Introduction to Code Generation

Note by Baris Aktemur:
Our slides are adapted from Cooper and Torczon's slides that they prepared for COMP 412 at Rice.

Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,- :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
      }
      return result;
 }
```

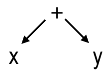
The Concept

- Assume an AST as input & ILOC as output
- Use a postorder treewalk evaluator (*visitor pattern* in OOD)
  - Visits & evaluates children
  - Emits code for the op itself
  - Returns register with result
- Bury complexity of addressing names in routines that it calls
  - *base*(), *offset*(), & *val*()
- Works for simple expressions
- Easily extended to other operators
- Does not handle control flow

Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
     case ×,÷,+,- :
        t1← expr(left child(node));
        t2← expr(right child(node));
        result ← NextRegister();
        emit (op(node), t1, t2, result);
        break;
     case IDENTIFIER:
        t1← base(node);
        t2← offset(node);
        result ← NextRegister();
        emit (loadAO, t1, t2, result);
        break;
     case NUMBER:
        result ← NextRegister();
        emit (loadI, val(node), none, result);
        break;
    }
    return result;
 }
```

Example:



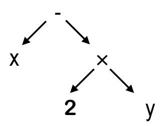
Produces:

expr("x") →		
loadI	@x	⇒ r1
loadAO	r_{arp},r1	⇒ r2
expr("y") →		
loadI	@y	⇒ r3
loadAO	r_{arp},r3	⇒ r4
NextRegister() → r5		
Emit(add,r2,r4,r5) →		
add	r2,r4	⇒ r5

Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
     case ×,÷,+,- :
        t1← expr(left child(node));
        t2← expr(right child(node));
        result ← NextRegister();
        emit (op(node), t1, t2, result);
        break;
     case IDENTIFIER:
        t1← base(node);
        t2← offset(node);
        result ← NextRegister();
        emit (loadAO, t1, t2, result);
        break;
     case NUMBER:
        result ← NextRegister();
        emit (loadI, val(node), none, result);
        break;
    }
    return result;
 }
```

Example:



Produces:

loadI	@x	⇒ r1
loadAO	r_{arp},r1	⇒ r2
loadI	2	⇒ r3
loadI	@y	⇒ r4
loadAO	r_{arp},r4	⇒ r5
mult	r3,r5	⇒ r6
sub	R2,r6	⇒ r7

Extending the Simple Treewalk Algorithm

Adding other operators

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls
- Handle assignment as an operator

Mixed-type expressions

- Insert conversions as needed from conversion table
- Most languages have symmetric & rational conversion tables
  - Original PL/I had asymmetric tables for BCD & binary integers

Typical Table for Addition

+	Integer	Real	Double	Complex
Integer	Integer	Real	Double	Complex
Real	Real	Real	Double	Complex
Double	Double	Double	Double	Complex
Complex	Complex	Complex	Complex	Complex

Generating Code in the Parser

Need to generate an initial IR form

- Chapter 4 talks about ASTs & ILOC
- Might generate an AST, use it for some high-level, near-source work such as type checking and optimization, then traverse it and emit a lower-level IR similar to ILOC for further optimization and code generation

The Big Picture

- Recursive treewalk performs its work in a bottom-up order
  - Actions on non-leaves occur after actions on children

Handling Assignment (just another operator)

lhs ← *rhs*

Strategy

- Evaluate *rhs* to a value *(an rvalue)*
- Evaluate *lhs* to a location *(an lvalue)*
  - *lvalue* is a register ⇒ move rhs
  - *lvalue* is an address ⇒ store rhs
- If *rvalue* & *lvalue* have different types
  - Evaluate *rvalue* to its "*natural*" type
  - Convert that value to the type of **lvalue*

Unambiguous scalars go into registers

Ambiguous scalars or aggregates go into memory

Keeping ambiguous values in memory lets the hardware sort out the addresses.

Handling Assignment

What if the compiler cannot determine the type of the rhs?

- Issue is a property of the language & the specific program
- For type-safety, compiler must insert a run-time check
  - Some languages & implementations ignore safety (*bad idea*)
- Add a *tag* field to the data items to hold type information
  - Explicitly check tags at runtime

Code for assignment becomes more complex

```
evaluate rhs
if type(lhs) ≠ rhs.tag
  then
     convert rhs to type(lhs) or
     signal a run-time error
lhs ← rhs
```

Choice between conversion & a runtime exception depends on details of language & type system

Much more complex than static checking, plus costs occur at runtime rather than compile time

Handling Assignment

Compile-time type-checking

- Goal is to eliminate the need for both tags & runtime checks
- Determine, at compile time, the type of each subexpression
- Use runtime check only if compiler cannot determine types

Optimization strategy

- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation

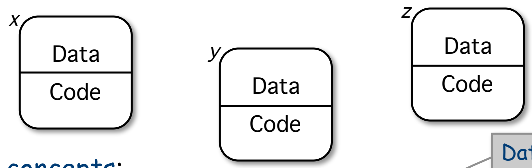
Can *design* the language so all checks are static

Object-Oriented Languages

What is an Object?

An object is an abstract data type that encapsulates data, operations and internal state behind a simple, consistent interface.

The Concept:



Elaborating the concepts:

- Each object has internal state
  - Data members are static (*lifetime of object*)
  - External access is through code members
- Each object has a set of associated procedures, or methods
  - Some methods are public, others are private
  - Locating a procedure by name is complex: dynamic binding
- Object's internal state leads to complex behavior

Data members, variables

Code members, or methods

OOLs & the Procedure Abstraction

What is the shape of an OOL's name space?

- Local storage in objects (*both public & private*)
- Storage defined in methods (*they are procedures*)
  - Local values inside a method
  - Static values with lifetimes beyond methods
- Methods shared among multiple objects
- Global name space for global objects and (*some?*) code

In some OOLs, everything is an object.

In others, variables co-exist with objects & inside objects.

Classes

- Objects with the same ~~state~~ members are grouped into a *class*
  - Same code, same data, same naming environment
  - Class members are static & shared among instances of the class
- Allows abstraction-oriented naming
- Should foster code reuse in both source & implementation

Implementing Object-Oriented Languages

So, what can an executing method access?

The fundamental question

- Names defined by the method
  - *And its surrounding lexical context*
- The receiving object's data members
  - Smalltalk terminology: *instance variables*
- The code & data members of the class that defines it
  - *And its context from inheritance*
  - Smalltalk terminology: *class variables and methods*
- Any object defined in the global name space

Inheritance adds some twists.

The method might need the address for any of these objects

Concrete Example: The Java Name Space

Code within a method M for object O of class C can see:

- Local variables declared within M (*lexical scoping*)
- All instance variables & class variables of C
- All public and protected variables of any *superclass* of C
- Classes defined in the same package as C or in any explicitly imported package
  - public class variables and public instance variables of imported classes
  - package class and instance variables in the package containing C
- Class declarations can be nested!
  - These member declarations hide outer class declarations of the same name (*lexical scoping*)
  - Accessibility: public, private, protected, package

class hierarchy

lexical

Both lexical nesting & class hierarchy at play

Superclass is an ancestor in the inheritance hierarchy

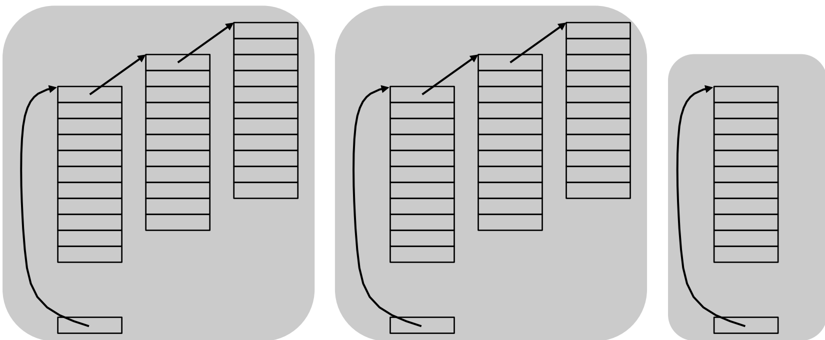
The Java Name Space

```
Class Point {
    public int x, y;
    public void draw();
}
Class ColorPoint extends Point {             // inherits x, y, & draw() from Point
    Color c;                                 // local data
    public void draw() {…}                   // override (hide) Point's draw
    public void test() { y = x; draw(); }    //  local code
}
Class C {                                    // independent of Point & ColorPoint
    int x, y;                                // local data
    public void m()                          // local code
    {
        Point p = new ColorPoint();          // uses ColorPoint, and, by inheritance
        y = p.x;                             // the definitions from Point
        p.draw();
    }
}
```

We will use and extend this example

OOL Symbol Tables

Conceptually



Lexical Hierarchy   Class Hierarchy   Global Scope

Search Order: lexical, class, global

Runtime Structures for OOLs

Object lifetimes are independent

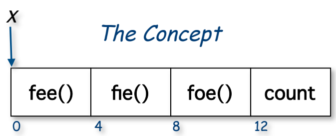
of method lifetimes,
of lifetimes of other objects …

- Each object needs an object record (OR) to hold its state
  - Independent allocation and deallocation
- Classes are objects, too
  - ORs of classes instantiate the class hierarchy

Object Records

- Static private storage for members
- Need fast, consistent access
  - Known constant offsets from OR pointer
- Provision for initialization

The Concept



Object Record Layout

Assume a Fixed-size OR

- Data members are at known fixed offsets from OR pointer
- Code members occur only in objects of class “class”
  - Code vector is a data-member of the class
  - Method pointers are at known fixed offsets in the code vector
  - Method-local storage kept in method’s AR, as in an ALL
- Variable-sized members ⇒ store descriptor to space in heap

Locating ORs

- For a receiver, the OR pointer is implicit
- For a receiver’s class, the receiver’s OR has a class pointer
- Top-level classes and static classes can be accessed by name
  - Mangle the class name & use it as a relocatable symbol
  - Handle nested classes as we would nested blocks in an ALL

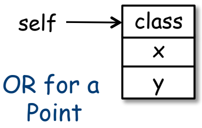
What About Inheritance?

Impact on OR Layout

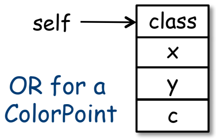
- OR needs slots for each member declared, all the way up the class hierarchy (class, superclass, super-superclass, …)
- Can use prefixing of storage to lay out the OR

Back to Our Java Example — Class Point

```
Class Point {
    public int x, y;
    …
}
```




```
Class ColorPoint extends Point {
    Color c;
    …
}
```



What happens if we cast a ColorPoint to a Point?

Take the word *extends* literally.

Open World versus Closed World

Prefixing assumes that the class structure is known when layout is performed. Two common cases occur.

Closed-World Assumption (Compile time)

- Class structure is known and closed prior to runtime
- Can lay out ORs in the compiler and/or the linker

Open-World Assumption (Interpreter or JIT)

- Class structure can change at runtime
- Cannot lay out ORs until they are allocated
  - Walk class hierarchy at allocation

C++ has a closed class structure.

Java as an open class structure.

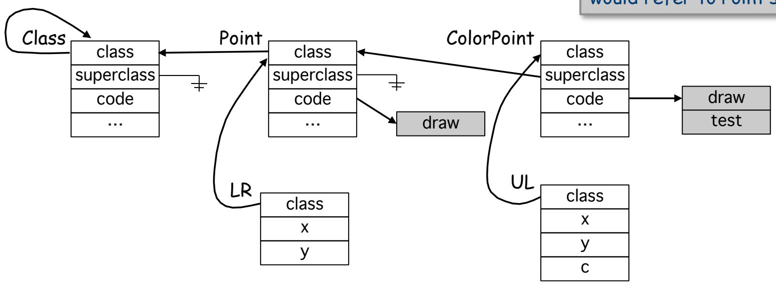
What About Code Members?

How does the language's runtime environment find the code for a given method invocation?

Closed Class Structure

- Mapping of names to methods is static and known (C++)
  - Fixed offsets & indirect calls
- Virtual functions force runtime resolution

If ColorPoint inherited draw from Point, its code vector would refer to Point's draw.



UL finds draw at offset 0 in ColorPoint's code vector