Interpreters

- Program executed immediately, rather than translating to machine code that can be executed later
- Advantages:
  - Quicker move from code to execution
  - Easier to write an interpreter that runs on many machines
  - Easier (often necessary) to have runtime checks
- Disadvantage:
  - Resulting code is SLOW!

Some Interpreter Approaches

- Single command parsed, executed
  - Examples: LISP, Scheme, Prolog
- Single command parsed, executed (possibly with optimization)
  - Example: most versions of SQL
- File partially compiled, resulting code interpreted
  - Example: Java
- File parsed, result executed in interpreter (possibly multiple times)
  - Examples: Perl, versions of awk

Single Command Interpreters

```
> (defun foo (x y)
    (cond ((zerop x) 0)
          (t (+ y (foo (- x 1) y)))))
 FOO DEFINED
> (foo 4 3)
 12
```

- Much like execution of commands in a shell
  - Read next command
  - Execute command
  - Repeat

Single Command Interpreters

- Generally scoping is much simpler
  - Generally a small number of scope layers
  - Example: foo declared globally, x and y are local to foo
  - As a result, most meaningful data is held globally
- Often these types of language allow you to
  - Save the current environment (including everything declared up to now)
  - Execute a set of commands as a batch
  - In some cases, allow compilation of command files

Optimization in Single Command Interpreters

- SQL (Standard Query Language for DBs) generally execute one command at a time

```
SELECT S.name, G.grade
FROM Student S, Class C, Grade G
WHERE (C.dept is "CS") and (C.num = 5641) and
          (C.cid = G.cid) and (S.sid = G.sid);
```

- Resulting query produces an initial plan for executing the query (an AST-like structure representing the query)
  - result is then optimized to take advantage of aspects of the DB (only need one cid from relation Class, look that up first, then up corresponding Grade entries, etc.)

Partial Compilation

- In Java, code is partially compiled (into byte code) that is low level, but not at machine level (since Java tries to be machine/OS independent)
- Resulting code is then interpreted at run-time (allowing the same byte code to be used across multiple platforms)
- Result is slow
  - One approach to speeding up – just in time compilers
    - As translation from byte code to actual machine code occurs keep track of translation and reuse when possible

Interpreting a Program File

- Some interpreters follow many of the early steps of a compiler (parsing/scanning, semantic analysis) but then go straight to execution rather than compiling
- Disadvantage: have to "recompile" every time
- Advantage: often can use the same "program" on multiple platforms
- Execution is generally done by interpreting the AST resulting from the semantic analysis step (as will be done in our project)

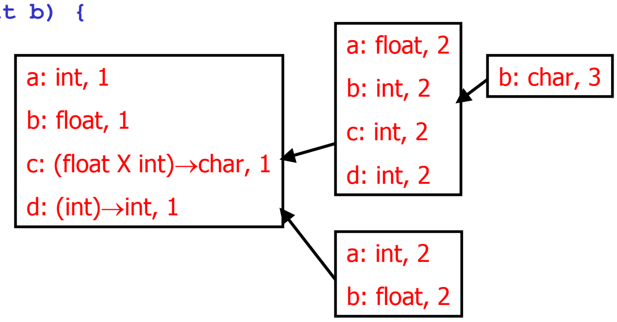
Key Issues in Executing a File

- How to manage memory/variables?
  - One approach – use a variant of the list of hashtables representation for a symbol table (keep all hashtables making a tree)
- How to execute each piece of code?
  - Surprisingly simple – often written in high level language with many similar features (e.g., implement an IF using an if command(s))
    - Need to represent variables that result from calculation/execution
  - How to deal with code that jumps out of a context (return statements, break, exceptions, etc.)?
    - Harder to deal with, often have to pass around flags used to control execution

A Scope Tree

```
Code:
int a;
float b;
char c (float a, int b) {
  int c;
  while (a < 100) {
    char b = ‘A’;
    a++;
  }
  int d;
  print(d);
}

int d (int a) {
  float b;
  c(1.0,2);
  d(3);
}
```



Simple Solution – Memory Management (No Recursion)

- Associate with each entry in the symbol table tree an appropriate amount of memory connected to that variable
- At scope entry (start of function or block), reset memory to initial value (as appropriate)
- Works if there is no recursion

Memory Management with Recursion

- May be many versions of a local variable/parameter during execution
  - Example:

```
int f1 (int v) {
  if (v = 1) return 1;
  else return v * f1(v – 1); }
```

  - One version of v for each recursive function call
    - But only the most recent v is ever active at one time
  - Idea: maintain memory locations associated with v as a stack/linked list
    - Top of stack is the most recent/current value of v
    - At scope entrance, go through entire scope, push new memory location for each variable
    - At scope exit, go through entire scope, pop the top of stack for each variable

Representing Values

- Need a mechanism for representing the result of calculations
  - Option 1: Single class with field indicating type of variable and corresponding memory for each possible type

```
enum PossibleValues { vchar, vint, vfloat, vstring};
class Value {
  PossibleValues vtype;
  void *vloc;
};
```

    - Works well for simple types, but not for complex/constructed types

Representing Values

- Option 2: Single base type with multiple possible extension types

```
class BaseValue {
};
class IntValue : public BaseValue {
  int ival;
};
```

  - Better for complex types, may want to have isa() field:

```
class BaseValue {
  virtual PossibleTypes isa() {}
};
class IntValue : public BaseValue {
  int ival;
  PossibleTypes isa() { return simple; }
};
```

  - But could simply track type during type checking (annotate node with type(s)) and determine from that

Representing Values

- Option 3: Simply pass void* to variable location, use type checking to determine context
  - Value type is known by operations applied/to be applied to it
  - More on this next

Execution Steps

- Depends partly on language
- Example:
  - LISP – declarations, function definitions, statement calls all mixed
    - Execution: get next item, execute
  - Executing a file that has been parsed depends on language, for example (Pascal):
    - Create variables and deal with global scope (set any initial global variables)
    - Execute "main" body statement (in C, C++ this would correspond to finding and executing the main function)

Execution Functions

- Generally execution done with (yet another!) AST tree traversal
- Usually implement two key functions (most nodes use only one or the other function)
  - ExprVal() – determine the value of an expression
  - StmtExec() – execute a statement (generally when the statement does NOT produce a value)

Evaluating Expressions, Simple

```
void *IntLitNode::ExprVal() {
  // ival of IntLitNode holds corresponding integer value
  void *new_node = (void *) new int(ival);
  return new_node;
}

void *IdentNode::ExprVal() {
  void *new_node = allocate space for type of variable;
  // copy current variable value to new space
  return new_node;
}
```

Evaluating Expressions, Operator

BinaryNode: fields – op, left_arg, right_arg

```
void *BinaryNode::ExprVal() {
  void *left_val =
          left_arg->ExprVal();
  void *right_val =
          right_arg->ExprVal();
  if (op is +)
   if (inputs are int) {
     *((int *) left_val) =
        *((int *) left_val) +
        *((int *) right_val);
     delete right_val;
     return left_val;
   }
   if (inputs are float) {
     *((float *) left_val) =
        *((float *) left_val +
        *((float *) right_val);
     delete right_val;
     return left_val;
   }
// NOTE: assumes coercion
//  done to guarantee all types
//  to operator the same
// Etc.
```

Evaluating Statements

IfElseNode: fields – if_expression, if_stmt, else_stmt (possibly null)

```
void IfElseNode::StmtExec() {
  void *if_value = if_expression->ExprVal();
  if (*((bool *) if_value))
    if_stmt->StmtExec();
  else
    if (else_stmt) else_stmt->StmtExec();
}
```

Executing Functions

- Allocate space for all variables in scope
- Calculate values of each argument and copy to corresponding variable
- Execute the body of the function (list of statements in the function)
- At function end eliminate the variables in the function scope (may need to copy values out depending on parameter passing mechanism)
- Complicating issue: return statements

Return Statements

- Return statements not only determine the resulting value of a function, but generally cause the function to end at that point
- Even worse, the return may be buried in several intervening scopes
- Key aspects:
  - Value returned
  - Stopping execution

Return Values

- Returns introduce complex issues for type checking (need to find function definition return is part of)
- Simple idea: for each function definition have a variable correspond to the return value of that function (add to the scope for that function)
- At function entrance, push a new copy of that variable as well
- Return statements are connected to that variable

Return Values

- Implementation (type checking/interpreting):
  - At function definition, insert into function scope a variable with a specific name not possible from normal language (e.g., __return_val), associate with variable return type of function
  - At return look up that name (__return_val) – find one for most closely nesting function
  - Type of expression corresponding to return should be the same as type for that variable
  - At function start set up variable (as with others in that scope)
  - When return statement encountered, copy variable into corresponding location

Halting Execution - Returns

- Execution generally ends after a return is encountered
  - Often buried within a further enclosing scope (or two)
- Idea: create a termination variable – pass the variable (by reference) down through the AST, setting it to true once a return is reached
  - Down side: every AST type needs to check that variable to determine if execution should continue

Further Complications

- Some languages allow statements that cause a context to end (break, continue)
  - Works somewhat like a return, but only within context (what if both return and break possible?)
    - One idea, generally only one active function for return and one active context to be broken (use two variables)
- But what about goto's and exceptions (a goto may cross multiple scope boundaries, and an exception may end several functions)
  - A lot depends on what is/is not possible in your language