

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

# Intermediate Representations & Symbol Tables

Copyright 2010, Pedro C. Diniz, all rights reserved.  
Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Intermediate Representations

```

graph LR
    SC[Source Code] --> FE[Front End]
    FE -- IR --> ME[Middle End]
    ME -- IR --> BE[Back End]
    BE --> TC[Target Code]
  
```

- 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

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Intermediate Representations

- Basic Idea:
  - Is target Machine independent
  - Common to many input languages

```

graph LR
    C[C/C++] --> IR[IR]
    J[Java] --> IR
    M[ML] --> IR
    IR --> MIPS
    IR --> PPC
    IR --> x86
  
```

- What are the desired characteristics of an IR?
  - Represent and preserve the semantics of the input program
  - Amenable to Analysis and Transformations

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Multi-Level IR?

- A Single IR is seldom the best solution for all uses...

```

graph LR
    IR1[IR1] --> H[High Level IR]
    IR1 --> L[Low Level IR]
    H --> IR2[IR2]
    L --> IR2
  
```

- 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)

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Three-Address Instructions IR

- Construct mapped to Three-Address Instructions
  - Register-based IR for Expression Evaluation
  - Infinite Number of Virtual Registers
  - 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:
 

Label:  $x = y \text{ op } z$  or  $\text{if exp goto L}$

  - Statements can have Symbolic Labels
  - Compiler inserts Temporary Variables (any variable with t prefix)
  - Type and Conversions dealt in other Phases of the Code Generation

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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.

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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)
    - the value n specifies that before this call there were n putparam instructions to load the values of the arguments.
    - the param x 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.)

Padra D'Áirí

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Function Call Example

Source Code	Three Address Instructions
<pre>           ...           y = p(a, b+1)           ...           int p(x,z){               return x+z;           }         </pre>	<pre>           ...           t1 = a           t2 = b + 1           putparam t1           putparam t2           y = call p, 2           ...           p: getparam z              getparam x              t3 = x + z              return t3         </pre>

Padra D'Áirí

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Loop Example

Source Code	Three Address Instructions
<pre> do   i = i + 1; while (a[i] &lt; v);         </pre>	<pre> L: t1 = i + 1    i = t1    t2 = i * 8    t3 = a[t2]    if t3 &lt; v goto L         </pre>

Padra D'Áirí

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Loop Example

Source Code	Three Address Instructions
<pre> do   i = i + 1; while (a[i] &lt; v);         </pre>	<pre> L: t1 = i + 1    i = t1    t2 = i * 8    t3 = a[t2]    if t3 &lt; v goto L         </pre>

Where did this come from?

Padra D'Áirí

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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
- The importance of different properties varies between compilers
  - Selecting an appropriate IR for a compiler is critical

Padra D'Áirí

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Types of Intermediate Representations

Three major categories

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

Examples: Trees, DAGs
- Linear**
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange

Examples: 3 address code, Stack machine code
- Hybrid**
  - Combination of graphs and linear code
  - Example: control-flow graph

Example: Control-flow graph

Padra D'Áirí

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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:

High level AST:  
Good for memory disambiguation

```

loadI 1    => r1
sub  r1, r1 => r2
loadI 10   => r3
mult r2, r3 => r4
sub  r4, r1 => r5
add  r4, r5 => r6
loadI @A   => r7
Add  r7, r6 => r8
load  r8    => rA[i,j]
          
```

Low level linear code:  
Good for address calculation

13

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Level of Abstraction

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

Low level AST

loadArray A, i, j

High level linear code

14

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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
  - $x \ 2 \ y \ * \ -$  in postfix form
  - $- \ * \ 2 \ y \ x$  in prefix form
- S-expressions are (essentially) ASTs

15

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Directed Acyclic Graph

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

$z \leftarrow x - 2 * y$   
 $w \leftarrow x / 2$

- Makes sharing explicit
- Encodes redundancy

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

16

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Stack Machine Code

Originally used for stack-based computers, now Java

- Example:
 

$x - 2 * y$	becomes	<pre> push x push 2 push y multiply subtract           </pre>
-------------	---------	---

Implicit names take up no space, where explicit ones do!

Advantages:

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

Useful where code is transmitted over slow communication links (*the net*)

17

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Three Address Code

Several different representations of three address code

- In general, three address code has statements of the form:
 
$$x \leftarrow y \ op \ z$$
 With 1 operator (*op*) and, at most, 3 names (*x*, *y*, & *z*)

Example:

$z \leftarrow x - 2 * y$  becomes  $t \leftarrow 2 * y$ ,  $z \leftarrow x - t$

Advantages:

- Resembles many machines
- Introduces a new set of names  $\{t\}$
- Compact form

18

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Three Address Code: Quadruples

Naïve representation of three address code

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

The original FORTRAN compiler used "quads"

```

load r1, y
loadI r2, 2
mult r3, r2, r1
load r4, x
sub r5, r4, r3
  
```

RISC assembly code

load	1	Y	
loadI	2	2	
mult	3	2	1
load	4	X	
sub	5	4	3

Quadruples

19

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

(1)	load	y	
(2)	loadI	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Implicit names take no space!

20

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Three Address Code: Indirect Triples

- List first triple in each statement
- Implicit name space
- Uses more space than triples, but easier to reorder

(100)	load	y	
(101)	loadI	2	
(102)	mult	(100)	(101)
(103)	load	x	
(104)	sub	(103)	(102)

- Major tradeoff between quads and triples is compactness versus ease of manipulation
  - In the past compile-time space was critical
  - Today, speed may be more important

21

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Static Single Assignment Form

- The main idea: each name defined exactly once
- Introduce  $\phi$ -functions to make it work

<p><b>Original</b></p> <pre> x ← -- y ← -- while (x &lt; k)   x ← x + 1   y ← y + x   </pre>	<p><b>SSA-form</b></p> <pre> x<sub>0</sub> ← -- y<sub>0</sub> ← -- if (x<sub>0</sub> &gt; k) goto next loop:   x<sub>1</sub> ← φ(x<sub>0</sub>, x<sub>2</sub>)   y<sub>1</sub> ← φ(y<sub>0</sub>, y<sub>2</sub>)   x<sub>2</sub> ← x<sub>1</sub> + 1   y<sub>2</sub> ← y<sub>1</sub> + x<sub>2</sub>   if (x<sub>2</sub> &lt; k) goto loop next:   ...   </pre>
--	---

Strengths of SSA-form

- Sharper analysis
- $\phi$ -functions give hints about placement
- (sometimes) faster algorithms

22

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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

```

graph TD
  A["a ← 2  
b ← 5"] --> B["a ← 3  
b ← 4"]
  A --> C["c ← a * b"]
  B --> D["if (x = y)"]
  D --> A
  D --> C
  
```

Basic blocks — Maximal length sequences of straight-line code

23

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Using Multiple Representations

```

Source Code → [Front End] → IR 1 → [Middle End] → IR 2 → [Middle End] → IR 3 → [Back End] → Target Code
  
```

- 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

24

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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

Paolo DiStasio

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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 (next...)
  - Overall storage layout
  - Overlap information
  - Virtual register assignments

Paolo DiStasio

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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 (*Fortran-like*)
  - Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes (*let*)
  - Procedure scope (typically) contains formal parameters

Paolo DiStasio

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## The Procedure as a Name Space

Why introduce lexical scoping?

- Provides a compile-time mechanism for binding "free" variables
- Simplifies rules for naming & resolves conflicts

How can the compiler keep track of all those names?

The 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  $x$ ?

The Answer:

- Lexically Scoped Symbol Tables

Paolo DiStasio

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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 are they store and at which offsets)
  - 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)

Paolo DiStasio

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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

Paolo DiStasio

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Sample Attributes

Name	Type	Meaning
Name	char string	Identifier's name
Type	enumerated	Source-level type identifier
Class	enumerated	Storage class
Volatile	boolean	Asynchronously accessed
Size	integer	Size of representation in bytes
Boundary	integer	Size of alignment in bytes
Register	boolean	true if stored in register
Displacement	integer	offset from frame

31

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Scope, Visibility and Lifetime

- **Scope:** Unit of *static* program structure that may have one or more variable declared in it.
- **Visibility:** Refers to what scopes a given variable's name refers to a particular instance of a variable.
- **Lifetime:** Execution period from the point when a variable first becomes visible until it is last visible.

32

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Storage Class

Prescribe: 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.

33

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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

We'll stay at the conceptual level

- 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.*

34

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## Example

```

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

```

```

B0: { int a, b, c
B1: { int n, b, x, w
B2: { int x, y, z
     ...
B3: { int x, a, v
     ...

```

35

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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.

36

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

### 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**

37

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

### 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

38

USC Viterbi School of Engineering CSCI 565 - Compiler Design Spring 2010

## 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

39