**Classes**

- simpleJava classes are equivalent to C structs

```java
class myclass {
    int x;
    int y;
    boolean z;
}
```

```c
struct mystruct {
    int x;
    int y;
    int z;
}
```

- What do we need to add to simpleJava classes to make them true objects?
To extend simpleJava to be a true object oriented language, classes will need methods.

New definition of classes:

```java
class <classname> {
    <List of instance variable declarations, method prototypes, & method definitions>
}
```

As in regular Java, instance variable declarations & method definitions can be interleaved.
class Point {
    int xpos;
    int ypos;
    Point(int x, int y) {
        xpos = x;
        ypos = y;
    }
    int getX() {
        return xpos;
    }
    int getY() {
        return ypos;
    }
    void setX(int x) {
        xpos = x;
    }
    void setY(int y) {
        ypos = y;
    }
}
10-3: Adding Methods to Classes

class Rectangle {

    Point lowerleftpt;
    Point upperrightpt;

    Rectangle(Point lowerleft, Point upperright) {
        lowerleftpt = lowerleft;
        upperrightpt = upperright;
    }
    int length() {
        return upperrightpt.getX() - lowerleftpt.getX();
    }
    int height() {
        return upperrightpt.getY() - lowerleftpt.getY();
    }
    int area() {
        return length() * height();
    }
}
}
10-4: “This” local variable

- All methods have an implicit “this” local variable, a pointer to the data block of the class
- Original version:
  ```java
  Point(int x, int y) {
    xpos = x;
    ypos = y;
  }
  ```
- Alternate version:
  ```java
  Point(int x, int y) {
    this.xpos = x;
    this.ypos = y;
  }
  ```
Compiler Changes for Methods

- Modify Abstract Syntax Tree
- Modify Semantic Analyzer
  - Classes will need function environments
  - “This” pointer defined for methods
- Modify Abstract Assembly Generator
  - Maintain the “this” pointer
Modifications to AST

• What changes are necessary to the AST to add methods to classes?
  • Which tree nodes need to be changed?
  • How should they be modified?
10-7: Modifications to AST

- ASTClass: contain prototypes and method definitions as well as instance variable declarations
  - Add abstract class ASTClassElem
    - ASTMethod, ASTMethodPrototype, ASTInstanceVarDef would be subclasses
- What would .jj file look like?
void classdef() :
{}
{
  <CLASSS> <IDENTIFIER>
    <LBRACE> classElems() <RBRACE>
}

def classElems() :
{
  
  (classElem())*

}

def classElem() :
{}
{

  <IDENTIFIER> <IDENTIFIER>
    (((<LBRACK> <RBRACK>)* <SEMICOLON>) |
    (<LPAREN> formals() <RPAREN>) (<SEMICOLON> | <LBRACE> statements() <RBRACE>))
}
10-9: Modifications to AST

- Change AST to allow method calls

  x.y.foo()
  z[3].y[4].bar(x.foo())
10-10: Modifications to AST

- x.foo(3,4)
• x.foo(3,4)
Modifications to AST

- A[3].foo()
Modifications to AST

- A[3].foo()

```
MethodCall(foo)
  ArrayVar
    BaseVar  IntegerLiteral(3)
      identifier(A)
```
10-14: Modifications to AST

- B[4].x.foo(5)
10-15: **Modifications to AST**

- $B[4].x.\text{foo}(5)$

```
MethodCall(bar)
  ClassVar Integer_Literal(5)
    ArrayVar identifier(x)
      BaseVar Integer_Literal(4)
        identifier(B)
```
Constructors have slightly different syntax than other functions

class Integer {
    int data;
    Integer(int value) {
        data = value;
    }
    int intValue() {
        return data;
    }
}
For the constructor

```java
Integer(int value) {
    data = value;
}
```

Create the abstract syntax

```java
Integer Integer(int value) {
    data = value;
    return this;
}
```
Modifications to AST

Integer(int value) {
    data = value;
}

- Create the abstract syntax

    Integer Integer(int value) {
        data = value;
        return this;
    }

- When will this not work?
• Constructors
  • AST needs to be modified to allow constructors to take input parameters
Without methods, the internal representation of a class contains only a variable environment.

How should this change if we add methods to classes?
10-21: Changes in Semantic Analysis

- Add Function Environment to the internal representation of class types

```java
class Integer {
    int data;

    Integer(int initvalue) {
        data = initvalue;
    }

    int intValue() {
        return data;
    }
}
```
10-22: Changes in Semantic Analysis

Type Environment:

KeyStack
- Integer
- void
- boolean
- int
- newscope

Integer

boolean

void

void

int

Variable Environment

CLASS TYPE

FUNCTION TYPE

Return Type

Parameters

newscope

intValue

FUNCTION TYPE

Return Type

Parameters

(new)
• What do you do now to analyze a class?
What do you do now to analyze a class?

- Create a new variable environment
- Analyze (visit) the instance variable declarations
  - Adding each variable to the new variable environment
- Create a new internal representation of a Class, using the new variable environment
- Add Class to the class environment
To analyze a Class Definition

- Create a new, local function & variable environment
- Build a new internal representation of a Class, using newly created (local) function & variable environments
- Add the class to the type environment
- Begin a new scope in the global variable environment & function environments
- Add “this” variable to the variable environment, using this class type
To analyze a Class Definition

- Analyze variables (adding to global variable environment \textit{and} the local variable environment)
- Analyze function definitions (adding to global function environment \textit{and} the local function environment)
- End scopes in global environments
To analyze a method call `x.foo()`

- Analyze variable `x`, which should return a class type
- Look up the function `foo` in the function environment of the variable `x`
- Return the type of `foo()`
class MethodExample {

    int instanceVar;

    MethodExample() {}  

    int foo(int x) {
        return x + instanceVar;
    }

    int bar(int x) {
        return foo(x);
    }
}

• What extra work do we need to do to allow instanceVar to be seen in foo?
What extra work do we need to do to allow instanceVar to be seen in foo?

- None! InstanceVar is already in the global variable environment.
- (We will need to do a little extra work to generate abstract assembly – why?)
What extra work do we need to do to allow bar to call the function foo?
What extra work do we need to do to allow bar to call the function foo?

- None!
- When we analyzed foo, we added the proper prototype to both the global function environment and the local function environment.
new MyClass(3,4)

- Look up “MyClass” in the type environment, to get the definition of the class
- Look up “MyClass” in the function environment for the class
- Check to see that the number & types of parameters match
- Return the type of MyClass
class SimpleClass {

    int x;
    int y;

    SimpleClass(int initialx, initialy) {
        x = initialx;
        y = initialy;
    }

    int average() {
        int ave;
        ave = (x + y) / 2;
        return ave;
    }

    void main {
        SimpleClass z;
        int w;
        z = new SimpleClass(3,4);
        w = z.average();
    }
}
To analyze class `SimpleClass`

- Create a new empty variable & function environment
- Create a new class type that contains these environments
- Begin a new scope in the global function & variable environments
- Add “this” to the global variable environment, with type `SimpleClass`
- Add $x$ and $y$ to *both* the local and global variable environment
- Add the prototype for the constructor to the local and global environments
To analyze class SimpleClass (continued)
  • Analyze the body of SimpleClass
  • Add prototype for average to both the local and global function environment
  • Analyze the body of average
  • End scope in global function & variable environments
To analyze the body of SimpleClass

- Begin a new scope in the global variable environment
- Add initialx and intialy to the global variable environment (both with type INTEGER)
- Analyze statement \( x = \text{initialx} \) using global environments
- Analyze statement \( y = \text{initialy} \) using global environments
- Analyze statement \( \text{return this;} \) using global environments
  - Added implicitly by the parser!
- End scope in the global variable environment
To analyze the body of average:

- Begin a new scope in the global variable environment.
- Add `ave` to the global variable environment with type `INTEGER`.
- Analyze the statement `ave = (x + y) / 2` using global environments.
- Analyze the statement `return ave` using global environments.
- End scope in local variable environment.
To analyze the body of main

- Begin a new scope in the variable environment
- Add \( z \) to the variable environment, with the type SimpleClass
- Analyze the statement
  \[ z = \text{new SimpleClass}(3,4); \]
To analyze the body of main (continued)

- \( z = \text{new SimpleClass}(3,4); \)
  - Look up SimpleClass in the type environment. Extract the function environment for SimpleClass
  - Look up SimpleClass in this function environment
  - Make sure the prototype for SimpleClass takes 2 integers
  - Look up the type of \( z \) in the global variable environment
  - Make sure the types match for the assignment statement
To analyze the body of main (continued)

- Analyze the statement \( w = z.\text{average}(); \)
  - Look up \( z \) in the variable environment, and make sure that it is of type \text{CLASS}.
  - Using the function environment obtained from the \text{CLASS} type for \( z \), look up the key \text{average}. Make sure that the function \text{average} takes zero input parameters.
  - Make sure the return type of \text{average} matches the type of \( w \).
- End scope in the variable environment
We will also need to make some changes in the AAT generator

- Maintain “this” pointer
  - Set the value of the “this” pointer at the beginning of a method call
- Access instance variables using the “this” pointer.
  - \( x = 3 \); produces different code if \( x \) is an instance variable or a local variable.
Activation records for methods will contain a “this” pointer.

“This” pointer will be the first item in the activation record.

Remainder of the activation record does not change.

“This” pointer is passed in as implicit 0th parameter.
10-44: Activation Record for Methods

Previous Activation Record

Current Activation Record

Input Parameter n
...
Input Parameter 2
Input Parameter 1
this pointer
Local Variables
Saved Registers

SP
FP
To set up an activation record (at the beginning of a method call)

- Save registers, as normal
- Set the FP to (SP + WORDSIZE)
  - So that the “this” pointer ends up in the correct activation record
  - Passed as the 0th parameter
- “this” is at location FP
- First local variable is at location FP-WORDSIZExE
- First input parameter is at location FP+WORDSIZExE
• Passing implicit “this” parameter
  • Each method call needs to be modified to pass in the implicit “this” parameter.

• Need to handle two types of method calls
  • Explicit method calls
    • x.foo(3,4)
  • Implicit Method Calls
    • Class contains methods foo and bar
    • foo calls bar (without using “this”)
10-47: Explicit Method Calls

- `x.foo(3,4)`

- **AST:**
  ```plaintext
  MethodCall(foo)
  ├── BaseVar
  │   └── identifier(x)
  │
  │   ├── Integer_Literal(4)
  │   └── Integer_Literal(3)
  └── MethodCall

- What should the Abstract Assembly be for this method call?
- (What should we pass as the “this” pointer?)
 Explicit Method Calls

- **AAT for x:**
  
  ```
  Memory
  Operator(-)
  Register(FP) Constant(x_offset)
  ```

- **AAT for x.foo(3,4)**

  ```
  CallExpression("foo")
  Memory Constant(3) Constant(4)
  Operator(-)
  Register(FP) Constant(x_offset)
  ```
class MyClass {
    void main() {
        int x;
        x = myfunction(3);
    }

    int foo(int y) {
        return y + 1;
    }
}

void bar() {
    int x;
    x = foo(7);
}

int myfunction(int a) {
    return a + 1;
}
Implicit Method Calls

- \( x = \text{myfunction}() \) in \text{main} is a function call – don’t need to pass in a “this” pointer
- \( x = \text{foo}(7) \) in \text{bar} is a method call – need to pass in a “this” pointer
- Add another field to \text{FunctionEntry}: Method bit
  - false if entry is a function (no need for “this” pointer)
  - true if entry is a method (need 0th parameter for “this” pointer)
class MethodCalling {

    int foo(int y) {
        return y + 1;
    }

    void bar() {
        int x;
        x = foo(7);
    }
}

10-51: Implicit Method Calls
We know `foo` is a method call
- method bit set to true in function entry for `foo`
- Need to pass in the “this” pointer as 0th parameter
- How can we calculate the “this” pointer to pass in?
• We know `foo` is a method call
  • method bit set to true in function entry for `foo`
• Need to pass in the “this” pointer as 0th parameter
• How can we calculate the “this” pointer to pass in?
  • Same as the “this” pointer of the current function
Any time a method is called implicitly, the “this” pointer to send in is:
10-55: Implicit Method Calls

- Any time a method is called implicitly, the "this" pointer to send in is:

Memory

| Register(FP) |
Implicit Method Calls

- Abstract Assembly for $\text{foo}(7)$
**Abstract Assembly for \( \text{foo}(7) \)**

- CallExpression("foo")
- Memory
  - Register(FP)
- Constant(7)
Constructor Calls

- Just like any other method call
- But... we need an initial “this” pointer
- No space has been allocated yet!
The AAT for a constructor call needs to:

- Allocate the necessary space for the object
- Call the constructor method, passing in the appropriate “this” pointer

What should the AAT for `new Integer(3)` be?
10-60: Constructor Calls

- The AAT for a constructor call needs to:
  - Allocate the necessary space for the object
  - Call the constructor method, passing in the appropriate “this” pointer
- What should the AAT for `new Integer(3)` be?

```
CallExpression("Integer")
  
  CallExpression("Allocate")
  Constant(Integer_Size)
  
  Constant(3)
```
class InstanceVsLocal {
    int instance;

    void method() {
        int local;

        local = 3;    /* line A */
        instance = 4;  /* line B */
    }
}

- Stack / heap contents during method?
- AAT for line A?
- AAT for line B?
Instance vs. Local Variables

- Instance variables and local variables are implemented differently.
- Need to know which variables are local, and which variables are instance variables (just like methods)
- Add instanceVar bit to VariableEntry
  - true for is instance variable
  - false for local variables / parameters
10-63: **Instance Variable Offsets**

- Keep track of two offsets (using two globals)
  - Local variable offset
  - Instance variable offset
- At the beginning of a class definition:
  - Set instance variable offset to 0
  - Insert “this” pointer into the variable environment, as a *local variable*
- At the beginning of each method
  - set the local variable offset to -\texttt{WORDSIZEx}
  - Remember the “this” pointer!
When an instance variable declaration is visited:
  - Add variable to local & global variable environments, using the instance variable offset, with instance bit set to true
  - Decrement instance variable offset by WORDSIZE

When a local variable declaration is visited:
  - Add variable to only the global variable environment, using the local variable offset, with instance bit set to false
  - Decrement local variable offset by WORDSIZE
For a base variable:

- If it is a local variable, proceed as before
- If it is an instance variable
  - Add an extra “Memory” node to the top of the tree
  - Need to do nothing else!
class InstanceVsLocal {
    int instance;

    void method() {
        int local;

        local = 3;
        instance = 4;
    }
}
• Insert `instance` to the global variable environment, with the “instance variable” bit set to 1, with offset 0

• Insert `local` to the global variable environment, with the “instance variable” bit set to 0, with offset `WORDSIZE` (remember “this” pointer!)

• Abstract Assembly for `local`: 
**AATs for Instance Variables**

- Insert `instance` to the global variable environment, with the “instance variable” bit set to 1, with offset 0
- Insert `local` to the global variable environment, with the “instance variable” bit set to 0, with offset `WORDSIZE` (remember “this” pointer!)
- Abstract Assembly for `local`:

```
Memory
  Operator(-)
    Register(FP)
    Constant(Wordsize)
```
AATs for Instance Variables

- Insert `instance` to the global variable environment, with the “instance variable” bit set to 1, with offset 0
- Insert `local` to the global variable environment, with the “instance variable” bit set to 0, with offset `WORDSIZEx` (remember “this” pointer!)
- Abstract Assembly for `instance`
• Insert instance to the global variable environment, with the “instance variable” bit set to 1, with offset 0

• Insert local to the global variable environment, with the “instance variable” bit set to 0, with offset WORDSIZE (remember “this” pointer!)

• Abstract Assembly for instance
10-71: Instance vs. Local Variables

class MyClass {

    int instance1;
    int instance2;

    void method(int param) {
        int local;

        local = instance1 - instance2 + param;
    }
}

• Stack / heap contents during method?
• AAT for assignment statement?
What about class variables and array variables?

```java
class Class1 {
    int x;
    int y[];
}
class Class2 {
    Class1 C1;
    int array[];
    Class1 C2[];
    void method() {
        array[2] = 3;
        C1.x = 3;
        C1.y[2] = 4;
    }
}
void main() {
    Class2 C2 = new Class2();
    C2.C1 = new Class1();
    C2.C1.y = new int[10];
    C2.array = new int[10];
    C2.C2 = new Class1[5];
    C2.C2[3].y = new int[5];
    C2.method();
}
```
Code Generation

- When methods are added to classes, what changes are necessary in the code generation stage?
10-74: Code Generation

- When methods are added to classes, what changes are necessary in the code generation stage?
- None!
  - The AAT structure is not changed
  - Prior modifications create legal AAT
  - Code Generator should work unchanged.
class Point {
    int xpos;
    int ypos;

    Point(int x, int y) {
        xpos = x;
        ypos = y;
    }

    int getX() {
        return xpos;
    }

    int getY() {
        return ypos;
    }

    void setX(int x) {
        xpos = x;
    }

    void setY(int y) {
        ypos = y;
    }
}

class Circle extends Point {
    int radiusval;

    Circle(int x, int y, int radius) {
        xpos = x;
        ypos = y;
        radiusval = radius;
    }

    int getRadius() {
        return radiusval;
    }

    void setRadius(int radius) {
        radiusval = radius;
    }
}
What changes are necessary to the lexical analyzer to add inheritance?
**Inheritance**

- What changes are necessary to the lexical analyzer to add inheritance?
  - Add keyword “extends”
  - No other changes necessary
What changes are necessary to the Abstract Syntax Tree for adding inheritance?
Inheritance

What changes are necessary to the Abstract Syntax Tree for adding inheritance?

- Add a “subclass-of” field to the class definition node
- “subclass-of” is a String
  - Examples for point, circle
• What changes are necessary to the Semantic Analyzer for adding inheritance?
10-82: Inheritance

- What changes are necessary to the Semantic Analyzer for adding inheritance?
  - Allow subclass access to all methods & instance variables of superclass
  - Allow assignment of a subclass value to a superclass variable
What changes are necessary to the Semantic Analyzer for adding inheritance?

- Add everything in the environment of superclass to the environment of the subclass
- Add a “subclass-of” pointer to internal representation of types
- On assignment, if types are different, follow the “subclass-of” pointer of RHS until types are equal, or run out of superclasses.
Case 1

class baseclass {
    int a;
    boolean b;
}
class subclass extends baseclass {
    boolean c;
    int d;
}

• baseclass contains 2 instance variables (a and b)
• subclass contains 4 instance variables (a, b, c and d)
Case 2

class baseclass2 {
    int a;
    boolean b;
}

class subclass2 extends baseclass2 {
    int b;
    boolean c;
}

- baseclass2 contains a, b
- subclass2 contains 4 instance variables, only 3 are accessible a, b (int), c
Case 2

class baseclass2 {
    int a;
    boolean b;
}
class subclass2 extends baseclass2 {
    int b;
    boolean c;
}

subclass2 contains 4 instance variables, only 3 are accessible a, b (int), c

How could we get at the boolean value of b?
• Case 3

class baseclass3 {
    int foo() {
        return 2;
    }
    int bar() {
        return 3;
    }
}

class subclass3 extends baseclass3 {
    int foo() {
        return 4;
    }
}
• When subclass A extends a base class B
  • Make clones of the variable & function environments of B
  • Start A with the clones
  • Add variable and function definitions as normal
To analyze a class A which extends class B

- begin scope in the global variable and function environments
- Look up the definition of B in the type environment
- Set superclass pointer of A to be B
- Add all instance variables in B to variable environment for B, and the global variable environment
- Add all function definitions in B to the function environment for A and the global function environment
To analyze a class A which extends class B (continued)

- Add “this” pointer to the variable environment of A
  - Overriding the old “this” pointer, which was of type B
- Analyze the definition of A, as before
- End scope in global function & variable environments
To analyze an assignment statement

- Analyze LHS and RHS recursively
- If types are not equal
  - If RHS is a class variable, follow the superclass pointer of RHS until either LHS = RHS, or reach a null superclass

- Use a similar method for input parameters to function calls
• What changes are necessary in the abstract assembly generator for adding inheritance?
What changes are necessary in the abstract assembly generator for adding inheritance?

At the beginning of a class definition, set the instance variable offset = size of instance variables in superclass, instead of 0.

When instance variables are added to subclass, use the same offsets that they had in superclass.

No other changes are necessary!
What changes are necessary in the code generator for adding inheritance?
• What changes are necessary in the code generator for adding inheritance?
• None – generate standard Abstract Assembly Tree
Adding inheritance without virtual functions can lead to some odd behavior.
10-97: **Inheritance**

class base {
    int foo() {
        return 3;
    }
}

class sub extends base {
    int foo() {
        return 4;
    }
}

void main() {
    base A = new base();
    base B = new sub();
    sub C = new sub();
    print(A.foo());
    print(B.foo());
    print(C.foo());
}

Adding inheritance without virtual functions can lead to some odd behavior

- Hard-to-find bugs in C++
- Why java does uses virtual functions
- Non-virtual (static, final) cannot be overridden
class super {
    int x;
    public int y;
    private int z;

    void foo() {
        x = 1;
        z = 2;
    }
}

class sub extends super {
    private int a;

    void bar() {
        z = 3;
        a = 4;
    }
}

void main () {
    super superclass;
    sub subclass;
    superclass = new super();
    subclass = new sub();
    superclass.x = 5;
    superclass.z = 6;
    subclass.y = 7;
    subclass.a = 8;
}
class super {
    int x;
    public int y;
    private int z;

    void foo() {
        x = 1;  /* Legal */
        z = 2;  /* Legal */
    }
}

class sub extends super {
    private int a;

    void bar() {
        a = 4;  /* Legal */
    }
}
Access Control

- Changes required in Lexical Analyzer
10-102: **Access Control**

- Changes required in Lexical Analyzer
  - Add keywords “public” and “private”
10-103: **Access Control**

- Changes required in Abstract Syntax Tree
10-104: Access Control

- Changes required in Abstract Syntax Tree
  - Add extra bit to methods and instance variables
    - public or private
10-105: **Access Control**

- Changes required in Semantic Analyzer
Access Control

- Changes required in Semantic Analyzer
  - Allow access to a variable within a class
  - Deny Access to variable outside of class
- How can we do this?
10-107: Access Control

- Changes required in Semantic Analyzer
  - Allow access to a variable within a class
  - Deny Access to variable outside of class
- Use the global variable environment to access variables inside class
- Use the local variable environment to access variables outside class

(examples)
10-108: **Access Control**

- When analyzing a public instance variable declaration
  - `public int y;`
  - Add `y` to both the local and global variable environment

- When analyzing a private instance variable declaration
  - `private int z;`
  - Add `z` to *only* the global variable environment
Access Control

- If we add \( z \) to only the global variable environment
  - When we access \( z \) from within the class, it will be found
  - When we access \( z \) from outside the class, it will *not* be found
  - Need to add a hack for getting this.x to work correctly ...
10-110: Access Control

- Changes required in the Assembly Tree Generator
  - Private variables are no longer added to the private variable environment
  - Can no longer use the size of the variable environment as the size of the class
  - Need to add a “size” field to our internal representation of class types
Access Control

- Changes required in the Code Generator
Changes required in the Code Generator
- We are still producing valid abstract assembly
- No further changes are necessary
Multiple functions (or methods in the same class) with the same name

Use the # and types of the parameters to distinguish between them

```
int foo(int x);
int foo(boolean z);
void foo(int x, int y);
```

Calls:

```
x = foo(3);
x = foo(true);
foo(3+5, foo(true));
```
Overloading Functions

- Just as in regular Java, can’t overload based on the *return type* of a function or method.
- Why not?
int foo(int x);
int foo(boolean y);

int bar(int x);
boolean bar(int x);

z = foo(bar(3));

• What should the compiler do?
Overloading Functions

- Changes required in the Lexical Analyzer
Overloading Functions

- Changes required in the Lexical Analyzer
  - Not adding any new tokens
  - No changes required
Changes required to the Abstract Syntax:
Overloading Functions

- Changes required to the Abstract Syntax:
  - None!
• Changes required to the Semantic Analyzer
  • Need to distinguish between:
    • `int foo(int a, int b)`
    • `int foo(boolean c, int d)`
Need to distinguish between:

- `int foo(int a, int b)`
- `int foo(boolean b, int d)`

We could use `fooointintint` and `fooboolanintint` as keys

Problems?
Overloading Functions

• `foo(3+4, bar(3,4));`
  - Need to convert `(3+4)` to “int”, `bar(3+4)` to “int” (assuming `bar` returns an integer)
  - Better solution?
Overloading Functions

- `foo(3+4, bar(3,4));`
  - Convert the pointer to the internal representation of an integer to a string
  - Append this string to "foo"
  - Use new string as key to define function
    - `foo13518761351876`
  - From `3+4` and `bar(3,4)`, we can get at the pointer to the internal representation of the type
Once we have expanded the key for functions to include the *types* of the input parameters, what further work is needed?
Once we have expanded the key for functions to include the \textit{types} of the input parameters, what further work is needed?

• None!
Recursive Classes

- Recursive classes allow for linked data structures

```java
class linkedList {
    int data;
    linkedList next;
}
```
Recursive Classes

- Changes necessary to allow recursive classes
  - Add keyword “null”
  - Add “null expression” to AST
  - Add class to type environment before class has been completely examined
  - Allow “null” expression to be used for any class value
10-128: Recursive Classes

Recursive Classes diagram with nodes labeled as follows:

- int
- boolean
- void
- linkedList
- KeyStack
- newscope
- data
- next
- class type
- integer type
- boolean type
- void type
- linkedList type
- KeyStack

The diagram shows the hierarchical and interrelated nature of these classes, indicating how they interact and connect within the structure.
Recursive Classes

- Modifications to Semantic Analyzer
  - On assignment – if LHS is a class, RHS may be null
  - For any function call – if formal is a class, actual may be null
  - Comparison operations: ==, != – If either side is a class, the other can be null
class super {
    int foo() {
        return 1;
    }
}

class sub {
    int foo() {
        return 2;
    }
}

void main() {
    super x = new sub();
    print(x.foo());
}
Virtual Methods

- If the language uses static methods (as described so far), the static type of the variable defines which method to use
  - In previous example, static methods would print out 1
  - C++ uses static methods (unless specified as “virtual”)

- If the language uses virtual methods, the type of the actual variable defines which method to use
  - In previous example, print out 2
  - Java uses only virtual methods (avoids some of the bizarre errors that can occur with C++)
class superclass {
    int x;
    void foo() {
        ...
    }
    void bar() {
        ...
    }
}

class subclass extends superclass {
    void main() {
        int y;
        void bar() {
            ...
        }
        void g() {
            ...
        }
    }
}

superclass a;
superclass a;
as = new subclass();
as = new subclass();
a.bar(); /* Point A */
a.bar(); /* Point B */
We need to generate the exact same code for:
- `a.bar()` at Point A
- `a.bar()` at Point B

Even though they will do different things at run time

Function pointers to the rescue!
10-134: Virtual Methods

- Previously, the data segment held only the instance variables of the class.
- Now, the data segment will also hold a pointer to a function table:
  - Only need one table / class (not one table / instance)
Virtual Methods

Data segment for variable a

Method Table

| x |

addres of foo
(superclass definition)

addres of bar
(superclass definition)

Data segment for variable b

Method Table

| x |

| y |

addres of foo
(superclass definition)

addres of bar
(subclass definition)

addres of g
(subclass definition)
Virtual Methods

• Function Environment
  • Previously, we needed to store the assembly language label of the function in the function environment
  • Now, we need to store the offset in the function table for the function
## Virtual Methods

### Environments for superclass

<table>
<thead>
<tr>
<th>Function Environment</th>
<th>Variable Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>value</td>
</tr>
<tr>
<td>foo</td>
<td>0</td>
</tr>
<tr>
<td>bar</td>
<td>4</td>
</tr>
<tr>
<td>key</td>
<td>value</td>
</tr>
<tr>
<td>x</td>
<td>4</td>
</tr>
</tbody>
</table>

### Environments for subclass

<table>
<thead>
<tr>
<th>Function Environment</th>
<th>Variable Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>value</td>
</tr>
<tr>
<td>foo</td>
<td>0</td>
</tr>
<tr>
<td>bar</td>
<td>4</td>
</tr>
<tr>
<td>g</td>
<td>8</td>
</tr>
<tr>
<td>key</td>
<td>value</td>
</tr>
<tr>
<td>x</td>
<td>4</td>
</tr>
<tr>
<td>y</td>
<td>8</td>
</tr>
</tbody>
</table>
**Virtual Methods**

- When a method `x.foo()` is called
  - Look up `x` in the function environment
    - Returns a class type, which contains a local function environment
  - Look up `foo` in the local function environment
    - Returns the offset of foo in the function table
  - Output appropriate code
Virtual Methods

- When a method `x.foo(3)` is called
  - Output appropriate code
    - Extend our AAT to allow *expressions* as well as labels for function calls

```
MethodCall
    Memory
    Constant(3)
    Memory
    foo_offset
    Memory
    x_offset
```

FP
class baseClass {
    int x;

    void foo(int x) {
        /* definition of foo */
    }

    void bar() {
        /* definition of bar */
    }
}

Virtual Methods Example
10-141: Virtual Methods Example
class extendedClass {
    int y;

    void bar() {
        /* definition of bar */
    }

    void g() {
        /* definition of g */
    }
}