CS 322: Languages and Compiler Design II

Mark P Jones, Portland State University

Spring 2015

Welcome to CS 322

Administrative Matters: see the syllabus!

Key topics

- Contact details, office hours, etc...
- Course objectives
- Prerequisites
- Textbook/background reading
- Assignments, exams, and grading
- Schedule
- Academic integrity
- Accommodations
- Subjective matters

Key topics

- Contact details, office hours, etc...
- Course objectives
- Prerequisites
- Textbook/background reading
- Assignments, exams, and grading
- Schedule
- Academic integrity
- Accommodations
- Subjective matters

These determine the topics that we teach and the questions/ exercises that we will ask you to do. Use them as your guide!

Key topics

- Contact details, office hours, etc...
- Course objectives
- Prerequisites
- Textbook/background reading
- Assignments, exams, and grading
- Schedule
- Academic integrity
- Accommodations
- Subjective matters

CS 321 with a grade C or higher is required for CS 322.

Key topics

- Contact details, office hours, etc...
- Course objectives
- Prerequisites
- Textbook/background reading
- Assignments, exams, and grading
- Schedule
- Academic integrity
- Accommodations
- Subjective matters

No set text, but plenty of background reading is available.

Key topics

- Contact details, office hours, etc...
- Course objectives
- Prerequisites
- Textbook/background reading
- Assignments, exams, and grading
- Schedule
- Academic integrity
- Accommodations
- Subjective matters

Multiple assignments, each one substantial, designed to span multiple weeks. Minimum grade requirements to pass. Additional "grading milestones" may be introduced.

Key topics

- Contact details, office hours, etc...
- Course objectives
- Prerequisites
- Textbook/background reading
- Assignments, exams, and grading
- Schedule
- Academic integrity
- Accommodations
- Subjective matters

Dates for midterm and final already fixed. No lecture on Memorial Day, but essential required labs in all ten weeks of the class.

Key topics

- Contact details, office hours, etc...
- Course objectives
- Prerequisites
- Textbook/background reading
- Assignments, exams, and grading
- Schedule
- Academic integrity
- Accommodations
- Subjective matters

No tolerance for cheating. Individual work only unless otherwise specified.

Planning for your success

- CS 322 is a **required class** for the CS major, but it will not be offered again until Winter 2016
- **I** want all of you to pass ... but to do that, **you** must demonstrate your understanding of the material and your ability to apply it through your work on the assignments and the exams
- There is **no other way** to pass
- Be sure you **make time for study**
- **Do not cheat!**
- If you don't understand any part of the course material, or the requirements of a particular assignment, **ask for help**

Reasons not to ask questions, Part 1

- **"As a CS major, I should be able to do this on my own"**
- You should certainly attempt to understand and tackle the problems on your own to begin with …
- But the option to ask for help:
  - Will be an explicit part of every assignment
  - Should strengthen, not compromise your learning
  - Is what you do in the real world, whether you're working with customers or other team members
  - Is what you do every time you Google, ask a class mate, check a text book, or consult the slides; why should talking to an instructor be any different?

Reasons not to ask questions, Part 2

- **"I'm very busy"**
- Plan your schedule carefully, and get started on assignments early so that you have time to ask questions
- Multiple office hours sessions every week, plus "by appointment" options
- D2L discussion forums and email are available 24/7
- If you don't have time to ask questions, this might not be a good time to take this course …

Reasons not to ask questions, Part 3

- **"I can see you're busy and I don't want to add to that!"**
- Sure, I'm busy … but your questions are important!
- Answering questions is an important part of my job!
- Posting on D2L will reach a wider audience:
  - more people can benefit from your questions
  - more people will be in a position to answer your questions
- Try to formulate questions as carefully and precisely as you can: this will likely improve your understanding, and will make the questions easier to understand and answer too!

Any questions?

CS 322: Languages and Compiler Design II

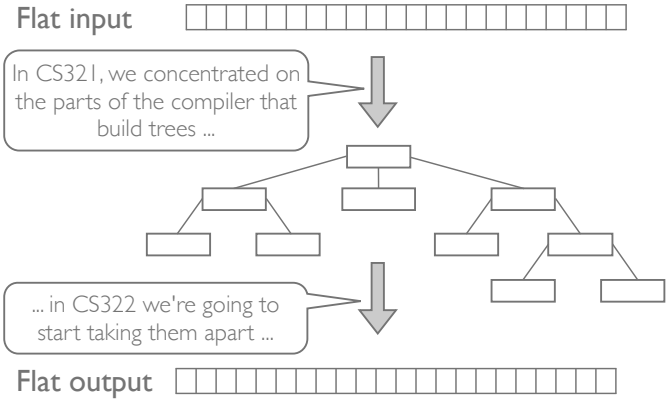
Mark P Jones, Portland State University

Spring 2015

Week 1: Interpreters

Working with (abstract syntax) trees: a quick review

Trees in Compilation



Representing trees in Java

The classes of an object-oriented language like Java provide a simple mechanism for representing trees:

```
class Tree {
    Tree left;
    Tree right;
    int  data;

    Tree(Tree left, Tree right, int data) {
        this.left  = left;
        this.right = right;
        this.data  = data;
    }
}
```

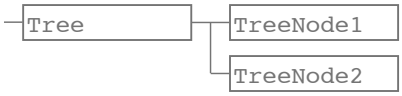
pointers to children

data specific to a node

constructors build new nodes

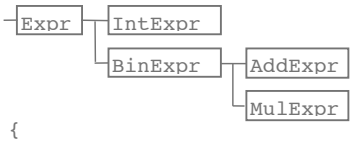
Multiple node types

- The tree structures that we use to represent abstract syntax can involve many different types of node:
  - The data that we need for an `if` statement is very different from the data we need for a +
- We can use inheritance to deal with this:



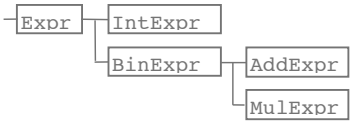
- Use `Tree` when an arbitrary tree is expected
- Use `TreeNode1` or `TreeNode2` to work with a particular type of node

Example




```
abstract class Expr { }

class IntExpr extends Expr {
   int value;
   ...
}

class BinExpr extends Expr {
   Expr left;
   Expr right;
   ...
}

class AddExpr extends BinExpr { ... }

class MulExpr extends BinExpr { ... }
```

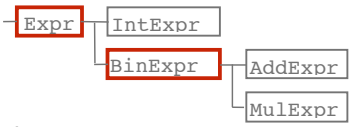

Adding functions




```
abstract class Expr {
    abstract int eval();
}

class IntExpr extends Expr { ...
   int value;
   int eval() { return value; }
}

class AddExpr extends BinExpr { ...
   int eval() { return left.eval() + right.eval(); }
}

class MulExpr extends BinExpr { ...
   int eval() { return left.eval() * right.eval(); }
}
```


More functions

- We don't have to define every function for every different type of tree node, so long as each one can inherit an appropriate definition
- For example, define the default behavior in the base class and then just describe special cases in other parts of the hierarchy




```
abstract class Expr { ...
    int size() {
        return 1;
    }
}
class BinExpr extends Expr { ...
    int size() {
        return left.size() + right.size();
    }
}
```


Which function gets called?

- If `e` has type `Expr` and we execute `e.eval()`, then which function gets called?
- It depends on how `e` was constructed:
  - If `e` was constructed using `new IntExpr()`, then the `IntExpr` version of `eval()` is called
  - If `e` was constructed using `new AddExpr()`, then the `AddExpr` version of `eval()` is called
  - ...
- The compile-time type doesn't tell us, but the run-time value will.

Interpreters and Compilers

Interpreters and compilers

In conventional English:

- **interpreter**: somebody that translates from one language to another.
  - Example: "I need an interpreter when I'm in Japan"
- **compiler**: somebody who collects, gathers, assembles, or organizes information or things.
  - Latin root: compilare, "plunder or plagiarize"

Interpreters and compilers

According to my dictionary:

- **in•ter•pret•er** (noun) Computing: a program that can analyze and execute a program line by line
- **com•pile** (verb) Computing (of a computer): convert (a program) into a machine-code or lower-level form in which the program can be executed

  Derivatives: **com•pil•er** (noun)

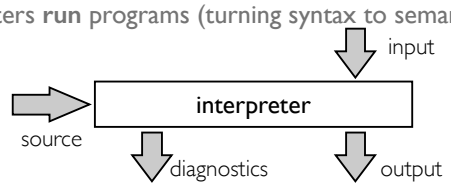
Interpreters and compilers

In computer science:

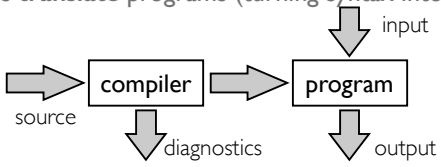
- An interpreter executes programs
  - An interpreter for L might be thought of as a function: $interp_L : L \rightarrow M$, where M is some set of meanings of programs
- A compiler translates programs
  - A compiler from L to L' might be thought of as a function $comp : L \rightarrow L'$

Interpreters and compilers

- Interpreters **run** programs (turning syntax to semantics)

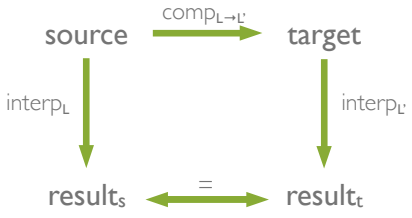


- Compilers **translate** programs (turning syntax into syntax)



Compiler correctness

We can capture the conditions for the correctness of a compiler by using the following "commuting diagram":



Or, in symbols: $\forall p.\ interp_L(p) = interp_{L'}(comp_{L\rightarrow L'}(p))$

Interpreter characteristics

Common (but not universal) characteristics:

- More emphasis on interactive use:
  - Use of a read-eval-print loop (REPL)
  - Examples: language implementations designed for educational or prototyping applications
- Less emphasis on performance:
  - Interpretive overhead that could be eliminated by compilation
  - Performance of scripting code, for example, is less of an issue if the computations that are being scripted are significantly more expensive

... continued

- Portability:
  - An interpreter is often more easily ported to multiple platforms than a compiler because it does not depend on the details of a particular target language
- Experimental platforms:
  - Specifying programming language semantics
  - More flexible language designs; some features are easier to implement in an interpreter than in a compiler

Examples

(Note: the choice between "interpreting" and "compiling" is an implementation decision, not a property of a language!)

- Programming languages:
  - Scripting languages: PHP, python, ruby, perl, bash, Javascript, ...
  - Educational languages: BASIC, Logo, ...
  - Declarative languages: Lisp, Scheme, ML, Haskell, Prolog, ...
  - Virtual machines: Java, Scala, C#, VB, Pascal (P-Code)
- Document description languages:
  - Postscript, HTML, ...
- Hardware:
  - A CPU executes/interprets machine language programs

Building an interpreter

How do you build an interpreter?

- We still need to read and analyze source programs, so interpreters use many of the same components that we find in a typical compiler front-end:
  - lexical analysis, parsing, type checking, ...
- Perhaps with less emphasis on abstract syntax, especially in older designs:
  - A simple interpreter could maintain a pointer to the current location in the source text and lex/parse/check/execute code on the fly ...
- But abstract syntax and code generation can still be put to good use in an interpreter ...

Interpreting Abstract Syntax

Code alert!

- Lots of code in the slides ahead
- Don't try to take it all in now
- Do try to understand the key details
- Do stop me and ask questions along the way!
- And note that the code is available on D2L so that you can study it more carefully later ... (e.g., in your lab session)

A simple source language

```
IExpr   ::=  Var(String)
         |   Int(int)
         |   Plus(IExpr, IExpr)
         |   Minus(IExpr, IExpr)

BExpr   ::=  LT(IExpr, IExpr)
         |   EqEq(IExpr, IExpr)

Stmt    ::=  Seq(Stmt, Stmt)
         |   Assign(String, IExpr)
         |   While(BExpr, Stmt)
         |   If(BExpr, Stmt, Stmt)
         |   Print(IExpr)
```

Abstract syntax!

Type correct!

A simple program in a concrete syntax

```
t := 0;
i := 0;
while (i < 11) {
  t := t + i;
  i := i + 1;
}
print t;
```


Abstract syntax in Java


```
abstract class IExpr {
}

class Var extends IExpr {
  private String name;
  Var(String name) { this.name = name; }
}

class Int extends IExpr {
  private int num;
  Int(int num) { this.num = num; }
}

class Plus extends IExpr {
  private IExpr l, r;
  Plus(IExpr l, IExpr r) { this.l = l; this.r = r; }
}
```


Abstract syntax in Java

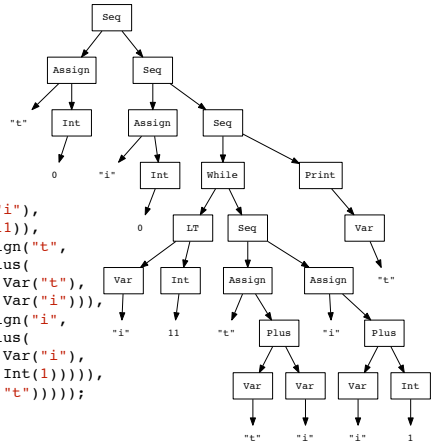

```
abstract class Stmt { }

class Seq extends Stmt {
  private Stmt l, r;
  Seq(Stmt l, Stmt r) { this.l = l; this.r = r; }
}
class Assign extends Stmt {
  private String lhs;
  private IExpr  rhs;
  Assign(String lhs, IExpr rhs) {
    this.lhs = lhs; this.rhs = rhs;
  }
}
class While extends Stmt {
  private BExpr test;
  private Stmt  body;
  While(BExpr test, Stmt body) {
    this.test = test; this.body = body;
  }
}
```


Our "simple program" in abstract syntax


```
Stmt s
  = new Seq(new Assign("t", new Int(0)),
    new Seq(new Assign("i", new Int(0)),
    new Seq(new While(new LT(new Var("i"), new Int(11)),
                      new Seq(new Assign("t",
                                         new Plus(new Var("t"),
                                                  new Var("i"))),
                              new Assign("i",
                                         new Plus(new Var("i"),
                                                  new Int(1))))),
            new Print(new Var("t")))));
```


Our "simple program" in abstract syntax


```
Stmt s
  = new Seq(
      new Assign("t",
                 new Int(0)),
    new Seq(
      new Assign("i",
                 new Int(0)),
    new Seq(new While(
              new LT(new Var("i"),
                     new Int(11)),
              new Seq(new Assign("t",
                        new Plus(
                          new Var("t"),
                          new Var("i"))),
                      new Assign("i",
                        new Plus(
                          new Var("i"),
                          new Int(1))))),
            new Print(new Var("t")))));
```



Adding printing code


```
abstract class IExpr {
  abstract String show();
}

class Var extends IExpr {
  ...
  String show() { return name; }
}

class Int extends IExpr {
  ...
  String show() { return Integer.toString(num); }
}

class Plus extends IExpr {
  ...
  String show() { return "(" + l.show() + " + " + r.show() + ")"; }
}
```

Printing statements

```
abstract class Stmt {
  abstract void print(int ind);

  static void indent(int ind) {
    for (int i=0; i<ind; i++) {
      System.out.print(" ");
    }
  }
}

class Assign extends Stmt {
  ...
  void print(int ind) {
    indent(ind);
    System.out.println(lhs + " = " + rhs.show() + ";");
  }
}
```


... continued

```
class Seq extends Stmt {
  ...
  void print(int ind) {
    l.print(ind);
    r.print(ind);
  }
}

class While extends Stmt {
  ...
  void print(int ind) {
    indent(ind);
    System.out.println("while (" + test.show() + ") {");
    body.print(ind+2);
    indent(ind);
    System.out.println("}");
  }
}
```


A "main" program

```
class Main {
  public static void main(String[] args) {
    Stmt s
      = ...

    System.out.println("Complete program is:");
    s.print(4);

    System.out.println("Done!");
  }
}
```

```
$ java Main
Complete program is:
    t = 0;
    i = 0;
    while ((i < 11)) {
      t = (t + i);
      i = (i + 1);
    }
    print t;
Done!
$
```


"Thanks for the memory"

If we want to run our programs, we need somewhere to store the values of variables:

```
import java.util.Hashtable;

class Memory {
  private Hashtable<String,Integer> store
    = new Hashtable<String,Integer>();

  int load(String name) {
    Integer i = store.get(name);
    return (i!=null) ? i.intValue() : 0;
  }

  void store(String name, int val) {
    store.put(name, new Integer(val));
  }
}
```

a "semantic object"

Evaluating expressions

```
abstract class IExpr { ...
  abstract int eval(Memory mem);
}
class Var extends IExpr { ...
  int eval(Memory mem) { return mem.load(name); }
}
class Int extends IExpr { ...
  int eval(Memory mem) { return num; }
}
class Plus extends IExpr { ...
  int eval(Memory mem) { return l.eval(mem) + r.eval(mem); }
}
class Minus extends IExpr { ...
  int eval(Memory mem) { return l.eval(mem) - r.eval(mem); }
}
```


Executing statements

```
abstract class Stmt { abstract void exec(Memory mem); ... }

class Seq extends Stmt { ...
    void exec(Memory mem) {
        l.exec(mem);
        r.exec(mem);
    }
}
class Assign extends Stmt { ...
    void exec(Memory mem) {
        mem.store(lhs, rhs.eval(mem));
    }
}
class While extends Stmt { ...
    void exec(Memory mem) {
        while (test.eval(mem)) {
            body.exec(mem);
        }
    }
}
```

An interpreter!

- Each `eval()` or `exec()` method associates:
  - a syntactic construct (i.e., a class in the abstract syntax) with
  - a corresponding semantics (i.e., some executable code)
- If there wasn't a clear and natural correspondence, we'd probably have the wrong semantics!

A "main" program

```
class Main {
  public static void main(String[] args) {
    Stmt s
     = ...;

    System.out.println("Complete program is:");
    s.print(4);

    System.out.println("Running on an empty memory:");
    Memory mem = new Memory();
    s.exec(mem);

    System.out.println("Done!");
  }
}
```

```
Complete program is:
    t = 0;
    i = 0;
    while ((i < 11)) {
      t = (t + i);
      i = (i + 1);
    }
    print t;
Running on an empty memory:
Output: 55
Done!
```


Quick review questions

- What would it take to make a general interpreter for this language (i.e., one that can take any program as input)?
- Why do we refer to this as an *interpreter*?
- What would it take to build a *compiler*?

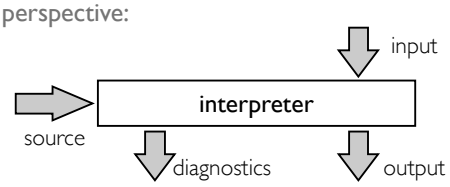
Building a Bytecode Interpreter

Teaser: what is this?

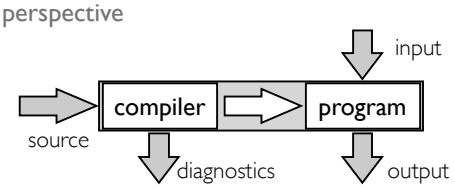
0, 0, 2, 0, 0, 0, 2, 1,
10, 1, 1, 1, 1, 0, 3, 2, 1,
1, 0, 0, 1, 3, 2, 0, 1, 0,
0, 11, 6, 0, 1, 1, 5, 11

Embedding a compiler

- Outside perspective:

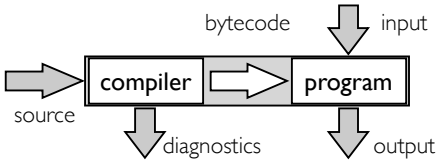


- Internal perspective



Bytecode

- Another common strategy for building an interpreter is to:
  - Generate an internal "bytecode" representation of the input program—a stream of bytes representing instructions for some (real or imagined) computer/machine code
  - Use a simulator for the machine code to execute the bytecodes



Benefits of bytecode

- Use of bytecode has the potential to:
  - reduce interpretive overhead
  - enable more compact representation
  - maintain portability
- But bytecode must still be interpreted, so it is typically not as fast as the corresponding native code might be ...

Back to our simple language

- Let's build a simple bytecode-based interpreter!
- To make it even simpler, we'll use `ints` rather than bytes

```
class Bytecode {
  private int maxCode  = 4000;
  private int prog[]   = new int[maxCode];
  private int nextCode = 0;

  private void emit(int code) {
    if (nextCode>maxCode) {
      System.out.println("... error ...");
      System.exit(1);
    }
    prog[nextCode++] = code;
  }
  ...
}
```


Instruction set

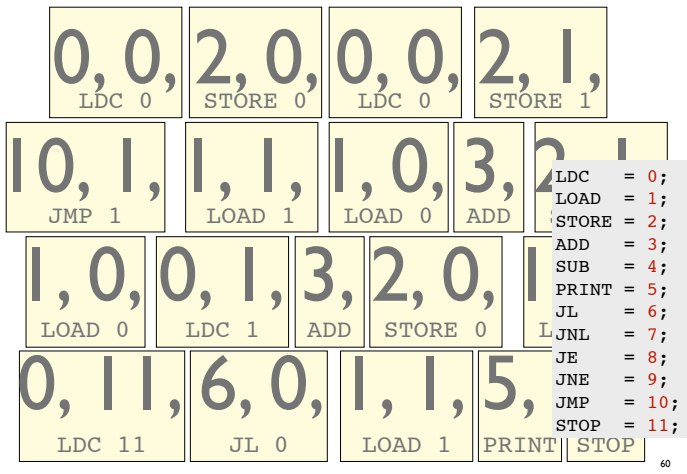
```
// Byte code instructions: --------------------------------
static final int LDC   = 0;    // LDC num
static final int LOAD  = 1;    // LOAD var
static final int STORE = 2;    // STORE var
static final int ADD   = 3;    // ADD
static final int SUB   = 4;    // SUB
static final int PRINT = 5;    // PRINT
static final int JL    = 6;    // JL  lab
static final int JNL   = 7;    // JNL lab
static final int JE    = 8;    // JE  lab
static final int JNE   = 9;    // JNE lab
static final int JMP   = 10;   // JMP lab
static final int STOP  = 11;   // STOP
```

Note: we get to choose the instructions because this code is for an abstract machine, not a real one

Emitting instructions

```
// Individual bytecode instructions: ----------------------
public void ldc(int n)         { emit(LDC);  emit(n); }
public void add()              { emit(ADD); }
public void sub()              { emit(SUB); }
public void jl(int lab)        { emit(JL);  emit(lab); }
public void jnl(int lab)       { emit(JNL); emit(lab); }
public void je(int lab)        { emit(JE);  emit(lab); }
public void jne(int lab)       { emit(JNE); emit(lab); }
public void jmp(int lab)       { emit(JMP); emit(lab); }
public void stop()             { emit(STOP); }
public void print()            { emit(PRINT); }

public void load(String var)  {
    emit(LOAD);  emit(location(var));
}
public void store(String var) {
    emit(STORE); emit(location(var));
}
```


What is this?



Emitting instructions

```
// Individual bytecode instructions: ----------------------
public void ldc(int n)         { emit(LDC);  emit(n); }
public void add()              { emit(ADD); }
public void sub()              { emit(SUB); }
public void jl(int lab)        { emit(JL);  emit(lab); }
public void jnl(int lab)       { emit(JNL); emit(lab); }
public void je(int lab)        { emit(JE);  emit(lab); }
public void jne(int lab)       { emit(JNE); emit(lab); }
public void jmp(int lab)       { emit(JMP); emit(lab); }
public void stop()             { emit(STOP); }
public void print()            { emit(PRINT); }

public void load(String var)  {
    emit(LOAD);  emit(location(var));
}
public void store(String var) {
    emit(STORE); emit(location(var));
}
```

What is a location?

Locations

```
// Mapping from variables to memory locations: -----------
  private Hashtable<String, Integer> vars
    = new Hashtable<String, Integer>();
  private int nextVar = 0;

  private int location(String var) {
    Integer i = vars.get(var);
    if (i==null) {
      vars.put(var, new Integer(nextVar));
      return nextVar++;
    }
    return i.intValue();
  }
```

- Compile-time mapping for variable names to memory locations; no need for run-time lookup
- This is one example of reduced interpretive overhead ...

Labels

- Suppose that we want to emit a forward jump instruction using something like `b.jmp(lab);`
- How will we know what value to use for the label until we've generated the code that comes after this instruction?
- Solution: we will need to track the use of the label `lab` and come back to "fixup" the jump instruction one we know the true address

... continued

```
// Generating labels: -------------------------------------
  private int labels[] = new int[1000];
  private int nextLab  = 0;

  public int newlabel() {
    return nextLab++;
  }

  public void atlabel(int l) {
    labels[l] = nextCode;
  }
```

use to "allocate" a label

use when we reach a label

- The address for a branch to label `l` will now be found in `labels[l]`
- In practice, it would be better to backpatch labels stored in code once and for all as soon as we reach the specified label

Displaying bytecode

```
// Display byte code instructions: ------------------------
public void dump() {
  int pc = 0;
  while (pc < nextCode) {
    System.out.print(pc + "\t");
    switch (prog[pc++]) {
      case LDC   : println("LDC   " + prog[pc++]);
                   continue;

      case LOAD  : println("LOAD  " + prog[pc++]);
                   continue;

      case STORE : println("STORE " + prog[pc++]);
                   continue;
      ...
    }
  }
}
```

actually uses `System.out.println()`

Executing bytecode

- Stack-based abstract machines are common targets for bytecode systems; in theory, the stack provides storage for an arbitrary number of temporary results

```
// Byte code execution: ---------------------------------
private int mem[]   = new int[100];
private int stack[] = new int[1000];
private int sp      = 0;

private void push(int n) { stack[sp++] = n; }
private int  pop()       { return stack[--sp]; }
```

- We also have a memory for storing variables ... note that there are no variable names here!

Executing instructions

```
public void exec() {
  int pc = 0;
  for (;;) {
    switch (prog[pc++]) {
      case LDC   : push(prog[pc++]);
                   continue;
      case LOAD  : push(mem[prog[pc++]]);
                   continue;
      case STORE : mem[prog[pc++]] = pop();
                   continue;
      case ADD   : push(pop() + pop());
                   continue;
      ...
    }
  }
}
```


... continued

```
case SUB   : { int r = pop();
               int l = pop();
               push(l - r);
             }
             continue;

...

case JMP   : pc = labels[prog[pc]];
             continue;

case STOP  : return;

case PRINT : System.out.println("Output: " + pop());
             continue;
```


Generating bytecode

- For expressions (IExpr)
  - IExpr
    ```
    abstract void bcgen(Bytecode b);
    ```
  - Var
    ```
    void bcgen(Bytecode b) {
        b.load(name);
    }
    ```
  - Int
    ```
    void bcgen(Bytecode b) {
       b.ldc(num);
    }
    ```
  - Plus
    ```
    void bcgen(Bytecode b) {
        l.bcgen(b); r.bcgen(b); b.add();
    }
    ```
  - Minus
    ```
    void bcgen(Bytecode b) {
        l.bcgen(b); r.bcgen(b); b.sub();
    }
    ```


... continued

- For statements (Stmt)
  - Stmt
    ```
    abstract void bcgen(Bytecode b);
    ```
  - Seq
    ```
    void bcgen(Bytecode b) {
        l.bcgen(b); r.bcgen(b);
    }
    ```
  - Assign
    ```
    void bcgen(Bytecode b) {
       rhs.bcgen(b); b.store(lhs);
    }
    ```
  - Print
    ```
    void bcgen(Bytecode b) {
        exp.bcgen(b); b.print();
    }
    ```


... continued

- While
  ```
  void bcgen(Bytecode b) {
      int l1 = b.newlabel(), l2 = b.newlabel();
      b.jmp(l2);
      b.atlabel(l1);
      body.bcgen(b);
      b.atlabel(l2);
      test.bcgenIfTrue(b, l1);
  }
  ```
- `test.bcgenIfTrue(b,label)` is a compilation scheme for `BExpr` that will produce bytecode that evaluates the `test` and jumps to the `label` if it is true

In our top-level driver

- Assuming that we have the abstract syntax for a statement in the variable prog, we can use the following to generate and run the corresponding bytecode

```
System.out.println("Generating bytecode:");
Bytecode b = new Bytecode();
s.bcgen(b);
b.stop();
b.dump();

System.out.println("Running on an empty memory:");
b.exec();
```

- Don't forget to output a stop instruction!

Testing


```
Generating bytecode:
0     LDC   0
2     STORE 0
4     LDC   0
6     STORE 1
8     JMP   24
10    LOAD  1
12    LOAD  0
14    ADD
15    STORE 1
17    LOAD  0
19    LDC   1
21    ADD
22    STORE 0
24    LOAD  0
26    LDC   11
28    JL    10
30    LOAD  1
32    PRINT
33    STOP
Running on an empty
memory:
Output: 55
Done!
```

No explicit labels

No references to variable names

Still the same result!

Possible refinements

- Byte encodings
  - 1byte for `ADD` instead of 4
  - 5 bytes for `LDC` instead of 8
- Compact encodings for common instructions
  - If we find that `LDC 0` occurs a lot in compiled programs, then we can add a custom instruction `LDC0` that has the same effect but only takes one byte
- Code pointers instead of single byte instructions
  - Interpreter can jump directly to code for each instruction
  - Reduces interpretive overhead by eliminating one step (at the expense of using more memory)
- Translate bytecode for frequently used functions in to native code at runtime ("just in time compilation")

Example: Java

- The engineers at Sun (now Oracle) designed:
  - A programming language, Java
  - A stack-based virtual machine called the JVM, with an instruction set of approximately 230 byte code instructions
- Compiler: `javac Main.java`
  - produces bytecode in `Main.class`
- Interpreter: `java Main`
  - Executes the bytecode in `Main.class`
  - Some JVMs use just-in-time compilation
  - Profile-driven just-in-time compilation (run-time code generation) has the potential to outperform statically compiled code (but that takes a lot of clever engineering!)

Counting programs

- Suppose a computer has A different addresses
- Each address stores one of B different "byte" values
- So, at any given point in time, the machine's memory can be in any one of $M = B^A$ possible states
- If programs are stored in memory, then there are at most M distinct programs
- (For example, on a 16 bit computer, with $A=2^{16}$, B=256, there are at most $M = 2.6 \times 10^{157,827}$ different programs)

Counting functions

- How many different functions are there that take an initial memory (one of the M possible values) and return just a single bit, either a 0 or a 1?
- Answer: there are 2^M such functions ... but at most M distinct programs
- Conclusion: there must be a lot of "uncomputable functions" (i.e., functions that are not described by a program)

Summary

In this lecture, we have seen:

- Interpreters execute programs, providing semantics for syntax
  - We can use an interpreter/evaluator to document the semantics of a language
- Interpreters can be implemented directly on program source text, on abstract syntax trees, or on intermediate code/ bytecode for an abstract/virtual machine
  - Bytecode generation follows the same principles as other forms of code generation, and helps to eliminate some causes of interpretive overhead