6.035

Fall 2000
Lectures 6 and 7
Intermediate Representation and Semantic Checking

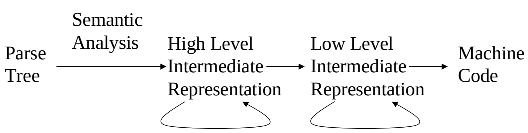
53

Outline

- Program Representation Goals
- Data Format in Running Code
- Compilation Tasks
- Symbol Tables
- High-Level Intermediate Representation
- Semantic Checks
- Low-Level Intermediate Representation

Program Representation Goals

- Enable Program Analysis and Transformation
  - Optimizations
- Structure Translation to Machine Code
  - Sequence of Steps



High Level IR

- Preserves Object Structure
- Suitable for Object-Oriented Optimizations
  - Inline Allocation of Objects
  - Optimizations of Dynamic Dispatch
- Preserves Structured Flow of Control
- Suitable for Loop Level Optimizations
  - Blocking for Cache
  - Loop Interchange, Fusion, Unrolling, etc.

Low Level IR

- Moves Data Model to Flat Address Space
- Eliminates Structured Control Flow
- Suitable for Low Level Compilation Tasks
  - Register Allocation
  - Instruction Selection

Alternatives

- There are many possible alternatives
- More or less language-specific
- These lectures present one way of doing it
  - Geared toward single-inheritance object-oriented languages

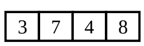
Examples of Object Representation and Program Execution

Example Vector Class

```
class vector {
   int v[];
   void add(int x) {
        int i;
        i = 0;
        while (i < v.length) { v[i] = v[i]+x; i = i+1; }
   }
}
```


Representing Arrays

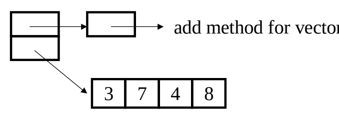
- Items Stored Contiguously In Memory
- Length Stored In First Word



- Color Code
  - Red - generated by compiler automatically
  - Blue, Yellow, Lavender - program data or code
  - Magenta - executing code or data

Representing Vector Objects

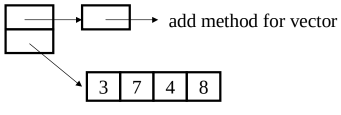
- First Word Points to Class Information
  - Method Table
- Next Words Have Object Fields
  - For vectors, First Word is Reference to Array



Invoking Vector Add Method

vect.add(1);

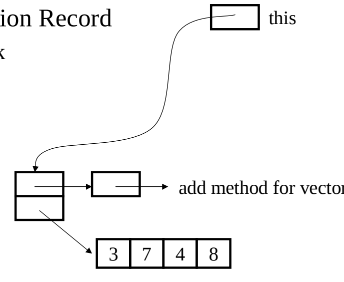
- Create Activation Record



Invoking Vector Add Method

vect.add(1);

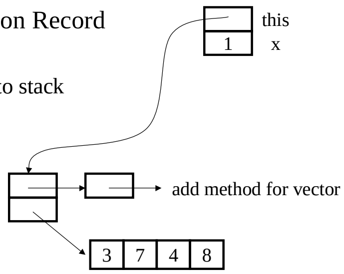
- Create Activation Record
  - this onto stack



Invoking Vector Add Method

vect.add(1);

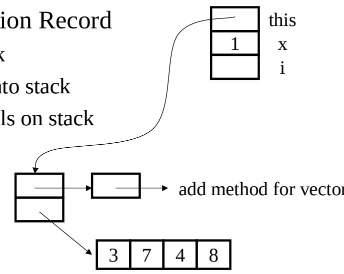
- Create Activation Record
  - this onto stack
  - parameters onto stack



Invoking Vector Add Method

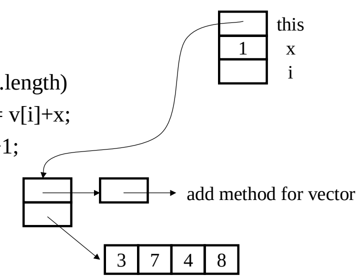
vect.add(1);

- Create Activation Record
  - this onto stack
  - parameters onto stack
  - space for locals on stack



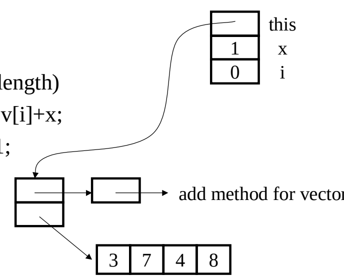
Executing Vector Add Method

```
void add(int x) {
        int i;
        i = 0;
        while (i < v.length)
                v[i] = v[i]+x;
                i = i+1;
  }
```



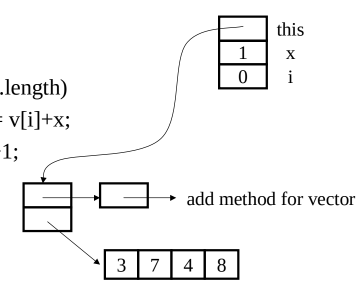
Executing Vector Add Method

```
void add(int x) {
        int i;
        i = 0;
        while (i < v.length)
                v[i] = v[i]+x;
                i = i+1;
  }
```



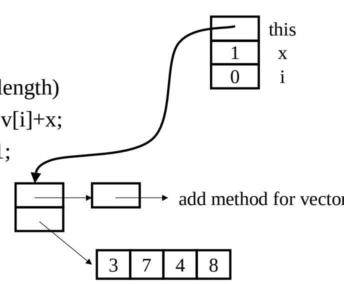
Executing Vector Add Method

```
void add(int x) {
        int i;
        i = 0;
        while (i < v.length)
                v[i] = v[i]+x;
                i = i+1;
  }
```



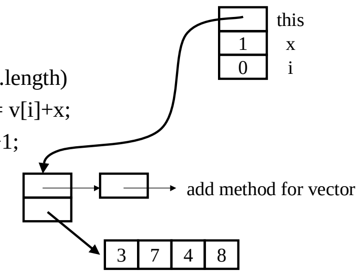
Executing Vector Add Method

```
void add(int x) {
        int i;
        i = 0;
        while (i < v.length)
                v[i] = v[i]+x;
                i = i+1;
  }
```



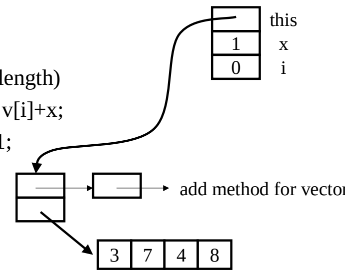
Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



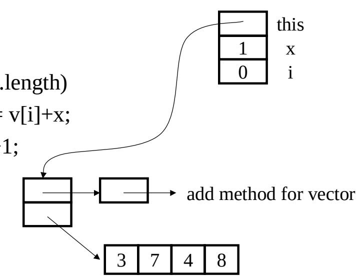
Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



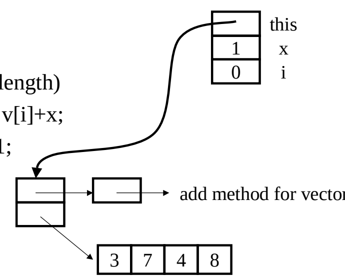
Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



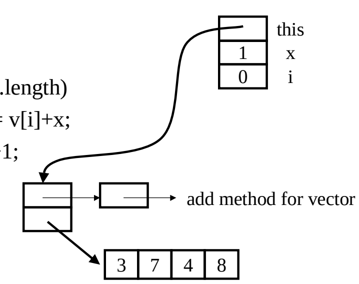
Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



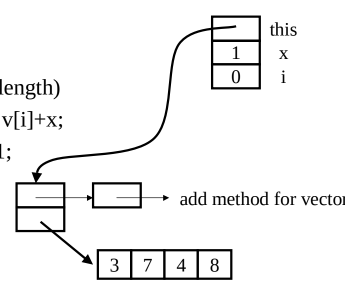
Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



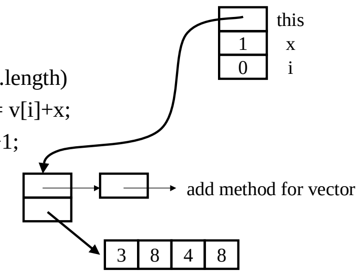
Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



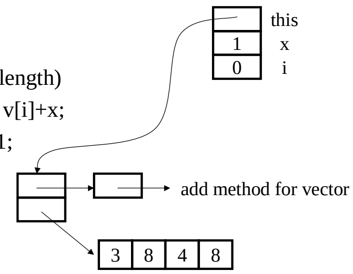
Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



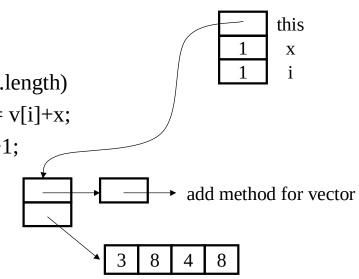
Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



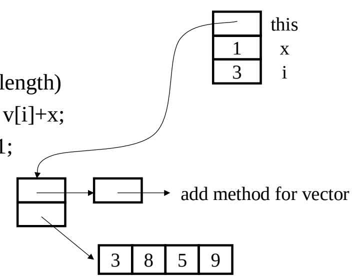
Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



Executing Vector Add Method

```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
        i = i+1;
}
```



Compilation Tasks

- Determine Format of Objects and Arrays in Memory
- Determine Format of Call Stack in Memory
- Generate Code to Read Values
  - this, parameters, array elements, object fields
- Generate Code to Compute New Values
- Generate Code to Write Values
- Generate Code for Control Constructs

Inheritance Example - Point Class

```
class point {
  int c;
  int getColor() { return(c); }
  int distance() { return(0); }
}
```

Point Subclasses

```
class cartesianPoint extends point{
   int x, y;
   int distance() { return(x*x + y*y); }
}
class polarPoint extends point {
   int r, t;
   int distance() { return(r*r); }
   int angle() { return(t); }
}
```


Dynamic Dispatch

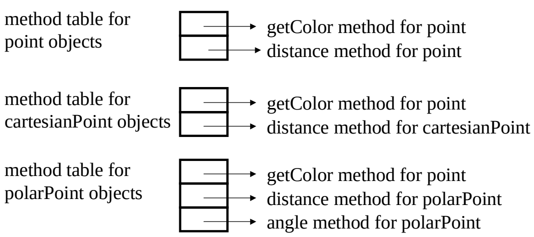
```
if (x == 0) {
   p = new point();
} else if (x < 0) {
   p = new cartesianPoint();
} else if (x > 0) {
   p = new polarPoint();
}
y = p.distance();
```

Which distance method is invoked?

- if p is a point return(0)
- if p is a cartesianPoint return(x*x + y*y)
- if p is a polarPoint return(r*r)
- Invoked Method Depends on Type of Receiver!

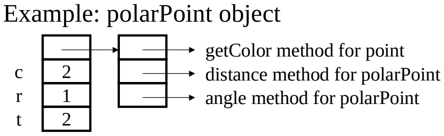
Implementing Dynamic Dispatch

- Basic Mechanism: Method Table

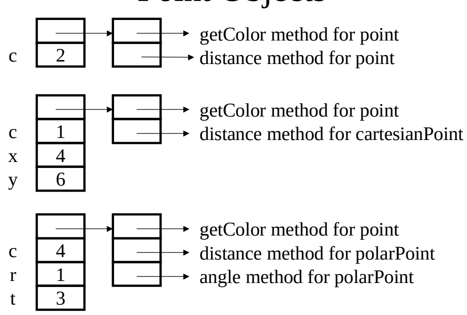


Implementing Object Fields

- Each object is a contiguous piece of memory
- Fields from inheritance hierarchy allocated sequentially in piece of memory
- First word is pointer to method table
- Example: polarPoint object



Point Objects



Invoking Methods

- Compiler Numbers Methods In Each Inheritance Hierarchy
  - getColor is Method 0, distance is Method 1, angle is Method 2
- Method Invocation Sites Access Corresponding Entry in Method Table
- Works For Single Inheritance Only
  - not for multiple inheritance, multiple dispatch, or interfaces

Compilation Tasks

- Determine Object Format in Memory
  - Fields from Parent Classes
  - Fields from Current Class
- Number Methods and Create Method Table
  - Methods from Parent Classes
  - Methods from Current Class
- Generate Code for Methods
  - Field, Local Variable and Parameter Accesses
  - Method Invocations

Symbol Tables - Key Concept in Compilation

- Compiler Uses Symbol Tables to Produce
  - Object Layout in Memory
  - Method Tables
  - Code to Access Object Fields, Local Variables, Parameters

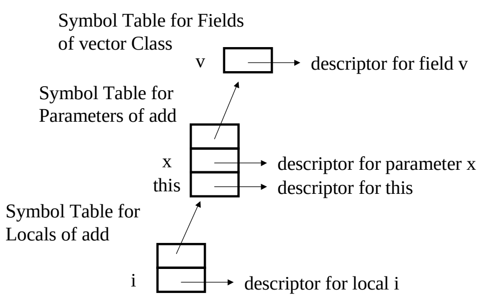
Symbol Tables During Translation From Parse Tree to IR

- Symbol Tables Map Identifiers (strings) to Descriptors (information about identifiers)
- Basic Operation: Lookup
  - Given A String, find Descriptor
  - Typical Implementation: Hash Table
- Examples
  - Given a class name, find class descriptor
  - Given variable name, find descriptor
    - local descriptor, parameter descriptor, field descriptor

Hierarchy In Symbol Tables

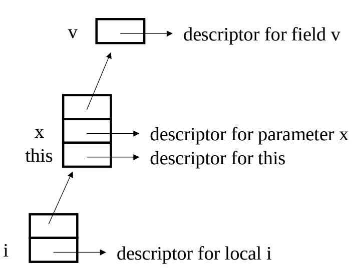
- Hierarchy Comes From
  - Nested Scopes
    - Local Scope Inside Field Scope
  - Inheritance
    - Child Class Inside Parent Class
- Symbol Table Hierarchy Reflects These Hierarchies
- Lookup Proceeds Up Hierarchy Until Descriptor is Found

Hierarchy in vector add Method



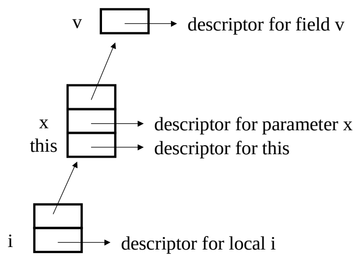
Lookup In vector Example

- v[i] = v[i]+x;



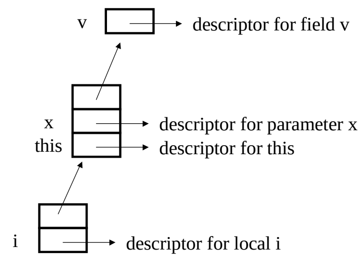
Lookup i In vector Example

- v[i] = v[i]+x;



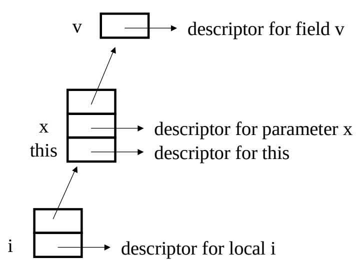
Lookup i In vector Example

- v[i] = v[i]+x;



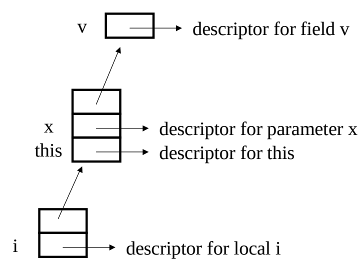
Lookup x In vector Example

- v[i] = v[i]+x;



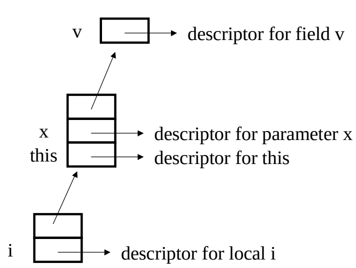
Lookup x In vector Example

- v[i] = v[i]+x;

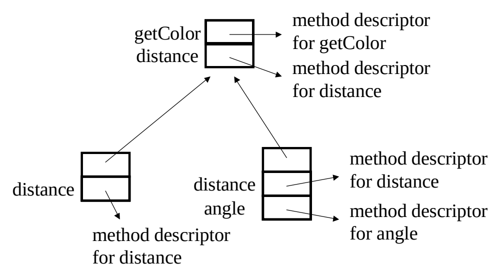


Lookup x In vector Example

- v[i] = v[i]+x;



Hierarchy in Method Symbol Tables for Points



Lookup In Method Symbol Tables

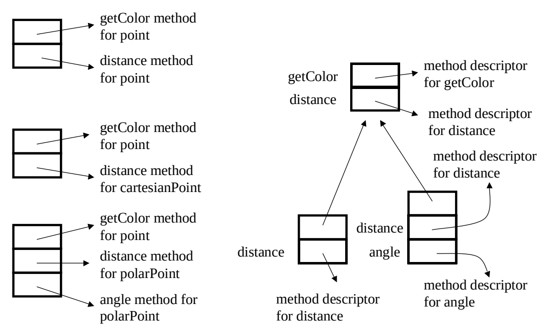
- Starts with method table of declared class of receiver object
- Goes up class hierarchy until method found
  - point p; p = new point(); p.distance();
    - finds distance in point method symbol table
  - point p; p = new cartesianPoint(); p.distance();
    - finds distance in point method symbol table
  - cartesianPoint p; p = new cartesianPoint(); p.getColor();
    - finds getColor in point method symbol table

Static Versus Dynamic Lookup

- Static lookup done at compile time for type checking and code generation
- Dynamic lookup done when program runs to dispatch method call
- Static and dynamic lookup results may differ!
  - point p; p = new cartesianPoint(); p.distance();
    - Static lookup finds distance in point method table
    - Dynamic lookup invokes distance in cartesianPoint class
    - Dynamic dispatch mechanism used to make this happen

Static and Dynamic Tables

- Static Method Symbol Table
  - Used to look up method definitions at compile time
  - Index is method name
  - Lookup starts at method symbol table determined by declared type of receiver object
  - Lookup may traverse multiple symbol tables
- Dynamic Method Table
  - Used to look up method to invoke at run time
  - Index is method number
  - Lookup simply accesses a single table element

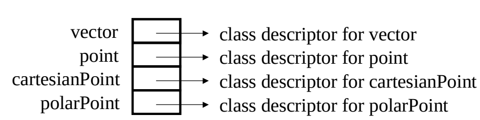


Descriptors

- What do descriptors contain?
- Information used for code generation and semantic analysis
  - local descriptors - name, type, stack offset
  - field descriptors - name, type, object offset
  - method descriptors
    - signature (type of return value, receiver, and parameters)
    - offset in method table
    - reference to local symbol table
    - reference to code for method

Program Symbol Table

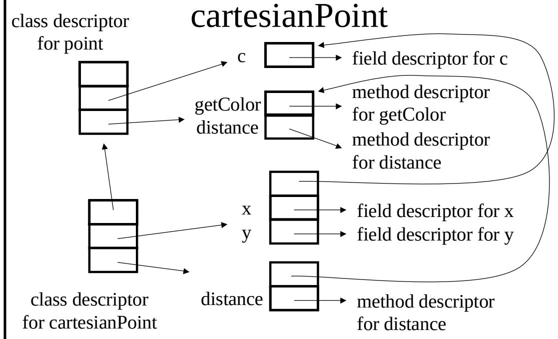
- Maps class names to class descriptors
- Typical Implementation: Hash Table



Class Descriptor

- Has Two Symbol Tables
  - Symbol Table for Methods
    - Parent Symbol Table is Symbol Table for Methods of Parent Class
  - Symbol Table for Fields
    - Parent Symbol Table is Symbol Table for Fields of Parent Class
- Reference to Descriptor of Parent Class

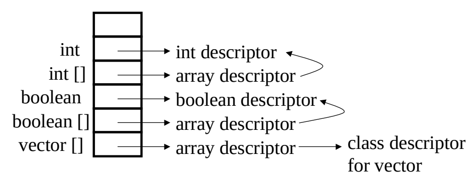
Class Descriptors for point and cartesianPoint



Field, Parameter and Local and Type Descriptors

- Field, Parameter and Local Descriptors Refer to Type Descriptors
  - Base type descriptor: int, boolean
  - Array type descriptor, which contains reference to type descriptor for array elements
  - Class descriptor
- Relatively Simple Type Descriptors
- Base Type Descriptors and Array Descriptors Stored in Type Symbol Table

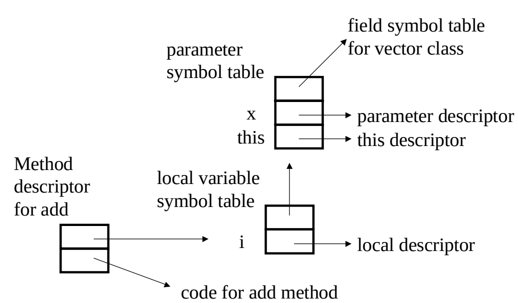
Example Type Symbol Table



Method Descriptors

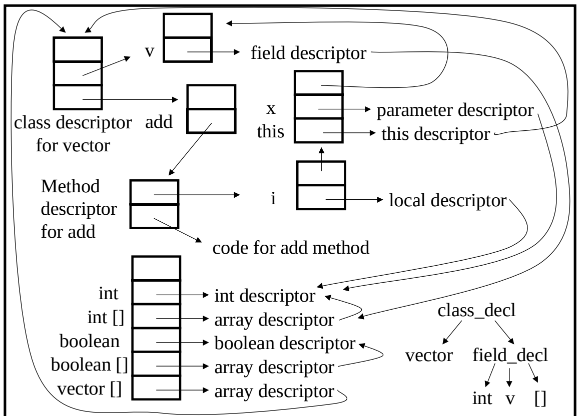
- Contain Reference to Code for Method
- Contain Reference to Local Symbol Table for Local Variables of Method
- Parent Symbol Table of Local Symbol Table is Parameter Symbol Table for Parameters of Method

Method Descriptor for add Method



Symbol Table Summary

- Program Symbol Table (Class Descriptors)
- Class Descriptors
  - ➜ Field Symbol Table (Field Descriptors)
    - ➜ Field Symbol Table for SuperClass
  - ➜ Method Symbol Table (Method Descriptors)
    - ➜ Method Symbol Table for Superclass
- Method Descriptors
  - ➜ Local Variable Symbol Table (Local Variable Descriptors)
    - ➜ Parameter Symbol Table (Parameter Descriptors)
      - ➜ Field Symbol Table of Receiver Class
- Local, Parameter and Field Descriptors
  - ➜ Type Descriptors in Type Symbol Table or Class Descriptors



Translating from Parse Trees to Symbol Tables

What is a Parse Tree?

- Parse Tree Records Results of Parse
- External nodes are terminals/tokens
- Internal nodes are non-terminals

class_decl::=‘class’ name ‘{’field_decl method_decl‘}’
field_decl::= ‘int’ name ‘[];’
method_decl::= ‘void’ name ‘(’ param_decl ‘) ’
‘{‘ var_decl stats ‘}’

Abstract Versus Concrete Trees

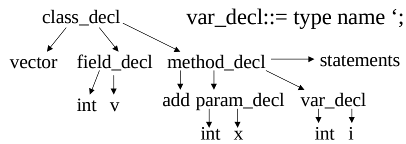
- Remember grammar hacks
  - – left factoring, ambuguity elimination, precedence of binary operators
- Hacks lead to a tree that may not reflect cleanest interpretation of program
- May be more convenient to work with abstract syntax tree (roughly, parse tree from grammar before hacks)

Building IR Alternatives

- Build concrete parse tree in parser, translate to abstract syntax tree, translate to IR
- Build abstract syntax tree in parser, translate to IR
- Roll IR construction into parsing

Example Grammar and Parse Tree

class_decl::=‘class’ name ‘{’field_decl method_decl‘}’
field_decl::= type name ‘[]’ ‘;’
method_decl::= ‘void’ name ‘(’ param_decl ‘) ’
‘{‘ var_decl statements ‘}’
param_decl::= type name ‘;’
var_decl::= type name ‘;’

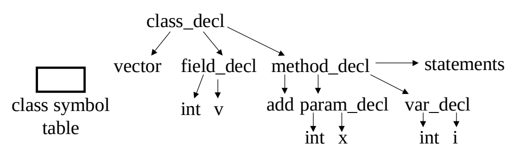


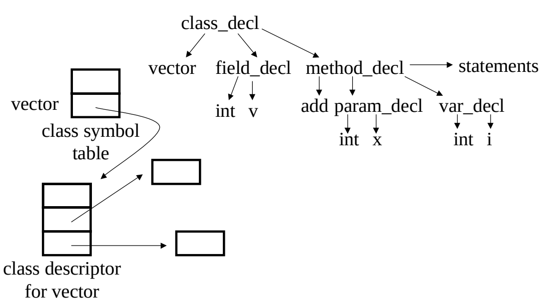
Parse Trees to Symbol Tables

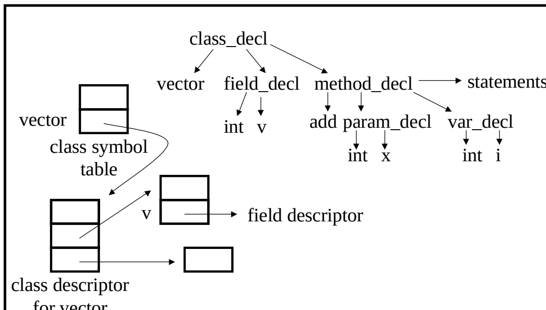
- Recursively Traverse Parse Tree
- Build Up Symbol Tables As Traversal Goes

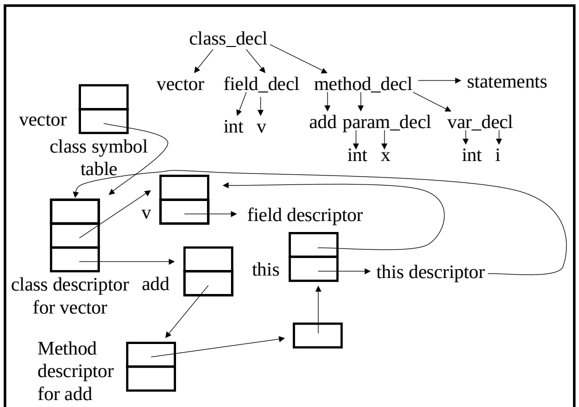
Traversing Class Declarations

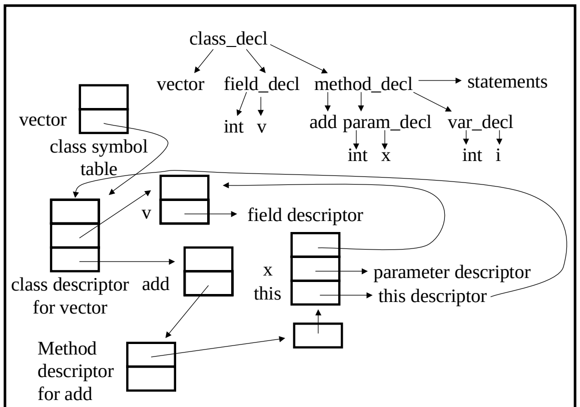
- Extract Class Name and Superclass Name
- Create Class Descriptor (field and method symbol tables), Put Descriptor Into Class Symbol Table
- Put Array Descriptor Into Type Symbol Table
- Lookup Superclass Name in Class Symbol Table, Make Superclass Link in Class Descriptor Point to Retrieved Class Descriptor
- Traverse Field Declarations to Fill Up Field Symbol Table
- Traverse Method Declarations to Fill Up Method Symbol Table

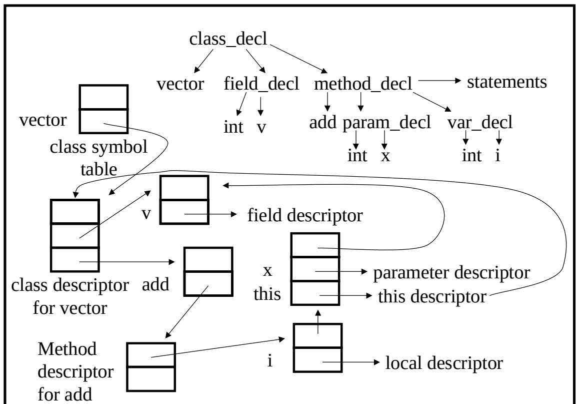


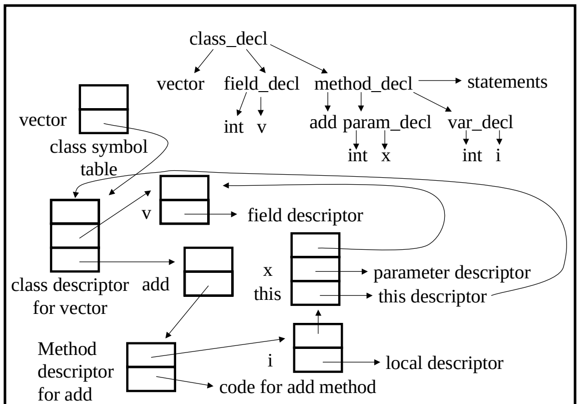








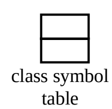


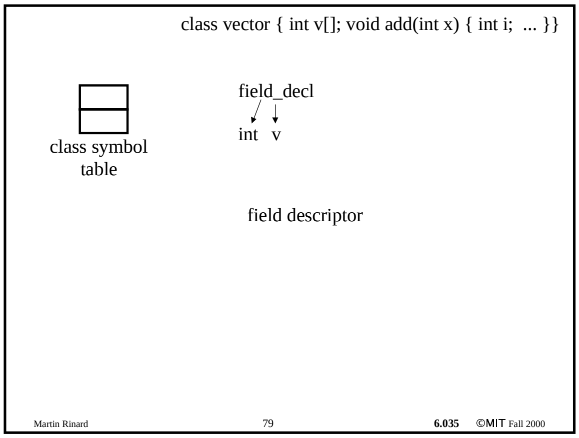


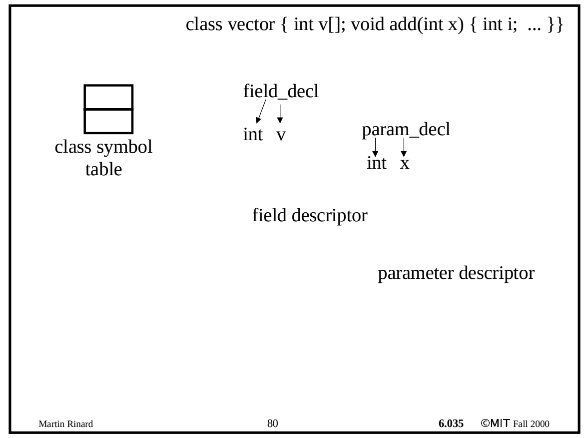
Eliminating Parse Tree Construction

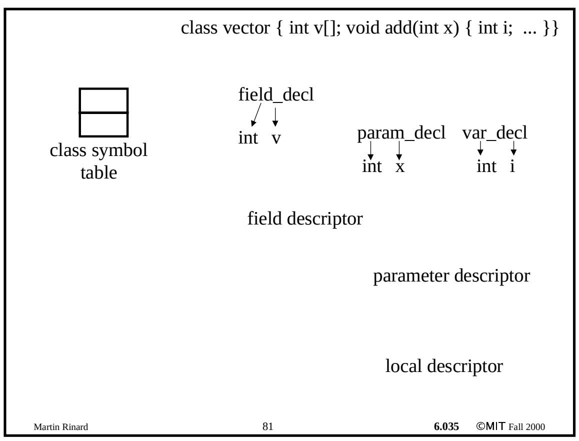
- Parser actions build symbol tables
  - Reduce actions build tables in bottom-up fashion
  - Actions correspond to activities that take place in top-down fashion in parse tree traversal
- Eliminates intermediate construction of parse tree - improves performance
- Also less code to write

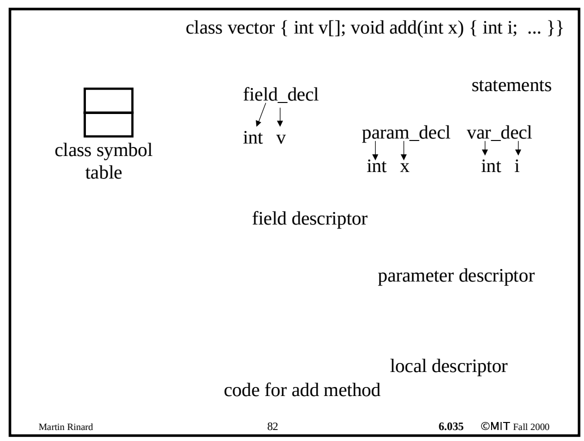
class vector { int v[]; void add(int x) { int i; ... }}

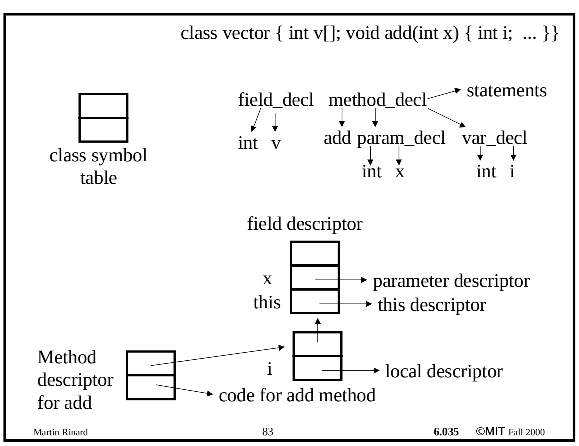


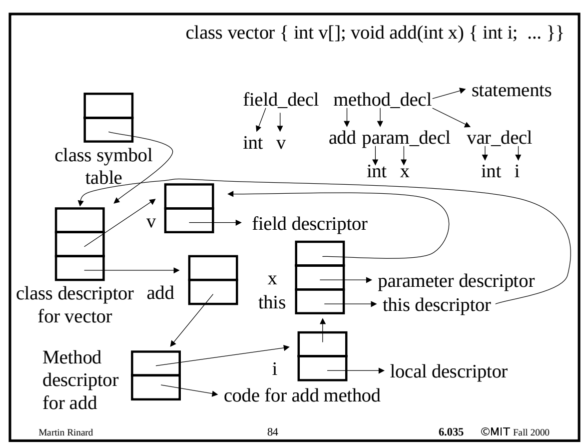

class vector { int v[]; void add(int x) { int i; ... }}
field_decl
int v
class symbol table
field descriptor
Martin Rinard
79
6.035 ©MIT Fall 2000


class vector { int v[]; void add(int x) { int i; ... }}
field_decl
int v
param_decl
int x
class symbol table
field descriptor
parameter descriptor
Martin Rinard
80
6.035 ©MIT Fall 2000


class vector { int v[]; void add(int x) { int i; ... }}
field_decl
int v
param_decl
int x
var_decl
int i
class symbol table
field descriptor
parameter descriptor
local descriptor
Martin Rinard
81
6.035 ©MIT Fall 2000


class vector { int v[]; void add(int x) { int i; ... }}
statements
field_decl
int v
param_decl
int x
var_decl
int i
class symbol table
field descriptor
parameter descriptor
local descriptor
code for add method
Martin Rinard
82
6.035 ©MIT Fall 2000


class vector { int v[]; void add(int x) { int i; ... }}
field_decl
method_decl
statements
int v
add
param_decl
var_decl
int x
int i
class symbol table
field descriptor
x
this
parameter descriptor
this descriptor
Method descriptor for add
i
local descriptor
code for add method
Martin Rinard
83
6.035 ©MIT Fall 2000


class vector { int v[]; void add(int x) { int i; ... }}
field_decl
method_decl
statements
int v
add
param_decl
var_decl
int x
int i
class symbol table
v
field descriptor
class descriptor for vector
add
x
this
parameter descriptor
this descriptor
Method descriptor for add
i
local descriptor
code for add method
Martin Rinard
84
6.035 ©MIT Fall 2000

Nested Scopes

- So far, have seen several kinds of nesting
  - Method symbol tables nested inside class symbol tables
  - Local symbol tables nesting inside method symbol tables
- Nesting disambiguates potential name clashes
  - Same name used for class field and local variable
  - Name refers to local variable inside method

Nested Code Scopes

- Symbol tables can be nested arbitrarily deeply with code nesting:

```
class bar {
  baz x;
  int foo(int x) {
    double x = 5.0;
    { float x = 10.0;
          { int x = 1; ... x ...}
      ... x ...
    }
    ... x ...
}
```

Note: Name clashes with nesting can reflect programming error. Compilers often generate warning messages if it occurs.

Representing Code in High-Level Intermediate Representation

Basic Idea

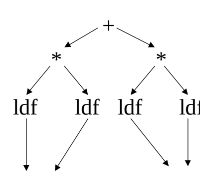
- Move towards assembly language
- Preserve high-level structure
  - object format
  - structured control flow
  - distinction between parameters, locals and fields
- High-level abstractions of assembly language
  - load and store nodes
  - access abstract locals, parameters and fields, not memory locations directly

Representing Expressions

- Expression Trees Represent Expressions
  - Internal Nodes - Operations like +, -, etc.
  - Leaves - Load Nodes Represent Variable Accesses
- Load Nodes
  - ldf node for field accesses - field descriptor
    - (implicitly accesses this - could add a reference to accessed object)
  - ldl node for local variable accesses - local descriptor
  - ldp node for parameter accesses - parameter descriptor
  - lda node for array accesses
    - expression tree for array
    - expression tree for index

Example

x*x + y*y

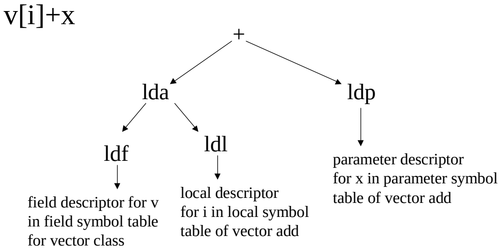


field descriptor for x in field symbol table for cartesianPoint class

field descriptor for y in field symbol table for cartesianPoint class

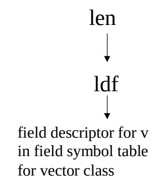
Example

v[i]+x



Special Case: Array Length Operator

- len node represents length of array
  - expression tree for array
- Example: v.length

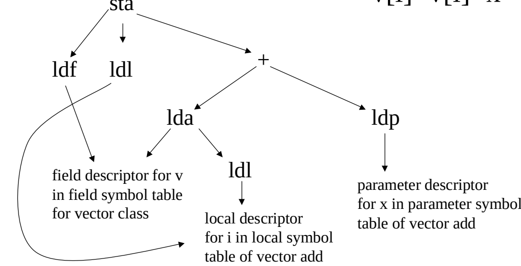


Representing Assignment Statements

- Store Nodes
  - stf for stores to fields
    - field descriptor
    - expression tree for stored value
  - stl for stores to local variables
    - local descriptor
    - expression tree for stored value
  - sta for stores to array elements
    - expression tree for array
    - expression tree for index
    - expression tree for stored value

Example

v[i]=v[i]+x

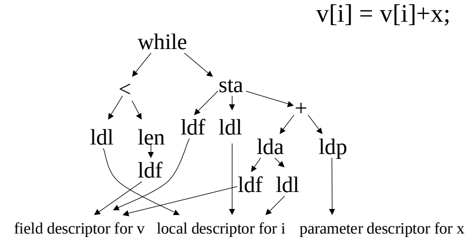


Representing Flow of Control

- Statement Nodes
  - sequence node - first statement, next statement
  - if node
    - expression tree for condition
    - then statement node and else statement node
  - while node
    - expression tree for condition
    - statement node for loop body
  - return node
    - expression tree for return value

Example

while (i < v.length)
v[i] = v[i]+x;



From Parse Trees to Intermediate Representation

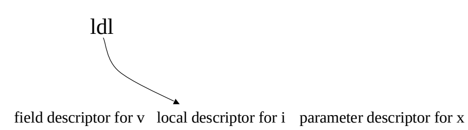
From Parse Trees to IR

- Recursively Traverse Parse Tree
- Build Up Representation Bottom-Up
  - Look Up Variable Identifiers in Symbol Tables
  - Build Load Nodes to Access Variables
  - Build Expressions Out of Load Nodes and Operator Nodes
  - Build Store Nodes for Assignment Statements
  - Combine Store Nodes with Flow of Control Nodes

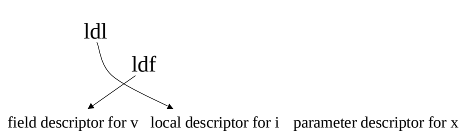

```
while (i < v.length)
  v[i] = v[i]+x;
```

field descriptor for v local descriptor for i parameter descriptor for x

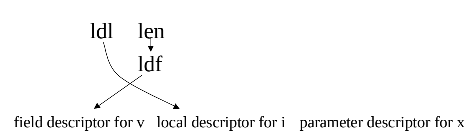

```
while (i < v.length)
  v[i] = v[i]+x;
```



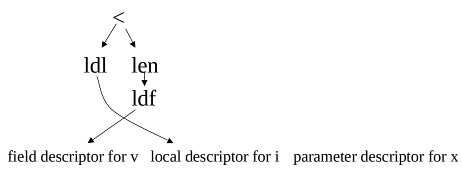

```
while (i < v.length)
  v[i] = v[i]+x;
```




```
while (i < v.length)
  v[i] = v[i]+x;
```

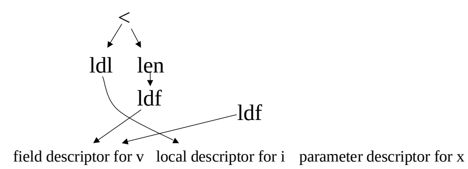


while (i < v.length)
v[i] = v[i]+x;



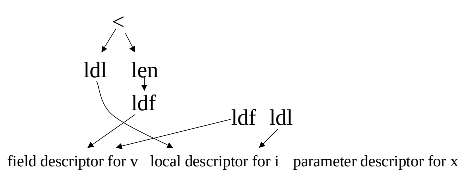
Martin Rinard 103 **6.035** ©MIT Fall 2000

while (i < v.length)
v[i] = v[i]+x;



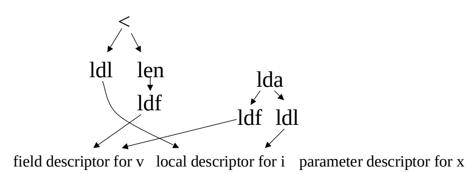
Martin Rinard 104 **6.035** ©MIT Fall 2000

while (i < v.length)
v[i] = v[i]+x;



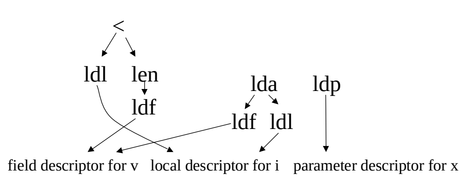
Martin Rinard 105 **6.035** ©MIT Fall 2000

while (i < v.length)
v[i] = v[i]+x;



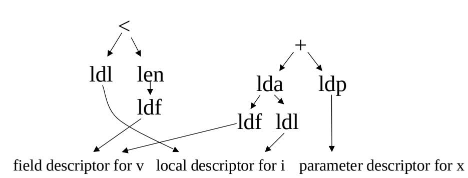
Martin Rinard 106 **6.035** ©MIT Fall 2000

while (i < v.length)
v[i] = v[i]+x;



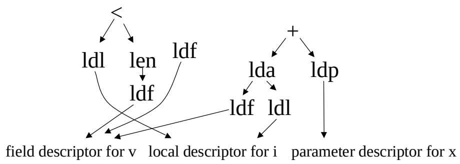
Martin Rinard 107 **6.035** ©MIT Fall 2000

while (i < v.length)
v[i] = v[i]+x;

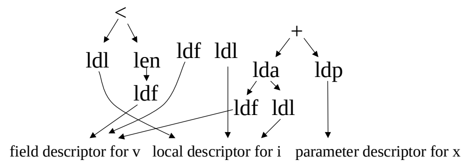


Martin Rinard 108 **6.035** ©MIT Fall 2000

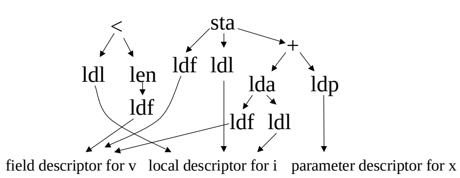
while (i < v.length)
v[i] = v[i]+x;



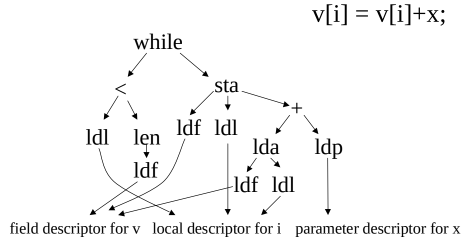
while (i < v.length)
v[i] = v[i]+x;



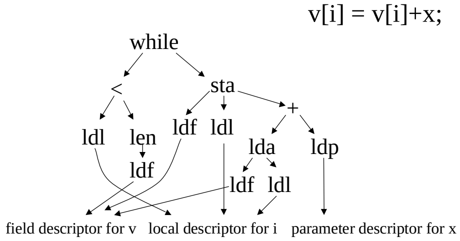
while (i < v.length)
v[i] = v[i]+x;



while (i < v.length)
v[i] = v[i]+x;

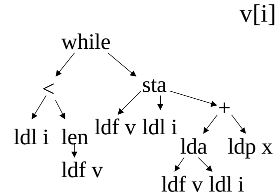


while (i < v.length)
v[i] = v[i]+x;



Abbreviated Notation

while (i < v.length)
v[i] = v[i]+x;



Semantic Analysis

Error Issue

- Have assumed no problems in building IR
- But are many static checks that need to be done as part of translation
- Called Semantic Analysis

Goal of Semantic Analysis

- Ensure that program obeys certain kinds of sanity checks
  - all used variables are defined
  - types are used correctly
  - method calls have correct number and types of parameters and return value
- Checked when build IR

Parameter Descriptors

- When build parameter descriptor, have
  - name of type
  - name of parameter
- What is the check? Must make sure name of type identifies a valid type
  - look up name in type symbol table
  - if not there, fails semantic check

Local Symbol Table

- When build local symbol table, have a list of local descriptors
- What to check for?
  - duplicate variable names
  - shadowed variable names
- When to check?
  - when insert descriptor into local symbol table
- Parameter and field symbol tables similar

Class Descriptor

- When build class descriptor, have
  - class name and name of superclass
  - field symbol table
  - method symbol table
- What to check?
  - Superclass name corresponds to actual class
  - No name clashes between field names of subclass and superclasses
  - Overridden methods match parameters and return type declarations of superclass

Load Instruction

- What does compiler have? Variable name.
- What does it do? Look up variable name.
  - If in local symbol table, reference local descriptor
  - If in parameter symbol table, reference parameter descriptor
  - If in field symbol table, reference field descriptor
  - If not found, semantic error

Add Operations

- What does compiler have?
  - two expressions
- What can go wrong?
  - expressions have wrong type
  - must both be integers (for example)
- So compiler checks type of expressions
  - load instructions record type of accessed variable
  - operations record type of produced expression
  - so just check types, if wrong, semantic error

Type Inference for Add Operations

- Most languages let you add floats, ints, doubles
- What are issues?
  - Types of result of add operation
  - Coercions on operands of add operation
- Standard rules usually apply
  - If add an int and a float, coerce the int to a float, do the add with the floats, and the result is a float.
  - If add a float and a double, coerce the float to a double, do the add with the doubles, result is double

Add Rules

- Basic Principle: Hierarchy of number types (int, then float, then double)
- All coercions go up hierarchy
  - int to float; int, float to double
- Result is type of operand highest up in hierarchy
  - int + float is float, int + double is double, float + double is double
- Interesting oddity: C converts float procedure arguments to doubles. Why?

Type Inference

- Infer types without explicit type declarations
- Add is very restricted case of type inference
- Big topic in recent programming language research
  - How many type declarations can you omit?
  - Tied to polymorphism

Equality Expressions

- If build expression A = B, must check compatability
- A compatable with B or B compatable with A
- Int compatable with Int
- Class C compatable with Class D if
  C inherits from D (but not vice-versa)

Method Invocations

- What does compiler have?
  - method name, receiver expression, actual parameters
- Checks:
  - receiver expression is class type
  - method name is defined in receiver's class type
  - types of actual parameters match types of formal parameters
  - What does match mean?
    - same type?
    - compatible type?

Semantic Check Summary

- Do semantic checks when build IR
- Many correspond to making sure entities are there to build correct IR
- Others correspond to simple sanity checks
- Each language has a list that must be checked

Broader Characterization of Program Errors and Approaches

Kinds of Program Errors

- Incorrect Program Execution in Any Context
  - Array index out of bounds, Divide by zero
  - Dereference a NULL or invalid pointer
  - Read uninitialized data
  - Type Errors
    - Add a pointer and a boolean
    - Dereference a character
  - Memory System Errors
    - Access Deallocated Memory
    - Memory Leak

Standard Requirements

- Variables declared before uses
- No variable name declared twice in same scope
- Formal (at declaration site) and actual (at call site) parameters match in number and type
- Types of return values and assigned variables match
- Plus other standard requirements

More Program Errors

- Incorrect in current program
  - Tons of programming errors
- Language implementation can and often does catch errors that are incorrect in any context
- You are on your own with programming errors
- Particularly nasty class of errors
  - Programming language implementation errors
  - Hardware errors

Safe Languages

- Safe Languages with Strong Typing (Java, ML)
  - All errors caught either statically or dynamically
    - Type errors caught statically
    - Others caught dynamically
    - Garbage collection required to avoid memory errors
- Safe Languages with Dynamic Typing (Scheme)
  - All errors caught dynamically

Dynamic Versus Static Typing

- Personality Issue
  - Some like to think about the future
    - Type system is a great framework for thinking about the future
  - Some like instant responses
    - Run the program and see what happens!
    - Interpreted languages, dynamic typing, nothing between the programmer and running the program

Unsafe Languages

- Some Errors Uncaught
- For others, Program Result Undefined (!!!)
- C
  - No array bounds checking
  - Pointer and Memory errors possible
  - Why use this language?
- Fortran
  - Result undefined if aliased parameters

Philosophy of Early C Compilers

- Compiler requires only information it needs to generate code.
  - No declaration? Variable is an integer!
  - No check of parameter types at call site against parameter types at declaration site
  - Pointers and integers are interchangable
  - Can cast pointers like crazy
  - Operator for every machine instruction
- Programmer knows what is going on, compiler should just get out of the way!

Philosophy of Current Compilers

- Programmers are prone to make silly errors
- Part of compiler’s job is to help find them
- Don’t let any potential type errors through!
- Looks like the trend is for even more checks

Type Checking Versus Type Inference

- Type information is redundant
  - Not required to execute program
  - Why write it down?
- Type checking
  - Programmer gives type declarations
  - Implementation checks that declarations are correct
- Type inference
  - Type declarations omitted
  - Implementation reconstructs types

Conversion to Low Level Intermediate Representation

- Convert Structured Flow of Control to Branch Flow of Control
  - Conditional Branches
- Convert Structured Model of Memory to Flat Memory Model
  - Stack Addressing of Locals
  - Flat Addressing of Fields
  - Flat Addressing of Arrays

Goal
Remain Largely Machine Independent

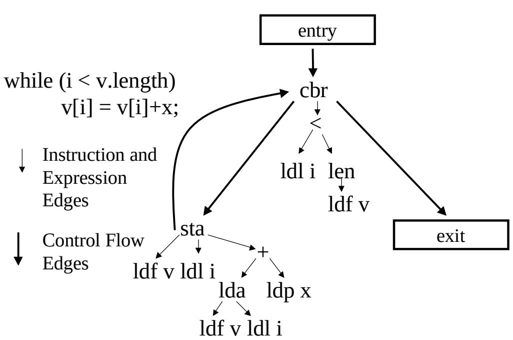
But
Move Closer to Standard Machine Model (flat address space, branches)

Converting Structured Flow of Control To Unstructured Flow of Control

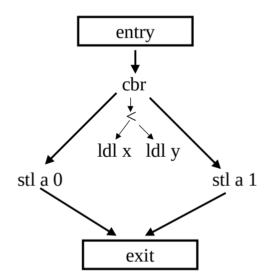
Program Representation

- Control Flow Graph (CFG)
  - CFG Nodes are Instruction Nodes
    - stl, sta, stf, cbr nodes are instruction nodes
    - ldl, lda, ldp, len, +, <, ... are expression nodes
  - CFG Edges Represent Flow of Control
  - Forks At Conditional Jump Instructions
  - Merges When Control Can Reach A Point Multiple Ways
  - Entry and Exit Nodes


```
while (i < v.length)
      v[i] = v[i]+x;
```

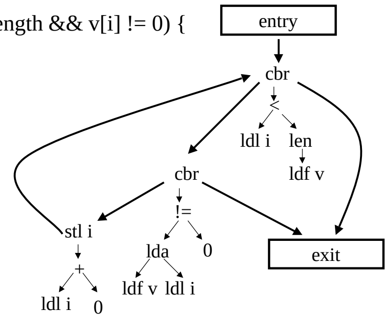



```
if (x < y) {
      a = 0;
} else {
      a = 1;
}
```



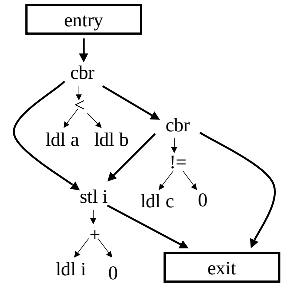
Short-Circuit Conditionals

```
while (i < v.length && v[i] != 0) {
        i = i+1;
}
```



More Short-Circuit Conditionals

```
if (a < b || c != 0) {
        i = i+1;
}
```



Routines for Destructuring Program Representation

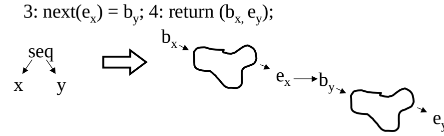
destruct(n)
- generates lowered form of structured code represented by n
- returns (b,e) - b is begin node, e is end node in destructed form

shortcircuit(c, t, f)
- generates short-circuit form of conditional represented by c
- if c is true, control flows to t node
- if c is false, control flows to f node
- returns b - b is begin node for condition evaluation

new kind of node - nop node

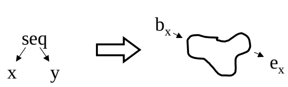
Destructuring Seq Nodes

destruct(n)
- generates lowered form of structured code represented by n
- returns (b,e) - b is begin node, e is end node in destructed form
- if n is of the form seq x y
  - 1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);
  - 3: next(e_x) = b_y; 4: return (b_x, e_y);



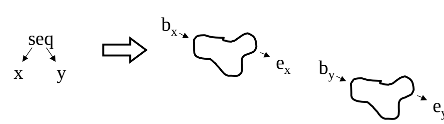
Destructuring Seq Nodes

destruct(n)
- generates lowered form of structured code represented by n
- returns (b,e) - b is begin node, e is end node in destructed form
- if n is of the form seq x y
  - 1: (b_x,e_x) = destruct(x);



Destructuring Seq Nodes

destruct(n)
- generates lowered form of structured code represented by n
- returns (b,e) - b is begin node, e is end node in destructed form
- if n is of the form seq x y
  - 1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);



Destructuring Seq Nodes

destruct(n)

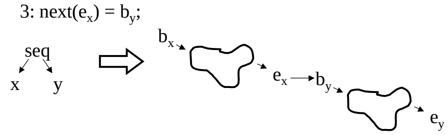
generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form seq x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

3: next$(e_x) = b_y$;



Destructuring Seq Nodes

destruct(n)

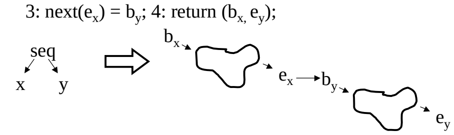
generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form seq x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

3: next$(e_x) = b_y$; 4: return (b_x, e_y);



Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n

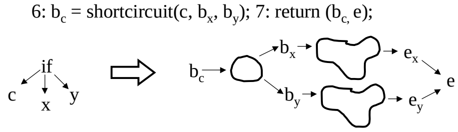
returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

3: e = new nop; 4: next(e_x) = e; 5: next(e_y) = e;

6: b_c = shortcircuit(c, b_x, b_y); 7: return (b_c, e);



Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form if c x y

1: (b_x,e_x) = destruct(x);

Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

Destructuring If Nodes

destruct(n)

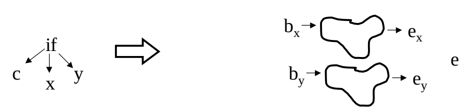
generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

3: e = new nop;

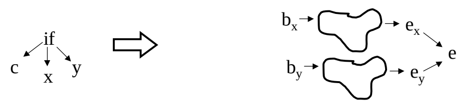


Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n
returns (b,e) - b is begin node, e is end node in destructed form
if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);
3: e = new nop; 4: next(e_x) = e; 5: next(e_y) = e;

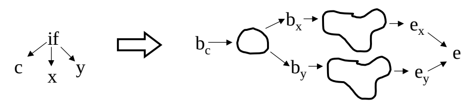


Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n
returns (b,e) - b is begin node, e is end node in destructed form
if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);
3: e = new nop; 4: next(e_x) = e; 5: next(e_y) = e;
6: b_c = shortcircuit(c, b_x, b_y);

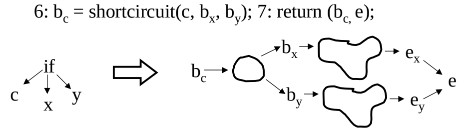


Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n
returns (b,e) - b is begin node, e is end node in destructed form
if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);
3: e = new nop; 4: next(e_x) = e; 5: next(e_y) = e;
6: b_c = shortcircuit(c, b_x, b_y); 7: return (b_c, e);

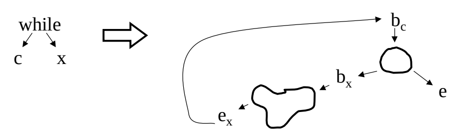


Destructuring While Nodes

destruct(n)

generates lowered form of structured code represented by n
returns (b,e) - b is begin node, e is end node in destructed form
if n is of the form while c x

1: e = new nop; 2: (b_x,e_x) = destruct(x);
3: b_c = shortcircuit(c, b_x, e); 4: next(e_x) = b_c; 5: return (b_c, e);



Destructuring While Nodes

destruct(n)

generates lowered form of structured code represented by n
returns (b,e) - b is begin node, e is end node in destructed form
if n is of the form while c x

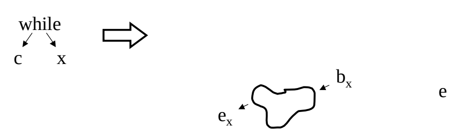
1: e = new nop;

Destructuring While Nodes

destruct(n)

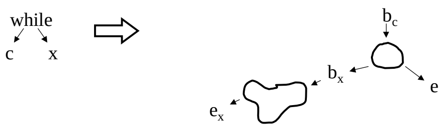
generates lowered form of structured code represented by n
returns (b,e) - b is begin node, e is end node in destructed form
if n is of the form while c x

1: e = new nop; 2: (b_x,e_x) = destruct(x);



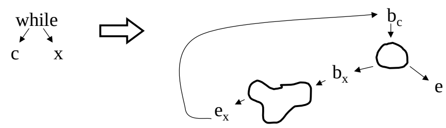
Destructuring While Nodes

destruct(n)
- generates lowered form of structured code represented by n
- returns (b,e) - b is begin node, e is end node in destructed form
- if n is of the form while c x
  - 1: e = new nop; 2: (b_x,e_x) = destruct(x);
  - 3: b_c = shortcircuit(c, b_x, e);



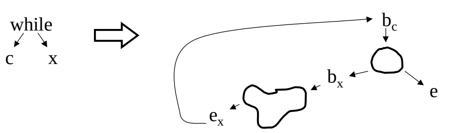
Destructuring While Nodes

destruct(n)
- generates lowered form of structured code represented by n
- returns (b,e) - b is begin node, e is end node in destructed form
- if n is of the form while c x
  - 1: e = new nop; 2: (b_x,e_x) = destruct(x);
  - 3: b_c = shortcircuit(c, b_x, e); 4: next(e_x) = b_c;



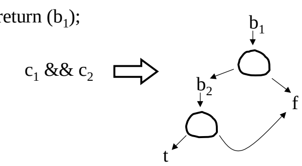
Destructuring While Nodes

destruct(n)
- generates lowered form of structured code represented by n
- returns (b,e) - b is begin node, e is end node in destructed form
- if n is of the form while c x
  - 1: e = new nop; 2: (b_x,e_x) = destruct(x);
  - 3: b_c = shortcircuit(c, b_x, e); 4: next(e_x) = b_c; 5: return (b_c, e);



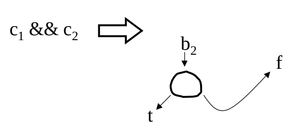
Shortcircuiting And Conditions

shortcircuit(c, t, f)
- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form c_1 && c_2
  - 1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, b_2, f);
  - 3: return (b_1);



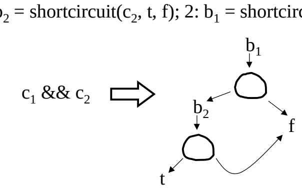
Shortcircuiting And Conditions

shortcircuit(c, t, f)
- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form c_1 && c_2
  - 1: b_2 = shortcircuit(c_2, t, f);



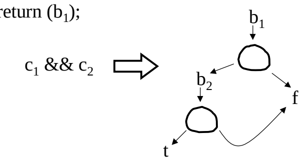
Shortcircuiting And Conditions

shortcircuit(c, t, f)
- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form c_1 && c_2
  - 1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, b_2, f);



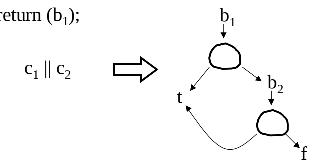
Shortcircuiting And Conditions

shortcircuit(c, t, f)
- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form c_1 && c_2
  - 1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, b_2, f);
  - 3: return (b_1);



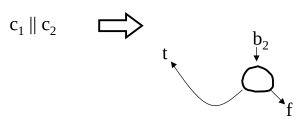
Shortcircuiting Or Conditions

shortcircuit(c, t, f)
- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form c_1 || c_2
  - 1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, t, b_2);
  - 3: return (b_1);



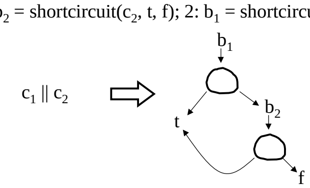
Shortcircuiting Or Conditions

shortcircuit(c, t, f)
- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form c_1 || c_2
  - 1: b_2 = shortcircuit(c_2, t, f);



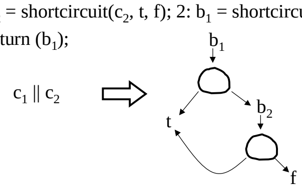
Shortcircuiting Or Conditions

shortcircuit(c, t, f)
- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form c_1 || c_2
  - 1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, t, b_2);



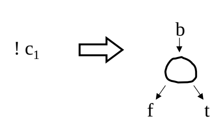
Shortcircuiting Or Conditions

shortcircuit(c, t, f)
- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form c_1 || c_2
  - 1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, t, b_2);
  - 3: return (b_1);



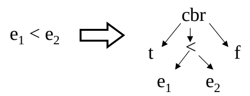
Shortcircuiting Not Conditions

shortcircuit(c, t, f)
- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form ! c_1
  - 1: b = shortcircuit(c_1, f, t); return(b);



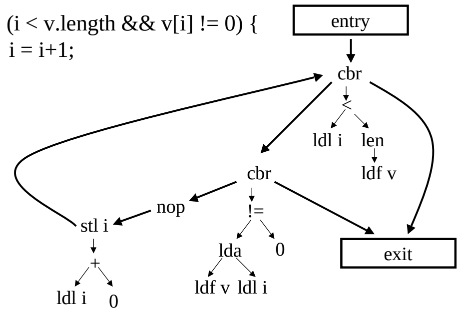
Computed Conditions

shortcircuit(c, t, f)
generates shortcircuit form of conditional represented by c
returns b - b is begin node of shortcircuit form
if c is of the form $e_1 < e_2$
1: b = new cbr($e_1 < e_2$, t, f); 2: return (b);

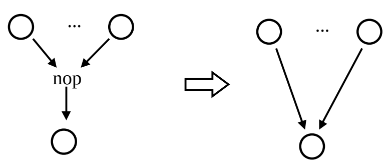


Nops In Destructured Representation

```
while (i < v.length && v[i] != 0) {
    i = i+1;
}
```



Eliminating Nops Via Peephole Optimization



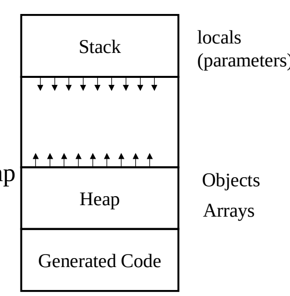
Converting to Flat Address Space

Memory Model for Target Machine

- Single flat memory
  - composed of words
  - byte addressable
- Nodes Model Load and Store Instructions
  - ld addr,offset - result is contents of memory at location addr+offset
  - st addr,offset,value - stores value in location addr+offset
  - Will replace lda, ldf, ldl nodes with ld nodes
  - Will replace sta, stf, stl nodes with st nodes

Memory Layout

- When is generated code set up?
- When does stack grow and shrink?
- When does the heap grow and shrink?

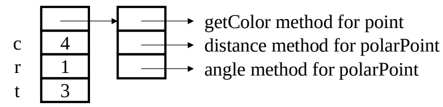


Parameters

- Most Machines Have Calling Conventions
  - First parameter in register 5,
  - Second parameter in register 6, ...
- Calling Conventions Vary Across Machines
- Will Assume Each Parameter is One Word
- Will Address Parameters by Number
  - ldp <parameter number>
  - this is parameter 0

Object and Array Layouts

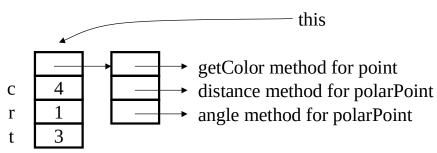
- Contiguous Allocation for Objects and Arrays
- Fields Laid Out Consecutively
  - Method Table in First Word



- Array Elements Laid Out Consecutively
  - Length in First Word 3 7 4 8

Accessing Fields

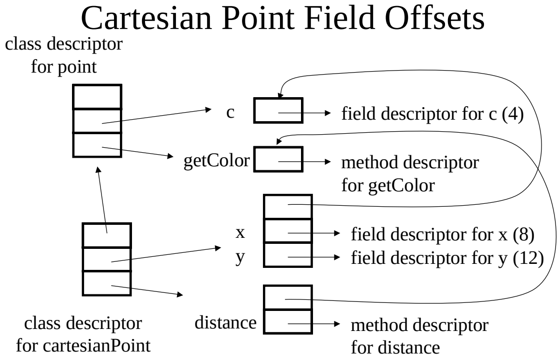
- Assume this points to start of object
- What is address of r field?
  - assume each field takes 4 bytes
- this+(2*4), or base+field offset



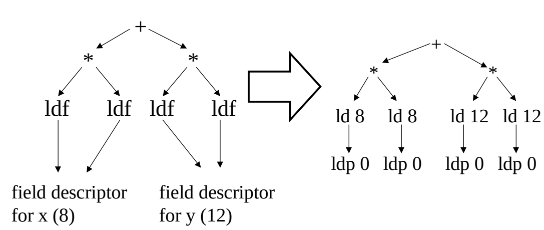
Converting ldf Nodes to ld Nodes

- Compute field offsets
  - traverse class hierarchy (field symbol tables)
  - offsets for subclass start where offsets for superclass end
  - store offsets in field symbol tables
- Use offsets to replace ldf nodes with ld nodes

Cartesian Point Field Offsets



Example Expression
x*x + y*y



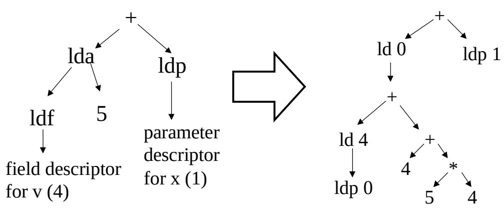
Accessing Array Elements

- Assume array variable points to start of array
- Array elements stored contiguously
- Don't forget length at front of array
- What is address of v[5]?
- Assume 4 byte integers
- (address in v) + 4 + (5*4)
- Array Base + 4 + (index * element size)

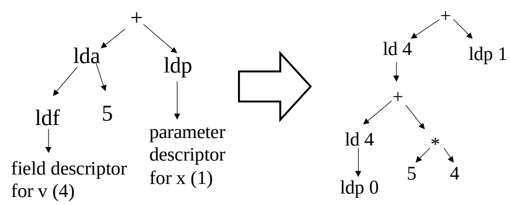
Converting lda Nodes to ld Nodes

- Compute Address of Array Element
  - Base + 4 + (index * element size)
- ld From that Address
- Offset of ld Node is 0
- Optimization
  - Put offset to skip length in ld instruction

Example: v[5]+x



Length Offset in ld Instruction



Local Variables

- Assume are allocated on call stack
- Address using offsets from call stack pointer
- Remember, stack grows down, not up, so offsets are all positive
- Special symbol sp contains stack pointer

Actions On Method Invocation

- Caller
  - Set up parameters using calling convention
  - Set up return address using calling convention
  - Jump to callee
- Callee
  - Allocate stack frame = Decrease stack pointer
  - Compute
  - Set up return value using calling convention
  - Deallocate stack frame = Increase stack pointer
  - Return to caller

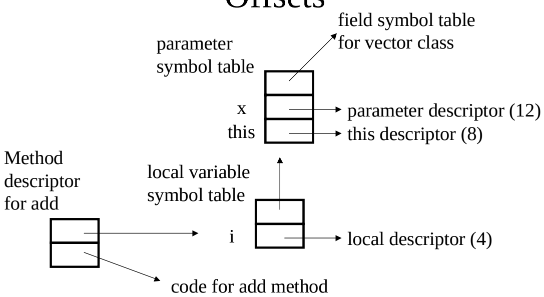
Stack Management

- Compute size of stack frame
  - allocated when enter method
  - deallocated when return
  - holds all local variables
  - plus space for all parameters
  - assume all locals and parameters are one word long
- Compute offsets of locals and parameters
  - store in local and parameter symbol tables
  - still use ldp nodes to access parameters

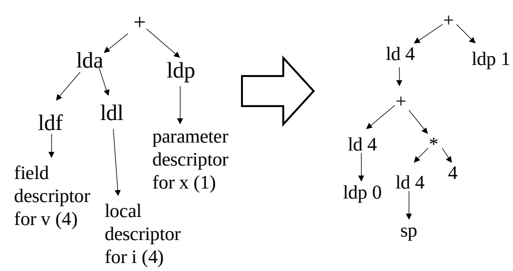
Eliminating ldl Nodes

- Use offsets in local symbol table and sp
- Replace ldl nodes with ld nodes

Example Local and Parameter Offsets

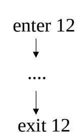


Example: v[i]+x



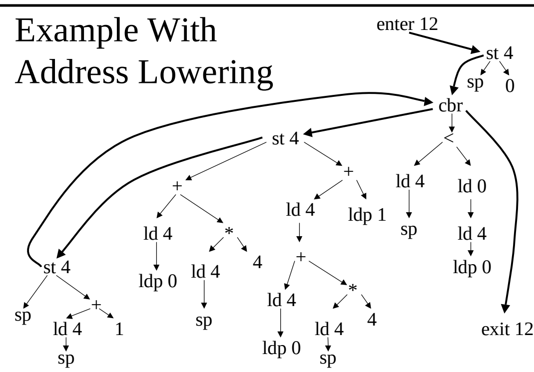
Enter and Exit Nodes for add Method

void add(int x) {
  int i;
  ...
}



- How big is stack frame for add?
  - 12 bytes (space for this, x, i)
  - assuming 4 byte words

Example With Address Lowering



Summary

- Field Accesses Translate To ld or st nodes
  - address is object pointer, offset is field offset
- Array Accesses Translate To ld or st nodes
  - address is array pointer + 4 + (index * element size)
  - Put length offset (4) in ld or st instruction
- Local Accesses Translate To ld or st nodes
  - address is sp, offset is local offset
- Parameter Accesses Translate To
  - lpd instructions - specify parameter number
- Enter and Exit Nodes Specify Stack Frame Size