Languages and Compilers (SProg og Oversættere)

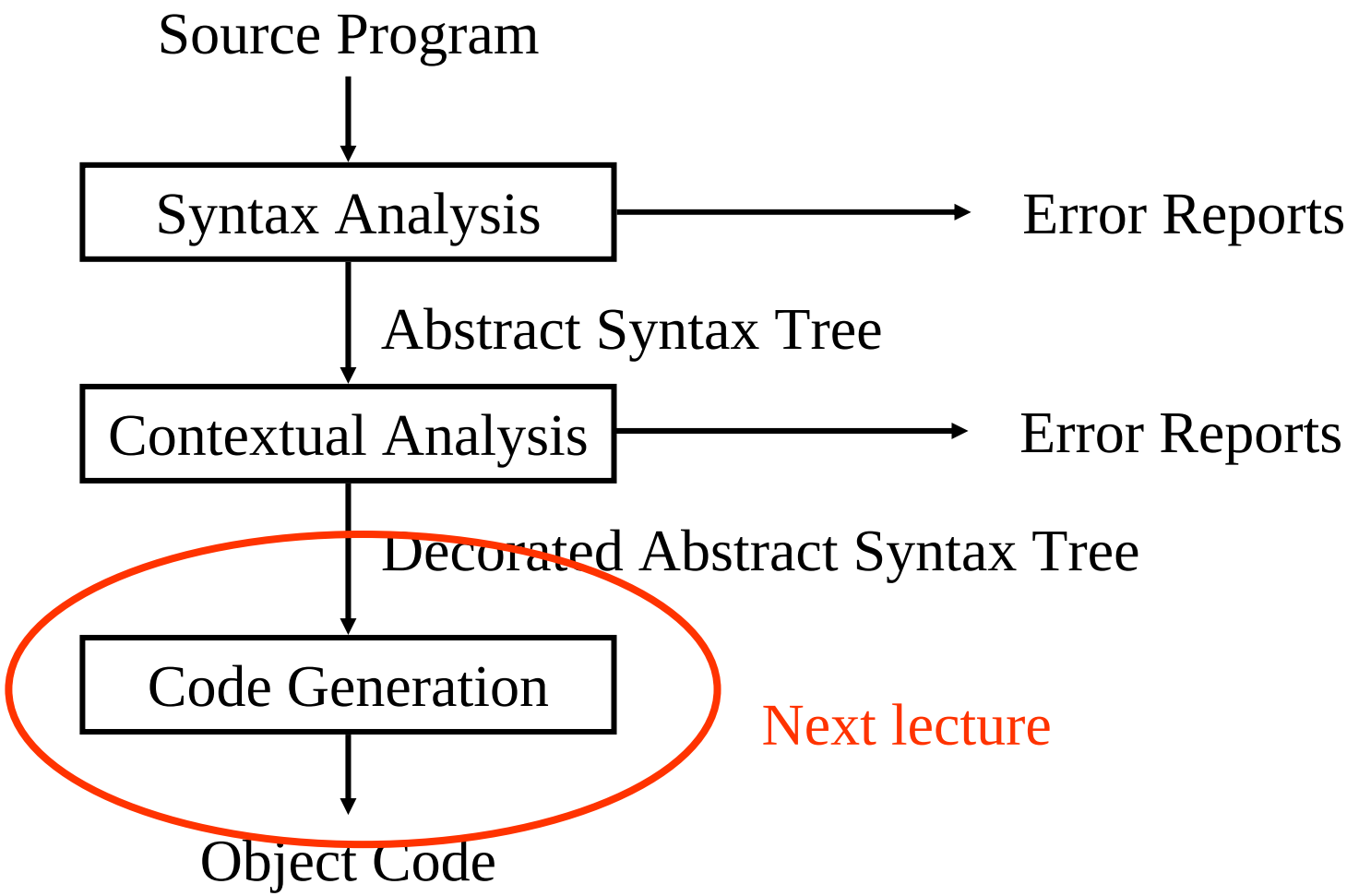
Bent Thomsen

Department of Computer Science

Aalborg University

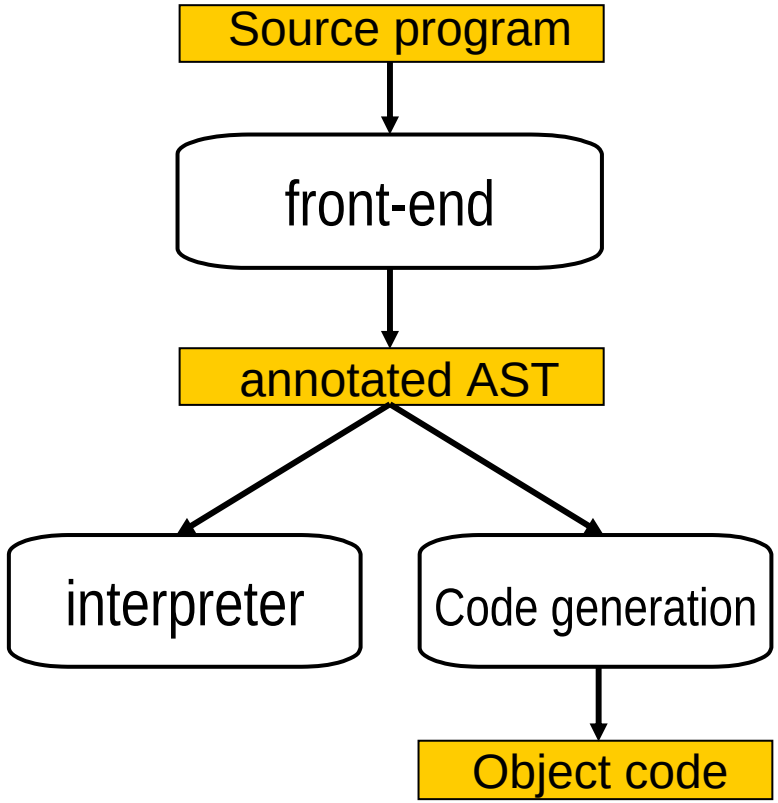
With acknowledgement to Norm Hutchinson whose slides this lecture is based on.

The “Phases” of a Compiler



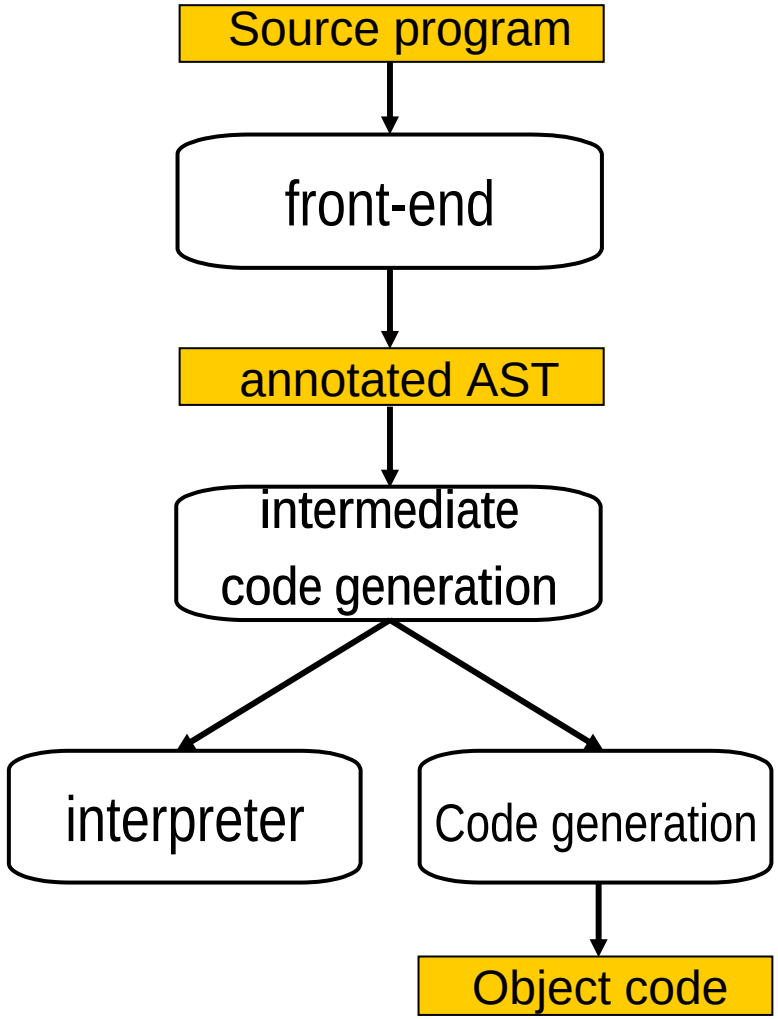
What's next?

- 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 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: 100x speed of compiled code
- iterative interpretation
  - operates on intermediate code
  - good error checking
  - slow: 10x

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)

```
1  public class HypoInstruction {
2    public byte op;        // op-code field
3    public short d;        // operand field
4
5    public static final byte
6      STOREop = 0,
7      ...
8  }
9
10 public class HypoState {
11   public static final short CODESIZE = 4096;
12   public static final short DATASIZE = 4096;
13
14   public HypoInstruction [] code = new HypoInstruction[CODESIZE];
15
16   public short [] data = new short[DATASIZE];
17
18   public short PC;
19   public short ACC;
20   public byte status;
21
22   public static final byte
23     RUNNING = 0, HALTED = 1, FAILED = 2;
24 }
```

Hypo Interpreter Implementation (2)

```
1  public class HypoInterpreter extends HypoState {
2    public void load () { ... }
3    public void emulate() {
4      PC = 0; ACC = 0; status = RUNNING;
5      do {
6        // fetch:
7        HypoInstruction instr = code[PC++];
8
9        // analyse:
10       byte op = instr.op;
11       byte d  = instr.d;
12
13       // execute:
14       switch (op) {
15         case STOREop: data[d] = ACC; break;
16         case LOADop:  ACC = data[d]; break;
17         ...
18       }
19     } while (status == RUNNING);
20 }
```

TAM

- The Triangle Abstract Machine is implemented as an itterative interpreter

Take a look at the file Interpreter.java

..\Triangle\tools-2.1\TAM\Interpreter.java

in the Triangle implementation.

TAM machine architecture

- TAM is a stack machine
  - There are no data registers as in register machines.
  - The temporary data are stored in the stack.
    - But, there are special registers (Table C.1 of page 407)
- TAM Instruction Set
  - Instruction Format (Figure C.5 of page 408)
  - op: opcode (4 bits)
    - r: special register number (4 bits)
    - n: size of the operand (8 bits)
    - d: displacement (16 bits)
- Instruction Set
  - Table C.2 of page 409

TAM Registers

No.	Mnemo.	Name	Behaviour
0	CB	code base	constant
1	CT	code top	constant
2	PB	primitive base	constant
3	PT	primitive top	constant
4	SB	stack base	constant
5	ST	stack top	changed by most instructions
6	HB	heap base	constant
7	HT	heap top	changed by heap instructions
8	LB	local base	changed by call and return
9	L1	local base 1	L1 = content(LB)
10	L2	local base 2	L2 = content(L1)
11	L3	local base 3	L3 = content(L2)
12	L4	local base 4	L4 = content(L3)
13	L5	local base 5	L5 = content(L4)
14	L6	local base 6	L6 = content(L5)
15	CP	code pointer	changed by all instructions

TAM Machine code

- Machine code are 32 bits instructions in the code store
  - op (4 bits), type of instruction
  - r (4 bits), register
  - n (8 bits), size
  - d (16 bits), displacement
- Example: LOAD (1) 3[LB]:
  - op = 0 (0000)
  - r = 8 (1000)
  - n = 1 (00000001)
  - d = 3 (0000000000000011)
- 0000 1000 0000 0001 0000 0000 0000 0011

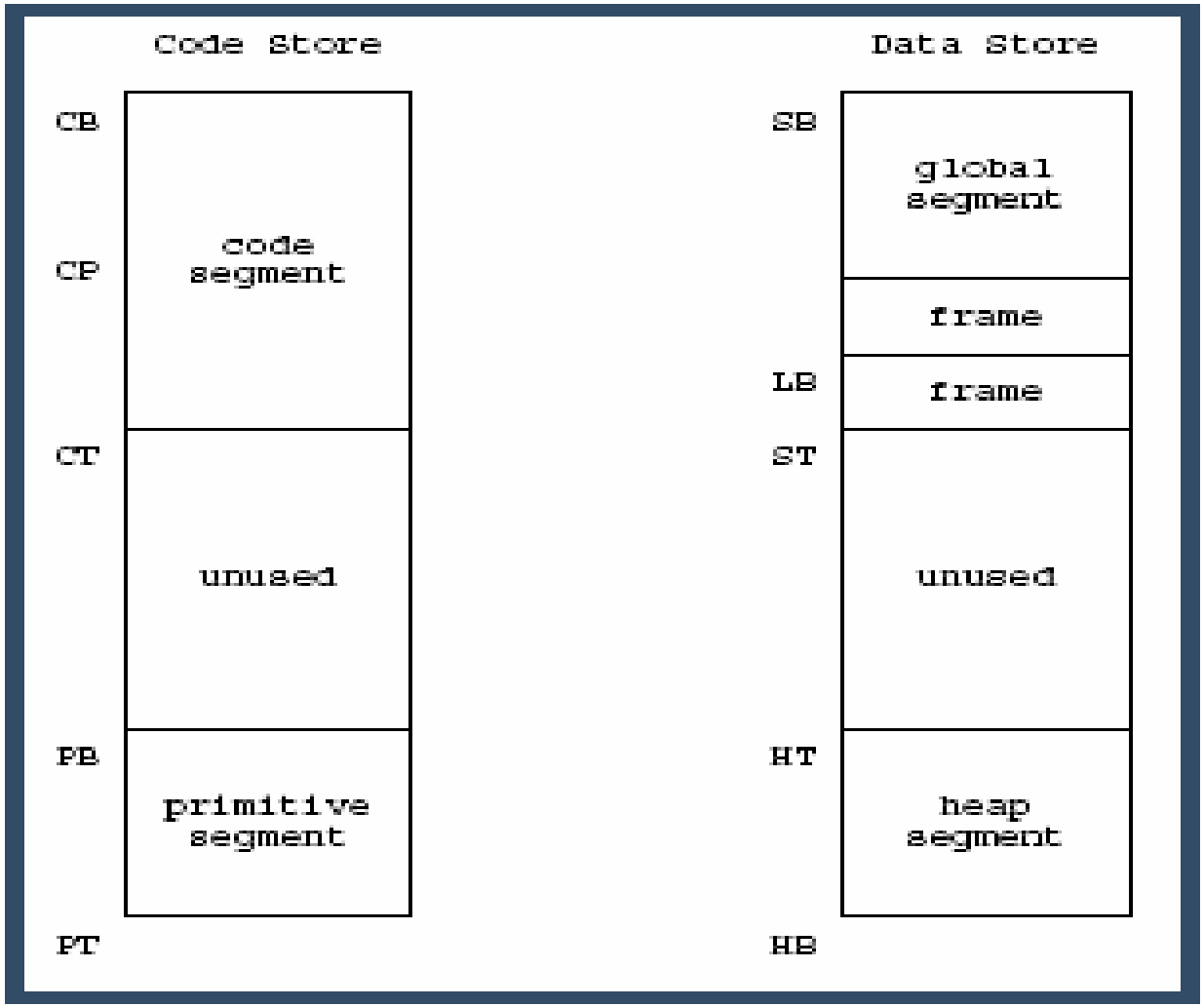
TAM Instruction set

Op.	Mnem.	Effect
0	LOAD(n) d[r]	Fetch an n-word object from the data address and push it onto the stack
1	LOADA d[r]	Push the data address onto the stack
2	LOADI(n)	Pop a data address from the stack, fetch an n-word object from that address, push it onto the stack
3	LOADL d	Push the one-word literal value d onto the stack
4	STORE(n) d[r]	Pop an n-word object from the stack, and store it at the data address
5	STOREI(n)	Pop an address from the stack, then pop an n-word object from the stack and store it at that address
6	CALL(n) d[r]	Call the routine at the code address using the address in register n as the static link
7	CALLI	Pop a closure (static link and code address) from the stack, then call the routine
8	RETURN(n) d	Return from the current routine; pop an n-word result from the stack, then pop the topmost frame, then pop d words of arguments, then push the result back
9	–	(unused)
10	PUSH d	Push d words (uninitialised) onto the stack
11	POP(n) d	Pop an n-word result from the stack, then pop d more words, then push the result back on the stack
12	JUMP d[r]	Jump to code address
13	JUMPI	Pop a code address from the stack, then jump to that address
14	JUMPIF(n) d[r]	Pop a one-word value from the stack, then jump to code address if and only if that value equals n
15	HALT	Stop execution of the program

TAM machine architecture

- Two Storage Areas
  - Code Store (32 bits words)
    - Code Segment: to store the code of the program to run
      - Pointed to by CB and CT
    - Primitive Segment: to store the code for primitive operations
      - Pointed to by PB and PT
  - Data Store (16 bits words)
    - Stack
      - global segment at the base of the stack
        - » Pointed to by SB
      - stack area for stack frames of procedure and function calls
        - » Pointed to by LB and ST
    - Heap
      - heap area for the dynamic allocation of variables
        - » Pointed to by HB and HT

TAM machine architecture



Global Variable and Assignment Command

- Triangle source code

```
! simple expression and assignment
let
    var n: Integer
in
      begin
      n := 5;
      n := n + 1
      end
```

- TAM assembler code

```
0: PUSH 1
1: LOADL 5
2: STORE (1) 0[SB]
3: LOAD (1) 0[SB]
4: LOADL 1
5: CALL add
6: STORE (1) 0[SB]
7: POP (0) 1
8: HALT
```

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

Recursive Interpreter for MiniTriangle

Representing MiniTriangle 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 MiniTriangle

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 MiniTriangle

```
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 MiniTriangle

```
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 MiniTriangle

```
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 MiniTriangle

```
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 MiniTriangle

```
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 MiniTriangle

```
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

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

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

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

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

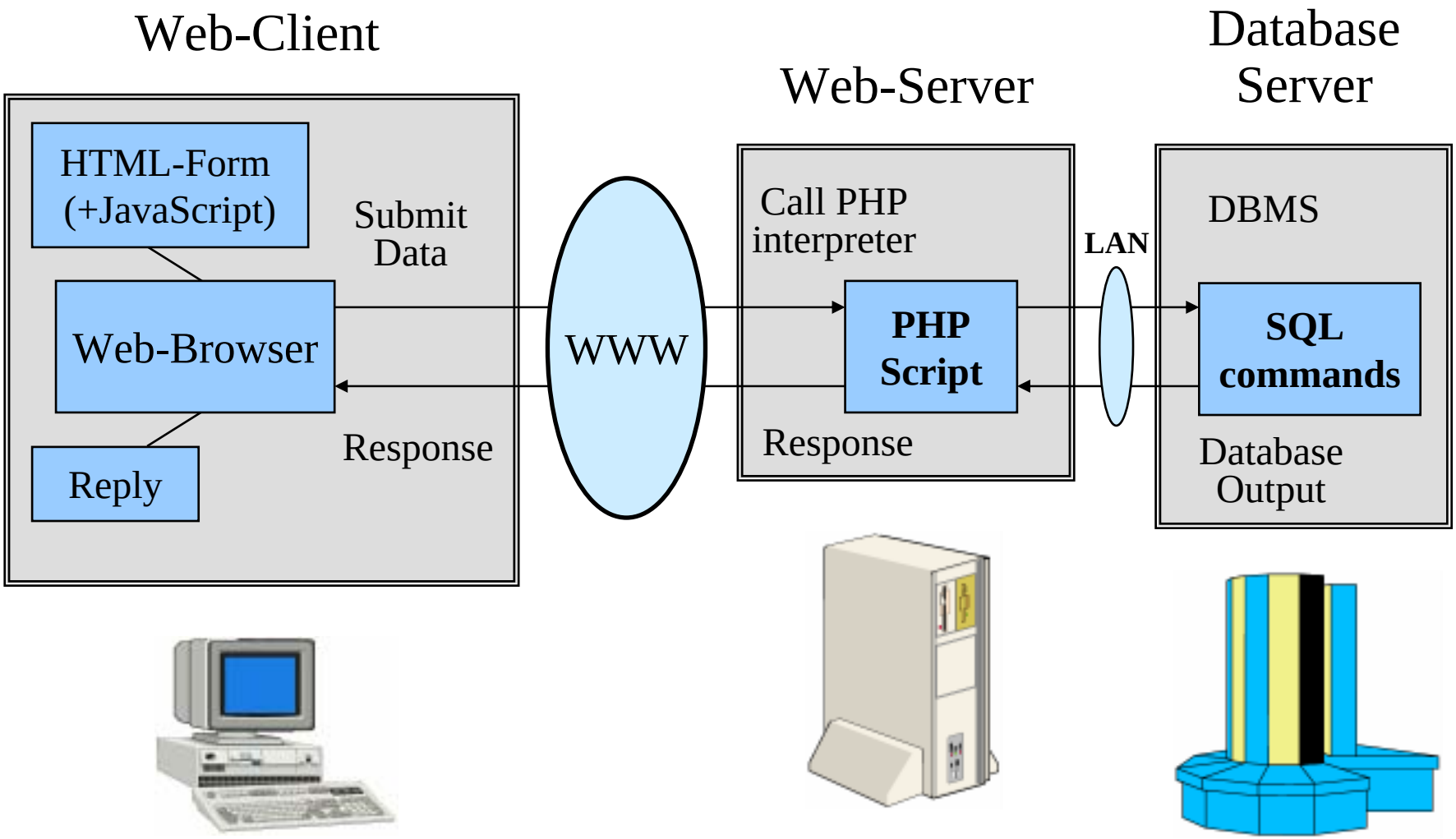
$$\mathcal{S}_{\mathrm{ds}}[\![\mathtt{while}\ b\ \mathtt{do}\ S]\!] = \mathrm{FIX}\ F$$

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

Recursive 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

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