Intermediate Representation (IR)

IR $\equiv$ data structures that encode all knowledge the compiler has derived about source program.

Also called Internal Representation.

IR is a fundamental design feature of a compiler system.
Determines compiler functionality, maintanability, speed, memory consumption.

A realistic compiler will have several IRs, including. multiple IRs at every point in the compilation flow.

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

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.

Design Goals for an IR?

What are the Key Design Goals for an IR?

Assume an ahead-of-time (AOT) optimizing compiler for modern languages and multicore superscalar architectures

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

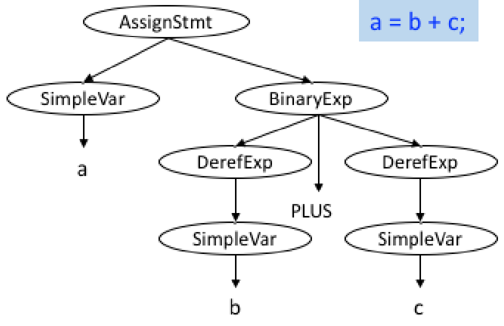
- CFG + 3-address code (SSA or non-SSA)
- CFG + 3-address code + expression DAG
- AST (for control flow) + 3-address code (for basic blocks)
- AST (for control flow) + expression DAG (for basic blocks)

Abstract syntax tree

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

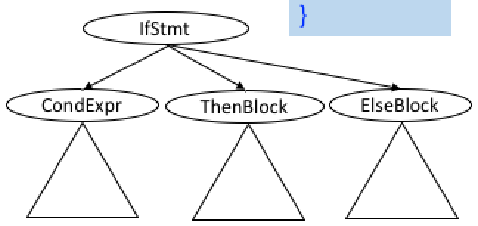
- Well-suited for source code
- Widely used in source-source translators
- Captures both control flow constructs and straight-line code explicitly
- Traversal and transformations are both relatively expensive
  - both are pointer-intensive
  - transformations are memory-allocation-intensive

Abstract syntax tree: Examples

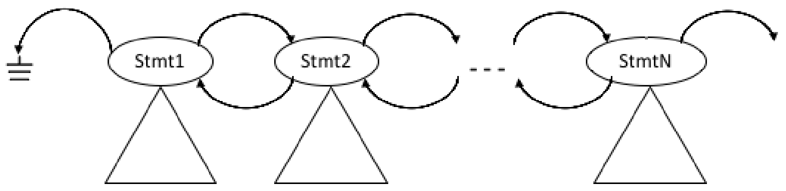


```
a = b + c;
```

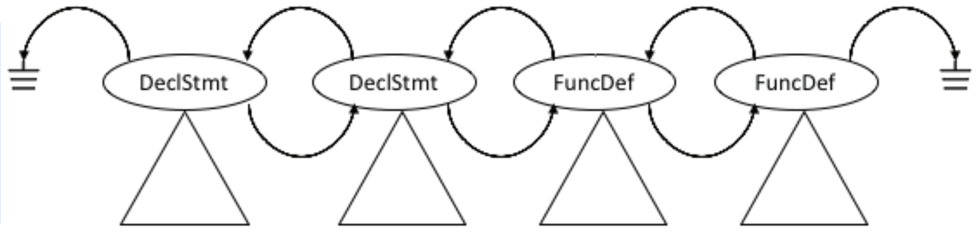
```
if (cond) {
   thenBlock;
} else {
   elseBlock;
}
```



```
{
  Stmt1;
  Stmt2;
  ...
  StmtN;
}
```



```
% cat file.c
int Gi;
float Gf;
void F1() {...}
void F2() {...}
```



University of Illinois at Urbana-Champaign

Directed acyclic graph

A Directed Acyclic Graph (DAG) is a graphical representation of symbolic expressions where any two provably equal expressions share a single node.

Each node can be thought of as a unique (symbolic) *value*.

Advantages

- sharing of values is explicit
- exposes redundancy (value computed twice)

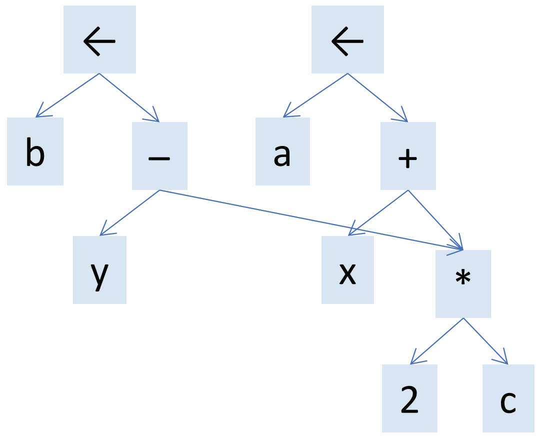
Disadvantages

- difficult to transform (e.g., delete a stmt)
- not useful for showing control flow structure

$\Longrightarrow$ Better for *analysis* than *transformation*

Example

a ← x + 2 * c;
b ← y – 2 * c;



Program Analysis Terminology

Informal Definitions:

Precise: A program analysis result is *precise* iff it represents only those behaviors that occur during *legal executions* of the program.

Conservative: A program analysis result is *conservative* if it represents a behavior that never occurs in *any legal execution* of the program.

Examples of conservative analysis results:

- Constant propagation fails to prove an expression is always zero
- Array bounds checking fails to prove an array index expression is always within bounds
- Alias analysis fails to prove that two pointer parameters are never aliased
- DAG construction fails to prove two expressions are always equal
- Control flow analysis fails to prove a given branch is never taken

Control Flow Graph: CFG

Definitions

Basic Block $\equiv$ 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 $\equiv$ the first statement of a basic block

Maximal Basic Block $\equiv$ a *maximal-length* basic block

CFG $\equiv$ a directed graph (usually for a single procedure) in which:

- Each node is a single 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:

```
stmtlist0
while (x < k)
   stmtlist1
stmtlist2
```

"do-while" loop in C:

```
stmtlist0
do
   stmtlist1
while (x < k);
stmtlist2
```

Examples 3 - Exceptions

"try-catch-finally" in Java:

```
stmtlist_0
try {
    $S_0$;                    // may throw
    $S_1$;                    // may throw
} catch (etype_1 e1) {
    $S_2$;                    // simple statement
} catch (etype_2 e2) {
    $S_3$;                    // simple statement
} finally {
    $S_4$;                    // simple statement
}
stmtlist_1
```

Dominance in Control Flow Graphs

Dominates $\equiv B_1$ 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;
    }
    ...  = X + Y;
```

Which assignments dominate (X+Y)?:

```
    X = 1;
    if (...) {
        Y = 4;
        ...  = X + Y;
    }
```

Static Single Assignment (SSA) Form

- 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:

$$\begin{array}{lll@{\qquad\qquad}lll}
V & \leftarrow & 4 & V_0 & \leftarrow & 4 \\
 & \leftarrow & V + 5 & & \leftarrow & V_0 + 5 \\
V & \leftarrow & 6 & V_1 & \leftarrow & 6 \\
 & \leftarrow & V + 7 & . & \leftarrow & V_1 + 7
\end{array}$$

- What about flow of control?
  Introduce ϕ-functions!

Static Single Assignment with Control Flow

- *2-way branch*:

```
if (...)
        X = 5;
else
        X = 3;

Y = X;
```

```
if (...)
        $X_0$ = 5;
else
        $X_1$ = 3;
$X_2 = \phi(X_0, X_1)$;
$Y_0 = X_2$;
```

- *While loop*:

```
     j = 1;
S: // while (j < x)
     if (j >= X)
          goto E;
     j = j+1;
     goto S
E:
     N = j;
```

```
     $j_5$ = 1;
S:        $j_2 = \phi(j_5, j_4)$;
     if ($j_2$ >= X)
          goto E;
     $j_4 = j_2+1$;
     goto S
E:
     N = $j_2$;
```

Definition of SSA Form

- **Definition (ϕ Functions)**:
  In a basic block B with N predecessors, $P_1,\ P_2,\ \ldots,\ P_N$,
  $$X = \phi(V_1, V_2, \ldots, V_N)$$
  assigns $X = V_j$ if control enters block B from P_j, $1 \leq j \leq N$.

- *Properties of ϕ-functions*:
  - ϕ is not an executable operation.
  - ϕ has exactly as many arguments as the number of incoming BB edges
  - Think about ϕ argument V_i as being evaluated on CFG edge from predecessor P_i to B

- **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

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 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?

So Where Do We Need Phi Functions?

Informal Conditions for Minimal SSA Form:

If basic block B contains an assignment to a variable V, then a ϕ 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 Φ inserted in the code!

Intuition for Placement Conditions:

(1) $\Longrightarrow$ the value of V computed in B reaches Z

(2) $\Longrightarrow$ 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 ϕ.

(3) $\Longrightarrow$ The ϕ for the value coming from B itself has not been placed in some earlier node on the path B $\rightarrow^{+}$ Z.

So Where Do We Need Phi Functions?

Optional material covered in CS 526

A constructive description

```
PhiFunctionPlacement
    Worklist <-- all assignments to scalars
    while (Worklist is not empty) {
        Remove one assignment, S, from Worklist;
        B <-- the basic block containing S;
        for (every basic block, Z, such that
                   B dominates some predecessor of Z, and
                   B is not a proper dominator of Z) {
            Place a Phi assignment at the start of block Z;
            Add this Phi assignment to WorkList;
        }
    }
```

- Does the inner (for) loop above compute exactly the set of nodes satisfying the Informal Conditions on the previous slide?
- (Definition) *Pruned SSA Form* $\equiv$ Minimal SSA form with unused Phi functions deleted.

Common Pitfalls

Repeated arguments

```
    if (p1) {
        X = 1;
        if (p2)
            goto E;
        Y = 1;
    } else {
        X = 2;
    }
E:
    printf("%d", X);
```

Arbitrary names in SSA form

```
    A = 1;
    if (p1) {
        B1 = 1;
    }
    B2 = phi(A, B1);
    printf("%d", B2);
```

More arbitrary names in SSA

```
    x1 = 0;
L1: x2 = phi(x1,y1);
    y1 = phi(x1,x2);
    if (y1 < 10)
       goto L1;
    exit();
```

Three address code

A term used to describe many different representations
Each statement $\equiv$ 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 syntatic structure of source

Example

```
    if (x > y)
        z = x - 2 * y
```

3-address code:

$$
\begin{array}{ll}
 & t_1 \leftarrow \texttt{load}\ x \\
 & t_2 \leftarrow \texttt{load}\ y \\
 & t_3 \leftarrow t_1\ \textit{gt}\ t_2 \\
 & \texttt{br}\ t_3\ L_2\ L_1 \\
L_1: & t_4 \leftarrow 2 * t_2 \\
 & t_5 \leftarrow t_1 - t_4 \\
 & z \leftarrow \texttt{store}\ t_5 \\
L_2: & \cdots
\end{array}
$$

Compilation Strategies

High-level Model

- *Retain high-level data types*: Structs, Arrays, Pointers, Classes
- Retain high-level control constructs (AST) OR 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
- No low-level architectural details

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)* *Muchnick, Chapter 21*

$$\begin{aligned} \textit{Code} &\equiv \text{2 different IRs} \\ \textit{Analysis info} &\equiv \text{CFG + dependence graph + ???} \end{aligned}$$

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)*

$$\begin{aligned} \textit{Code} &\equiv \text{Low-level IR (+ optional high-level IR with SSA)} \\ \textit{Analysis info} &\equiv \text{CFG + “intervals” + value graph + dataflow graphs} \end{aligned}$$

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

Examples of Real Systems (continued)

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

Mid-level Code	$\equiv$	CFG + Mostly 3-address IR in SSA form
Low-level Code	$\equiv$	CFG + Mostly 3-address Machine IR
Analysis info	$\equiv$	Value Numbering + Points-to graph + Call graph

Basic blocks: doubly linked list of LLVM (or Machine IR) instructions

Example 4: dHPF Compiler (*Fortran90 + HPF)* `dhpf.cs.rice.edu`

Code	$\equiv$	AST
Analysis info	$\equiv$	CFG + SSA + Value DAG + Call Graph

EXTRA SLIDES

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:
  $\Rightarrow$ difficult to do reordering transformations
- cannot “reuse” expressions easily (must store and re-load)
  $\Rightarrow$ difficult to express optimized code

Example

$$x - 2 * y - 2 * z$$

Stack machine code:

```
push x
push 2
push y
multiply
push 2
push z
multiply
add
subtract
```

Storage Formats for Three Address Code

Size vs. *Ease of Reordering* vs. *Locality*

Quadruples

`x - 2 * y`

`load`	t_1	y	–
`loadi`	t_2	2	–
`mult`	t_3	t_2	t_1
`load`	t_4	x	–
`sub`	t_5	t_3	t_4

- 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

`x - 2 * y`

Order	*Code*			
		op	arg1	arg2
(103)	(100)	load	y	
(100)	(101)	loadi	2	
(101)	(102)	mult	(100)	(101)
(102)	(103)	load	x	
(104)	(104)	sub	(103)	(102)

- index is implicit name
- easier to reorder stmts
- more expensive to traverse

Linked list

- explicit names
- easy to reorder
- costly to traverse

XIL and YIL: The Intermediate Languages of TOBEY

O'Brien et al., IR'95.

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

    $\Longrightarrow$ 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
  - Passes may be used in different orders, and repeated
- *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

XIL and YIL: The Intermediate Languages of TOBEY

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”
  - $\Longleftarrow$ Want dataflow optimization of machine-specific computations
  - $\Longleftarrow$ 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 $\equiv$ 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