2. Execution mechanisms Part I: Interpretation

What is an interpreter?
Take 1

- First-cut definition: An interpreter for a source language L is a mechanism for the direct execution of all programs from L

  + Allows for interpreters to be hardware as well as software (good)

    Both software and hardwares can be interpreter.

    - Example: a RISC-V chip is an interpreter of RISC-V instructions

  — Does not exclude dynamic compilation (bad..?)

What is an interpreter? Take 2

- A software interpreter for a source language L is a self-contained, complete program for the execution of all programs from L (i.e., a machine code executable that does not generate additional machine code)

Interpreter directly executes the source program, it will not produce any execute machine code

    - Excludes hardware, unfortunately should be called by using command to be called.

What is an interpreter? Take 3

Does not take into account how the rest of codes are configured.

An interpreter for a source language L is a mechanism for the direct execution of all programs from L, which executes each element of the source program in turn without reference to other elements

Some consequences

- Performance is typically uniform and predictable (e.g., every execution of the same node is the same and has the same performance, modulo micro-architectural effects in the host machine [caching, branch prediction, etc.])
- Typically slow, as there is no scope for optimization across time or program space

Contrast: compilation

looks like a function

- A compiler transforms a program in a source language S to an equivalent program in a target language T ($S \neq T$).

- It does not execute the source program at all (cf. interpretation)

Source code has been compiled to

machine code (can be seen as another language)

The machine code can be executed.

Interpretation is usually preceded by some kind of compilation

- It is rare that the source program of any non-trivial language is executed directly by an interpreter; usually it is transformed by a parser or compiler into some intermediate representation

- The IR removes lexical noise such as comments, white space and other formatting

    - Don't want to penalize commentary and formatting by increasing execution time In some case, interpreter should skip the comment, which will increase execution time.

- Lexemes are either condensed into "atoms" (identifiers, constants) or abstracted/combined into operations (keywords, operators)

- Elements are reordered into execution order (e.g., operators in an expression)

- See later sections for examples

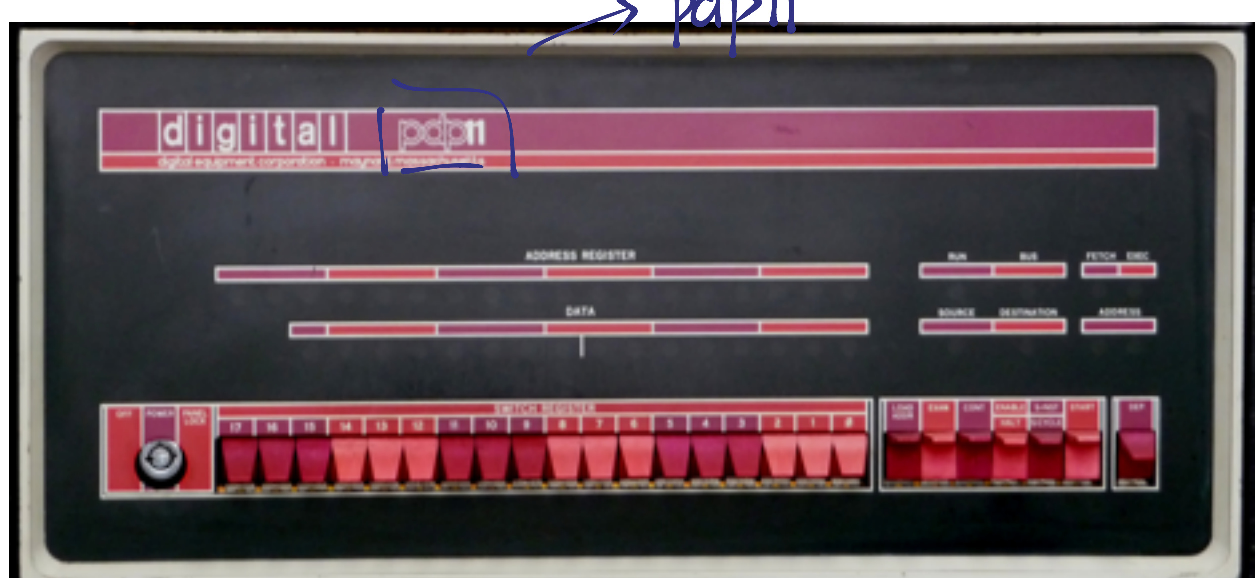
The Bottom Line

- **All** execution ultimately bottoms out at hardware interpretation

- Unless you like writing interpreters in machine code, you'll need to compile your interpreter to run it

(and you'll need one of these)

pdp11



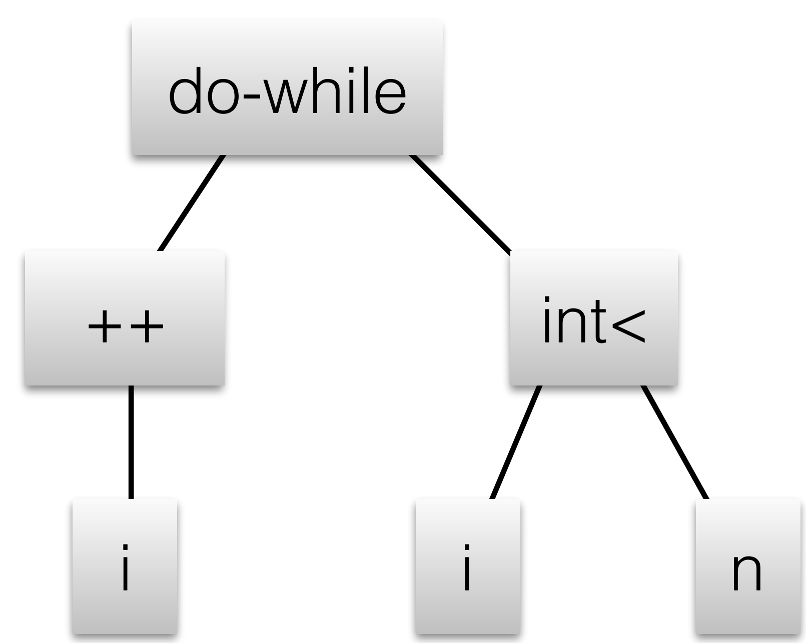
The interpreter illusion

- One common goal of an advanced VM implementation is to preserve the illusion of interpretation while maximizing performance (e.g., by compiling on-the-fly).
- Can be challenging! For example, compilers commonly reorder actions.
- The trick is to carefully define/discover what is observable, and what is not
    - Or: “cheat, but don’t get caught”
- Very similar in spirit to many micro-architectural considerations (when reordering instructions and memory effects)

Interpretation technique #1: AST interpreters for high-level languages

- AST = Abstract Syntax Tree
- The tree produced by a parser of a high-level language compiler

```
do {
  i++;
} while (i < n);
```



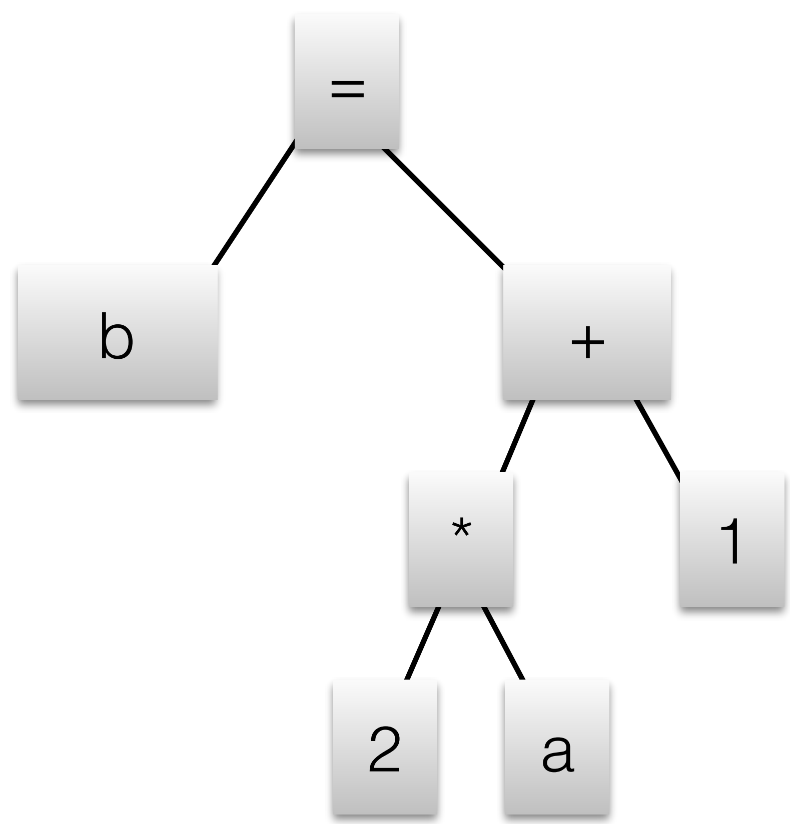
Example: interpreting a simple expression language

- Language elements: variables (single letter) holding an integer; integer constants; expressions involving simple arithmetic; assignments

3+4
b=2*a+1
x*x+x+5

example input

b=2*a+1



Simple expression AST interpreter (in C)

don't recommend like this

```
typedef struct exp Exp; /* convenient shorthand */

struct exp {
  enum {ADD, SUB, MUL, DIV, CONST, VAR, ASSGN} tag;
  union {
    int val; /* CONST */
    char var; /* VAR */
    struct {Exp *l; Exp *r} exp; /* ADD, SUB, MUL, DIV */
    struct {char var; Exp *rhs} assgn; /* ASSGN */
  } u;
};

int vars[26];

int eval(Exp *e) {
  switch (e->tag) {
  case CONST: return e->u.val;
  case VAR: return vars[e->u.var];
  case ADD: return eval(e->u.exp.l)+eval(e->u.exp.r);
  /* ditto SUB, MUL and DIV */
  case ASSGN: return vars[e->u.assgn.var]= eval(e->u.assgn.rhs); }
```

Adding statements

```
typedef struct stmt Stmt;
struct stmt {
  enum {SEQ} tag;
  union {
    struct {unsigned n; Stmt *seq[];} seq;
  } s;
};

void evals(Stmt *s) {
  switch (s->tag) {
    case SEQ: {int i;
               for (i= 0; i < s->s.seq.n; i++)
                 evals(s->s.seq.seq[i]);
               break;}
  }
}
```

Implementation using objects

- ASTs and interpreters are a natural fit for object-oriented programming.
- Each kind of node is a class
  - Use inheritance for better factoring
- Use method dispatch for evaluation

Once again, in Java

```
class ASTNode { …stuff common to all nodes…}

abstract class ExprNode extends ASTNode {
  abstract int eval();
}

class ConstNode extends ExprNode { int val; int eval() { return val; } … }

class AddNode extends ExprNode {
  ExprNode left, right;
  int eval() { return left.eval()+right.eval(); … }
…}

class VarNode extends ExprNode {
  int val;
  void set(int v) { val=v; }
  int eval() { return val; } …
…}

class AssignNode extends ExprNode {
  VarNode var; ExprNode rhs;
  int eval() { int rhsVal=rhs.eval(); var.set(rhsVal); return rhsVal; }
…}

// all trivial constructors elided
```

Control flow is easy… sometimes

- …when the semantics are local and equivalent in the implementation language. Example: consider interpreting a do-while expression.
  1. Add the tag and suitable AST node.

  ```
  struct {Stmt *do_part; Exp *expr;} dowhile;
  ```

  2. Extend the interpreter loop:

  ```
  case DOWHILE:
     do
       evals(s->s.dowhile.do_part)
     while (eval(e.dowhile.expr));
  ```

Lab assignments #1 and #2

1. Get familiar with Feeny

2. Write a Feeny AST interpreter, take some measurements. Design some extensions.

But sometimes not so easy…

- Example: **`break`** from within a loop
- Why doesn’t this work?

```
void evals(Stmt *s) {
  switch (s->tag) {
  …
  case BREAK: break;
```

evals(⟦do{…break; …. }while …⟧) ⇒

do evals(⟦{…break; ….}⟧); while (…)

evals(⟦{…break; ….}⟧) ⇒ for (…) evals(⟦break⟧);

evals(⟦break⟧) ⇒ break; *// two levels removed*

// from where it needs to be!

Can not determine where to go!

Solutions to control flow problems

- Add a break-out path for each possible enclosing node type
  - Messy when you need to return a value *and* a break-out indication if the implementation language can't return a pair
- Use host language exceptions

Implementing break in C using *longjmp*()

```
#include <setjmp.h>
jmpbuf j;
…
case DOWHILE:
  if (setjmp(j)==0)
     do
       evals(s->s.dowhile.do_part)
     while (eval(e.dowhile.expr));
  break;
case BREAK: longjmp(j,0);
```

looks like function

Evaluate will leave a "goto label" statement here.

Break, using exceptions in Java

Use try-catch structure to leave original control flow.

```
class DoWhileNode … {
  Statement do_part;
  Expr expr;

  void eval() throws BreakException {
    try {
      do {
        do_part.eval();
      } while (expr.eval()!=0);
    catch (BreakException b) {};
  }

class BreakNode … {
  void eval() throws BreakException {
    throw new BreakException(); // slow!
  }
}
```

But it will cost more time. really slow.

Performance

Let's look at what it takes to interpret b=2*a+1

evaluate tree

eval(⟦b=2*a+1⟧)

  eval(⟦2*a+1⟧)

    eval(⟦2*a⟧)

      eval(⟦2⟧) → *i*

      eval(⟦a⟧) → *j*

      $i * j \rightarrow k$

    eval(⟦1⟧) → *l*

    $k + l \rightarrow m$

  assign *m* to ⟦b⟧

- Every eval() is a call, tag fetch, switch and return
- Max stack depth is 4 frames

Performance

- [demo stepping through machine code of an example]
- How many machine instructions per node?
- Predictability of control flow?
- How many loads just to walk the tree?

Memory

- The footprint of an AST can be large, especially with 64-bit pointers
- Traversing an AST can be like a random walk through address space
- Both effects *could* be addressed by careful structure design and placement
- Serialization of the IR as a distribution format would have to address this anyway — but usually source is distributed (or Bytecode)