Languages and Compilers (SProg og Oversættere)

Lecture 14 – 2
Interpreters

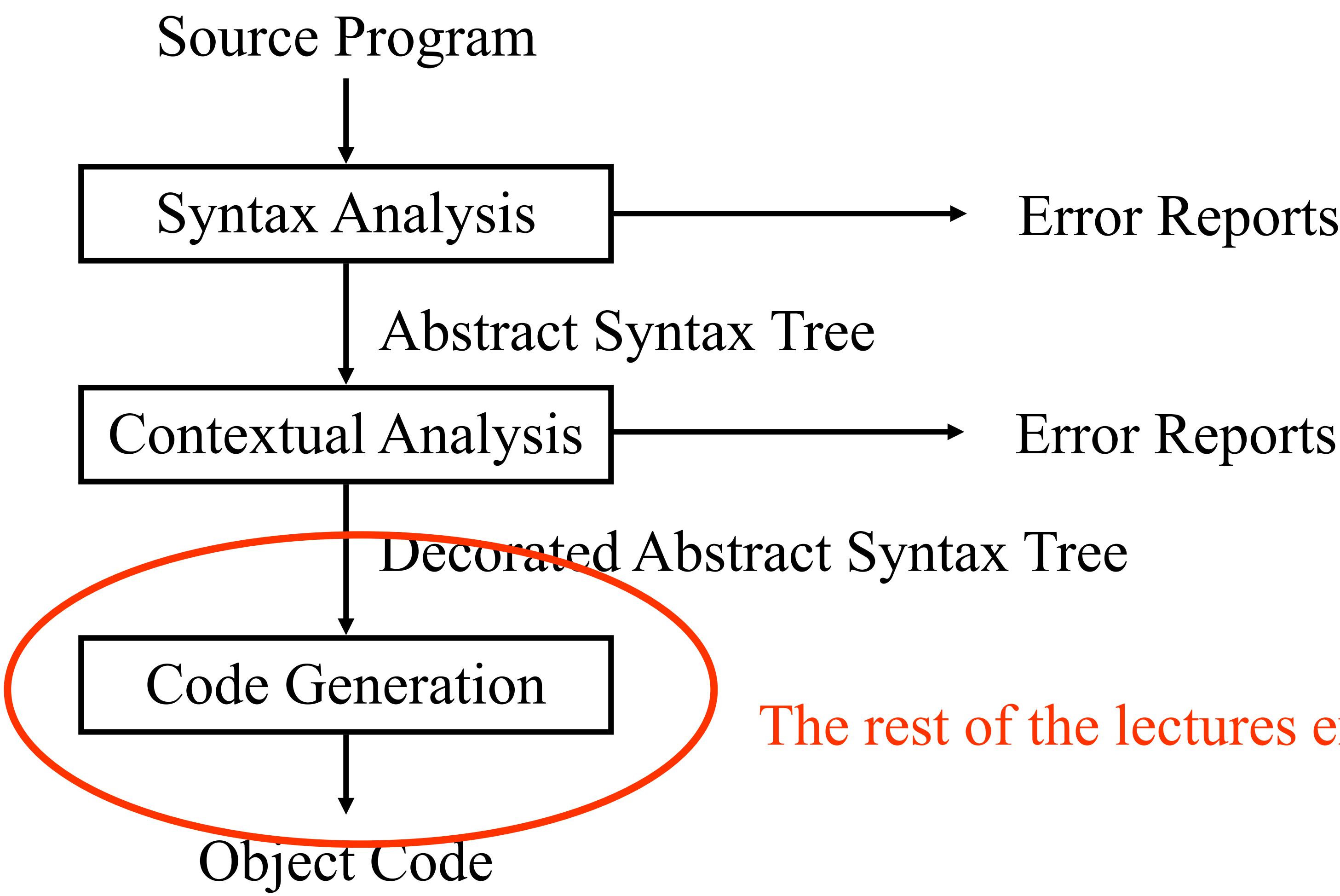
Bent Thomsen
Department of Computer Science
Aalborg University

With acknowledgement to Norm Hutchinson whose slides this lecture is based on.

Learning goals

- To get an undertanding of interpretation
  - Recursive interpretation
  - Iterative interpretation

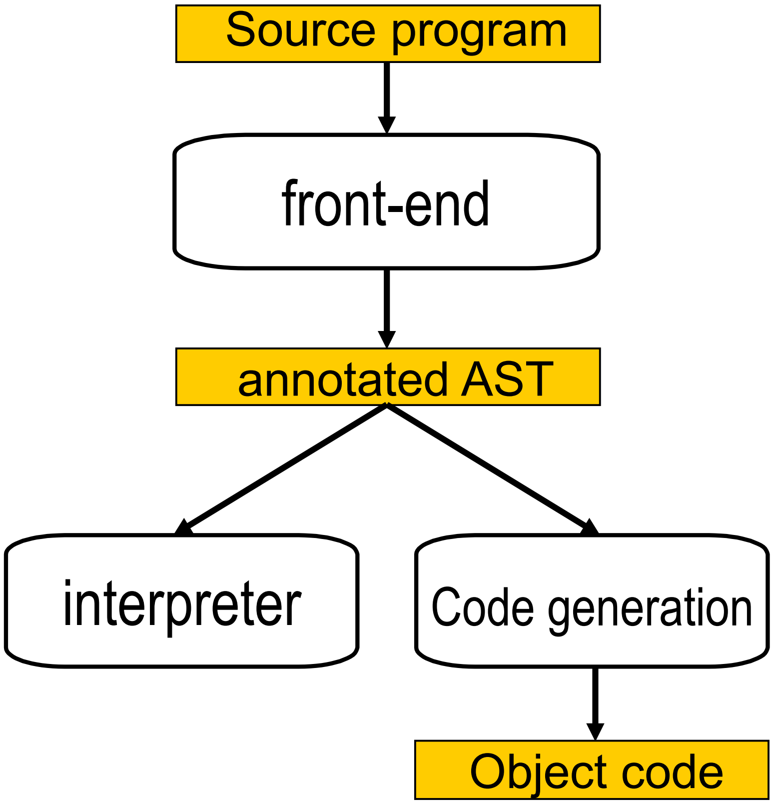
The “Phases” of a Compiler



The rest of the lectures except one

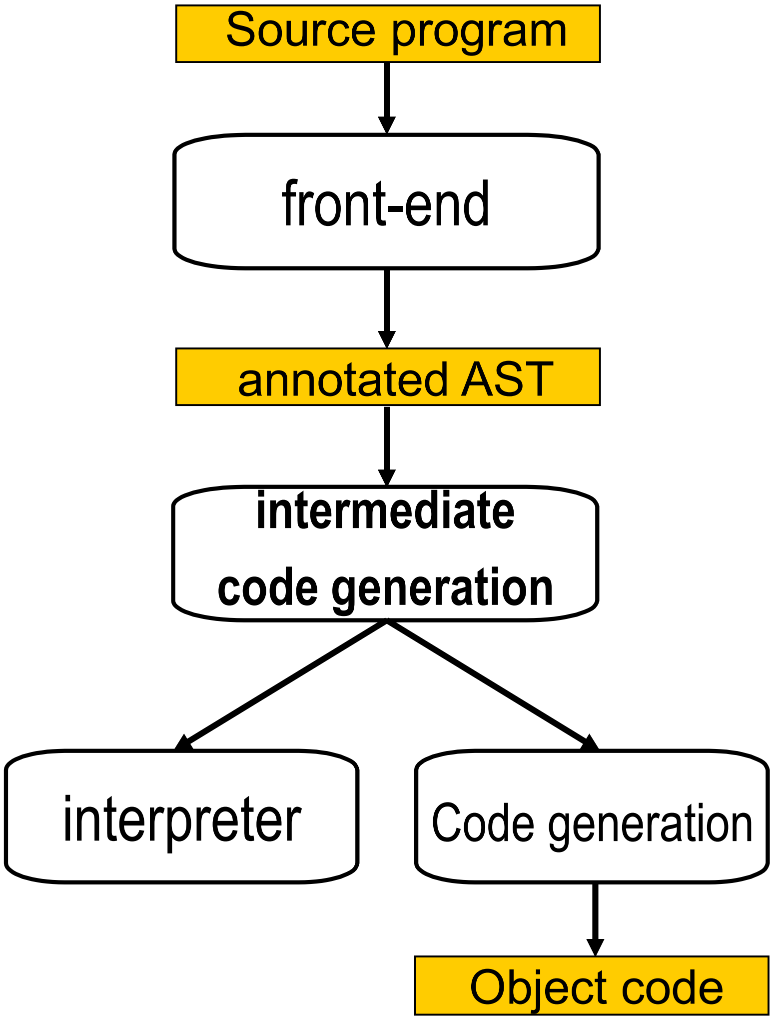
What's next?

- interpretation
- code generation
  - code selection
  - register allocation
  - instruction ordering



What's next?

- intermediate code
- interpretation
- code generation
  - code selection
  - register allocation
  - instruction ordering



Intermediate code

- language independent
  - no (or few) structured types,
    only basic types (char, int, float)
  - no structured control flow,
    only (un)conditional jumps

- linear format
  - Java byte code

The usefulness of Interpreters

- Quick implementation of new language
  - Remember bootstrapping
- Testing and debugging
- Portability via Abstract Machine
- Hardware emulation

Interpretation

- recursive interpretation
  - operates directly on the AST [attribute grammar]
  - simple to write
  - thorough error checks
  - very slow: speed of compiled code 100 times faster
- iterative interpretation
  - operates on intermediate code
  - good error checking
  - slow: 10x

Recursive interpretation

- Two phased strategy
  - Fetch and analyze program
    - Recursively analyzing the phrase structure of source
    - Generating AST
    - Performing semantic analysis
      - Recursively via visitor
  - Execute program
    - Recursively by walking the decorated AST

Change the calc.cup

```
terminal           PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer  NUMBER;
non terminal Integer expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
expr  ::= expr:e1 PLUS expr:e2
        {: RESULT = new Integer(e1.intValue()+ e2.intValue()); :}
       | expr:e1 MINUS expr:e2
        {: RESULT = new Integer(e1.intValue()- e2.intValue());  :}
       | expr:e1 TIMES expr:e2
        {: RESULT = new Integer(e1.intValue()* e2.intValue());  :}
       | expr:e1 DIVIDE expr:e2
        {: RESULT = new Integer(e1.intValue()/ e2.intValue());  :}
       | LPAREN expr:e RPAREN {: RESULT = e;    :}
       | NUMBER:e {: RESULT= e; :}
```

Recursive Interpreter for Mini Triangle

Representing Mini Triangle values in Java:

```
public abstract class Value { }

public class IntValue extends Value {
        public short i;
}

public class BoolValue extends Value {
        public boolean b;
}

public class UndefinedValue extends Value { }
```

Recursive Interpreter for Mini Triangle

A Java class to represent the state of the interpreter:

```
public class MiniTriangleState {
       public static final short DATASIZE = …;

       //Code Store
       Program program; //decorated AST
       //Data store
       Value[] data = new Value[DATASIZE];
       //Register …
       byte status;
       public static final byte  //status value
              RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

Recursive Interpreter for Mini Triangle

```
public class MiniTriangleProcesser
        extends MiniTriangleState implements Visitor {

        public void fetchAnalyze () {
                //load the program into the code store after
                //performing syntactic and contextual analysis
        }
        public void run () {
                … // run the program
        public Object visit…Command
                        (…Command com, Object arg) {
                //execute com, returning null (ignoring arg)
        }
        public Object visit…Expression
                        (…Expression expr, Object arg) {
                //Evaluate expr, returning its result
        }
        public Object visit…
}
```

Recursive Interpreter for Mini Triangle

```
public Object visitAssignCommand
                    (AssignCommand com, Object arg) {
      Value val = (Value) com.E.visit(this, null);
      assign(com.V, val);
      return null;
}

public Objects visitCallCommand
                    (CallCommand com, Object arg) {
      Value val = (Value) com.E.visit(this, null);
      CallStandardProc(com.I, val);
      return null;
}

public Object visitSequentialCommand
                    (SequentialCommand com, Object arg) {
      com.C1.visit(this, null);
      com.C2.visit(this, null);
      return null;
}
```

Recursive Interpreter for Mini Triangle

```
public Object visitIfCommand
                    (IfCommand com, Object arg) {
       BoolValue val = (BoolValue) com.E.visit(this, null);
       if (val.b) com.C1.visit(this, null);
       else       com.C2.visit(this, null);
       return null;
}

public Object visitWhileCommand
                    (WhileCommand com, Object arg) {
       for (;;) {
              BoolValue val = (BoolValue) com.E.visit(this, null)
              if (! Val.b) break;
              com.C.visit(this, null);
       }
       return null;
}
```

Recursive Interpreter for Mini Triangle

```
public Object visitIntegerExpression
                    (IntegerExpression expr, Object arg){
      return new IntValue(Valuation(expr.IL));
}
public Object visitVnameExpression
                    (VnameExpression expr, Object arg) {
      return fetch(expr.V);
}
…
public Object visitBinaryExpression
                    (BinaryExpression expr, Object arg){
      Value val1 = (Value) expr.E1.visit(this, null);
      Value val2 = (Value) expr.E2.visit(this, null);
      return applyBinary(expr.O, val1, val2);
}
```

Recursive Interpreter for Mini Triangle

```
public Object visitConstDeclaration
                    (ConstDeclaration decl, Object arg){
        KnownAddress entity = (KnownAddress) decl.entity;
        Value val = (Value) decl.E.visit(this, null);
        data[entity.address] = val;
        return null;
}
public Object visitVarDeclaration
                    (VarDeclaration decl, Object arg){
        KnownAddress entity = (KnownAddress) decl.entity;
        data[entity.address] = new UndefinedValue();
        return null;
}
public Object visitSequentialDeclaration
                    (SequentialDeclaration decl, Object arg){
        decl.D1.visit(this, null);
        decl.D2.visit(this, null);
        return null;
}
```

Recursive Interpreter for Mini Triangle

```
Public Value fetch (Vname vname) {
        KnownAddress entity =
               (KnownAddress) vname.visit(this, null);
        return data[entity.address];
}
Public void assign (Vname vname, Value val) {
        KnownAddress entity =
               (KnownAddress) vname.visit(this, null);
        data[entity.address] = val;
}
Public void fetchAnalyze () {
        Parser parse = new Parse(…);
        Checker checker = new Checker(…);
        StorageAllocator allocator = new StorageAllocator();
        program = parser.parse();
        checker.check(program);
        allocator.allocateAddresses(program);
}
Public void run () {
        program.C.visit(this, null);
}
```

Recursive Interpreter and Semantics

- Code for Recursive Interpreter is very close to a denotational semantics
- (see chapter 14 p. 211-221 in Transitions and Trees)

$$\mathcal{S}_{\mathrm{ds}}[\![x := a]\!]s = s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$\mathcal{S}_{\mathrm{ds}}[\![\mathtt{skip}]\!] = \mathrm{id}$$

$$\mathcal{S}_{\mathrm{ds}}[\![S_1 ; S_2]\!] = \mathcal{S}_{\mathrm{ds}}[\![S_2]\!] \circ \mathcal{S}_{\mathrm{ds}}[\![S_1]\!]$$

$$\mathcal{S}_{\mathrm{ds}}[\![\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2]\!] = \mathrm{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\mathrm{ds}}[\![S_1]\!], \mathcal{S}_{\mathrm{ds}}[\![S_2]\!])$$

$$\mathcal{S}_{\mathrm{ds}}[\![\mathtt{while}\ b\ \mathtt{do}\ S]\!] = \mathrm{FIX}\ F$$

$$\text{where } F\ g = \mathrm{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\mathrm{ds}}[\![S]\!], \mathrm{id})$$

Recursive Interpreter and Semantics

- Code for Recursive Interpreter can be derived from big step semantics

$$[\text{plus}_{\text{bss}}] \qquad \frac{s \vdash a_1 \rightarrow_a v_1 \quad s \vdash a_2 \rightarrow_a v_2}{s \vdash a_1 + a_2 \rightarrow_a v} \qquad \text{hvor } v = v_1 + v_2$$

```
public Object visitBinaryExpression
                    (BinaryExpression expr, Object arg){
      Value val1 = (Value) expr.E1.visit(this, null);
      Value val2 = (Value) expr.E2.visit(this, null);
      return applyBinary(expr.O, val1, val2);
}
```

Recursive Interpreter and Semantics

- Code for Recursive Interpreter can be derived from big step semantics

$$[\mathrm{ass}_{\mathrm{bss}}] \qquad \langle x := a, s\rangle \rightarrow s[x \mapsto v] \qquad \text{hvor } s \vdash a \rightarrow_a v$$

```
public Object visitAssignCommand
                    (AssignCommand com, Object arg) {
      Value val = (Value) com.E.visit(this, null);
      assign(com.V, val);
      return null;
}

Public void assign (Vname vname, Value val) {
      KnownAddress entity =
            (KnownAddress) vname.visit(this, null);
      data[entity.address] = val;
}
```

Recursive Interpreters

- Usage
  - Quick implementation of high-level language
    - LISP, SML, Prolog, … , all started out as interpreted languages
  - Scripting languages
    - If the language is more complex than a simple command structure we need to do all the front-end and static semantics work anyway.
    - Web languages
      - JavaScript, PhP, ASP where scripts are mixed with HTML or XML tags

Iterative interpretation

- Follows a very simple scheme:

```
Initialize
Do {
    fetch next instruction
    analyze instruction
    execute instruction
} while (still running)
```

- Typical source language will have several instructions
- Execution then is just a big case statement
  - one for each instruction

Iterative Interpreters

- Command languages
- Query languages
  - SQL
- Simple programming languages
  - Basic
- Virtual Machines

Mini-Shell

Script	::=	Command*
Command	::=	Command-Name Argument* end-of-line
Argument	::=	Filename
	|	Literal
Command-Name	::=	**create**
	|	**delete**
	|	**edit**
	|	**list**
	|	**print**
	|	**quit**
	|	Filename

Mini-Shell Interpreter

```
Public class MiniShellCommand {
      public String    name;
      public String[]  args;
}

Public class MiniShellState {
      //File store…
      public …

      //Registers
      public byte status; //Running or Halted or Failed

      public static final byte // status values
            RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

Mini-Shell Interpreter

```
Public class MiniShell extends MiniShellState {
        public void Interpret () {
                … // Execute the commands entered by the user
                  // terminating with a quit command
        }
        public MiniShellCommand readAnalyze () {
                … //Read, analysze, and return
                  //the next command entered by the user
        }
        public void create (String fname) {
                … // Create empty file wit the given name
        }
        public void delete (String[] fnames) {
                … // Delete all the named files
        }
        …
        public void exec (String fname, String[] args) {
                … //Run the executable program contained in the
                … //named files, with the given arguments
        }
}
```

Mini-Shell Interpreter

```
Public void interpret () {
      //Initialize
      status = RUNNING;
      do {
            //Fetch and analyse the next instruction
            MiniShellCommand com = readAnalyze();

            // Execute this instruction
            if (com.name.equals(“create”))
                  create(com.args[0]);
            else if (com.name.equals(“delete”))
                  delete(com.args)
            else if …

            else if (com.name.equals(“quit”))
                  status = HALTED;
            else status = FAILED;
      } while (status == RUNNING);
}
```

Hypo: a Hypothetic Abstract Machine

- 4096 word code store
- 4096 word data store
- PC: program counter, starts at 0
- ACC: general purpose register
- 4-bit op-code
- 12-bit operand
- Instruction set:

Op-code	Instruction	Meaning
0	STORE d	word at address d $\leftarrow$ ACC
1	LOAD d	ACC $\leftarrow$ word at address d
2	LOADL d	ACC $\leftarrow$ d
3	ADD d	ACC $\leftarrow$ ACC + word at address d
4	SUB d	ACC $\leftarrow$ ACC − word at address d
5	JUMP d	PC $\leftarrow$ d
6	JUMPZ d	PC $\leftarrow$ d, if ACC = 0
7	HALT	stop execution

Hypo Interpreter Implementation (1)

```
public class HypoInstruction {
  public byte op;         // op-code field
  public short d;         // operand field

  public static final byte
    STOREop = 0,
     ...
}

public class HypoState {
  public static final short CODESIZE = 4096;
  public static final short DATASIZE = 4096;

  public HypoInstruction [] code = new HypoInstruction[CODESIZE];

  public short [] data = new short[DATASIZE];

  public short PC;
  public short ACC;
  public byte status;

  public static final byte
    RUNNING = 0, HALTED = 1, FAILED = 2;
}
```

Hypo Interpreter Implementation (2)

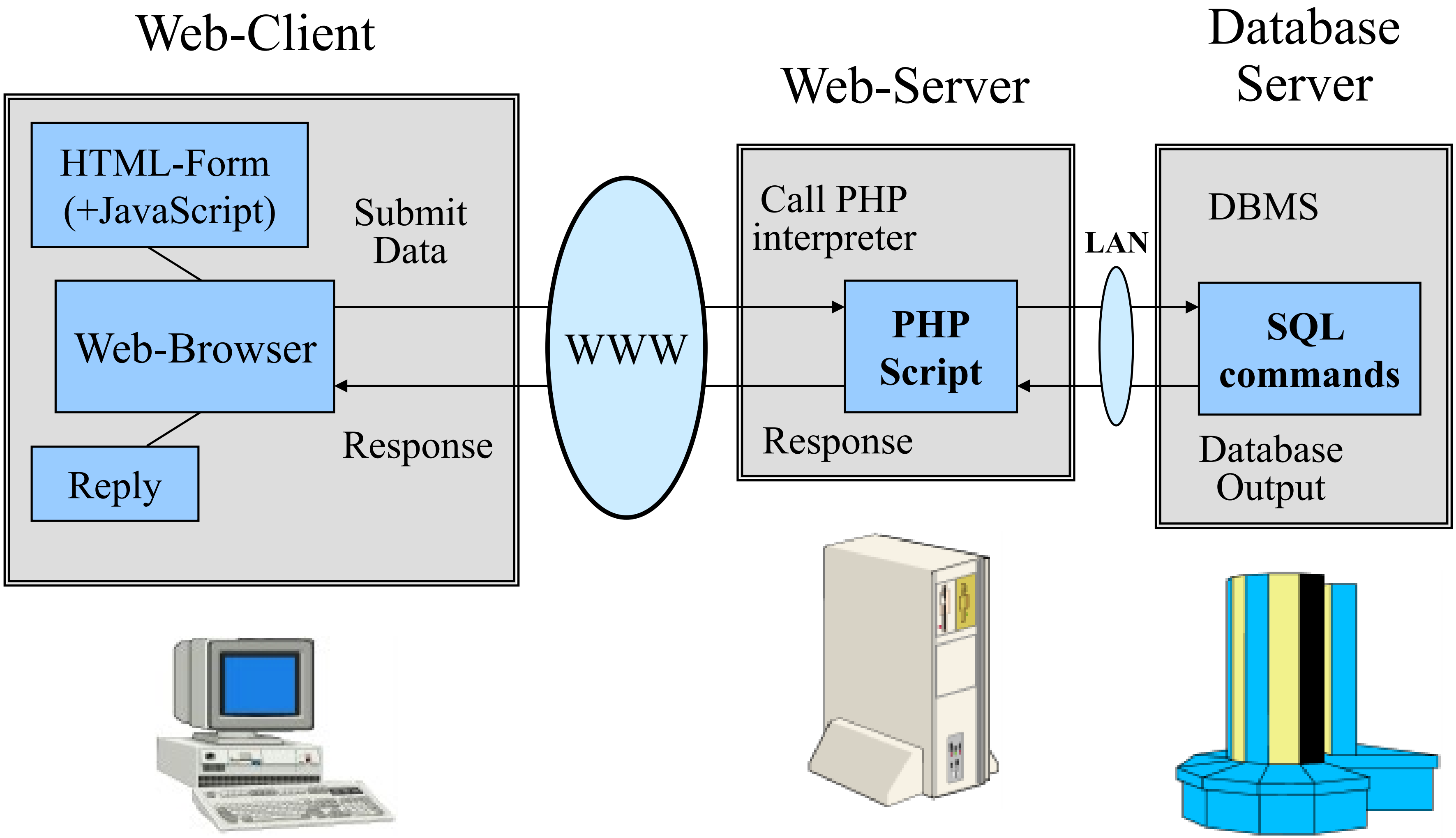
```
public class HypoInterpreter extends HypoState {
  public void load () { ... }
  public void emulate() {
    PC = 0; ACC = 0; status = RUNNING;
    do {
      // fetch:
      HypoInstruction instr = code[PC++];

      // analyse:
      byte op = instr.op;
      byte d  = instr.d;

      // execute:
      switch (op) {
        case STOREop: data[d] = ACC; break;
        case LOADop:  ACC = data[d]; break;
        ...
      }
    } while (status == RUNNING);
}
```

Other iterative interpreters

- Java Virtual Machine (JVM)
- .Net CLR
- Dalvik VM

- Note: LLVM is not a traditional virtual machine !
  - However LLVM provides an IR that can be used for further compilation

Interpreters are everywhere on the web



Interpreters versus Compilers

Q: What are the tradeoffs between compilation and interpretation?

Compilers typically offer more advantages when

- programs are deployed in a production setting
- programs are "repetitive"
- the instructions of the programming language are complex

Interpreters typically are a better choice when

- we are in a development/testing/debugging stage
- programs are run once and then discarded
- the instructions of the language are simple
- the execution speed is overshadowed by other factors
    - e.g. on a web server where communications costs are much higher than execution speed

What can you do in your project now

- Build a recursive interpreter!