Intermediate Representations & Symbol Tables
Intermediate Representations

- Front End - Produces an Intermediate Representation (IR)
- Middle End - Transforms the IR into an equivalent IR that runs more efficiently
- Back End - Transforms the IR into native Code

- IR Encodes the compiler’s Knowledge of the Program
- Middle End usually consists of Several Passes
Intermediate Representations

- Basic Idea:
  - Is Target Machine Independent
  - Common to many Input Languages

  \[
  \begin{array}{c}
  \text{C/C++} \\
  \text{Java} \\
  \text{ML} \\
  \end{array}
  \quad
  \begin{array}{c}
  \text{IR} \\
  \text{MIPS} \\
  \text{PPC} \\
  \text{x86} \\
  \end{array}
  \]

- What are the desired characteristics of an IR?
  - Represent and Preserve the Semantics of the Input Program
  - Amenable to Analysis and Transformations
Multi-Level IR?

• A Single IR is seldom the best solution for all uses…

  - Multi-Level IR
    - High-Level IR close to AST
    - Low-Level IR close to machine instructions
    - Typically flows One-Way
  
  • High-Level IR facilitates:
    - Source-to-Source Transformations
    - High-Level Program Analysis and Transformations (e.g., loop unrolling)

  • Low-Level IR facilitates:
    - Target Architecture Transformations & Analysis (e.g., register windows, predication, speculation)
Intermediate Representations

• Decisions in IR design affect the speed and efficiency of the compiler

• Some important IR Properties
  – Ease of generation
  – Ease of manipulation
  – Procedure size
  – Freedom of expression
  – Level of abstraction

• Importance of different properties varies
  – Selecting an appropriate IR for a compiler is critical
Types of Intermediate Representations

Three Major Categories

• Structural
  – Graphically oriented
  – Heavily used in source-to-source translators
  – Tend to be large

• Linear
  – Pseudo-code for an abstract machine
  – Level of abstraction varies
  – Simple, compact data structures
  – Easier to rearrange

• Hybrid
  – Combination of graphs and linear code

Examples:

- Trees, DAGs
- 3 address code
- Stack machine code
- Control-Flow Graph
Level of Abstraction

• The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.

• Two different representations of an array reference:

  ```
  loadI 1 => r_1
  sub  r_j, r_1 => r_2
  loadI 10 => r_3
  mult r_2, r_3 => r_4
  sub  r_i, r_1 => r_5
  add  r_4, r_5 => r_6
  loadI @A => r_7
  Add  r_7, r_6 => r_8
  load  r_8 => r_{Ai}
  ```

High level AST: Good for memory disambiguation

Low level linear code: Good for address calculation
Level of Abstraction

- Structural IRs are usually considered high-level
- Linear IRs are usually considered low-level
- Not necessarily true:

```
load

+ @A

* +

10 -

j j 1

loadArray A,i,j

Low level AST

High level linear code
```
Abstract Syntax Tree

An abstract syntax tree is the procedure’s parse tree with the nodes for most non-terminal nodes removed

```
x - 2 * y
```

- Can use linearized form of the tree
  - Easier to manipulate than pointers
    ```
    x 2 y * -  
    - * 2 y x
    ```
    in postfix form
    in prefix form

- S-expressions are (essentially) ASTs
Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value.

- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!
Stack Machine Code

Originally used for stack-based computers, now Java

- Example:
  \[ x - 2 * y \]

  becomes

  push x
  push 2
  push y
  multiply
  subtract

Advantages

- Compact form
- Introduced names are *implicit*, not *explicit*
- Simple to generate and execute code

Useful where code is transmitted over slow communication links
Three Address Code

Several different representations of three address code

• In general, three address code has statements of the form:
  \[ x \leftarrow y \text{ op } z \]
  With 1 operator \((\text{op})\) and, at most, 3 names \((x, y, \& z)\)

Example:
  \[ z \leftarrow x - 2 * y \]
  becomes

Advantages:

• Resembles many machines
• Introduces a new set of names
• Compact form
Three Address Code: Quadruples

Naïve representation of three address code

- Table of k * 4 small integers
- Simple record structure
- Easy to reorder
- Explicit names

RISC assembly code

<table>
<thead>
<tr>
<th>Quadruples</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1, y</td>
</tr>
<tr>
<td>loadI r2, 2</td>
</tr>
<tr>
<td>mult r3, r2, r1</td>
</tr>
<tr>
<td>load r4, x</td>
</tr>
<tr>
<td>sub r5, r4, r3</td>
</tr>
</tbody>
</table>

Quadruples

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1</td>
<td>y</td>
</tr>
<tr>
<td>loadI</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>x</td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
Three-Address Instructions IR

• Constructs Mapped to Three-Address Instructions
  – Register-based IR for Expression Evaluation
  – Infinite Number of Virtual Registers (later we will lift this assumption)
  – Still Independent of Target Architecture
  – Parameter Passing Discipline either on Stack or via Registers

• Addresses and Instructions
  – Symbolic Names are addresses of the corresponding source-level variable.
  – Various constants, such as numeric and offsets (known at compile time)

• Generic Instruction Format:
  \[ \text{Label: } x = y \text{ op } z \text{ or if exp goto } L \]
  – Statements can have Symbolic Labels
  – Compiler inserts Temporary Variables as any variable with “t” prefix
  – Registers (limited in number) with prefix “r”
  – Type and Conversions dealt in other Phases of the Code Generation
Three-Address Instructions

• Assignments:
  - $x = y \text{ op } z$ (binary operator)
  - $x = \text{ op } y$ (unary)
  - $x = y$ (copy)
  - $x = y[i]$ and $x[i] = y$ (array indexing assignments)
  - $x = \text{ phi } y z$ (Static Single Assignment instruction)

• Memory Operations:
  - $x = &y; x = *y$ and $*x = y$; for assignments via pointer variables.
Three-Address Instructions

• Control Transfer and Function Calls:
  - `goto L` (unconditional);
  - `if (a relop b) goto L` (conditional) where `relop` is a relational operator consistent with the type of the variables `a` and `b`;
  - `y = call p, n` for a function or procedure call instruction to the name or variable `p`
    • `p` might be a variable holding a set of possible symbolic names (a function pointer).
  • Possible Parameter Passing Mechanism – a Stack
    - The value `n` specifies that before this call there were `n put` instructions to load the values of the arguments.
    - The `param` instruction specifies a specific value in reverse order (i.e., the `param` instruction closest to the call is the first argument value).
  • Later we will talk about parameter passing disciplines (Run-Time Env.)
Function Call Example

Source Code

```c
int p(x, z) {
    return x + z;
}

y = p(a, b+1)
```

Three Address Instructions

```c
| t1 = a |
| t2 = b + 1 |
| putparam t1 |
| putparam t2 |
| y = call p, 2 |
| p: getparam z |
| getparam x |
| t3 = x + z |
| return t3 |
| return |
```
Loop Example

Source Code

do
    i = i + 1;
while (a[i] < v);

Three Address Instructions

L: t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a[t2]
    if t3 < v goto L
Loop Example

Source Code

\[
\text{do} \\
\quad i = i + 1; \\
\text{while } (a[i] < v); \\
\]

Three Address Instructions

\[
\begin{align*}
L: & \quad t1 = i + 1 \\
& \quad i = t1 \\
& \quad t2 = i \times 8 \\
& \quad t3 = a[t2] \\
& \quad \text{if } t3 < v \text{ goto } L
\end{align*}
\]

Where did this come from?
Control-Flow Graph

Models the transfer of control in the procedure

• Nodes in the Graph are Basic Blocks
  – Can be represented with quads or any other linear representation

• Edges in the Graph represent Control Flow

Example

```
if (x = y)
  a ← 2
  b ← 5
  c ← a * b
  a ← 3
  b ← 4
```

Basic blocks — Maximal length sequences of straight-line code
Static Single Assignment Form

• The main idea:
  – Each Name Defined Exactly Once
  – Each Use (of a Name) refers to Exactly One Name

• Introduce $\phi$-functions to make it work

Original

```
x ← ...
y ← ...
while (x < k)
  x ← x + 1
  y ← y + x
```

SSA-form

```
x_0 ← ...
y_0 ← ...
if (x_0 >= k) goto next
loop:
x_1 ← \phi(x_0, x_2)
y_1 ← \phi(y_0, y_2)
x_2 ← x_1 + 1
y_2 ← y_1 + x_2
if (x_2 < k) goto loop
next:
```

Strengths of SSA-form

• Sharper analysis

• $\phi$-functions give hints about placement of transformed code
Static Single Assignment Form

SSA Form:
- Each Name Defined Exactly Once
- Each use refers to Exactly One Name

Original

\[ x \leftarrow \ldots \]
\[ y \leftarrow \ldots \]
while \((x < k)\)
\[ x \leftarrow x + 1 \]
\[ y \leftarrow y + x \]

IR

\[ x \leftarrow \ldots \]
\[ y \leftarrow \ldots \]
\[ \text{if} (x \geq k) \text{ goto next} \]

loop: \[ x \leftarrow x + 1 \]
\[ y \leftarrow y + x \]
\[ \text{if} (x < k) \text{ goto loop} \]
Static Single Assignment Form

**Original**

\[
\begin{align*}
x & \leftarrow \ldots \\
y & \leftarrow \ldots \\
\text{while } (x < k) & \\
& x \leftarrow x + 1 \\
& y \leftarrow y + x
\end{align*}
\]

**SSA Form:**
- Each Name Defined Exactly Once
- Each use refers to Exactly One Name

**SSA-form**

\[
\begin{align*}
x_0 & \leftarrow \ldots \\
y_0 & \leftarrow \ldots \\
\text{if } (x_0 \geq k) & \text{ goto next} \\
\text{loop: } x_1 & \leftarrow \phi(x_0, x_2) \\
y_1 & \leftarrow \phi(y_0, y_2) \\
x_2 & \leftarrow x_1 + 1 \\
y_2 & \leftarrow y_1 + x_2 \\
\text{if } (x_2 < k) & \text{ goto loop} \\
\text{next: } & \ldots
\end{align*}
\]
Static Single Assignment Form

**Original**

\[
x \leftarrow \ldots
\]
\[
y \leftarrow \ldots
\]
while \((x < k)\)
\[
x \leftarrow x + 1
\]
\[
y \leftarrow y + x
\]

**IR**

\[
x \leftarrow \ldots
\]
\[
y \leftarrow \ldots
\]
if \((x \geq k)\) goto next

**SSA Form:**

- Each Name Defined Exactly Once
- Each use refers to Exactly One Name

**SSA-form**

\[
x_0 \leftarrow \ldots
\]
\[
y_0 \leftarrow \ldots
\]
if \((x_0 \geq k)\) goto next

loop:
\[
x_1 \leftarrow \phi(x_0, x_2)
\]
\[
y_1 \leftarrow \phi(y_0, y_2)
\]
\[
x_2 \leftarrow x_1 + 1
\]
\[
y_2 \leftarrow y_1 + x_2
\]
if \((x_2 < k)\) goto loop

next:
\[
\ldots
\]
Using Multiple Representations

- Repeatedly lower the level of the Intermediate Representation
  - Each intermediate representation is suited towards certain optimizations
- Example: the Open64 compiler
  - WHIRL intermediate format
    - Consists of 5 different IRs that are progressively more detailed
Memory Models

Two major Models:

• Register-to-Register model
  – Keep all values that can legally be stored in a register in registers
  – Ignore machine limitations on number of registers
  – Compiler back-end must insert loads and stores

• Memory-to-Memory model
  – Keep all values in memory
  – Only promote values to registers directly before they are used
  – Compiler back-end can remove loads and stores

• Compilers for RISC machines usually use register-to-register
  – Reflects programming model
  – Easier to determine when registers are used
The Rest of the Story…

Representing the code is only part of an IR

Other necessary components:

• Symbol Table
• Constant Table
  – Representation, type
  – Storage class, offset
• Storage Map
  – Overall storage layout
  – Overlap information
  – Virtual register assignments
The Procedure as a Name Space

Each procedure creates its own *Name Space*

- Any name (almost) can be declared locally
- Local names obscure identical non-local names
- Local names cannot be seen outside the procedure
  - Nested procedures are “inside” by definition
- We call this set of rules & conventions “lexical scoping”

Examples

- C has global, static, local, and *block* scopes
  - Blocks can be nested, procedures (in C) cannot
- Scheme has global, procedure-wide, and nested scopes
  - Procedure scope (typically) contains formal parameters
The Procedure as a Name Space

Why introduce Lexical Scoping?

– A compile-time mechanism for binding “free” variables
– Simplifies rules for naming & resolves conflicts

How can the compiler keep track of all those names?

Problem:

– At point \( p \), which declaration of \( x \) is current?
– At run-time, where is \( x \) found?
– As parser goes in & out of scopes, how does it delete \( x \)?

Answer:

• Lexically Scoped Symbol Tables
The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of actual parameters
- Control returns to call site, immediately after invocation
The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call
• Invoked at a call site, with some set of actual parameters
• Control returns to call site, immediately after invocation

```
int p(a,b,c)
    int a, b, c;
    {
        int d;
        d = q(c,b);
        ...
    }
... s = p(10,t,u);
...```

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

```
int p(a,b,c)
    int a, b, c;
    {
        int d;
        d = q(c,b);
        ...
    }

int q(x,y)
    int x,y;
    {
        return x + y;
    }
```

```
... s = p(10,t,u);
...
```
The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

```plaintext
int p(a,b,c)  
  int a, b, c;  
  {  
    int d;  
    d = q(c,b);  
    ...  
  }

int q(x,y)  
  int x,y;  
  {  
    return x + y;  
  }

s = p(10,t,u);  
...  
...```

...
The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation

```c
int p(a, b, c)
int a, b, c;
{
    int d;
    d = q(c, b);
    ...
}
```

```c
int q(x, y)
int x, y;
{
    return x + y;
}
```

```c
s = p(10, t, u);
```
The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call
• Invoked at a call site, with some set of actual parameters
• Control returns to call site, immediately after invocation

```
int p(a,b,c)
int a, b, c;
{
    int   d;
    d = q(c,b);
    ...
}

int q(x,y)
int x,y;
{
    return x + y;
}
```

• Most languages allow recursion
The Procedure as a Control Abstraction

Implementing procedures with this behavior

- Requires code to save and restore a “return address”
- Must map actual parameters to formal parameters \((c \rightarrow x, b \rightarrow y)\)
- Must create storage for local variables (&, maybe, parameters)
  - \(p\) needs space for \(d\) (&, maybe, \(a, b, \& c\))
  - where does this space go in recursive invocations?

Compiler emits code that causes all this to happen at run time
The Procedure as a Control Abstraction

Implementing procedures with this behavior

- Must preserve \( p \)'s state while \( q \) executes
  - recursion causes the real problem here

- Strategy: Create unique location for each procedure activation
  - Can use a “stack” of memory blocks to hold local storage and return addresses

Compiler emits code that causes all this to happen at run time
The Procedure as a Name Space

Each Procedure Creates its own Name Space:

• Any name (almost) can be declared locally
• Local names obscure identical non-local names
• Local names cannot be seen outside the procedure
  – Nested procedures are “inside” by definition

• We call this set of rules & conventions “lexical scoping”

Examples:

• C has global, static, local, and block scopes
• Blocks can be nested, procedures cannot
• Scheme has global, procedure-wide, and nested scopes
  – Procedure scope (typically) contains formal parameters
The Procedure as a Name Space

Why introduce Lexical Scoping?
– A compile-time mechanism for binding “free” variables
– Simplifies rules for naming & resolves conflicts

How can the Compiler keep track of all those Names?

Problem:
– At point $p$, which declaration of $x$ is current?
– At run-time, where is $x$ found?
– As parser goes in & out of scopes, how does it delete $x$?

Answer: Lexically Scoped Symbol Tables
Why Do We Need a Symbol Table?

• Bind Names (Symbols) to program entities (e.g., variables and procedures)
  – Set of possible values, i.e. their type
  – Storage associated with names i.e., where they are stored and at which offsets (of the activation record or global table – see next lecture)
  – Visibility, i.e. where they can be referenced

• Usage of a Symbol Table
  – Verification (e.g., number and type of procedure arguments)
  – Code Generation (e.g. to select type of instructions)
  – Debugging (e.g., memory to symbol association)
Symbol Table Structure

• Attributes of Symbol in a Symbol Table
  – Name
  – Type
  – Storage Class
  – Scope, Visibility and Lifetime

• Structure and Attributes Depends on the Language
  – PASCAL allows nested lexical scoping
  – Lisp allows dynamic scoping
## Sample Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>char string</td>
<td>Identifier’s name</td>
</tr>
<tr>
<td>Type</td>
<td>enumerated</td>
<td>Source-level type identifier</td>
</tr>
<tr>
<td>Class</td>
<td>enumerated</td>
<td>Storage class</td>
</tr>
<tr>
<td>Volatile</td>
<td>boolean</td>
<td>Asynchronously accessed</td>
</tr>
<tr>
<td>Size</td>
<td>integer</td>
<td>Size of representation in bytes</td>
</tr>
<tr>
<td>Boundary</td>
<td>integer</td>
<td>Size of alignment in bytes</td>
</tr>
<tr>
<td>Register</td>
<td>boolean</td>
<td>true if stored in register</td>
</tr>
<tr>
<td>Displacement</td>
<td>integer</td>
<td>offset from frame</td>
</tr>
</tbody>
</table>
Scope, Visibility and Lifetime

• Scope: Unit of *static* program structure that may have one of more variable declared in it.

• Visibility: Refers to what scopes a given variable’s name refers to a particular instance of variable.

• Lifetime: Execution period from the point when a variable first becomes visible until it is last visible.
Storage Class

Storage Class: Scope, Visibility and Lifetime of Symbols

- **Global:** Visible throughout the entire program execution;
  Lifetime encompasses whole execution.

- **File, Module:** Visible in all of a given file;
  Lifetime encompasses whole execution.

- **Automatic:** Visible and live with activation of a scope

Modifiers: How Values can be Changed and Retained

- **Volatile:** Can be modified asynchronously
- **Static:** Retains values across lifetime boundaries.
Lexically-Scoped Symbol Tables

The Problem:
- The compiler needs a distinct record for each declaration
- Nested lexical scopes admit duplicate declarations

The Interface:
- `insert(name, level)` – creates record for `name` at `level`
- `lookup(name, level)` – returns pointer or index
- `delete(level)` – removes all names declared at `level`

Many implementation schemes have been proposed
- Hash table implementation is tricky, detailed, & fun

Symbol tables are compile-time structures the compiler use to resolve references to names. We’ll see the corresponding run-time structures that are used to establish addressability later.
Example

```
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure r {
            int x, y, z
            ....
        }
        procedure s {
            int x, a, v
            ...
        }
    }
    ... q ...
}
```

```
B0: {
    int a, b, c
B1: {
    int v, b, x, w
B2: {
    int x, y, z
    ....
    }
B3: {
    int x, a, v
    ...
    }
    ...
    r ...
    s
    }
    ...
}
```
Lexically-Scoped Symbol Tables

High-level Idea
• Create a new table for each scope
• Chain them together for lookup

“Sheaf of tables” implementation
• `insert()` may need to create table
  it always inserts at current level
• `lookup()` walks chain of tables &
  returns first occurrence of name
• `delete()` throws away table for level
  \( p \), if it is top table in the chain

If the compiler must preserve the table (for, say, the debugger), this idea is actually practical.
Individual tables can be hash tables.
Implementing Lexically Scoped Symbol Tables

Stack Organization

Implementation

- insert() creates new level pointer if needed and inserts at nextFree
- lookup() searches linearly from nextFree–1 forward
- delete() sets nextFree to the equal the start location of the level deleted.

Advantage
- Uses much less space

Disadvantage
- Lookups can be expensive
Implementing Lexically Scoped Symbol Tables

Threaded Stack Organization

Implementation
- **insert()** puts new entry at the head of the list for the name
- **lookup()** goes direct to location
- **delete()** processes each element in level being deleted to remove from head of list

Advantage
- **lookup** is fast

Disadvantage
- **delete** takes time proportional to number of declared variables in level
Nested Procedures & Symbol Tables

1. program sort(input, output);
2. var a: array [0..10] of integer;
3. x, i: integer;
4. procedure readarray;
5. var i: integer;
6. begin ... a ... end { readarray };
7. procedure exchange(i, j: integer);
8. begin
9. x := a[i]; a[i] := a[j]; a[j] := x;
10. end { exchange };
11. procedure quicksort(m, n: integer);
12. var k, v: integer;
13. function partition(y, z: integer): integer;
14. var i, j: integer;
15. begin ... a ...
16. ... v ...
17. ... exchange(i, j); ...
18. end { partition }
19. begin ... end { quicksort }
20. begin ... end { sort }
Summary

• Intermediate Representations
  – Common Types and Implementations
  – Static-Single-Assignment (SSA) form

• The Procedure Abstraction
  – Scope, Visibility and Lifetime

• Symbol Tables
  – General Structure and Implementation