numbering + Points-to graph + Call graph

**Basic blocks**: doubly linked list of LLVM instructions

Example 4: dHPF Compiler (Fortran90 + HPF)

**Code**
- AST

**Analysis info**
- CFG + SSA + Value DAG + Call Graph

XIL and YIL: The Intermediate Languages of TOBEY

Key Design Assumptions in XIL

- Low-level IR with no source-level semantic assumptions
- Must be capable of supporting multiple targets
- All loads, stores, and addressing computations must be exposed "from front-end onwards."
- "Main disadvantage": Slower compile time due to larger code volume
- Loops and source-level branches are lowered to compares, and conditional branches to labels
- Loop structure and induction vars. must be recovered via program analysis
- Some "exotic" or complex macro instructions, expanded by Macro Expansion phase:
  - String operations; multi-dim array refs; unlimited args; unlimited size for immediate operands
  - Formal identities:
    - Identities found by hashing: hash(op, arg1, ..., argn)
    - All defs of a symbolic register must be formally identical
    - A symbolic register is name of a unique value
  - Dataflow optimizations operate on symbolic registers (including loads and stores)

XIL and YIL: The Intermediate Languages of TOBEY

Structural Design Assumptions in XIL

- Code representation:
  - Doubly linked list of pointers to instructions
  - Instructions live in a separate (unordered) table: Computation Table
  - More complex than just triples: complex operands; multiple results
- Analysis representations:
  - DAG representation of symbolic expressions
  - Control-flow graph
  - Symbol information: types, line numbers, literal value table
  - IR allows flexible ordering of compiler passes
  - Structure stays fixed throughout optimization and code generation
- Computation Table (CT): Enforces formal identities
  - Uses the hash function so each instruction is entered only once
  - Symbolic registers are simply pointers to unique instructions in CT
- Exception: By client request. Called "non-canonical" instructions
XIL and YIL: The Intermediate Languages of TOBEY

Key Design Assumptions in YIL

- Require higher-level abstractions (than XIL) to support:
  - Dependence analysis for array subscripts
  - Loop transformations: memory hierarchy opts, auto-par, auto-vec
- YIL abstractions can be constructed from XIL (instead of separate generator from front-end)
  - This is unusual: Most compilers successively “lower” the IR
- Adding a layer of structural abstraction over XIL is better than designing a brand new IR:
  - YIL links back to XIL to share expression DAGs in CT
  - YIL exploits XIL functionality for manipulating expressions

Structural Design of YIL

- Code representation:
  - “Statement graph”: doubly linked list of statement nodes
  - Nodes for Loop, If, Assign, Call
  - Loops and loop-nests are explicit
  - Assign node represents a store and all computations feeding it

- Analysis representations:
  - SSA form for variables (probably scalars only)
  - Explicit use-def chains for all variables
  - Dependence graph with dependence distances
  - Links to expression DAGs and symbol information of XIL

- Loop optimizations focus on “unimodular transformations”. Described by a loop transformation matrix
- SSA form is updated incrementally by many optimizations (that don’t change control flow)

XIL and YIL: The Intermediate Languages of TOBEY

Critique of XIL

- Reasonable design for the “very back end”
  - Want dataflow optimization of machine-specific computations
  - Want rich symbolic expression manipulation

But . . .
- XIL also serves as “mid-level” optimizer, i.e., many machine-independent opts
- Code volume is a significant cost
- Many such optimizations require both XIL and YIL features
- Unclear if XIL preserves important type information
- E.g., structures, arrays, pointers
- These are needed for pointer and dependence analysis (important for both dataflow opts and scheduling)

XIL and YIL: The Intermediate Languages of TOBEY

Critique of Hierarchical IL (XIL+YIL)

- Hierarchical = two separate simultaneous ILs:
  - YIL is not a full-fledged IL with complete analysis, optimization suite
  - YIL relies on XIL for dataflow opts, low-level opts

- Lack of dataflow opts in YIL could be a weakness:
  - Many high-level optimizations depend on good low-level opts
  - E.g., Dep. analysis needs pointer analysis, which needs extensive low-level opts
  - Also, many high-level opts. must be followed by good low-level opts

- Interprocedural optimization (IPO) important for both high-level and low-level opts
- Unclear how IPO can work with the XIL / YIL dichotomy
- Code volume of XIL could slow down IPO